Capítulo 1

Novas Tecnologias de Middleware: Rumo à Flexibilização e ao Dinamismo

Fábio M. Costa
Instituto de Informática
Universidade Federal de Goiás
fmc@inf.ufg.br

Fabio Kon
Departamento de Ciência da Computação
Universidade de São Paulo
kon@ime.usp.br

1.1 Introdução

Recentes avanços em redes de computadores viabilizaram o surgimento de ambientes computacionais cujos componentes encontram-se fisicamente distribuídos. Modernas tecnologias de redes permitem que a comunicação entre tais componentes se dê com níveis de qualidade de serviço suficientes para se emular o comportamento de ambientes centralizados. Isto, por sua vez, tornou possíveis novas formas de se estruturar aplicações e sistemas computacionais, de maneira a aproveitar os benefícios de ambientes distribuídos. Em particular, a possibilidade de particionar os diversos elementos funcionais de uma aplicação, de forma que cada um seja executado, em paralelo, em um nó de computação diferente, representa uma solução efetiva e barata para se obter um melhor desempenho. Pode-se ainda citar benefícios como compartilhamento de recursos, maior disponibilidade dos serviços da aplicação, potencial para tolerância a falhas, e melhor escalabilidade.

Entretanto, o desenvolvimento de aplicações em ambientes distribuídos apresenta complicações não existentes em sistemas centralizados. Isto se deve, principalmente, ao fato de que os diversos componentes das aplicações encontram-se dispersos em nós de computação independentes, de forma que a interação entre os mesmos requer comunicação através da rede. Ainda como conseqüência desta distribuição, outras questões importantes devem ser consideradas no desenvolvimento de aplicações, tais como: heterogeneidade (nos níveis de hardware, sistemas operacionais e linguagens de programação), independência de falhas (falhas em um determinado nó não necessariamente afetam os demais), concorrência e segurança. Na ausência de uma infra-estrutura de suporte adequada, tais questões tendem a tornar o design e implementação de aplicações distribuídas uma tarefa de complexidade extrema, limitando o seu uso em situações reais.

Tecnologias de middleware para suporte a comunicação foram então propostas com o objetivo de fornecer a infra-estrutura necessária para facilitar o desenvolvimento de aplicações distribuídas. Tipicamente, plataformas de middleware se interpõem entre as aplicações e seus sistemas operacionais subjacentes, fornecendo uma interface de programação uniforme para os desenvolvedores de aplicações. A Figura 1.1 ilustra este papel integrador que o middleware exerce em ambientes distribuídos e heterogêneos.

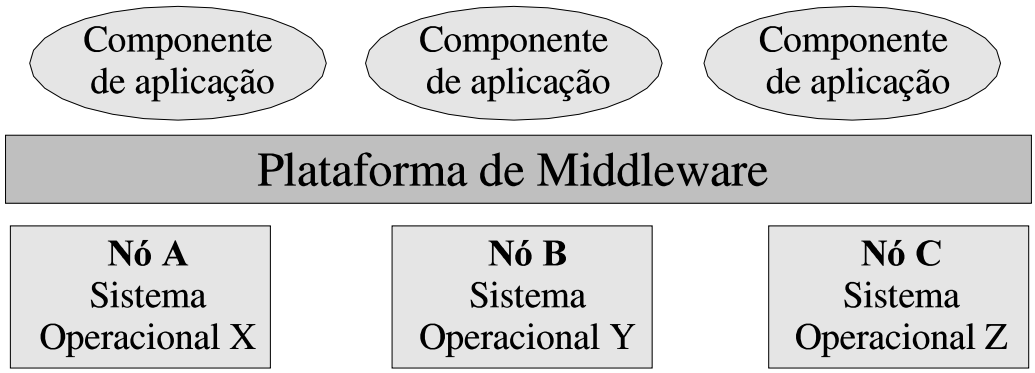

Figura 1.1: O papel integrador de plataformas de middleware

Em sua forma mais básica, tal como em tecnologias como CORBA, COM e Java/RMI, uma plataforma de middleware permite que a comunicação entre os diversos componentes em nível de aplicação seja realizada de maneira transparente do ponto de vista do desenvolvedor de aplicações. Aspectos como comunicação remota, diferentes modos de acesso e diferentes formatos de dados são escondidos do programador. Em tal cenário, o desenvolvimento de aplicações distribuídas tende, idealmente, a assumir um grau de complexidade semelhante ao caso de aplicações centralizadas, uma vez que o desenvolvedor precisa se concentrar apenas nos problemas específicos da aplicação (sendo que problemas oriundos de distribuição são tratados transparentemente pela plataforma).

Além disso, plataformas de middleware típicas oferecem também um conjunto de serviços padrão, comuns a diversas categorias de aplicações, com o objetivo de agregar valor às interações entre os componentes das aplicações e, novamente, liberar o programador da tarefa de programar tais serviços manualmente. São exemplos disto serviços como segurança, controle de transações distribuídas e resolução de nomes, entre outros.

A ampla disseminação de tecnologias de middleware e os recentes avanços obtidos nesta área tem, portanto, favorecido a adoção de arquiteturas distribuídas para aplicações e serviços. Em particular, destacam-se os modelos de arquiteturas cliente-servidor, *peer-to-peer*, e *web services*, cada um apropriado para uma determinada categoria de aplicações, tais como sistemas de informações organizacionais, disseminação e troca de informações e comércio eletrônico. Em decorrência da adoção destes tipos de arquiteturas (e das tecnologias de middleware subjacentes), ocorreu um considerável amadurecimento da área e, conseqüentemente, uma crescente sofisticação de aplicações distribuídas existentes. Ocorreu também o surgimento de novas categorias de aplicações explorando as potencialidades de ambientes distribuídos (p. ex., aplicações de multimídia distribuída e aplicações móveis). Em ambos os casos, houve um dramático aprimoramento nos requisitos em relação ao suporte de middleware.

Em primeiro lugar, notou-se a necessidade de tecnologias de middleware capazes de oferecer maior flexibilidade aos programadores de aplicações, no sentido de permitir a seleção dos serviços de suporte mais apropriados (e necessários) a cada caso específico. Isto se deveu à constatação de que não é possível obter desempenho ótimo para qualquer tipo de aplicação utilizando uma configuração fixa de middleware. Visto de outra forma, a capacidade de se configurar flexivelmente uma plataforma de middleware tem o potencial de tornar mais eficiente o suporte oferecido às aplicações, devido ao uso otimizado de

recursos tais como memória, processamento e largura de banda. Neste sentido, o uso de tecnologias de componentes de software como a base para novas tecnologias de middleware, em particular, CORBA 3, J2EE, e .NET, surge como uma clara resposta, embora em tais tecnologias, componentes sejam primordialmente usados para a configuração de aplicações (e não da própria plataforma de middleware).

Em segundo lugar, observou-se uma demanda emergente por um suporte de middleware mais dinâmico, que permitisse a adaptação, em tempo de execução, do serviço fornecido. O uso de adaptação dinâmica permite que a configuração de uma plataforma de middleware seja otimizada como resposta imediata a variações nos requisitos das aplicações e nas condições do ambiente de execução. Por exemplo, no caso de aplicações de multimídia distribuída, uma redução consistente na largura de banda disponível na rede, pode ser tratada com a adaptação dos mecanismos de comunicação (com vistas à sua otimização para as novas condições), permitindo assim reduzir ou suavizar o impacto da queda de qualidade na rede. Para aplicações que não toleram interrupções no seu funcionamento (para a realização de reconfigurações seguidas por um reinício da aplicação), adaptação dinâmica mostra-se como a única alternativa viável.

Neste contexto, o suporte oferecido pelas tecnologias de middleware atuais ainda é incipiente. Contudo, várias abordagens têm sido propostas para se preencher esta lacuna. Em comum, a maioria destas abordagens emprega técnicas de reflexão computacional, que mantêm uma representação interna da configuração do sistema, a qual pode ser manipulada de forma que mudanças na representação são refletidas no sistema real (e vice-versa), como veremos na Seção 1.6. Embora o emprego abrangente de reflexão computacional (de forma a permitir a adaptação de aspectos arbitrários da plataforma) exija uma reestruturação do design de middleware, a adoção de elementos isolados (*ad hoc*) com características reflexivas em tecnologias atualmente disponíveis aponta a abordagem geral de reflexão como um caminho concreto para a próxima geração de plataformas de middleware.

1.1.1 Visão geral do curso

O restante do texto está dividido em seis seções, que procuram cobrir desde os aspectos básicos de middleware, com ênfase em middleware orientado a objetos, até os avanços e tendências mais recentes nesta área. O tratamento dado está centrado nos requisitos gerados por aplicações emergentes de middleware em relação ao suporte configurável e adaptável por parte das plataformas subjacentes.

A Seção 2 oferece uma introdução aos fundamentos de middleware orientado a objetos. Os conceitos fundamentais de processamento distribuído aberto são revistos, provendo a base para o estudo de middleware. Plataformas de middleware são então apresentadas como uma camada de software que oferece infra-estrutura para o desenvolvimento de aplicações em um contexto de objetos distribuídos.

A Seção 3 examina tecnologias de middleware tradicionais. Ênfase é dada ao estudo de uma arquitetura representativa, CORBA, mostrando como os conceitos de middleware orientado a objetos são realizados neste padrão. Exemplos de código-fonte típico são também providos com o intuito de ilustrar os conceitos abordados. A seção também apresenta uma breve introdução às arquiteturas de Java/RMI e COM/DCOM, incluindo comparações com o modelo de CORBA.

A Seção 4 discute aplicações recentes de middleware, com ênfase nos requisitos por

elas gerados. Em particular, são discutidas aplicações que se tornaram factíveis face aos recentes avanços nas tecnologias subjacentes de ambientes distribuídos. Ênfase é dada a aplicações de multimídia distribuída e aplicações envolvendo mobilidade e ubiqüidade.

A Seção 5 apresenta e discute avanços recentes relacionados às três tecnologias de middleware estudadas na Seção 3. Em particular, CORBA 3, J2EE e .NET são considerados. Aspectos específicos destas tecnologias são descritos, seguidos por uma breve comparação. O estudo dá atenção especial às características destas tecnologias voltadas para configurabilidade da plataforma, tais como interceptadores portáteis, políticas de controle, programação declarativa de serviços, e o uso de modelos de componentes de software. Finalmente, a seção oferece uma avaliação destas tecnologias com respeito à satisfação dos requisitos de aplicações discutidos na Seção 4.

A Seção 6, introduz novas abordagens para o design e implementação de middleware voltadas para o suporte a configurabilidade e adaptabilidade dinâmica. Duas abordagens complementares são estudadas: o uso de modelos de componentes como a base para a configuração de middleware e o emprego de reflexão computacional para a provisão de adaptabilidade dinâmica. Os conceitos fundamentais destas abordagens são examinados, seguidos por exemplos de arquiteturas representativas.

Finalmente, a Seção 7 apresenta uma discussão do estado atual das tecnologias de middleware, considerando as tendências para desenvolvimentos futuros. A discussão se baseia nos resultados apresentados no curso, bem como em perspectivas para o uso futuro de middleware.

1.2 Middleware: Conceitos Básicos

Esta seção aborda aspectos fundamentais de sistemas distribuídos, com o intuito de situar o papel desempenhado por plataformas de middleware.[1] Em particular, a Seção 1.2.1 examina as características de ambientes distribuídos com alto grau de heterogeneidade, autonomia, evolução e mobilidade, genericamente denominados de ambientes de processamento distribuído aberto. Em tais ambientes, os problemas oriundos da distribuição tornam-se mais complexos em virtude da necessidade de compatibilizar, possivelmente em tempo de execução, recursos e serviços desenvolvidos de maneira independente. Plataformas de middleware devem então lidar com estes problemas adicionais, de modo a fornecer um ambiente uniforme para o desenvolvimento de aplicações distribuídas.

1.2.1 Processamento Distribuído Aberto

Nesta seção discutimos inicialmente os conceitos inerentes a sistemas distribuídos em geral. Em seguida, as características adicionais introduzidas por ambientes distribuídos abertos são consideradas, formando a base para uma análise dos requisitos para plataformas de middleware. A discussão é complementada por um estudo do Modelo de Referência para Processamento Distribuído Aberto da ISO/IEC/ITU-T (RM-ODP) [17], que normaliza a terminologia e o contexto para sistemas distribuídos em ambientes abertos.

[1] Note que estamos interessados no estudo de middleware para o suporte à comunicação distribuída no contexto de sistemas orientados a objetos. Outros tipos de middleware considerados na literatura, tais como middleware de transações e middleware orientado a mensagens estão fora do escopo deste curso.

Características Fundamentais de Sistemas Distribuídos

Um sistema distribuído pode ser definido simplesmente como "uma coleção de computadores independentes que se mostra aos seus usuários como um sistema único e coerente" [56]. Esta definição implica a existência de uma rede de computadores subjacente, bem como de uma infra-estrutura de suporte que mascara a existência desta rede do ponto de vista dos usuários e programadores de aplicações. Idealmente, o desenvolvimento de aplicações com o uso de uma tal infra-estrutura pode ser feito sem a preocupação de se lidar com os problemas advindos da distribuição do sistema em rede. Embora o estudo de técnicas para se resolver tais problemas não seja o objetivo deste curso, esta seção os considera do ponto de vista de requisitos em relação a plataformas de sistemas distribuídos.

Em primeiro lugar, os elementos que compõem um sistema distribuído podem estar fisicamente dispersos, o que pressupõe a capacidade de *interação remota* entre eles. Contudo, é importante que a infra-estrutura do sistema distribuído torne transparentes aspectos como a localização dos componentes do sistema, bem como os mecanismos subjacentes que permitem que os mesmos se comuniquem. Em particular, o usuário ou programador de aplicações não deve se preocupar em ter que localizar um componente para só então interagir com o mesmo. Além disso, a interação entre componentes locais e remotos deve se dar utilizando as mesmas primitivas de comunicação.

Como conseqüência da distribuição, os diversos componentes de um sistema distribuído podem operar de maneira *concorrente*. Isto gera a demanda por mecanismos de coordenação, no nível da infra-estrutura, capazes de manter a coerência do sistema distribuído como um todo. Novamente, é importante que tal suporte seja provido transparentemente, de forma que o programador de aplicações não necessite lidar com os mesmos de maneira explícita.

Outra característica inerente a sistemas distribuídos é a *impossibilidade de se ter uma visão precisa do estado global do sistema* num determinado instante. Isto se deve a limitações físicas no tempo de propagação de mensagens na rede, que impedem que informações sobre o estado dos componentes do sistema sejam comunicadas instantaneamente. Entretanto, pode ser necessário que a infra-estrutura do sistema distribuído ofereça mecanismos que permitam a manutenção de uma representação aproximada do estado global do sistema, de forma a dar aos usuários e programadores de aplicação a ilusão de um sistema único e coerente.

Relacionada a isto, outra característica particular a sistemas distribuídos diz respeito ao *assincronismo* das atividades de comunicação e processamento no sistema, em virtude da ausência de um relógio global que as dirija. Desta forma, não se pode assumir que atividades relacionadas dentro do sistema ocorram no mesmo instante cronológico. Neste sentido, pode ser necessário que a infra-estrutura do sistema distribuído ofereça mecanismos de sincronização (físicos ou lógicos) que introduzam comportamento síncrono, de forma a facilitar o desenvolvimento de aplicações.

Finalmente, sistemas distribuídos caracterizam-se pela possibilidade de *falhas parciais*, uma vez que um determinado componente do sistema pode parar de funcionar independentemente dos demais componentes. Conseqüentemente, o sistema como um todo pode continuar funcionando, a despeito da ausência da funcionalidade provida pelo componente falho. Cabe à infra-estrutura do sistema distribuído mascarar, se necessário, a ausência do componente falho (por exemplo, através da sua substituição), tornando assim transparente a ocorrência do problema (do ponto de vista do usuário ou do programador de aplicações).

Sistemas Distribuídos Abertos

Tradicionalmente, sistemas distribuídos eram construídos sobre plataformas de computação homogêneas, dentro de domínios organizacionais bem definidos e com administração centralizada. A disseminação de tecnologias de comunicação inter-redes, contudo, tornou possível a existência de sistemas distribuídos envolvendo múltiplos domínios organizacionais autônomos. Desta forma, novas características foram introduzidas além daquelas presentes em sistemas distribuídos convencionais.

Uma das características determinantes de sistemas distribuídos abertos refere-se à *heterogeneidade* das tecnologias de computação subjacentes. Uma vez que diferentes partes do sistema estão sob o controle de autoridades distintas, não há garantias de que cada uma utilize as mesmas tecnologias. Assim, é comum em tais sistemas a co-existência de tecnologias diferentes (ou mesmo incompatíveis) nos níveis de hardware, sistemas operacionais, protocolos de rede, linguagens de programação e mesmo aplicações.

Um ambiente distribuído aberto é também caracterizado por um elevado potencial para *evolução* e *mobilidade*. Durante o seu tempo de vida, um sistema distribuído aberto pode sofrer mudanças que envolvam a substituição de tecnologias ou funcionalidades, motivada, por exemplo, pela necessidade de maior desempenho ou por mudanças nos requisitos. Em conseqüência, acentua-se o grau de heterogeneidade no sistema. Com relação à mobilidade, é comum em ambientes distribuídos abertos o deslocamento físico das entidades envolvidas, tais como fontes de informação, nós de processamento, usuários, programas e dados.

Um sistema distribuído aberto deve, portanto, incluir, além dos mecanismos existentes em sistemas distribuídos convencionais, facilidades para acomodar as características acima descritas de maneira transparente às aplicações. Em particular, o sistema deve se encarregar de compatibilizar os diversos elementos heterogêneos que o compõem, de maneira a uniformizar métodos de acesso, formatos de dados, etc., permitindo assim que o sistema como um todo seja *interoperável*. De igual modo, um sistema distribuído aberto deve prover meios para que novos elementos sejam adicionados ou removidos, inclusive em tempo de execução, também garantindo a sua interoperabilidade com os demais elementos do sistema. Além disso, o sistema deve ser capaz de preservar a continuidade e a consistência dos serviços oferecidos face à mobilidade de seus elementos.

Nota-se, portanto, uma demanda ainda maior por infra-estruturas de suporte que ofereçam mecanismos para satisfazer tais requisitos (além daqueles existentes em sistemas distribuídos convencionais), de forma a tornar viável o desenvolvimento de aplicações em ambientes distribuídos abertos. Neste contexto, encontra-se a motivação fundamental para a adoção de plataformas de middleware como a forma de prover tal infra-estrutura. A seguir, examinamos um modelo de referência padrão para plataformas de sistemas distribuídos abertos, provendo a base para o estudo das tecnologias específicas apresentadas neste curso.

1.2.2 O Modelo de Referência ISO RM-ODP

No início da década de 90 a ISO [2] e a ITU-T [3] conjuntamente iniciaram um processo de padronização com vistas à definição de um conjunto de regras e padrões para guiar

[2] International Organization for Standardization

[3] International Telecommunication Union - Telecommunication Standardizadion Sector

e uniformizar o desenvolvimento de arquiteturas de sistemas distribuídos abertos. Como resultado deste processo, foi desenvolvido o Modelo de Referência para Processamento Distribuído Aberto (RM-ODP). O modelo identifica os conceitos e os elementos componentes de sistemas distribuídos abertos, fornecendo uma terminologia e princípios arquiteturais para o seu uso. Em particular, RM-ODP destaca-se por adotar uma visão integrada do processo de desenvolvimento de sistemas distribuídos abertos, na qual os diversos aspectos e níveis de abstração envolvidos (por exemplo, desde a análise de requisitos até a implementação e implantação) são considerados. Além disso, RM-ODP introduz um modelo orientado a objetos para sistemas distribuídos, segundo o qual os diversos elementos de um sistema são representados em termos de objetos e interações entre os mesmos.

Atualmente, RM-ODP engloba um conjunto de padrões e especificações relacionadas. Os princípios básicos de sistemas ODP são definidos em quatro especificações complementares, que formam o núcleo do modelo de referência:

1. Introdução e visão geral, que descreve a abordagem adotada, os conceitos básicos e o contexto para sistemas ODP [17]

2. Fundamentos, que define, de forma analítica, os conceitos utilizados para a descrição de sistemas de processamento distribuído arbitrários [19]

3. Arquitetura, que define a abordagem para estruturação de sistemas de processamento distribuído, bem como as características que qualificam tais sistemas como *abertos* [18]

4. Semântica arquitetural, que contém uma formalização dos conceitos de sistemas de processamento distribuído aberto definidos na parte 2 do padrão (Fundamentos) [20].

Além destas especificações básicas, o processo de padronização de RM-ODP também resultou na definição de padrões para diversos aspectos de sistemas ODP. Notadamente, destacam-se o modelo de referência para interfaces e *bindings* [22], e padrões para as funções de *trading* [21] e repositório de tipos [23].

Embora seja factível o desenvolvimento de plataformas de middleware completamente baseadas em RM-ODP, este não é o objetivo do processo de padronização. Ao invés disso, RM-ODP oferece um arcabouço genérico de conceitos arquiteturais e terminologia que permitam o surgimento de padrões em áreas específicas. RM-ODP deve então ser considerado como um meta-padrão para processamento distribuído aberto, fornecendo um ponto de referência para a definição de padrões concretos para plataformas de middleware [5]. Um exemplo claro deste uso do modelo de referência para ODP é o padrão industrial CORBA, desenvolvido pela OMG (ver Seção 1.3), que incorpora, direta ou indiretamente, diversos elementos desenvolvidos ou recomendados no contexto de RM-ODP. Além disso, outro uso do modelo de referência para ODP tem sido como uma base conceitual para se entender e avaliar tecnologias de middleware existentes. A seguir, examinamos os principais elementos do modelo de RM-ODP que podem ser utilizados para esta finalidade.

Pontos de Vista

Dada a complexidade inerente de sistemas distribuídos abertos, faz-se necessário adotar uma abordagem que permita tornar gerenciáveis os diversos problemas envolvidos. RM-ODP adota uma abordagem baseada no conceito de *pontos de vista*, que permite que os

diferentes aspectos ou áreas de interesse envolvidos no desenvolvimento de um sistema ODP sejam tratados de maneira independente. Um ponto de vista em RM-ODP pode ser pensado como uma projeção sobre o sistema como um todo, a qual considera apenas os aspectos relevantes para um determinado público de interesse, abstraindo-se de outros aspectos que não contribuam para o entendimento da projeção. Um sistema ODP é normalmente especificado segundo vários pontos de vista diferentes, cada um fornecendo uma uma visão completa do sistema de acordo com sua respectiva área de interesse. O padrão também define regras para se garantir a conformidade entre especificações do mesmo sistema definidas em pontos de vista diferentes. Como resultado, durante o design ou manutenção de um sistema ODP, pode-se escolher o ponto de vista mais apropriado, sendo que mudanças realizadas em um determinado ponto de vista têm efeito no sistema como um todo (inclusive podendo ser propagadas para outros pontos de vista que definam conceitos relacionados).

O Modelo de Referência para Processamento Distribuído Aberto define cinco pontos de vista:

1. Ponto de vista de Empresa, que considera o papel de um sistema ODP no contexto (organização ou empresa) onde ele está inserido. Este ponto de vista modela o sistema com base em contratos que expressam as obrigações dos vários participantes do ambiente distribuído, em termos de seus respectivos papéis, escopos de atuação, e de políticas que governam suas interações.

2. Ponto de vista de Informação, que modela um sistema ODP em termos dos elementos de informação que o mesmo mantém, do fluxo destas informações na organização e do processamento necessário para sua manipulação. O modelo resultante é expresso em termos de esquemas, de forma similar ao que acontece em metodologias de modelagem de dados. O modelo de informação permite, portanto, que os vários componentes de um sistema ODP tenham uma visão comum da semântica das informações manipuladas pelo sistema.

3. Ponto de vista Computacional, que permite a modelagem de um sistema ODP com base na sua decomposição funcional em termos de objetos que interagem através de interfaces bem definidas. Em RM-ODP, objetos são definidos como entidades que encapsulam comportamento e estado, sendo que o estado de um objeto somente pode ser alterado como o resultado de interações com o objeto (via interface) ou por meio de ações internas do objeto[4]. O ponto de vista computacional fornece, portanto, a base para a estruturação lógica de um sistema distribuído (embora independentemente dos mecanismos subjacentes de suporte a distribuição), em termos de unidades funcionais distintas. Um aspecto fundamental do modelo computacional refere-se ao conceito de *binding*, que modela as interações entre objetos através de suas interfaces e governa as propriedades a serem observadas por tais interações.

4. Ponto de vista de Engenharia, que aborda os mecanismos e funções específicos necessários para suportar a interação entre os objetos em um sistema ODP. Este ponto de vista permite modelar um sistema em termos da infra-estrutura e dos recursos necessários para a criação e manutenção de seus objetos componentes, bem

[4]Diferentemente de outros modelos de objetos, o ponto de vista computacional define que um objeto pode ter várias interfaces, cada uma expondo um sub-conjunto do comportamento global do objeto.

como dos mecanismos necessários para realizar as associações (*bindings*) entre as interfaces destes objetos.

5. Ponto de vista de Tecnologia, que considera o desenvolvimento de um sistema ODP em termos da identificação e implantação das tecnologias de hardware e software necessárias para atender aos requisitos e implementar as funcionalidades expressas nos demais pontos de vista.

Cada ponto de vista introduz uma linguagem específica, a qual define os conceitos e regras utilizados na modelagem e especificação de sistemas ODP de acordo com aquele ponto de vista. Entretanto, tais linguagens não implicam sintaxes ou notações específicas, mas fornecem um ponto de referência para a definição de linguagens concretas a serem usadas em padrões específicos de middleware.

Transparências de Distribuição

O Conceito de transparência é fundamental em sistemas distribuídos abertos, conforme visto na Seção 1.2.1. Transparências representam o modelo de abstração fundamental em RM-ODP, permitindo que os desenvolvedores de sistemas ODP se concentrem apenas nos aspectos relevantes, sendo que demais aspectos do sistema são mantidos de forma invisível. Em geral, o desenvolvedor necessita lidar apenas com os aspectos funcionais de um sistema ou aplicação (isto é, aspectos diretamente relacionados com a lógica da aplicação), enquanto que soluções padronizadas e reutilizáveis são empregadas de maneira transparente para a implementação dos aspectos não-funcionais.

RM-ODP define uma série de transparências de distribuição, descritas abaixo.

- Transparência de localização: abstrai a localização física de um objeto ou interface em um ambiente distribuído, normalmente através de nomes ou identificadores lógicos, os quais são utilizados em lugar de endereços físicos. Juntamente com a transparência de acesso, constitui a base essencial para um sistema distribuído aberto, pois permitem a interação entre objetos em ambientes heterogêneos.

- Transparência de falhas: mascara, do ponto de vista de um objeto, a ocorrência de falhas (bem como a possível recuperação) em outros objetos do sistema ou mesmo do próprio objeto em questão. Introduz-se assim um nível de tolerância a falhas sem a necessidade de intervenção direta do desenvolvedor do sistema.

- Transparência de migração: mascara, do ponto de vista de um objeto, a capacidade de o sistema mudar a sua localização física.

- Transparência de relocação: mascara a mudança de localização de uma interface em relação a objetos que dela façam uso.

- Transparência de replicação: mascara o uso de um grupo de objetos compatíveis para o suporte a uma única interface.

- Transparência de persistência: permite que um objeto seja desativado e reativado de maneira transparente ao programador (de forma que seu tempo de vida seja independente do programa que o implementa).

- Transparência de acesso: abstrai as diferenças entre métodos de acesso a objetos e interfaces em um sistema distribuído heterogêneo. Isto é feito através da compatibilização das diferentes representações de dados e mecanismos de invocação presentes no sistema.

- Transparência de transação: mascara a existência de mecanismos de coordenação distribuída utilizados para manter a consistência das interações em uma configuração de objetos.

Note-se que a abordagem para transparência empregada em RM-ODP é seletiva, no sentido de que um determinado sistema ODP é livre para escolher o conjunto de transparências de que necessita. Desta forma, pode-se evitar o custo adicional relacionado ao suporte de transparências desnecessárias. Na prática, desenvolvedores de sistemas ODP podem selecionar as transparências apropriadas através de programação declarativa, sendo que requisitos específicos de transparência podem ser declaradas para objetos individuais, grupos de objetos, ou para um sistema ODP como um todo.

Funções ODP

Com o objetivo de implementar a infra-estrutura básica necessária para sistemas distribuídos abertos, bem como as transparências especificadas, RM-ODP define um número de funções comuns, divididas em quatro grupos:

- Funções de gerenciamento, que oferecem suporte para as diversas fases do ciclo de vida de objetos (criação, desativação, reativação, movimentação e recuperação). Oferecem também facilidades para o gerenciamento de recursos no sistema.

- Funções de coordenação, que consideram os requisitos de coordenação de atividades distribuídas e o gerenciamento de grupos de objetos. São exemplos disto as funções de notificação de eventos, *checkpointing* e recuperação, as funções de grupo de objetos e de replicação, a função de migração, e as funções de transação.

- Funções de repositório, que oferecem serviços comuns para a manutenção e manipulação de informações no contexto de um sistema ODP. Em particular, destacam-se as funções de repositório de tipos (que mantém informações sobre os tipos de interfaces existentes no sistema, bem como relacionamentos entre os mesmos), e a função de *trader* (que facilita a publicação de informações sobre os serviços disponíveis em um ambiente distribuído aberto).

- Funções de segurança, que definem aspectos como controle de acesso, auditoria, autenticação, integridade, confidenciabilidade e gerenciamento de chaves.

Em geral, tais funções são realizadas através de objetos (ou grupos de objetos), os quais são especificados utilizando os próprios conceitos definidos em RM-ODP (por exemplo, através dos vários pontos de vista e do re-uso de outras funções ODP mais básicas).

1.3 Tecnologias Tradicionais de Middleware

Para uma melhor compreensão dos recentes avanços em middleware, é necessário um entendimento básico dos principais padrões e tecnologias nos quais estes avanços se baseiam.

Isto permite conhecer os mecanismos e serviços fundamentais para suporte a sistemas distribuídos abertos, os quais servem de base para os recursos mais sofisticados introduzidos recentemente. Permite também um melhor entendimento das limitações que tecnologias convencionais de middleware apresentam em relação a novos requisitos das aplicações e à evolução das tecnologias subjacentes. Desta forma, espera-se prover uma motivação para o estudo das tecnologias recentemente introduzidas, bem como para a busca por futuros avanços.

O estudo se concentra nas três tecnologias ou padrões de middleware mais amplamente difundidas atualmente, a saber CORBA, Java RMI e DCOM, com ênfase na primeira delas. Uma análise crítica destas tecnologias é também apresentada, tomando-se o modelo RM-ODP como ponto de referência.

1.3.1 CORBA

CORBA (*Common Object Request Broker Architecture*) é o resultado de um esforço iniciado em 1989 pelo consórcio OMG (*Object Management Group*) que reúne desenvolvedores e usuários de middleware, com o objetivo de padronizar tecnologias para objetos distribuídos. O processo utilizado pelo OMG consiste, predominantemente, na submissão, avaliação e adoção de tecnologias comprovadas no âmbito da indústria e na sua adaptação no contexto padrão de CORBA. No decorrer de seu desenvolvimento, a padronização de CORBA foi também alinhada ao modelo de referência RM-ODP, sendo que vários elementos e conceitos deste padrão foram adotados no contexto de CORBA e vice-versa.

A especificação de CORBA define um modelo de objetos distribuídos e uma arquitetura de serviços para o suporte a este modelo. Ênfase é dada aos aspectos de interoperabilidade e portabilidade em ambientes heterogêneos, de forma que aplicações distribuídas sejam independentes de linguagens e plataformas de hardware e software específicas. CORBA não se refere a uma plataforma de middleware em particular, mas fornece as definições padrão necessárias para que diferentes fabricantes de middleware desenvolvam plataformas capazes de interoperar em ambientes distribuídos abertos. Isto se dá através da especificação de padrões para as interfaces e a semântica dos diversos componentes e serviços do middleware, sendo que, obedecidas tais especificações, fabricantes individuais têm liberdade para definir os detalhes internos de cada implementação. A versão estável mais recente da especifição CORBA é a 2.6 [42]. No momento, o OMG está finalizando a especificação de CORBA 3 [48] que traz uma série de novas funcionalidades como, por exemplo, o modelo de componentes de CORBA (ver Seção 1.5.2).

O Modelo de Objetos

CORBA adota um modelo de objetos distribuídos baseado nos conceitos de cliente e servidor e de *objetos remotos*. Neste modelo, a implementação de um objeto reside no espaço de endereçamento de um servidor, enquanto que a interface através da qual o objeto oferece seus serviços pode ser instanciada e invocada remotamente.

Um objeto em CORBA possui um identificador único, implementado através do conceito de *referência de objeto* (que também inclui, embora de forma opaca ao cliente, informações sobre a localização do objeto, protocolos de acesso, etc.). Através de uma referência de objeto, clientes (que não necessariamente precisam ser objetos) podem chamar métodos (também conhecidos como operações) em um objeto CORBA específico, de maneira transparente em relação aos aspectos de distribuição.

Objetos (ou, mais precisamente, tipos de objetos) são especificados através de suas interfaces, utilizando-se a *Linguagem de Definição de Interfaces* (IDL) de CORBA. IDL provê uma sintaxe precisa para a definição dos aspectos que constituem de uma interface, tais como operações, parâmetros, atributos e exceções, bem como dos tipos de dados básicos necessários (por exemplo, inteiro, string, enumeração, struct e seqüência). IDL define, portanto, o núcleo do modelo de objetos (ou meta-modelo) de CORBA. Aspectos de semântica de interfaces (no nível de aplicações), contudo, não são representáveis em IDL.

O modelo de programação de CORBA é complementado por um conjunto de mapeamentos de IDL para linguagens de programação específicas, nas quais clientes e objetos são implementados. Por exemplo, mapeamentos padrão foram definidos para as linguagens C, C++, Java, Smalltalk, Python e COBOL. Um mapeamento de linguagem permite que os conceitos definidos em IDL sejam corretamente representados na linguagem de implementação escolhida pelos programadores de objetos e clientes. Desta forma, permite-se a interoperabilidade entre clientes e objetos implementados em linguagens diferentes, uma vez que as interações entre os mesmos são representadas segundo o modelo comum especificado em IDL.

Uma vez definida uma especificação IDL, o programador utiliza-se de um *compilador IDL* para gerar o código-fonte para uma linguagem de programação específica. O código gerado é uma tradução das interfaces definidas na especificação IDL para a linguagem de programação na qual o programador implementará o cliente e/ou o servidor. Por exemplo, se o objetivo é implementar um servidor em C++ e um cliente em Java, o programador passa o mesmo arquivo IDL por dois compiladores diferentes para gerar o mapeamento para C++ e para Java. O compilador IDL gera dois componentes principais: o *stub*, que é ligado em tempo de compilação ao cliente que irá utilizar a interface e o *skeleton*, que é ligado, também em tempo de compilação, ao servidor que implementará a interface.

Exemplos de definições em CORBA IDL são mostrados na Figura 1.2 que ilustra os conceitos mais importantes disponíveis na linguagem através de um cenário de aplicação simplificado, que represeta uma biblioteca hipotética.

O exemplo ilustra a definição de quatro *tipos de objetos* através de quatro interfaces diferentes: `Pessoa`, `Usuario`, `Livro` e `EstoqueDeLivros`, todas elas incluídas em um único módulo denominado `Biblioteca`. O exemplo também mostra como tipos auxiliares podem ser definidos (no caso, estruturas, seqüências, e enumerações). A definição da interface `Usuario` ilustra o uso de herança em CORBA IDL, através do qual objetos deste tipo também herdam as características definidas na interface base `Pessoa`. Desta forma, CORBA permite uma forma de polimorfismo na qual objetos implementando uma interface derivada podem substituir objetos que implementam a interface base. Note, que estamos abordando aqui apenas herança de interfaces e não herança de implementação. Quando o programador escreve as classes concretas que implementarão as interfaces, ele pode decidir se usa ou não herança de implementação.

Interfaces, como mostrado no exemplo, podem possuir atributos, que definem características dos objetos que possuem aquela interface. Note que esses atributos não são equivalente a variáveis membras de uma classe em uma linguagem como C++. Na verdade, o compilador de IDL mapeia a definição dos atributos para um par de métodos de acesso, *get* e *set*, ou apenas *get* no caso de atributos *readonly*. Cabe então ao programador escrever a implementação dos métodos de acesso da forma que for mais apropriada.

Finalmente, interfaces têm sua funcionalidade definida através de *operações* (tais como

```
module Biblioteca {
    exception NaoDisponivel {};
    exception MultaNaoPaga {};
    struct Endereco {
        string rua;
        short  numero;
        string complemento;
        string cidade_estado;
    };
    interface Pessoa {
        readonly attribute string nome;
        attribute Endereco endereco;
    };
    typedef sequence<string> TitulosDeLivros;
    interface Usuario : Pessoa {
        attribute TitulosDeLivros livros_emprestados;
        void livro_solicitado (in string titulo);
        void registra_multa (in string livro, in float valor);
    };
    interface Livro {
        enum Estado { emprestado, disponivel, reservado, desaparecido};
        readonly attribute string titulo;
        readonly attribute string autor;
        readonly attribute Estado estado;
        boolean reserve (in Usuario usuario);
    };
    struct Exemplares {
        Livro livro;
        short numero_de_exemplares;
    };
    typedef sequence<Exemplares> Prateleira;
    typedef sequence<Prateleira> Prateleiras;
    interface EstoqueDeLivros {
        readonly attribute Prateleiras prateleiras;
        Livro busca_livro (in string titulo);
        void adiciona_livro (in Livro livro, in Prateleira local);
        void retira_livro (in Usuario usuario, in Livro livro)
              raises (NaoDisponivel, MultaNaoPaga);
        void devolve_livro (in Usuario usuario, in Livro livro)
              raises (MultaNaoPaga);
    };
};
```

Figura 1.2: Examplos de definições em CORBA IDL

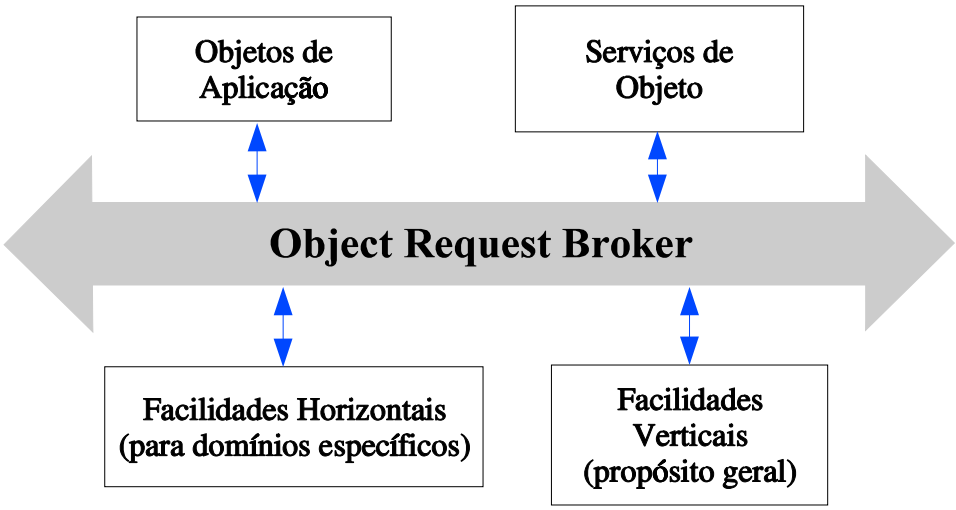

Figura 1.3: A arquitetura de gerenciamento de objetos do OMG

`busca_livro` e `retira_livro` na interface `EstoqueDeLivros`), os quais devem ser contemplados pela respectiva implementação de objeto. Dois tipos básicos de operações são possíveis em IDL: *requisição e resposta*, ou *request-reply* (com semântica de chamada síncrona, do tipo *no-máximo-uma-vez*) e *unidirecional*, ou *oneway* (com semântica de chamada assíncrona, ou seja, o cliente não é bloqueado como resultado da requisição e, portanto, não recebe nenhuma resposta do objeto remoto).

Note que IDL não especifica nada sobre o funcionamento interno dos objetos, ficando a cargo do desenvolvedor definir detalhes como os algoritmos e mecanismos utilizados para implementar cada operação e a linguagem de implementação. A única restrição é que as características e serviços definidos na interface do objeto sejam corretamente implementadas.

A Arquitetura de Objetos da OMG

CORBA forma parte de uma arquitetura para objetos distribuídos desenvolvida pela OMG e denominada Object Management Architecture (OMA) [41]. A Figura 1.3 provê uma visão geral desta arquitetura, mostrando as principais categorias de elementos que dela fazem parte.

No centro da OMA está o *Object Request Broker* (ORB), cuja definição é objetivo primordial da especificação de CORBA. O ORB é a peça chave no suporte ao modelo de objetos de CORBA. Ele é responsável por efetuar a comunicação entre os objetos distribuídos, de maneira transparente em relação à distribuição física e à heterogeneidade do sistema. Uma discussão detalhada da arquitetura do ORB é apresentada abaixo.

Os demais componentes da arquitetura OMA são geralmente implementados através de objetos CORBA, os quais se comunicam e oferecem seus serviços através do ORB. **Objetos de aplicação** representam os blocos funcionais dos quais são constituídas as aplicações em um ambiente distribuído. Como o nome indica, estes objetos são específicos

de cada aplicação, sendo que sua definição está fora do escopo do padrão. O único requisito em relação aos mesmos é que tenham suas interfaces especificadas em CORBA IDL e que sejam instanciados como objetos CORBA.

Serviços de objeto (ou *Common Object Services*) representam blocos básicos para a construção dos aspectos não-funcionais das aplicações (ou seja, aqueles aspectos que não fazem parte da lógica específica da aplicação). Serviços de objeto são, portanto, independentes de aplicação. Alguns dos principais serviços de objeto já definidos são enumerados a seguir.

- Serviço de Nomes, pelo qual objetos podem receber nomes de alto nível (isto é, legíveis ao ser humano), os quais são mapeados para os identificadores de objetos de uso interno do ORB. Por exemplo, um objeto associado a uma impressora pode receber o nome "`/blocoB/impressoras/hp2`" para representar que a impressora localizada no Bloco B e de nome hp2 pode ser acessada através deste objeto.

- Serviço de Negociação (*Trading*), que permite que objetos divulguem ofertas de serviços e atributos associados a eles, bem como que clientes consultem a disponibilidade de serviços com base em seus atributos. Por exemplo, um cliente pode utilizar este serviço para localizar uma impressora que imprima arquivos Postscript, que seja colorida, que imprima mais do que 10 páginas por minuto e cujo custo de impressão seja o mínimo possível dentre todas as ofertas satisfazendo estes critérios. A especificação deste serviço foi definida de acordo com o padrão desenvolvido no contexto de RM-ODP.

- Serviço de Transações, que permite a constituição de transações distribuídas envolvendo múltiplos objetos.

- Serviço de Persistência, que fornece facilidades para o armazenamento persistente de objetos de forma transparente às aplicações.

- Serviços de Segurança, que fornecem mecanismos para autenticação, autorização, auditoria e comunicação segura, em ambientes distribuídos heterogêneos.

Além de vários serviços de uso geral como os descritos acima, a OMA também define uma série de interfaces para determinados domínios de aplicação específicos (*Domain Interfaces* ou *Vertical Facilities*), tais como serviços de suporte para aplicações de comércio eletrônico, aplicações médicas e aplicações financeiras. Em geral, a implementação destas interfaces faz uso dos Serviços de Objeto definidos no padrão.

Principais Elementos de CORBA

CORBA define uma arquitetura de ORB constituída por vários elementos necessários para implementar os diversos aspectos do modelo. Estes elementos são padronizados em termos de suas interfaces e do comportamento deles esperado. Seus detalhes de implementação interna, contudo, são deixados a cargo de cada fabricante. Uma visão geral da arquitetura de um sistema CORBA é fornecida na Figura 1.4, sendo que seus elementos são descritos a seguir.

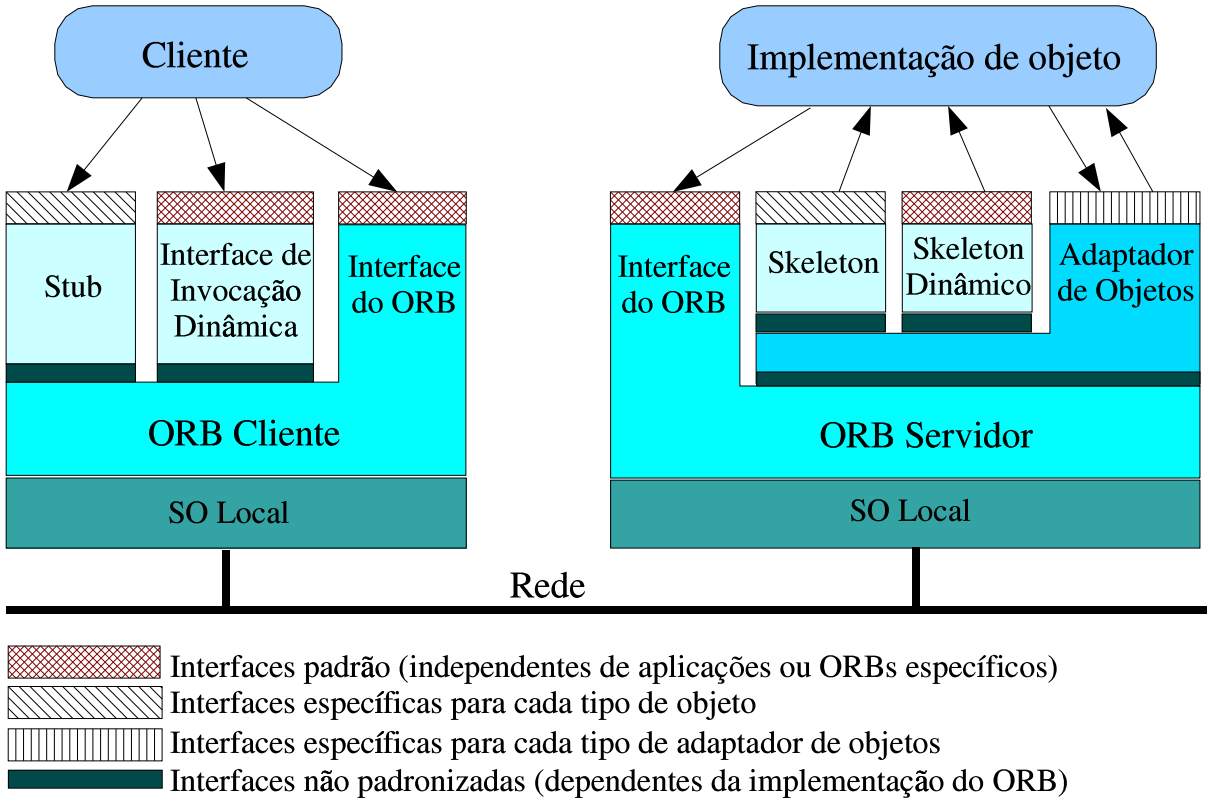

Figura 1.4: Organização geral de um sistema CORBA

ORB Core. O núcleo do ORB tem como papel principal prover um canal de comunicação entre clientes e objetos. O ORB fornece uma interface que permite executar operações tais como: iniciar e terminar o ORB, converter referências de objetos em strings (e vice-versa), montar listas de argumentos para uso com a interface de invocação dinâmica (ver a seguir), etc.

Stub. São os componentes responsáveis por converter requisições geradas pelos clientes em um formato apropriado para sua condução através da rede. Isto envolve a seriação (*marshaling*) dos argumentos da requisição (ou seja, a transformação dos dados a serem passados como argumento em um formato padronizado para comunicação via rede), bem como a de-seriação (*unmarshaling*) dos valores devolvidos. Além disso, stubs são também responsáveis por converter requisições (e valores devolvidos) entre a linguagem de implementação do cliente e a notação utilizada internamente pelo ORB. Tipicamente, *stubs* são gerados automaticamente a partir de definições de interface especificadas em IDL, por meio do compilador que implementa o mapeamento entre IDL e a linguagem de implementação do cliente. Um determinado *stub* é, portanto, específico para uma determinada interface IDL e para um determinado ORB. Em tempo de execução, o *stub* cria na máquina do cliente, objetos chamados *proxy* que funcionam como representantes do objeto CORBA remoto na máquina do cliente. As chamadas de método do cliente são,feitas no objeto *proxy* que as redireciona, através do ORB, ao objeto remoto apropri-

ado. Assim, quando um cliente recebe uma referência remota para um objeto CORBA (chamada de IOR), ela é traduzida, no contexto do cliente, para uma referência local ao objeto *proxy* implementado na linguagem de programação do cliente (por exemplo, Java, C++ ou Smalltalk).

Interface de Invocação Dinâmica (DII). Para os casos em que a interface do objeto a ser chamado não é conhecida a priori, o uso de requisições através de stubs não é possível. Nestes casos, o programa cliente tem a possibilidade de construir dinamicamente uma requisição CORBA e utilizar a mesma para fazer chamadas ao objeto através da DII. O papel desempenhado pela DII é semelhante àquele desempenhado por stubs (por exemplo, *marshaling* e *unmarshaling*), embora genérico e independente de interfaces particulares. Um aspecto importante de requisições dinâmicas é a obtenção, em tempo de execução, de informações que descrevem a interface do objeto a ser invocado, de forma a montar requisições apropriadamente. Isto é feito através do Repositório de Interfaces, que é examinado abaixo.

Esqueleto. O *skeleton* é o componente responsável por receber requisições do ORB e convertê-las para o formato esperado pela implementação do objeto, através da de-seriação dos argumentos da requisição (de forma a utilizá-los, por exemplo, para chamar o método apropriado na implementação), bem como da seriação dos valores a serem devolvidos para o cliente. Note-se que isto inclui também a tradução das requisições para a linguagem de implementação do objeto. Como no caso de *stubs*, *skeletons* são específicos para cada interface IDL e são tipicamente gerados automaticamente por um compilador IDL.

Esqueleto Dinâmico. Analogamente à DII, a interface de esqueleto dinâmico (DSI) permite a manipulação de requisições do lado do servidor de maneira genérica, independente de tipos de interface específicos. Esta funcionalidade é tipicamente utilizada por pontes (*bridges*) para permitir a interação entre objetos em ORBs diferentes. Em geral, um objeto CORBA que desempenha o papel de ponte é capaz de receber e tratar requisições quaisquer através da DSI, de forma a encaminhá-las ao objeto apropriado que reside em um outro ORB, possivelmente utilizando outro tipo de middleware.

Adaptador de Objetos. É o elemento responsável por compatibilizar o modelo de programação no qual objetos são implementados com o modelo de objetos de CORBA (por exemplo, através do mapeamento do conceito de objeto CORBA para a linguagem de implementação). Funções específicas de um adaptador de objetos são a criação e ativação de objetos CORBA, a geração de referências interoperáveis para objetos (IORs), e o despacho de requisições (através de um esqueleto) para o objeto apropriado.

Algumas destas funções são diretamente visíveis às implementações dos objetos, podendo ser invocadas através da interface do adaptador. CORBA permite que vários tipos diferentes de adaptadores de objetos estejam presentes, com o objetivo de acomodar as necessidades específicas das diversas categorias de aplicações. Entretanto, um tipo padrão, chamado de Adaptador de Objetos Portátil (POA), é definido na especificação, sendo que toda implementação de CORBA deve oferecê-lo. A arquitetura, as funções e a interface do POA são definidas de maneira precisa e completa, de forma a facilitar a portabilidade de implementações de objetos entre diferentes implementações do POA. Graças ao POA,

é possível desenvolver uma aplicação distribuída independentemente de um ORB específico. Pode-se iniciar o desenvolvimento utilizando-se um ORB e, se o desempenho não for satisfatório, pode-se mudar para outro ORB CORBA com muita facilidade.

Outra característica importante do POA é sua flexibilidade de configuração, representada pelo uso de políticas que governam aspectos como a ativação de objetos (por exemplo, ativação sob demanda, compartilhamento de threads e persistência) e a criação de referências de objeto. Assim, introduz-se um certo grau de configurabilidade no comportamento do adaptador de objetos e, por consegüinte, do próprio ORB.

Além dos componentes mostrados na Figura 1.4, CORBA também define outros dois componentes básicos:

- *Repositório de Interfaces*, que fornece um serviço para a obtenção de informações sobre interfaces em tempo de execução. Mais precisamente, o repositório de interfaces permite a representação das diversas estruturas definidas em CORBA IDL, bem como de relacionamentos elementares entres os mesmos. Em outras palavras, o Repositório de Interfaces provê um serviço básico de gerenciamento de meta-informações em ambientes CORBA. Implementações do repositório são tipicamente constituídas por um ou mais objetos CORBA representando definições IDL e implementando interfaces para acesso a tais definições. Seu principal uso é voltado para o suporte à construção de requisições dinâmicas, conforme descrito acima.

- *Repositório de Implementações*, que mantém um registro com informações necessárias para a ativação de objetos. Dada uma particular referência de objeto, o repositório de implementações permite encontrar informações sobre como localizar e instalar a implementação do objeto correspondente. Por exemplo, o repositório pode conter a especificação do arquivo executável ou biblioteca a ser carregada, o número da porta à qual o objeto instanciado será associado, e a forma como a instanciação deve proceder (tal como em um processo diferente ou em um já existente). Este serviço é usado, particularmente, pelo adaptador de objetos, para a obtenção das informações necessárias para a criação de objetos.

O Modelo de Desenvolvimento em CORBA

O desenvolvimento de clientes e objetos em CORBA segue um padrão geral bem definido, o qual é sumarizado na Figura 1.5.

O processo tem início com a definição das interfaces dos objetos (juntamente com definições de tipos e estruturas auxiliares) em CORBA IDL. A compilação destas definições gera, como produtos, *stubs* e esqueletos, bem como as estruturas correspondentes para armazenamento no Repositório de Interfaces. A geração dos *stubs* é feita por um compilador que implementa o mapeamento de CORBA IDL para a linguagem de implementação do cliente, enquanto que os esqueletos são gerados por um compilador que realiza o mapeamento para a linguagem de implementação do objeto.

Do lado cliente, o programador deve fornecer a implementação da aplicação, a qual faz uso dos serviços disponíveis na interface do objeto. Esta implementação e os stubs que ela utiliza devem então ser compilados e liagados, formando assim o cliente.

Do lado servidor, o programador deve fornecer as implementações das características do objeto, conforme definidas em suas interfaces. Tal implementação é feita utilizando-se construções da linguagem de implementação do objeto, segundo o mapeamento de

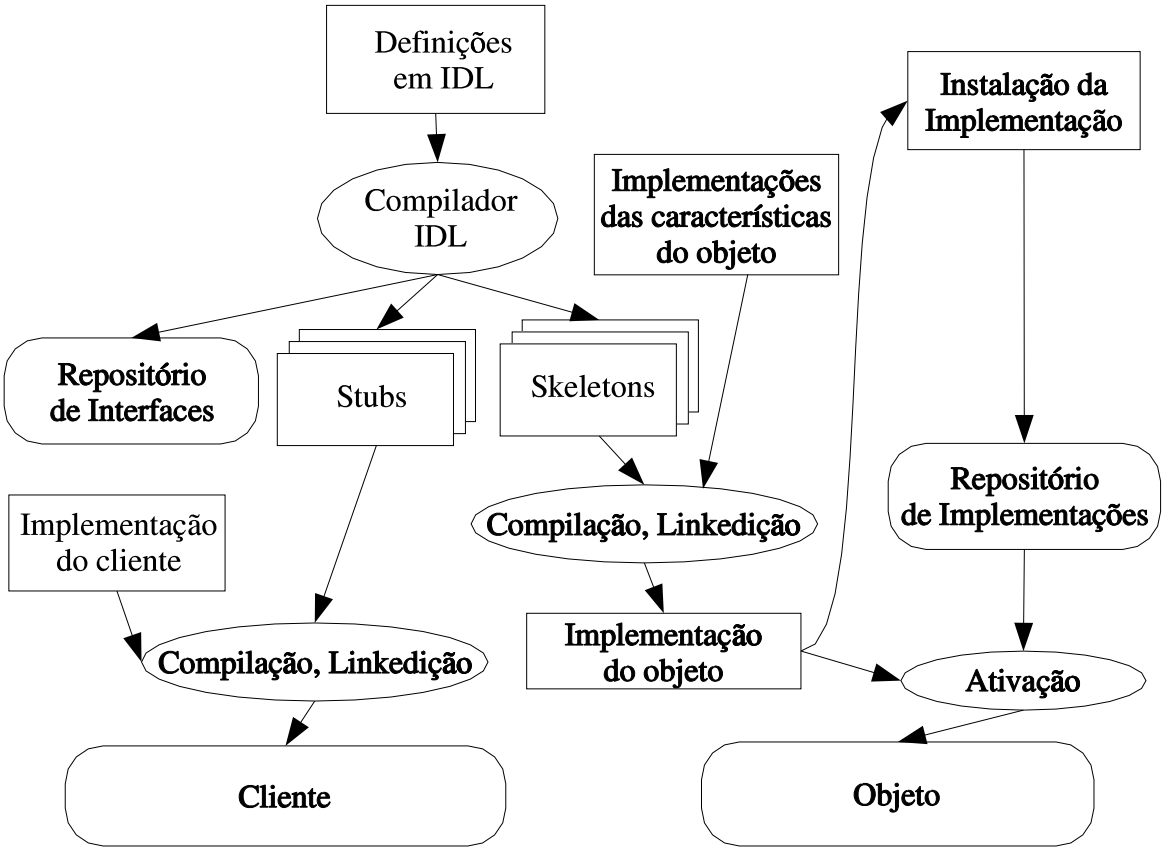

Figura 1.5: O processo de desenvolvimento de clientes e objetos em CORBA

CORBA IDL. Por exemplo, em Java, uma interface IDL é implementada através de uma classe com métodos que correspondem às operações da interface. Esta implementação deve então ser compilada, juntamente com os respectivos esqueletos, os quais são ligados para formar a implementação do objeto. A entidade de linguagem que contém a implementação de objeto é denominada servente (*servant*). Informações sobre a implementação do objeto são então submetidas para instalação no Repositório de Implementações. Finalmente, através do processo de ativação (realizado pelo POA em conjunto com o repositório de implementações), um objeto CORBA é criado e disponibilizado para receber requisições dos clientes.

Outros Aspectos de CORBA

Além da arquitetura básica descrita acima, a especificação de CORBA também define aspectos importantes para uso do modelo em diversas circunstâncias diferentes. Alguns destes aspectos são opcionais, enquanto que outros são obrigatórios.

Interoperabilidade. Uma vez que os detalhes internos de implementação do ORB são deixados a cargo de cada fabricante, é provável que ORBs diferentes utilizem mecanismos ou protocolos de comunicação distintos. Isto dificulta a interação entre objetos executados

em ORBs diferentes. Este problema foi resolvido na versão 2.2 de CORBA através da definição de um protocolo padrão a ser utilizado para interações entre ORBs, denominado de *General Inter-ORB Protocol* (GIOP). GIOP define os tipos de mensagens trocadas entre ORBs (por exemplo, para a condução de requisições, respostas e notificações de erro), bem como o formato preciso destas mensagens. Entretanto, note que GIOP é um protocolo abstrato, sendo que diferentes realizações concretas do mesmo são possíveis, dependendo do protocolo de transporte subjacente. Notadamente, a implementação de GIOP sobre TCP/IP é a mais comum e é conhecida como IIOP (*Internet Inter-ORB Protocol*). Outro aspecto importante para a interoperabilidade entre ORBs distintos é a definição de referências de objeto que possam ser interpretadas corretamente, independentemente do ORB em que foram geradas. A especificação de Referências de Objeto Interoperáveis (IORs) cumpre tal papel, fornecendo um formato padronizado para a especificação das informações necessárias sobre interfaces.

Interceptadores Portáteis. Até recentemente, a especificação de CORBA não permitia a configuração da implementação interna do ORB. Este problema foi parcialmente resolvido com a inclusão de um mecanismo de interceptadores, que possibilita a introdução e configuração de serviços internos ao ORB. Este mecanismo funciona através da interceptação do fluxo normal de execução do ORB, em pontos bem definidos, de forma a modificar a forma como requisições são conduzidas (por exemplo, para a introdução de novas propriedades não-funcionais como autenticação, criptografia, compressão, etc.). A atual especificação de interceptadores em CORBA, denominada de Interceptadores Portáteis, define dois tipos de interceptadores:

- Interceptadores de Requisições, que realizam interceptação do envio e recepção de requisições e respostas, permitindo a inspeção de informações sobre as requisições e a inclusão de comportamentos ou serviços adicionais para a sua manipulação.
- Interceptadores de Referência de Objeto, que permitem alterar a forma como IORs são geradas, tornando assim possível modificar as políticas que regem a criação de objetos.

CORBA Messaging. Tradicionalmente, CORBA provê apenas os dois estilos de interação, exemplificados na seção 1.3.1. A especificação *CORBA Messaging* estende este modelo de interações com o suporte a requisições assíncronas do tipo *callback* (o cliente provê uma operação especial através da qual o resultado de uma requisição pode ser devolvido) e *polling* (o cliente chama explicitamente uma interface específica para receber o resultado de uma requisição). Estes estilos de interação são implementados através do pré-processamento de definições IDL, de forma a gerar as operações necessárias para o seu suporte. Portanto, o meta-modelo de CORBA (como definido por IDL) não é alterado. Dois aspectos importantes do suporte a requisições assíncronas em CORBA são a persistência das requisições (que significa que mensagens são armazenadas pelo middleware caso o objeto destino não esteja ativo no momento) e a Qualidade de Serviço (que permite a definição de propriedades a serem obedecidas durante a execução das requisições, tais como prioridades e prazos).

minimumCORBA. O uso de CORBA em ambientes computacionais com recursos limitados (por exemplo em dispositivos móveis ou em sistemas embutidos) é dificultado pelo

considerável custo (por exemplo, em termos de memória, processamento e comunicação) que uma implementação completa do padrão representa. Em geral, as bibliotecas que implementam o ORB exigem vários megabytes de memória o que normalmente está além da capacidade de computadores de mão, sistemas embutidos, etc. *minimumCORBA* define um subconjunto dos componentes de CORBA mais apropriado para implantação em tais ambientes, através da omissão de características não essenciais. Em particular, os aspectos dinâmicos de CORBA (DII e DSI) são omitidos, juntamente com a maior parte do Repositório de Interfaces e o suporte a interceptadores. Além disso, diversas funcionalidades não essenciais são omitidas das interfaces do ORB e do POA. Note-se, contudo, que minimumCORBA preserva interoperabilidade com CORBA, o que implica no suporte à especificação completa de CORBA IDL. Como veremos na Seção 1.6, nós acreditamos que a escolha da OMG não foi das mais felizes pois são justamente os aspectos dinâmicos os mais relevantes para os ambientes computacionais do futuro envolvendo computação ubíqua.

Tempo-Real CORBA. Esta é uma extensão opcional de CORBA, para uso em sistemas de tempo real. O objetivo é prover meios para garantir a previsibilidade, de uma extremidade à outra de um sistema, das atividades do middleware, segundo restrições temporais impostas à sua execução. Isto envolve o uso de mecanismos para gerenciar a alocação de recursos às tarefas, para o controle de prioridades de execução, para o escalonamento de threads em tempo real, entre outros. Note-se que o uso de tais mecanismos pressupõe algum tipo de suporte subjacente oferecido por um sistema operacional de tempo real ou por extensão de tempo real em sistemas tradicionais como Linux, Solaris ou NT.

Tolerância a Falhas CORBA. Esta extensão de CORBA objetiva prover um suporte robusto para aplicações que necessitam de altos níveis de confiabilidade. A abordagem baseia-se na replicação de objetos em *grupos de objetos*, bem como em técnicas para a detecção e recuperação de falhas com base na redundância assim obtida. O uso de grupos de objetos é implementado de maneira transparente através de um tipo especial de IOR, denominado de referência de grupo de objetos interoperáveis (IOGR). Uma IOGR contém múltiplas referências para as diferentes réplicas do objeto alvo, as quais representam alternativas a serem usadas em caso de falhas.

Uma excelente referência para os principais aspectos da arquitetura CORBA é o livro escrito por Hening e Vinoski [16].

1.3.2 Outras Tecnologias Relevantes

Embora a ênfase aqui tenha sido no estudo de CORBA como exemplo representativo de middleware para ambientes distribuídos abertos, é importante reconhecer a existência de outras tecnologias também amplamente utilizadas. Em particular, esta seção apresenta uma breve descrição das características mais importantes de duas destas tecnologias, DCOM e Java RMI.

Distributed COM (DCOM)

DCOM [34] é a extensão de COM (Common Object Model), o modelo de objetos da Microsoft, para ambientes distribuídos. Essencialmente, DCOM adota os mesmos meca-

nismos de comunicação entre objetos empregados em COM, adicionando a possibilidade de que interações sejam independentes da localização física dos objetos.

De forma semelhante ao que acontece em CORBA, DCOM adota um modelo no qual objetos podem ser representados remotamente através de suas interfaces. Entretanto, ao invés de usar uma representação de interfaces baseada em uma IDL, DCOM utiliza interfaces binárias, que consistem de tabelas de ponteiros para as respectivas implementações de cada um dos métodos de uma interface. Uma vantagem disto é que interfaces se tornam independentes de linguagem, dispensando portanto o uso de mapeamentos de linguagem, como em CORBA.[5] Cada interface em DCOM possui um identificador único, que também é utilizado para identificar o componente que implementa a interface. Entretanto, um componente pode implementar várias interfaces.

De acordo com a arquitetura geral de DCOM, clientes e objetos interagem através da interposição de proxies, que realizam o *marshaling* e *unmarshaling* das requisições, sendo que a comunicação subjacente é normalmente realizada através de RPCs. Como em CORBA, DCOM provê tanto requisições estáticas quanto dinâmicas. Requisições dinâmicas são implementadas através do uso de bibliotecas de tipos, as quais permitem a descoberta, em tempo de execução, da assinatura dos métodos em uma dada interface. Além disso, todo componente em DCOM implementa uma interface especial, *IUnknown*, que permite a descoberta das interfaces implementadas pelo componente.

Além do suporte básico para interação entre clientes e objetos, DCOM também fornece um conjunto de serviços de distribuição, tais como serviço de nomes, de controle de concorrência, de controle do ciclo de vida de objetos, de persistência e de segurança. Entretanto, vários destes serviços são, na verdade, implementados pelo sistema operacional subjacente (Windows), o que penaliza a portabilidade de aplicações desenvolvidas em DCOM para fora de ambientes Microsoft. Apesar de a Microsoft argumentar que DCOM é, em tese, uma plataforma de middleware independente de sistema operacional, só há implementações robustas de DCOM para a plataforma Microsoft.

Java RMI

Java é um sofisticado ambiente de middleware para desenvolvimento de sistemas distribuídos que é composto por uma linguagem de programação (a linguagem Java), uma máquina virtual (JVM), um mecanismo para chamada remota de métodos (RMI) e uma série de serviços distribuídos. Java RMI é fortemente integrado à linguagem Java [53]. Isto significa que clientes e objetos precisam ser implementados em Java, seguindo o modelo de programação adotado pela linguagem. Os problemas de heterogeneidade são, portanto, inteiramente resolvidos pela máquina virtual Java. A homogeneidade resultante desta abordagem faz com que não haja necessidade de uma linguagem separada para definições de interfaces (como IDL); as definições de interface são fornecidas diretamente em Java.

Como em CORBA, clientes Java RMI chamam métodos em um objeto através de chamadas locais a métodos de um *stub*, o qual representa localmente o objeto remoto. Objetos recebem requisições remotas através de esqueletos. *Stubs* e esqueletos são gerados automaticamente a partir das classes que implementam os objetos através do compilador RMI, o `rmic`. Desta forma, antes de fazer uso de um dado objeto, o middleware deve

[5]Embora interfaces em DCOM sejam normalmente definidas utilizando-se uma IDL, chamada MIDL, tais definições são usadas apenas para produzir a representação binária da interface, que é a representação utilizada pelo programador.

obter a implementação (em formato de *byte-code* Java) do *stub* apropriado de forma a efetuar a sua carga no sistema de execução. A implementação do *stub* pode ser carregada dinamicamente tanto do sistema de arquivos local quanto de um servidor remoto em qualquer ponto da Internet, dada a sua URL. Note que isto difere da abordagem de CORBA e DCOM de gerar o *stub* localmente a partir da representação da interface do objeto.

Clientes normalmente obtêm referências para objetos remotos através de um serviço de nomes provido por Java RMI, denominado de *registry*. Este serviço também provê interfaces para que objetos servidores possam se registrar, de forma a se tornarem acessíveis a possíveis clientes. O *registry* é mais limitado do que o Serviço de Nomes de CORBA e só permite que objetos presentes na mesma máquina que o *registry* a ele se registrem. Além disso, quando o cliente solicita a resolução de um nome, ele precisa indicar explicitamente o endereço da máquina onde está o *registry*.

Ao contrário de CORBA, o fato de que *byte-code* Java possa ser interpretado em qualquer arquitetura de software e hardware facilita a migração de objetos de uma máquina para outra em sistemas heterogêneos. Em Java RMI, objetos podem ser passados como argumentos por valor (em CORBA, eles são passados por referência). Quando um objeto Java é transmitido para uma outra JVM através de RMI, o seu estado é convertido em uma seqüência de bytes (processo de seriação) e enviado para a JVM remota. Ao receber a requisição RMI, a JVM remota carrega o código da classe correspondente e instância uma cópia do objeto original, de-seriando o seu estado. A partir daí, alterações em qualquer uma das duas cópias dos objetos não são sincronizadas.

Note-se que o objetivo de Java RMI é prover o suporte básico para distribuição. Outros serviços mais avançados são implementados em cima de RMI, tais como Jini (para o suporte a sistemas distribuídos capazes de se configurarem expontaneamente), JNDI (uma interface comum para acesso a diversos tipos de serviços de nomes e de diretório) e, notadamente, Enterprise Java Beans, que oferece serviços como controle de transações, persistência e segurança para servidores baseados em componentes. EJB é examinado na Seção 1.5.

1.3.3 Comparação com os Princípios de RM-ODP

O modelo RM-ODP provê um ponto de referência conveniente para a análise crítica de tecnologias de middleware. Nesta seção, os principais aspectos da arquitetura de CORBA, DCOM e Java RMI são considerados sob a perspectiva dos conceitos de pontos de vista, transparências, funções ODP e estilos de interação, definidos em RM-ODP.

Pontos de vista

Em relação à abordagem de pontos de vista, de maneira geral pode-se afirmar que as três tecnologias aqui estudadas se concentram, predominantemente, no ponto de vista computacional. Nos três casos, as funcionalidades disponíveis para o programador se limitam ao acesso, muitas vezes implícito, aos serviços necessários para prover a interação entre clientes e servidores. O modelo adotado favorece a construção de aplicações e serviços de alto nível com base em configurações de objetos, de maneira independente da forma na qual tais objetos serão distribuídos. Decisões sobre a distribuição dos objetos, quando necessárias, se aplicam apenas no uso de serviços de que lidam com a configuração do sistema, como a instanciação e instalação de objetos. Mesmo assim, o modelo não

oferece uma visão clara dos mecanismos subjacentes. Esta é uma característica inerente destas tecnologias, uma vez que se baseiam na padronização de interfaces, muitas vezes adotando uma abordagem do tipo "caixa preta" para implementação dos mecanismos necessários para prover tais interfaces. Em outras palavras, o ponto de vista de engenharia não é visível ao programador, de forma que se torna difícil a configuração personalizada da plataforma. Apesar disto, nota-se atualmente um movimento no sentido de se ampliar a visão que o programador tem da engenharia da plataforma. Um exemplo disto é a adoção de mecanismos de interceptadores e de políticas em CORBA, que permitem, ainda que estaticamente, um controle explícito sobre os serviços internos do ORB como parte da especificação de aplicações distribuídas. Entretanto, estes recursos carecem da estrutura e da completude proporcionadas pela abordagem de pontos de vista.

Por outro lado, com relação aos pontos de vista de empresa, informação e tecnologia, nota-se uma omissão generalizada. Os modelos de CORBA, DCOM ou Java RMI não consideram a especificação de aplicações com base nos aspectos tratados nestes pontos de vista.

Transparências

Em geral, as transparências básicas de localização e acesso especificadas em RM-ODP são observadas em CORBA, DCOM e Java RMI. Entretanto, apenas CORBA realiza estas transparências em um ambiente distribuído completamente heterogêneo, ao passo que DCOM e Java RMI assumem um ambiente homogêneo no que se refere a sistema operacional e linguagem de programação, respectivamente.

Quanto às demais transparências ODP, nota-se as seguintes situações:

- migração: em geral omitida das especificações, sendo deixada a cargo de implementações proprietárias; é facilitada em ambientes Java graças à interpretação do *byte-code* pela JVM;

- relocação: em CORBA, implementada através do mecanismo de desvio de requisições (pelo ORB) presente no protocolo GIOP (isto é, através do uso de mensagens de resposta com status do tipo LOCATION_FORWARD que permite que requisições a um ORB sejam redirecionadas para outro ORB);

- replicação: em CORBA, suporte padrão para transparência de replicação é oferecido apenas para a finalidade de tolerância a falhas;

- transação: mecanismos para controle de transações são em geral oferecidos, embora seu uso seja predominantemente explícito (isto é, não-transparente);

- falhas: oferecida no contexto da especificação de Tolerância a Falhas CORBA;

- persistência: tanto CORBA quanto DCOM fornecem serviços para o armazenamento e recuperação de objetos, sendo que seu uso transparente é normalmente possível; em Java, o mecanismo padrão de seriação, pode ser utilizado para persistência.

Funções ODP

Pode-se dizer que este aspecto é relativamente bem contemplado nas três tecnologias estudadas acima, embora RM-ODP prescreva um repertório mais extenso de funções de suporte a distribuição. Isto é de se esperar, uma vez que a maior parte dos serviços especificados como funções ODP constituem pré-requisitos essenciais para sistemas distribuídos abertos.

Em CORBA, boa parte dos serviços de objeto possuem funções ODP equivalentes. Em particular, no caso do Serviço de Negociação (*Trading*), CORBA adota o mesmo padrão que foi desenvolvido no contexto de RM-ODP. Em DCOM, o caso é semelhante, embora as especificações dos serviços não tenham nenhuma relação direta com RM-ODP. Além disso, em vários casos, DCOM simplesmente exporta os serviços providos pelo sistema operacional Windows. No caso de Java RMI, a maior parte dos serviços de distribuição similares às funções ODP é definida fora da especificação de RMI. Entretanto, o fato de tais serviços estarem fortemente ligados ao contexto de Java (assim como RMI) caracteriza-os como parte da plataforma.

Estilos de Interação

RM-ODP define três estilos fundamentais de interação:

- operacional: usado para modelar interações do tipo cliente/servidor, baseado em mecanismos de chamada de métodos;
- *stream*: usado para interações que envolvem o fluxo contínuo de elementos de informação entre produtores e consumidores, obedecendo uma certa relação temporal;
- sinal: usado para representar, de forma discreta, a emissão e recepção de mensagens primitivas por objetos no sistema distribuído.

Em geral, os mecanismos de interação tradicionais presentes em CORBA, DCOM e Java RMI são baseados na chamada de métodos remotos, portanto obedecendo ao estilo operacional proposto em RM-ODP. Além disso, estas plataformas geralmente oferecem variações deste estilo de interação, tais como requisições assíncronas e *callbacks*, as quais não são diretamente prescritas em RM-ODP. Entretanto, o estilo de interação por sinais provido por RM-ODP pode ser utilizado para implementar tais variações. Por outro lado, nenhuma das três tecnologias estudadas fornece meios para o tratamento explícito de interações primitivas segundo o estilo de sinais.

No caso de *streams*, não se observa mecanismos para sua realização como parte do modelo de programação de CORBA, DCOM ou Java RMI. Entretanto, especificações complementares destas tecnologias procuram abordar este aspecto. Em particular, o modelo de *streams* de áudio e vídeo definido pela OMG [40] procura suprir esta lacuna no contexto de CORBA, embora apenas recursos para o controle de *streams* sejam providos, sendo que a real comunicação é feita por meio de mecanismos externos (soquetes TCP ou UDP/IP). Isto significa que o suporte a *streams* não é integrado ao modelo de programação, tal como ocorre no modelo computacional de RM-ODP, que permite a declaração explícita do estilo das interações providas por cada interface.

As tecnologias examinadas nesta seção constituem exemplos representativos da primeira geração de plataformas de middleware orientado a objetos. Em geral, pode-se dizer que o objetivo original destas tecnologias, que consiste na provisão de soluções para os problemas básicos de distribuição, foi satisfatoriamente atingido. Isto é comprovado pela larga aceitação dos modelos propostos como base para a construção de aplicações. A próxima seção, entretanto, apresenta importantes novas aplicações para as quais os modelos de middleware de primeira geração não se mostram adequados. Em particular, os requisitos destas aplicações apontam para a necessidade de modelos de middleware mais flexíveis e que ofereçam melhor suporte para o desenvolvimento de aplicações complexas. Conforme visto na Seção 1.5, os recentes avanços em tecnologias de middleware contemplam estes requisitos apenas parcialmente, sendo que novas abordagens para o design e implementação de middleware flexível precisam ser adotadas para fornecer uma solução mais abrangente (conforme discutido na seção 1.6).

1.4 Novas Aplicações de Middleware e seus Requisitos

A primeira geração de tecnologias de middleware orientado a objetos, representada por CORBA, Java RMI e DCOM, foi motivada pela necessidade de prover suporte adequado para o desenvolvimento de aplicações do tipo cliente/servidor. Esta categoria de aplicações representa, fundamentalmente, uma re-estruturação de aplicações mais convencionais (isto é, com arquiteturas centralizadas), de forma a adaptá-las a ambientes distribuídos abertos. Conseqüentemente, o desenvolvimento de arquiteturas tradicionais de middleware foi pautado pelos requisitos típicos destas aplicações. Esta influência se manifesta em aspectos tais como o estilo de interação predominante e a prioridade em ocultar os detalhes de implementação da plataforma (como uma abordagem simples para tornar as aplicações independentes de plataforma).

A disponibilidade e os recentes avanços em tecnologias de middleware (bem como nas tecnologias subjacentes de hardware, redes e sistemas operacionais), entretanto, motivaram a concepção de novas categorias de aplicações. Estas aplicações emergentes apresentam requisitos adicionais (algumas vezes conflitantes) em relação àqueles apresentados por aplicações mais convencionais. Por exemplo, o estilo operacional de interações, baseado na chamada de métodos remotos, nem sempre se mostra conveniente para aplicações fora do contexto cliente/servidor. Além disso, a definição comumente adotada para o conceito de "abertura", que se restringe à padronização das interfaces da plataforma (ignorando sua implementação interna), tem como conseqüência um suporte de middleware rígido que, apesar de adequado a aplicações tradicionais, é incapaz de se adaptar aos requisitos de novas aplicações.

Nesta seção, ênfase é dada à identificação dos requisitos de duas categorias importantes de aplicações emergentes, a saber: multimídia distribuída e computação ubíqua. Os requisitos destas aplicações são contrastados com o suporte oferecido por tecnologias de middleware convencionais, com o objetivo de motivar os recentes e futuros avanços nesta área.

1.4.1 Aplicações de Multimídia Distribuída

Esta classe de aplicações se caracteriza pelo uso integrado de múltiplas mídias (por exemplo, texto, gráficos, imagens, áudio e vídeo) na comunicação interativa entre os diferentes

elementos de uma aplicação e entre os seus usuários [1]. Vários dos problemas encontrados no desenvolvimento de aplicações distribuídas mais tradicionais são também encontrados no caso de aplicações de multimídia distribuída, sendo que as mesmas soluções podem ser adotadas. Serviços de nomes, de *trading* e de segurança, bem como transparências de localização e de acesso, são exemplos disto. Entretanto, outros aspectos comuns de plataformas de middleware convencionais não se adequam ao modelo de interação baseado em multimídia, o que representa um desafio para as novas tecnologias de processamento distribuído aberto [5].

Suporte para Mídias Contínuas

O termo *mídia contínua* se refere a mídias de informação cujo conteúdo varia em função do tempo. São exemplos deste tipo de mídia: áudio, vídeo e animações gráficas. O uso deste tipo de mídia em sistemas distribuídos requer suporte para transferências de dados ininterruptas entre os produtores e consumidores da informação, por períodos de tempo relativamente longos. Além disso, uma relação temporal entre elementos de dados transferidos sucessivamente deve ser obedecida, de forma a manter a constância do fluxo de informações. Para ilustrar este requisito, considere uma conversação baseada em voz, a qual pode ser representada através de uma seqüência de pacotes, cada um contendo um conjunto de amostras sucessivas do sinal de áudio digitalizado. A transferência desta seqüência de pacotes deve ser feita de tal forma que o intervalo entre pacotes sucessivos (conforme observado pelo receptor) seja aproximadamente constante, observando ainda uma determinada taxa de transferência, de modo que o conteúdo de voz possa ser corretamente reconstituído pelo receptor. Se cada pacote contém 100 ms de áudio digitalizado, então o receptor deverá receber, em média, 10 pacotes por segundo para reproduzir o som de forma adequada, sem interrupções.

O modelo de comunicação operacional, através da chamada (síncrona ou assíncrona) de métodos, por outro lado, é voltado para a transmissão de unidades discretas de informação, sem qualquer dependência ou relação explícita entre as mesmas. Este modelo é, portanto, inapropriado para a transmissão de mídias contínuas, o que demanda o desenvolvimento de modelos mais apropriados para satisfazer os requisitos deste tipo de mídia em sistemas distribuídos. Em particular, o estilo de interações baseado em *streams* proposto em RM-ODP [17] se mostra naturalmente adequado para este contexto. Entretanto, as plataformas de middleware mais usadas atualmente não provêem suporte a este modelo.

Qualidade de Serviço

Para que aplicações de multimídia distribuída possam apresentar uma qualidade satisfatória para os seus usuários, é necessário garantir a qualidade dos serviços utilizados para a transmissão dos dados multimídia. Esta qualidade é expressa, principalmente, em termos dos seguintes requisitos:

- **Temporização:** o controle de variáveis como o atraso fim-a-fim (*end-to-end delay*) na transmissão de pacotes e a variação deste atraso (ou *jitter*) são fundamentais para se preservar a interatividade e a correta fluência de informações representando mídias contínuas (por exemplo, a comunicação de voz requer atrasos fim-a-fim não superiores a 250ms, sendo que variações deste atraso não devem ultrapassar 10ms);

- **Volume de transmissão:** cada tipo ou formato de mídia apresenta requisitos característicos com relação à quantidade dados transmitidos e recebidos por segundo, sendo que o suporte de comunicação deve garantir uma vazão (ou *throughput*) mínima, de forma a não degradar a qualidade da transmissão;
- **Confiabilidade:** o suporte de comunicação deve garantir um limite máximo no número de elementos de informação perdidos ou corrompidos por segundo, de forma que a inteligibilidade da informação multimídia transmitida não seja comprometida.

O gerenciamento de Qualidade de Serviço (QoS) é portanto uma característica fundamental que plataformas de middleware devem possuir para prover suporte adequado para multimídia distribuída. Funções comuns neste contexto incluem a especificação e negociação de qualidade de serviço, o gerenciamento de recursos (de forma que os recursos necessários sejam garantidos para fornecer a qualidade de serviço requisitada por cada aplicação) e o controle de admissão (novas aplicações somente são admitidas caso o sistema tenha recursos suficientes para oferecer a qualidade por elas requisitada, sem prejudicar as demais aplicações já em execução). Além disso aspectos dinâmicos como a monitoração e a renegociação dos níveis de qualidade de serviço reais devem estar presentes. Desta forma, viabiliza-se a existência de mecanismos para a manutenção e, se necessário, para o ajuste em tempo de execução do nível de qualidade de serviço fornecido a uma determinada aplicação (por exemplo, caso os recursos disponíveis no sistema se tornem escassos). Note que, em certos casos, isto pode envolver a reconfiguração dos mecanismos internos da plataforma (por exemplo, pela substituição de alguns de seus componentes por outros mais eficientes), o que não é, em geral, permitido por arquiteturas de middleware convencionais.

Sincronização em Tempo Real

A integração e o uso interativo de múltiplas mídias distintas, como descrito acima, são características determinantes de aplicações de multimídia distribuída. Em geral estas características são definidas com base em medidas de tempo real. O aspecto da interatividade implica, principalmente, na manutenção de sincronismo intra-mídia, de forma a manter as propriedades de temporização de cada mídia transmitida (por exemplo, para garantir a fluência em transmissões de voz ou de vídeo). A integração de múltiplas mídias, por sua vez requer sincronização inter-mídia, de forma a preservar restrições de tempo real entre fluxos de mídia relacionados (por exemplo, para a correta sobreposição dos fluxos de voz e vídeo utilizados para transmitir o discurso de uma pessoa).

Estes aspectos têm sido tradicionalmente considerados como parte da implementação das aplicações. Entretanto, como se tratam de requisitos comuns a aplicações de multimídia distribuída em geral, seria natural que mecanismos básicos de suporte a sincronização de tempo real fossem implementados pela própria plataforma de middleware. Tais mecanismos poderiam então ser re-utilizados (e reconfigurados) pelo programador de aplicações.

Comunicação Multi-Ponto

Várias aplicações de multimídia distribuída (tais como tele-conferência e TV interativa) envolvem a participação de grupos de usuários, os quais interagem através da geração e consumo de fluxos de mídia. Isto requer que o modelo de programação forneça primitivas

de comunicação multi-ponto, facilitando assim o desenvolvimento de aplicações. Em particular, o suporte a comunicação multi-ponto deve observar os requisitos descritos acima quando usados para transmissão multimídia.

1.4.2 Computação Móvel e Ubíqua

Os recentes avanços em tecnologias de redes sem fio e dispositivos computacionais portáteis tornaram possível a concepção de ambientes distribuídos cujos componentes não são ligados a uma localização fixa. Um PDA (*Personal Digital Assistant*) ou computador de mão dotado de uma interface para rede sem fio, por exemplo, permite o deslocamento físico, dentro da área de cobertura de sua rede local, sem perda de conexão (embora a qualidade da conexão possa deteriorar à medida em que o dispositivo se afasta da estação-base da rede). Além disso, caso o dispositivo seja movido para fora da área de alcance de sua rede original, o mesmo pode, automaticamente, se conectar a outra rede local ou então se desconectar completamente (se não houver cobertura na área para onde o dispositivo foi levado). Há também os casos de dispositivos com mais de uma interface de rede, os quais podem se conectar a redes de tipos diferentes em momentos diferentes. Por exemplo, um PDA com uma interface para redes locais IEEE 802.11b e outra para redes GSM, de longa distância, pode se reconfigurar para usar a segunda interface caso seu usuário o leve para uma área fora da cobertura da rede local. Obviamente, haverá uma queda substancial na largura de banda disponível (de algo em torno de 2-11Mbps para cerca de 9600bps), haverá também uma mudança no padrão de ocorrência de erros; entretanto, a continuidade do acesso é preservada. Do ponto de vista de sistemas distribuídos, este tipo de ambiente representa um desafio fundamental, uma vez que os componentes das aplicações distribuídas devem manter seu serviço a despeito do elevado potencial para mobilidade e da conectividade variável. Plataformas de middleware para ambientes móveis devem, portanto, prover meios para lidar com tais problemas, o que geralmente não se observa em tecnologias de middleware convencionais.

Para complementar o tipo de cenário descrito acima, o desenvolvimento de tecnologias de baixo custo para redes sem fio, notadamente BlueTooth [6], promete concretizar a visão de ambientes de computação ubíqua originalmente introduzida por Mark Weiser [57]. Neste contexto, pequenos dispositivos computacionais (fixos ou móveis) permeiam o ambiente de forma imperceptível, com o objetivo de facilitar as tarefas do usuário. No futuro próximo, escolas, escritórios, hospitais, ruas, rodovias, estações de metrô e até nossas casas serão populadas por milhares de dispositivos computacionais trabalhando em conjunto inter-conectados através de diferentes tipos de redes. Além dos requisitos típicos apresentados por sistemas móveis, dispositivos usados para computação ubíqua tendem a ser severamente restritos com relação aos recursos disponíveis. Em geral, a capacidade de armazenamento, processamento e comunicação em tais dispositivos é ordens de magnitude menor do que em sistemas fixos. Desta forma, uma das prioridades para o desenvolvimento de aplicações em tais ambientes é a economia de recursos. Não obstante, tais aplicações são quase sempre distribuídas e, assim como aplicações convencionais, necessitam de uma infra-estrutura com mecanismos e serviços que facilitem a comunicação entre seus componentes. Plataformas de middleware convencionais, contudo, não se adequam a ambientes com tais características, em virtude de seu design rígido, que geralmente assume uma disponibilidade considerável de recursos.

Segundo [7], os requisitos a serem satisfeitos por middleware para sistemas distribuí-

dos móveis (e, por extensão, para sistemas usados em computação ubíqua), podem ser agrupados em três categorias fundamentais, como descrito abaixo.

Baixa Carga Computacional

Plataformas de middleware para computação móvel e ubíqua devem se adequar à escassez de recursos típica de tais ambientes. Para tanto, é fundamental que os mecanismos da plataforma não sobrecarreguem os dispositivos nos quais a plataforma é executada. Isto significa que propriedades não-funcionais e transparências normalmente providas por middleware podem ter que ser omitidas caso não sejam consideradas essenciais. Este requisito implica na habilidade de se configurar a plataforma de middleware para cada ambiente ou dispositivo específico, de forma a reduzir o consumo de recursos ao mínimo necessário.

Comunicação Assíncrona

Em ambientes móveis, dispositivos podem observar conectividade intermitente. Como exemplo, um PDA pode, em determinado instante, estar conectado a uma rede local de alta velocidade, enquanto que, num momento seguinte, o dispositivo pode sair da área de cobertura da rede, passando para uma rede de baixo desempenho (como GSM) ou mesmo ficando completamente desconectado. Além disso, é comum a tais dispositivos a desconexão intencional, por exemplo, através da desativação (*shutdown*) do dispositivo. No contexto de sistemas de processamento distribuído, uma implicação fundamental disto é que componentes (ou objetos) hospedados em dispositivos móveis podem não estar disponíveis no momento em que outros componentes tentarem fazer uso dos mesmos. Isto significa que mecanismos de comunicação síncrona, que exigem a disponibilidade simultânea de ambos os componentes comunicantes não são apropriados. Mais apropriado seria a provisão, pela plataforma de middleware, de mecanismos de comunicação assíncrona, que permitem o armazenamento intermediário das informações transferidas. Observe que, devido ao requisito anterior, é essencial que tais mecanismos não gerem uma carga desnecessária nos dispositivos móveis.

Sensibilidade ao Contexto

Este requisito resulta da dinamicidade de ambientes móveis, que faz com que mudanças no contexto de um componente da aplicação sejam relativamente freqüentes durante o seu tempo de vida. Por exemplo, o conjunto de serviços disponíveis para um determinado componente, (após conexões, desconexões e deslocamentos do dispositivo hospedeiro e de outros dispositivos no sistema), bem como o seu nível e tipo de conectividade, podem variar ao longo do tempo. O serviço provido por uma plataforma de middleware deve, portanto, ser capaz de se adaptar ao contexto atual da aplicação e seus componentes (por exemplo, através do uso de mecanismos mais leves de comunicação quando utilizando conexões de baixa velocidade). Note que isto pode exigir conhecimento específico de cada aplicação (por exemplo, para definir a melhor forma de lidar com mudanças de contexto não previstas de antemão), o que significa que a plataforma de middleware deve, idealmente, prover mecanismos para que as aplicações direcionem o tipo de adaptação a ser realizado. Como um outro exemplo, considere uma aplicação cujos componentes podem migrar de um dispositivo para outro no sistema. Em tais casos, o suporte oferecido pela plataforma de middleware ao componente deve se adaptar ao contexto de cada dispositivo

hospedeiro. Em resumo, a dinamicidade de contexto em ambientes móveis requer maior flexibilidade e adaptabilidade do que oferecido por plataformas de middleware convencionais.

Note que é comum a existência de aplicações que combinam características de ambientes móveis (e computação ubíqua) com multimídia distribuída. Do ponto de vista de middleware, isto significa que uma combinação dos requisitos das duas categorias de aplicação deve ser contemplada pela plataforma. Por exemplo, pode ser necessário prover o gerenciamento de qualidade de serviço em ambientes com recursos limitados. Nestes casos, o gerenciamento de qualidade de serviço deve garantir o uso disciplinado dos recursos do sistema, ao mesmo tempo em que deve, ele mesmo, ser implementado de forma a consumir o mínimo de recursos. Como resultado desta combinação de requisitos, restringe-se ainda mais a aplicabilidade de tecnologias convencionais de middleware, necessitando novas tecnologias mais apropriadas.

1.4.3 Outras Aplicações Emergentes e seus Requisitos

A difusão de ambientes distribuídos de larga escala, em particular no contexto da teia mundial (*World-Wide Web*), favoreceu a re-estruturação de várias aplicações distribuídas existentes, muitas vezes resultando em categorias de aplicações completamente novas. O tipo de suporte exigido para o desenvolvimento de tais aplicações, entretanto, muitas vezes está além da capacidade de infra-estruturas convencionais de middleware. Exemplo disso é o modelo de serviços oferecidos através da teia (*Web Services*), os quais se caracterizam por serem componentes de aplicação acessíveis por meio de protocolos e interfaces padrão da teia. São inúmeras as aplicações às quais o modelo de serviços *web* se aplica, entre elas comércio eletrônico e acesso a informações distribuídas. Estas classes de aplicações compartilham um conjunto de requisitos comuns, tais como aqueles relacionados abaixo.

- *Integração com a Internet:* o acesso a serviços *web* deve ser feito com o uso de tecnologias padrão tais como HTTP e XML. Isto significa que plataformas de middleware devem dispor de mecanismos que permitam a integração efetiva de componentes de aplicação que utilizam tais tecnologias (por exemplo, permitindo que um cliente qualquer utilize um vocabulário XML pré-definido para ter acesso a serviços no contexto da paltaforma).

- *Elevada disponibilidade:* serviços *web* devem, em geral, estar disponíveis continuamente (24 horas por dia, 7 dias por semana, 365 dias por ano), sendo que períodos de interrupção de funcionamente têm se tornado cada vez mais custosos e menos aceitáveis. Plataformas de middleware devem, portanto prover altos níveis de robustez e tolerância a falhas. Além disso, uma plataforma deve também permitir a reconfiguração e evolução da aplicação (ou da própria plataforma) sem a necessidade de interrupção do serviço.

- *Desenvolvimento rápido:* devido a pressões de mercado, a implementação de serviços *web* requer métodos altamente eficientes, que permitam acelerar o ciclo de desenvolvimento. Do ponto de vista de middleware, isto significa que a plataforma deve prover mecanismos eficazes para o reuso de componentes já existentes, bem como para o uso transparente dos serviços e aspectos não-funcionais providos pela plataforma.

Observe que outros tipos de aplicações, não necessariamente voltadas para uso na Internet, apresentam requisitos semelhantes. Em particular, elevada disponibilidade e desenvolvimento rápido têm se tornado aspectos cada vez mais importantes. O design de plataformas de middleware deve, portanto, permitir altos níveis de robustez e reutilização, ao mesmo tempo em que deve oferecer a flexibilidade necessária para que eventuais reconfigurações sejam feitas sem a necessidade de interrupções na operação das aplicações.

1.5 Recentes Avanços em Tecnologias de Middleware

As tecnologias de middleware examinadas na Seção 1.3 têm em comum o fato de que sua ênfase está em facilitar o acesso aos serviços em um ambiente distribuído. Os mecanismos implementados pela plataforma são voltados para permitir transparência de distribuição, de forma que clientes possam utilizar serviços em objetos sem se preocupar com questões como localização e diferentes formas de acesso. Por outro lado, o desenvolvimento da parte servidora das aplicações (isto é, dos objetos que provêem serviços para os clientes) não é priorizado nestas tecnologias. Em particular, os desenvolvedores de objetos devem explicitamente lidar com aspectos como a configuração e uso de serviços padrão, bem como com o controle do ciclo de vida dos objetos. Além disso, a construção de configurações de objetos é dificultada por não haver um padrão para a implantação de objetos, bem como para explicitar as interações e dependências entre os mesmos. Isto torna o desenvolvimento de serviços uma atividade complexa e que muitas vezes resulta em um baixo potencial para reutilização e portabilidade.

Apesar desta orientação original tomada pelas tecnologias de middleware convencionais, constata-se que o desenvolvimento de objetos provedores de serviços responde pela maior parte do custo total de desenvolvimento de aplicações. Isto é especialmente válido no caso de serviços cujo acesso é feito via *web* (veja a seção 1.4), para as quais a parte cliente geralmente se baseia em protocolos e interfaces padronizados, o que aumenta ainda mais a proporção do tempo gasto na implementação de serviços em relação ao tempo gasto com a implementação de clientes. Note que um rápido ciclo de desenvolvimento tem se tornado, em geral, um requisito cada vez mais importante para o desenvolvimento de aplicações.

Com base na observação desta deficiência básica, procurou-se estender as tecnologias de middleware convencionais mais difundidas com um suporte mais adequado para a implementação de serviços. No caso específico das tecnologias estudadas na Seção 1.3, observa-se a seguinte evolução dos padrões: CORBA 3 [43, 39] (e seu modelo de componentes) foi desenvolvido para suceder o modelo de objetos padrão de CORBA; J2EE [54](e Enterprise Java Beans, ou EJB) foi desenvolvido para estender e adequar o modelo de programação de Java/RMI para ambientes servidores; e .NET [35] se propõe a ser o sucessor do modelo de objetos padrão da Microsoft (COM/DCOM). Em comum, estas novas tecnologias empregam o conceito de componentes de software como os blocos básicos para a construção de aplicações, utilizando abordagens semelhantes, embora nem sempre compatíveis. O restante desta seção é dedicado ao estudo destas tecnologias, com ênfase no padrão CORBA 3.

1.5.1 CORBA 3

A versão 3 da especificação de CORBA [48] começou a ser desenvolvida pela OMG em 1998, em paralelo com o desenvolvimento da versão 2. A intenção aparente foi de continuar evoluindo o padrão comercial (permitindo o aperfeiçoamento de implementações existentes), ao mesmo tempo em que outra força tarefa se concentraria no futuro de CORBA. Visto de outra forma, o termo CORBA 3 refere-se a um conjunto de especificações construído em cima de CORBA 2. Na prática, esta abordagem paralela fez com que várias especificações compreendidas por CORBA 3 fossem sendo incorporadas nas versões intermediárias de CORBA 2. Assim, ao terminar o ciclo de desenvolvimento da versão 2, todas as características planejadas para CORBA 3 estariam incluídas, tornando mais ameno o processo de transição entre as versões.

O conjunto de especificações representado por CORBA 3 pode ser dividido em três categorias [49], conforme descrito abaixo.

- *Integração com a Internet e Java:* que inclui um mapeamento reverso de Java para IDL (para permitir o uso de objetos Java no contexto de CORBA), a especificação de *firewall* (para permitir que requisições CORBA possam ser devidamente tratadas quando atravessando uma firewall), e a especificação do uso de URLs para referenciar objetos CORBA;

- *Controle de Qualidade de Serviço:* que inclui o suporte para mensagens assíncronas, bem como MinimumCORBA, tolerância a falhas, e tempo real;

- *O Modelo de Componentes de CORBA:* que incrementa o modelo de objetos de CORBA com o suporte padronizado para componentes de software.

Como visto na Seção 1.3, os dois primeiros itens acima já são parte da especificação atual de CORBA 2 (versão 2.6). O modelo de componentes, por outro lado, teve sua versão final publicada em novembro de 2001, e constitui agora o elemento distintivo de CORBA 3. De fato, o termo CORBA 3 tem sido largamente utilizado para se referir ao CCM. Por sua importância, devotamos a próxima seção ao estudo deste modelo de componentes.

1.5.2 O Modelo de Componentes de CORBA

O Modelo de Componentes de CORBA, ou CCM [43], representa um novo modelo de programação construído com base no modelo de objetos original de CORBA. Como tal, o CCM provê um modelo de programação de mais alto nível, baseado no conceito de componentes como elementos de software com alto potencial de reutilização [55]. CCM fornece suporte padronizado para o empacotamento de componentes e para a montagem de aplicações com base em configurações de componentes. Além disso, a implantação e a instalação de componentes e configurações são também feitas de acordo com mecanismos padrão, o que favorece sua portabilidade para implementações diferentes do modelo. CCM é portanto uma tecnologia particularmente voltada para a implementação do lado servidor de aplicações distribuídas, de modo a propiciar maior produtividade.

As novas características introduzidas pelo CCM são construídas com base nos conceitos convencionais de CORBA, através de uma extensão de CORBA IDL para a declaração

de componentes (conhecida como IDL3), e através do mapeamento dos conceitos introduzidos nestas extensões em termos da IDL convencional (aqui chamada de IDL2). O modelo de CCM, incluindo o suporte de tempo de execução para componentes, é portanto implementado em termos do ORB e de outros serviços tradicionalmente providos por CORBA.

A especificação do CCM se divide em cinco modelos e um meta-modelo [32], conforme descritos abaixo.

- *Modelo abstrato:* especifica os meios para a definição de tipos de componentes, incluindo a definição das interfaces providas e usadas pelos componentes. Isto é feito através de IDL3, que introduz novos meta-tipos para o suporte à definição de componentes. Em geral, podemos dizer que o modelo abstrato corresponde à essência do próprio CCM.

- *Modelo de programação:* introduz o *Framework de Implementação de Componentes* (CIF), que permite definir a forma como os aspectos funcionais de um componente interagem com os serviços providos pela plataforma (aspectos não-funcionais). A linguagem de definição de implementação de componentes (CIDL) é utilizada para especificar tais aspectos.

- *Modelo de empacotamento (packaging):* define a forma como tipos e implementações de componentes são empacotados para distribuição. Um componente é distribuído sob a forma de um arquivo ZIP contendo as várias partes da definição do componente, juntamente com um descritor em XML que especifica como a infra-estrutura do CCM deve criar e gerenciar o componente. Este modelo permite o empacotamento tanto de componentes individuais quanto de configurações inteiras de componentes.

- *Modelo de implantação:* define o processo de implantação de componentes (ou configurações de componentes) em um ambiente distribuído. Isto inclui a instalação de componentes (a partir de pacotes de componentes), a criação de instâncias específicas dos componentes, e sua interconexão para formar aplicações completas.

- *Modelo de execução:* especifica o ambiente de execução para instâncias de componentes, o qual é baseado no conceito de *container*. Um container desempenha o papel fundamental de encapsular o componente e esconder dele o uso e gerenciamento de aspectos não-funcionais, tais como ciclo de vida, persistência, transações e segurança. Isto significa que, do ponto de vista do componente, o uso dos serviços de objeto CORBA correspondentes a estes aspectos é tornado transparente.

- *Meta-modelo:* define os conceitos utilizados nos modelos acima descritos, utilizando UML (dentro do contexto da *Meta-Object Facility*). Em particular, o meta-modelo define as construções de linguagem presentes em IDL3 e CIDL, e permite a geração automática de repositórios de tipos de componentes.

A seguir, examinamos mais de perto as características que definem componentes no contexto do CCM, bem como os modelos de programação e de execução para componentes.

Definindo Componentes CORBA

Componentes (ou mais precisamente, tipos de componentes) são definidos em IDL3 por meio da palavra-chave `component` e das construções relacionadas. A Figura 1.6 apresenta um trecho de código em IDL3 que contém definições de componentes para o exemplo da biblioteca introduzido na Seção 1.3 (ver Figura 1.2). Uma representação pictórica destes tipos de componentes, por sua vez, é apresentada na Figura 1.7. No texto que se segue, discutimos cada um dos aspectos envolvidos nas definições.

```
component CompUsuario {
    provides Usuario usuario;
};

valuetype LivroReservado : Components::EventBase {
    public string titulo;
};
component CompLivro suports Livro {
    // Fonte de eventos:
    publishes LivroReservado livro_reserv;
};

component MinhaBiblioteca suports EstoqueDeLivros {
    // Atributo:
    attribute long maximo_livros;
    // Facetas:
    provides Emprestimo emprestimo;
    provides Admin admin;
    // Receptáculos:
    uses multiple Usuario usuario;
    uses multiple Livro livro;
    // Destino de eventos:
    consumes LivroReservado livro_reserv;
    // Outras definições do componente:
    ...
};
```

Figura 1.6: Exemplo de definição de componentes em CORBA IDL3

Portas. Conforme visto na Figura 1.6, IDL3 permite explicitar as interações de um componente com outros elementos do sistema, tais como clientes ou outros componentes, através do mecanismo de *portas*. Três tipos de portas são definidos:

- *Facetas:* interfaces providas pelo componente, sendo que um mesmo componente pode prover zero ou mais interfaces, cada uma apresentando uma visão diferente do componente para os seus clientes. No nosso exemplo, o (tipo de) componente `MinhaBiblioteca` provê duas facetas distintas: `empréstimo` (para uso rotineiro dos usuários da biblioteca, para a busca, retirada e devolução de livros) e `admin` (que representa a visão que os administradores têm da biblioteca, e que permite a adição de novos livros ao acervo).

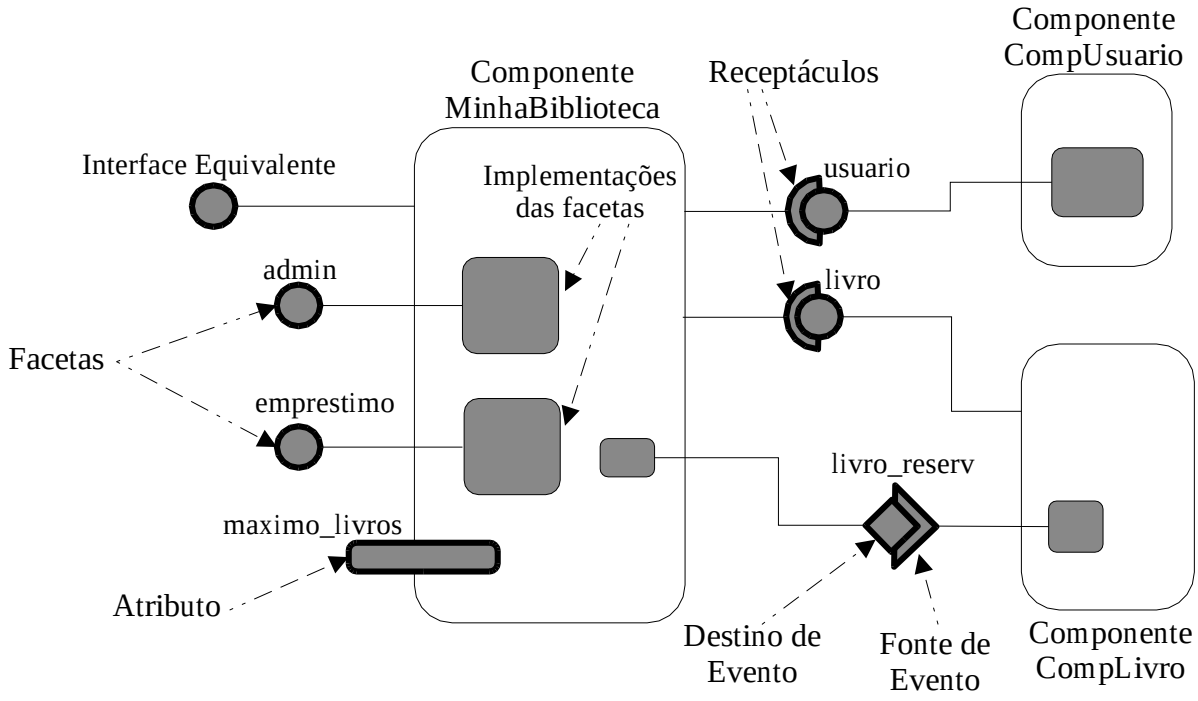

Figura 1.7: Uma configuração de componentes CORBA

- *Receptáculos:* representam os tipos de interfaces exigidas pelo componente. Através de um receptáculo, um componente pode se conectar (isto é, obter referências de objeto) a interfaces providas por outros componentes, de modo a formar montagens (*assemblies*) de componentes. No nosso exemplo, o componente `MinhaBiblioteca` declara dois receptáculos: `usuario` (para ser capaz de informar a entrega de livros ou o registro de multas aos usuários) e `livro` (para ter acesso aos atributos que definem uma determinada instância de `Livro`). Note que os dois receptáculos declarados têm cardinalidade múltipla, o que significa que várias instâncias dos componentes `CompLivro` e `CompUsuario` podem estar conectadas a eles ao mesmo tempo. No caso oposto, de receptáculos simples (sem a palavra-chave `multiple`), apenas um componente poderia estar conectado de cada vez.

- *Eventos:* permite representar o modelo de comunicação do tipo *publish/subscribe*, segundo o qual componentes podem interagir entre si de maneira fracamente acoplada. Componentes declaram seu interesse em produzir ou consumir eventos através da definição de fontes e destinos de eventos. Então, quando um evento é produzido, ele é distribuído para todas as instâncias de componentes que se inscreveram para receber notificações do mesmo. No exemplo, o componente `MinhaBiblioteca` está inscrito para receber notificações de eventos do tipo `LivroReservado`, o qual é produzido por componentes do tipo `CompLivro`.[6]

[6]Observe que tipos de eventos (como `LivroReservado`) são implementados através de *value types* (segundo a especificação de *Objects by Value* [42]), o que significa que eventos podem conduzir informações adicionais, tal como no exemplo, onde o evento informa o título do livro reservado.

É interessante observar que IDL3, através da definição de portas, pode ser vista como uma linguagem de configuração de componentes, no sentido de linguagens como Darwin [31], uma vez que permite a descrição explícita de configurações de componentes, através das conexões entre os mesmos.

Atributos. Além de portas para interação com outros componentes, CCM também permite a definição de atributos para componentes, os quais são normalmente usados para configuração de componentes. No exemplo da Figura 1.6, o atributo `maximo_livros` define, durante a sua configuração, o número máximo de livros que a biblioteca pode ter. Atributos são acessados através de operações do tipo *get* e *set*.

Interfaces equivalentes. O acesso a um componente CORBA pode também ser feito através de sua *interface equivalente*, a qual é definida implicitamente e tem o mesmo nome que o (tipo do) componente. Esta interface permite que clientes que não aderem ao modelo de CCM possam acessar o componente utilizando o modelo de objetos convencional de CORBA. Do ponto de vista de tais clientes, uma referência ao componente se iguala a uma referência para a interface equivalente do mesmo. As características específicas de aplicação disponíveis em interfaces equivalentes são especificadas através de herança a partir das interfaces implementadas (*supported interfaces*) pelo componente. Como exemplo, na Figura 1.6, a interface equivalente do componente `MinhaBiblioteca` é derivada da interface `EstoqueDeLivros`, que é a interface original para acesso às funcionalidades da biblioteca (ver Figura 1.2 na Seção 1.3). Desta forma, clientes elaborados para a versão não componentizada podem interagir, sem necessidade de alterações no seu código, com a versão baseada em componentes.

Navegação entre as interfaces de um componente. Um cliente pode, a partir de uma dada interface de um componente (o que inclui a interface equivalente), obter referências para as demais interfaces do mesmo. Desta forma, é possível descobrir dinamicamente as facetas providas por um dado componente.

Homes. Este elemento do modelo permite padronizar o gerenciamento do conjunto de instâncias de um tipo de componente. Em particular, o *Home* de um componente permite gerenciar o ciclo de vida de suas instâncias, com operações para criar, buscar e destruir instâncias. *Homes* correspondem a um novo meta-tipo, introduzido pelo modelo de componentes de CORBA, o qual pode ser declarado através da palavra-chave `home`, tal como no exemplo mostrado na Figura 1.8, que define um *Home* para o tipo de componente `MinhaBiblioteca`, que fornece uma operação para a criação de instâncias (declarada através da palavra-chave `factory`) e outra para a busca de instâncias específicas do componente (declarada com a palavra-chave `finder`). (Note que, neste caso, a busca é feita através de uma chave primária unicamente associada a cada instância do componente.) Desta forma, para utilizar um componente, um cliente deve primeiramente obter uma referência para o seu *Home*, a partir do qual pode criar ou encontrar uma instância apropriada do mesmo.

```
valuetype MinhaBibliotecaKey : Components::PrimaryKeyBase {
    public long identifier;
};
home MinhaBibliotecaHome
  manages MinhaBiblioteca
  primaryKey MinhaBibliotecaKey {
    factory create_minha_biblioteca (in long id)
      raises (Components::DuplicateKeyValue,
              Components::InvalidKey);
    finder search_minha_biblioteca (in long id)
      raises (Components::UnknownKeyValue,
              Components::InvalidKey);
};
```

Figura 1.8: Exemplo de definição de *Home*

Implementando Componentes

O principal objetivo do modelo de programação do CCM é prover meios para descrever os aspectos não-funcionais dos componentes, permitindo a geração automática desta parte da implementação, de forma que o desenvolvedor de componentes necessite codificar apenas a parte funcional dos mesmos [32][7]. A integração destas duas partes é descrita no CIF (o *framework* de implementação de componentes CORBA). A especificação da estrutura de uma implementação de componente é feita em CIDL, que também permite descrever o estado persistente do componente, se apropriado. O CIF utiliza descrições em CIDL para gerar esqueletos (*skeletons* que automatizam boa parte do comportamento básico de componentes, incluindo navegação, ativação, gerenciamento de estado e ciclo de vida, etc.

A implementação de um componente é definida através de *executores*, que descrevem o comportamento dos diversos elementos que constituem um componente. Dois tipos de executores são providos: executor de *Home* (que fornece a implementação para o *Home* do componente, sendo, em sua maior parte, gerado automaticamente) e executor de componente (que fornece a implementação para as diversas facetas do componente, sendo conceitualmente semelhante a serventes no contexto do POA). Uma implementação de componente é então definida em termos de uma *composição* destes elementos, bem como, opcionalmente, da declaração do gerenciamento de estado do componente.

Quatro tipos de componentes são definidos no CIF, de acordo com o modelo de persistência de estado adotado (que é definido pelo tipo de container que hospeda o componente):

- *Serviço:* componentes sem estado e sem identidade, cujo ciclo de vida é limitado ao tempo necessário para servir uma única invocação (embora seja também possível manter *pools* de componentes de serviço, os quais são usados sob demanda);

- *Sessão:* componentes com estado transiente e cuja identidade não é persistente; seu ciclo de vida é associado a uma seqüência de chamadas feitas pelo cliente (tal como no caso de componentes iteradores);

[7] Note, contudo, que o único aspecto não-funcional explicitamente contemplado na atual especificação do CIF diz respeito à persistência do estado de componentes.

- *Processo:* componentes com estado estado e identidade persistentes; este tipo é normalmente utilizado para representar processos de negócios (por exemplo, uma cesta de compras em uma aplicação de comércio eletrônico);

- *Entidade:* similar ao tipo anterior, mas cujos clientes têm conhecimento explícito da persistência do componente; geralmente utilizados para modelar entidades de negócio, tais como os clientes de uma empresa, contas-correntes, etc.

O Modelo de Execução: Containers

Containers são o mecanismo fundamental para se prover o encapsulamento de componentes, bem como o acesso transparente a serviços de objetos, tais como persistência, transações, segurança e notificação de eventos. Um container representa o ambiente de tempo de execução para uma (e somente uma) instância de componente. Um container também provê recursos de baixo nível, tais como memória e processador, para o componente nele hospedado. O mecanismo de containers pode, portanto, ser visto como um *framework* para integrar os serviços de objeto acima mencionados ao comportamento funcional de componentes, tendo como base os descritores de componentes especificados durante o empacotamento do componente.

Além de conter uma instância de componente, um container também possui uma instância do POA especializada para o tipo de componente hospedado (e especificada no descritor do componente). O container permite automatizar aspectos como a ativação, o gerenciamento e o uso de serviços do POA. O *framework* de containers define, ainda, as seguintes características:

- Interfaces externas para que clientes possam utilizar o componente hospedado;

- Interfaces locais para acesso a serviços do sistema por parte do componente hospedado;

- *Callbacks* para gerenciamento de instâncias de componentes (tal como para ativar ou desativar um componente);

- Funcionalidade para a navegação entre as interfaces do componente hospedado (através de sua interface *Home*);

- Conexão das facetas e receptáculos do componente, de forma que o mesmo possa operar corretamente;

- Canais de eventos para transmitir e receber eventos.

Observe que o acesso às diversas características de um componente expostas por um container, bem como aos serviços de objeto CORBA é realizado através do ORB, como ilustrado na Figura 1.9. A figura também mostra os principais elementos de um container, como descritos acima. Neste contexto, clientes preparados para o uso de componentes podem fazer uso de todos os serviços providos através do container (por exemplo, para a navegação das interfaces de um componente). Por outro lado, clientes convencionais podem ainda fazer chamadas ao componente (através do uso de sua interface equivalente, como descrito na Seção 1.5.2), embora sem os benefícios do container.

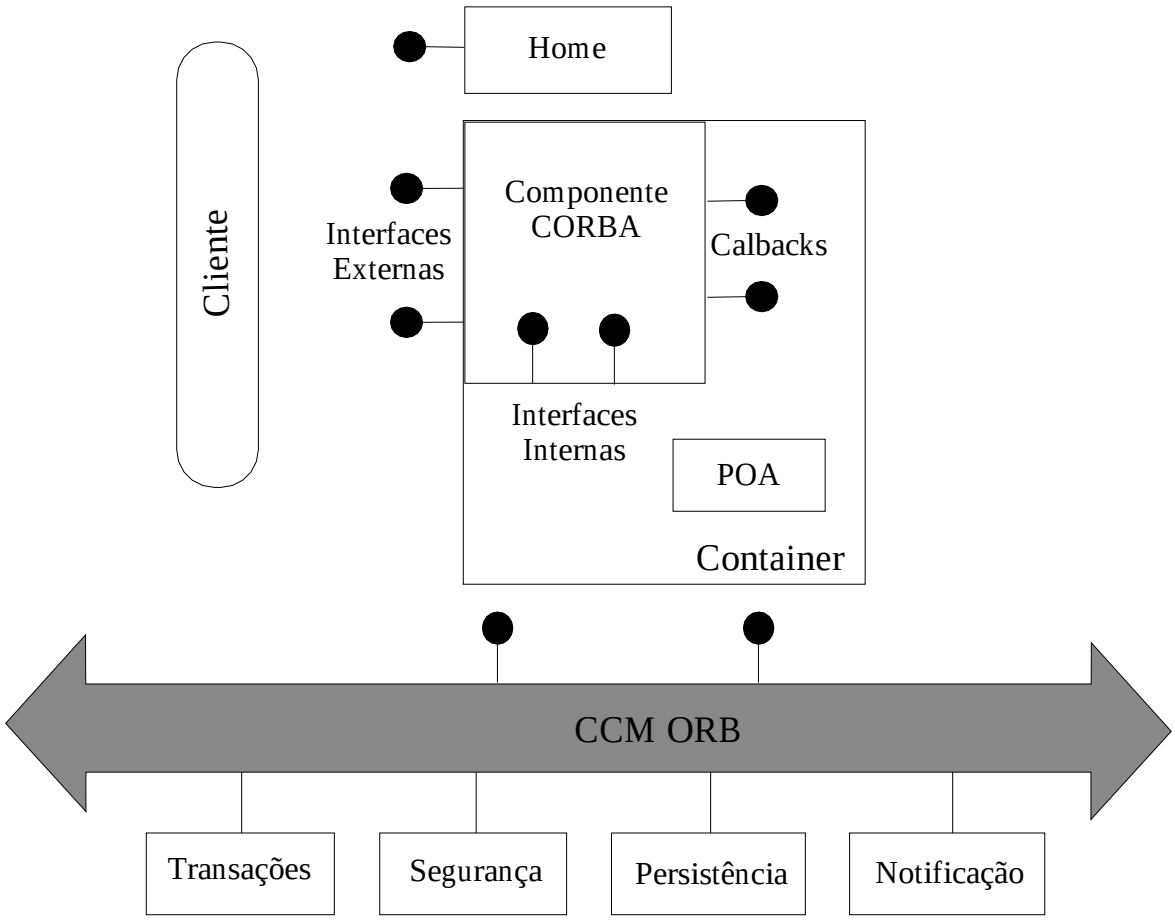

Figura 1.9: A arquitetura de programação com containers.

Considerações Gerais

Como se pode concluir da discussão acima, CCM pode ser visto como um conjunto de *frameworks* para a definição, implementação, empacotamento, implantação e execução de componentes no ambiente de CORBA. Isto é possível através da tradução das definições CCM em temos de IDL convencional, segundo mapeamentos padrão. Desta forma, as funcionalidades de alto nível providas pelo CCM são implementadas sob a forma de interfaces IDL, através das quais tais funcionalidades são acessadas pelas entidades envolvidas no modelo (clientes, containers, componentes, ferramentas, etc.). Detalhes sobre estes mapeamentos podem ser encontrados na especificação do CCM [43], enquanto que [32] provê uma boa visão geral dos mesmos. A grande vantagem do modelo, contudo, está na geração automática de implementações para as interfaces que definem funcionalidades padrão do modelo. Como resultado, o desenvolvimento de componentes torna-se uma atividade mais eficiente, uma vez que o desenvolvedor deve apenas codificar os aspectos funcionais da aplicação, sendo que aspectos não-funcionais devem apenas ser especificados de maneira declarativa (nos descritores de componentes, em XML).

Entretanto, algumas críticas são pertinentes em relação ao CCM. Em particular, a definição estática de containers (de acordo com a especificação atual) penaliza a flexi-

bilidade do modelo. Por exemplo, não é possível definir novas categorias de containers, que permitam prover ambientes de execução diferentes daqueles atualmente oferecidos. Além disso, o conjunto de aspectos não-funcionais (serviços de objetos) suportados por containers é fixo, o que impede a inclusão de novos serviços que se façam necessários. Seria interessante poder contar com containers adaptáveis, os quais permitissem a adição (estática ou dinâmica) de novos aspectos não funcionais a serem gerenciados dentro do *framework*. Note que esta inflexibilidade do modelo atual também reduz sua aplicabilidade fora do ambiente de servidores. Por exemplo, seria interessante poder estruturar aplicações móveis em termos de componentes CORBA, mas isto é praticamente inviabilizado em virtude da impossibilidade de se configurar a plataforma CCM para economia de recursos.

Finalmente, vale observar que atualmente ainda são poucas as implementações do Modelo de Componentes de CORBA. A primeira implementação completa, chamada de OpenCCM [33], teve sua primeira versão estável publicada apenas recentemente. Isto significa que experiências práticas com o CCM estão apenas começando, sendo que deficiências do modelo (como, por exemplo, com relação ao seu desempenho) podem vir a ser identificadas. CCM se encontra, portanto, em um estágio semelhante àquele em que CORBA se encontrava na primeira metade da década de 90 e, certamente, pode-se esperar um considerável amadurecimento do padrão nos próximos anos. Apesar disto, vale ressaltar que o alinhamento e compatibilização do CCM com o modelo de componentes Enterprise Java Beans (ver Seção 1.5.3), cuja aceitação é ampla atualmente, significa que pelo menos uma parte do padrão já foi comprovada pelo uso e pode ser considerada relativamente estável. Outras partes do padrão, que estendem o modelo de EJB, contudo, ainda necessitam ter sua efetividade comprovada na prática.

1.5.3 J2EE

A plataforma J2EE (*Java 2 Enterprise Edition*) [54, 2] é uma extensão do modelo original de Java que provê um ambiente de suporte para o desenvolvimento de servidores de aplicações distribuídas. As características introduzidas por J2EE podem ser divididas em dois grupos, descritos a seguir.

- Uma infra-estrutura (ou servidor) de tempo de execução para hospedar e gerenciar aplicações. Semelhante ao CCM, esta infra-estrutura é baseada em *containers*, que gerencia componentes de aplicação e provê acesso padronizado aos serviços de suporte.

- Um conjunto de APIs que estendem Java e são usadas na construção de aplicações. Estas APIs definem o modelo de programação para aplicações J2EE, e provêem um padrão para o acesso a serviços de suporte subjacentes (evitando assim o uso de interfaces de serviço proprietárias). Dentre as APIs de J2EE encontram-se:

  - JDBC: para acesso a bancos de dados relacionais
  - Enterprise Java Beans (EJB): o *framework* de componentes de J2EE
  - Java Servlets: para a construção de aplicações *web* dinâmicas
  - JavaServer Pages (JSP): que permite o desenvolvimento de aplicações *web* baseadas em templates

- Java Message Service (JMS): que provê serviços de middleware orientado a mensagens
- Java Transaction API (JTA): para o suporte a transações distribuídas
- Java API for XML parsing (JAXP)
- Java Connector Architecture (JCA): para integração de aplicações legadas com componentes J2EE
- Java Authentication and Authorizadion Service (JAAS)
- Java Interface Definition Language (IDL) API: permite que componentes J2EE invoquem objetos de CORBA via IIOP
- RMI-IIOP API: uma implementação de Java RMI sobre IIOP. É o protocolo padrão para uso entre containers em J2EE.
- Java Naming and Directory Interface (JNDI): provê uma forma padronizada para acesso a diferentes serviços de nomes e de diretório distribuído.

A maioria das APIs de J2EE são especificadas independentemente de sua implementação, sendo que esta última pode ser provida por terceiros. Através do uso destas APIs padrão, aplicações se tornam independentes da implementação dos serviços por elas utilizados, o que favorece sua portabilidade. Entretanto, uma discussão destas APIs está fora do contexto deste curso. No que se segue, nos concentramos em uma breve descrição da arquitetura do ambiente de execução de J2EE.

A arquitetura de J2EE. Uma plataforma J2EE típica provê suporte para vários containers, os quais por sua vez fornecem o ambiente de execução para os componentes da apliação. Como no CCM, a infra-estrutura provida por containers em J2EE facilitam o desenvolvimento de componentes, através do uso declarativo de serviços, que evita a necessidade de tratamento explícito aos aspectos não-funcionais das aplicações. Em outras palavras, um container provê, para seus componentes, um ambiente com certas propriedades não-funcionais agregadas. Esta característica é implementada através de interceptadores em nível de requisições (*requests*), que permitem modificar (ou personalizar) a forma como requisições são manipuladas. Containers podem, portanto, ser vistos como *wrappers* de componentes, uma vez que requisições são primeiro interceptadas pelo container, antes de serem repassadas para os componentes-destino). Diferentemente do CCM, J2EE especifica quatro tipos de componentes com finalidades e uso distintos:

- *Web containers:* que hospedam servlets e páginas JSP;
- *EJB containers:* que hospedam componentes EJB;
- *Applet containers:* que hospedam applets de Java;
- *Containers de clientes de aplicação:* que hospedam aplicações Java convencionais.

Dentre estes quatro tipos, apenas EJB containers podem ser diretamente mapeados para CCM, sendo que os demais são específicos para acomodar outros elementos típicos de ambientes Java (note que os dois últimos tipos de containers permitem encapsular também o lado cliente das aplicações). Componentes em um container acessam serviços externos através das APIs de J2EE. Clientes, por sua vez, fazem acesso a componentes através

dos respectivos containers dos componentes. Dois tipos de clientes são previstos: clientes *web* (que utilizam serviços providos por *web containers*, através do protocolo HTTP) e clientes EJB (que podem ser tanto clientes de aplicação, como componentes *web* ou EJB, e utilizam serviços providos por EJBs via RMI-IIOP ou via chamadas locais, no caso de clientes EJB no mesmo container).

A arquitetura de containers. Um container engloba um ou mais componentes de aplicação (que podem ser servlets, páginas JSP, EJBs, etc., dependendo do tipo do container), juntamente com os descritores de implantação de cada um. O container em si é constituído pelas partes descritas a seguir.

- *Contrato de componente:* um conjunto de APIs especificadas pelo container e que os componentes devem implementar ou estender para que possam ser gerenciados no contexto do container. Exemplos disto são interfaces com operações para criar, inicializar, desativar, reativar e destruir componentes.

- *APIs de serviço do container:* que provê acesso, na forma de proxies, a serviços externos ao container. Exemplos disto são as APIs padrão de J2EE listadas acima.

- *Serviços declarativos:* são serviços que o container interpõe entre clientes e componentes, conforme especificado no descritor de implantação de cada componente. Desta forma, é possível agregar aspectos não-funcionais a um componente sem a necessidade de programar tais aspectos.

- *Outros serviços do container:* tais como gerenciamento do ciclo de vida de componentes, "pooling" de recursos, e "clustering" (que permite distribuir os componentes de um container em mais de uma máquina hospedeira, de forma a permitir o balanceamento de carga).

Tecnologias de componentes utilizadas. J2EE utiliza dois tipos distintos de componentes: componentes *web* e componentes EJB. Componentes *web* são baseados em duas tecnologias distintas:

- *servlets* (pequenos programas Java que permitem estender programaticamente a funcionalidade de servidores *web*), que permitem construir aplicações que rodam em servidores *web*, de forma a embutir parte da lógica da aplicação no processo de requisição-resposta de HTTP;

- *JavaServer Pages*, que permitem embutir componentes de aplicação em páginas *web*, de forma a permitir a geração dinâmica de conteúdo de maneira programática. Para que possam receber requisições de clientes, componentes JSP são compilados pelo container para gerar servlets equivalentes.

A invocação destes componentes é feita através de HTTP, sendo que as respostas a requisições são encapsuladas em HTML ou XML. Note que, para ambos os tipos de componentes, o container se comporta como um servidor *web* (uma vez que pode responder a requisições em HTTP).

Componentes EJB, por outro lado, representam uma tecnologia mais robusta e escalável para o desenvolvimento de aplicações distribuídas. EJBs podem ser definidos

como unidades de software reutilizáveis contendo a lógica da aplicação. (Aspectos não-funcionais, tais como controle de transações, por sua vez, são providos automaticamente pelo container, conforme descrito acima.) Três tipos de EJBs são definidos:

- *Session beans:* semelhante a componentes de sessão no CCM, são objetos transientes criados para servir uma seqüência de requisições do cliente, após o que são destruídos;

- *Entity beans:* semelhante a componentes de entidade no CCM, são objetos persistentes e com identidade única, usados para modelar entidades de dados permanentes;

- *Message-driven beans:* permitem a integração do Java Message Service (para o suporte a mensagens assíncronas) com o modelo de componentes, de forma que EJBs possam receber mensagens através do JMS.

Componentes EJB são, de modo geral, comparáveis a componentes CORBA. De fato, a especificação do CCM pode ser vista como um super-conjunto de EJB, sendo que a interoperabilidade entre EJB e CCM faz parte da especificação deste último. Note, contudo, que CCM define o modelo de componentes em cima do modelo de objetos de CORBA, enquanto que EJBs são suportados pelo modelo de objetos de Java, o que faz com que EJBs sejam dependentes de linguagem.

1.5.4 .NET

.NET [35] é o membro mais novo da família de tecnologias objetos proposta pela Microsoft. A plataforma é construída sobre a experiência adquirida com o uso de COM/DCOM, procurando eliminar deficiências identificadas nestas tecnologias anteriores, bem como introduzir novos conceitos. Os objetivos gerais desta tecnologia são o suporte a computação distribuída, componentização de software, suporte a serviços corporativos e, em especial, suporte a serviços *web*. Alguns pontos fundamentais da abordagem adotada em .NET são discutidos a seguir.

No centro da especificação de .NET está o *Common Language Runtime* (CLR), que define um modelo de objetos (também referido como um sistema de tipos) normalizado, em termos do qual componentes e serviços são definidos. Implementações de componentes em uma determinada linguagem são mapeados para o CLR, o qual provê o ambiente de execução para os componentes. Em tese (pelo menos), este mapeamento permite que partes distintas de um mesmo componente sejam implementadas em linguagens diferentes, desde que haja um mapeamento de cada linguagem utilizada para o CLR. Na prática, contudo, apenas um sub-conjunto de cada linguagem pode geralmente ser completamente mapeado para o CLR, o que impõe uma barreira natural à idéia de desenvolvimento em várias linguagens. Note que este conceito não se confunde com interoperabilidade entre componentes escritos em linguagens diferentes, o que é suportado por .NET de maneira semelhante como acontece em DCOM (ver Seção 1.3).

No que diz respeito à construção de aplicações a partir de componentes, .NET faz uso intenso de meta-informações, em XML, para explicitamente descrever as dependências entre componentes relacionados. Desta forma, ao se remover, incluir ou substituir um componente em uma configuração, é possível definir quais outros componentes da configuração são afetados (por dependerem do componente alterado), de forma que os mesmos

possam também ser alterados (por exemplo, substituídos por versões mais recentes) caso necessário.

Para a comunicação entre componentes, o protocolo preferido é SOAP (*Simple Object Access Protocol*), que consiste em um vocabulário (DTD) XML capaz de conduzir chamadas de métodos remotos, bem como seus parâmetros e resultados. Isto favorece o uso de .NET na Internet, em particular, para o suporte e acesso a serviços *web*. Entretanto, apesar de sua portabilidade, o uso de SOAP é criticado por penalizar a eficiência das interações entre componentes (uma vez que chamadas são conduzidas, em última análise, através de mensagens textuais, que devem ser interpretadas através de parsing).

Uma outra crítica a .NET diz respeito à virtual ausência de suporte para interoperabilidade em ambientes heterogeneidade no nível de sistemas operacionais. Isto se deve não apenas ao fato de não existirem implementações atuais de .NET para outros sistemas operacionais fora do contexto Microsoft, mas também ao fato de que o CLR e os serviços disponíveis neste ambiente são fortemente vinculados aos sistemas operacionais da família MS Windows.

1.5.5 Avaliação

Com relação às aplicações de middleware discutidas na seção 1.4 e seus requisitos, podemos fazer algumas observações sobre o estado atual de tecnologias de middleware mais difundidas. No que tange ao suporte para rápido desenvolvimento e integração com a Internet, estas tecnologias têm demonstrado a maturidade necessária, através da separação bem-sucedida entre aspectos funcionais e não-funcionais durante o desenvolvimento de aplicações. Além disso, o requisito de elevada disponibilidade da plataforma e das aplicações por ela suportadas tem sido satisfeito com a adoção de modelos de componentes mais robustos, que incorporam aspectos como o controle de transações e segurança.

Entretanto, pode-se afirmar que os requisitos gerados por aplicações de multimídia distribuída e computação ubíqua continuam não sendo satisfeitos pela geração atual de plataformas de middleware. Isto se deve, principalmente à falta de flexibilidade destas plataformas, no que diz respeito a configurabilidade. Em geral, os únicos mecanismos existentes que permitem um certo grau de configurabilidade do serviço da plataforma são baseados na seleção dos serviços providos aos componentes e no conceito de interceptadores. Contudo, este mecanismo não são poderosos o suficiente para realizar alterações na configuração interna da plataforma, de modo a adequá-la a ambientes com características distintas (que podem variar, por exemplo, de servidores de aplicações a dispositivos embutidos usados para computação ubíqua). Além disso, o aspecto de adaptação dinâmica continua sem suporte padrão nestas tecnologias. É interessante notar que a adoção do paradigma de componentes permite alto grau de flexibilidade na configuração de aplicações, sendo que a sua aplicação no design e implementação da própria plataforma oferece oportunidades semelhantes, conforme discutido na próxima seção.

1.6 Novas Abordagens: Rumo à Flexibilização

Como vimos na seção 1.4, os novos ambientes computacionais e suas novas aplicações trazem consigo um alto grau de dinamismo. Isto faz com que as tecnologias estáticas e monolíticas do middleware tradicional se tornem inconvenientes. Na seção 1.6.1, discutimos a importância do desenvolvimento das gerações futuras de middleware através

da tecnologia de componentes. Esta nova forma de organização interna do middleware (baseada em componentes) pode ser estruturada de forma ainda melhor através de técnicas de Reflexão Computacional, descritas na seção 1.6.2.

1.6.1 Middleware Baseado em Componentes

A pesquisa em sistemas orientados a objetos e o seu intenso uso na indústria levou ao desenvolvimento da "programação baseada em componentes". Como vimos na seção 1.5, esta não é uma alternativa à orientação a objetos mas sim uma complementação aos conceitos fundamentais de objetos. Ela acentua a ênfase na busca de módulos de software independentes que possam ser reutilizados em diferentes contextos e combinados de diferentes modos a fim de implementar sistemas complexos.

Sistemas baseados em componentes são construídos através da combinação de vários componentes de software, normalmente armazenados em arquivos no sistema de arquivos local ou em um repositório de componentes acessível através da rede. Programadores determinam quais componentes farão parte do sistema e como será o inter-relacionamento entre eles.

Implementações convencionais de middleware CORBA, Java e DCOM são monolíticas, ou seja, todo o mecanismo interno do middleware é apresentado como uma caixa preta e estarão presentes no sistema queira o usuário ou não. Implementações tradicionais de CORBA são distribuídas como bibliotecas de carga dinâmica que ocupam vários megabytes de memória e que incluem todos os mecanismos definidos pelo padrão CORBA para o ORB. Mesmo que a aplicação necessite apenas de 10% destes mecanismos (e a maioria das aplicações utiliza apenas uma pequena parcela da funcionalidade do ORB), todo o código é carregado no processo da aplicação. Isto é particularmente grave para máquinas com recursos escassos como PDAs, computadores portáteis e sistemas embutidos.

Outra limitação de middleware convencional é que os protocolos e mecanismos utilizados pelo ORB são, em geral, fixos, não podendo se adaptar a diferentes contextos. CORBA e Java RMI quase sempre usam o protocolo TCP/IP para transportar os dados através da rede. Mas este protocolo é sabidamente ineficiente em uma série de situações como em redes sem fio, onde um protocolo como WTCP [51] seria muito mais eficiente. Aplicações que envolvem a transferência de multimídia também não funcionam bem com TCP/IP; neste caso um protocolo específico como VDP [10], para a Internet, ou o xbind [9], para redes ATM, seriam mais apropriados.

O uso da tecnologia de componentes para a construção de middleware pode ajudar a resolver os dois problemas descritos acima. Primeiramente, se o programador é capaz de especificar quais são os serviços do ORB dos quais ele necessitará, o middleware pode carregar apenas aquelas componentes que implementam aquele serviço. O Middleware se configura em tempo de inicialização para que os componentes utilizados pela aplicação estejam disponíveis. Em segundo lugar, se cada aspecto interno do middleware é implementado por um componente diferente, é possível desenvolver versões alternativas para cada tipo de componente, implementando diferentes mecanismos e utilizando diferentes protocolos. O desenvolvedor, então, poderia escolher quais os mecanismos mais apropriados para a sua aplicação.

Um dos primeiros trabalhos a abordar a idéia de middleware baseado em componentes foi o sistema Quarterware [50] desenvolvido na Universidade de Illinois em Urbana-

Champaign em 1997. Quarterware funciona como um arcabouço (*framework*) C++ a partir do qual pode-se construir diferentes instâncias de middleware. Utilizando este arcabouço, é possível desenvolver componentes responsáveis pelo transporte dos dados através da rede utilizando diferentes protocolos de comunicação (TCP, UDP, VDP, WTCP, etc.), diferentes políticas de concorrência, etc. Os mecanismos de *marshaling* e *unmarshaling* são encapsulados em componentes oferecendo diferentes tipos de seriação como o CDR usado em CORBA e o padrão definido pela Sun para Java RMI. Desta forma, os desenvolvedores de Quarterware desenvolveram pequenas implementações de middleware que era capaz de interagir com objetos CORBA e com objetos Java RMI.

A principal limitação de Quarterware era que a sua configuração era feita no momento da carga do middleware na memória. Uma vez determinadas quais as componentes formariam o sistema, não era mais possível reconfigurá-lo. Pesquisas posteriores no mesmo grupo resolveram este problema com LegORB [45] e UIC [44].

UIC (*Universally Interoperable Core*) define um esqueleto baseado em componentes abstratos que encapsulam as funcionalidades encontradas nos principais ORBs, ou seja, protocolos de comunicação, mecanismos para estabelecimento de conexões, seriação e de-seriação, chamada de métodos, escalonamento, geração de referências para objetos, interface de programação para o cliente e para o servidor, gerenciamento de memória e estratégias para concorrência. Implementações concretas destes componentes abstratos são carregadas dinamicamente de forma a atender aos requisitos de aplicações específicas. UIC define diferentes personalidades atendendo às especificações de diferentes padrões de middleware. No momento, a única personalidade em estágio avançado de desenvolvimento é UIC-CORBA que é capaz de interagir com ORBs CORBA utilizando muito menos recursos do que ORBs tradicionais.

Além da configuração inicial no momento em que o middleware é iniciado, UIC permite que os mecanismos internos do middleware sejam reconfigurados em tempo de execução para adaptar o sistema a variações no ambiente. Esta *reconfiguração dinâmica* é desejável, por exemplo, quando um computador móvel deixa de estar conectado a uma rede sem fio e passa a utilizar uma conexão por ondas de rádio. Se o objetivo é otimizar o desempenho do sistema, o melhor a fazer é utilizar um protocolo como TCP enquanto a conexão por fio estiver disponível e chavear o sistema para WTCP quando a conexão sem fio passa a ser utilizada. Outras aplicações podem desejar efetuar outras reconfigurações dinâmicas no middleware quando ocorrem mudanças na disponibilidade de recursos (como memória e processador) ou quando ocorrem mudanças no padrão de acesso do sistema pelos seus usuários.

A vantagem de ORBs baseados em componentes para sistemas com recursos limitados pode ser ilustrada pelo tamanho do código das bibliotecas implementando um cliente minimal e um cliente/servidor minimal em UIC-CORBA [44]. O tamanho do middleware para um cliente minimal capaz de enviar requisições simples para um servidor CORBA padrão é de 29KB no Windows CE, 72KB no Windows 2000 e 18KB no PalmOS. Já o tamanho do UIC-CORBA contendo tanto funcionalidades minimais para o cliente quanto para um servidor simples é de 48.5KB no Windows CE, 100KB no Windows 2000 e 31KB no PalmOS. Estes números chegam a ser até 30 vezes menores do que implementações de middleware tradicional monolítico.

1.6.2 Middleware Reflexivo

Como vimos nos capítulos anteriores, nos últimos 10 anos, uma série de tecnologias de middleware foram desenvolvidas com o intuito de facilitar o desenvolvimento de aplicações complexas, principalmente no contexto de sistemas distribuídos. O middleware fica no meio do caminho, entre o núcleo do sistema operacional e a aplicação (daí o seu nome) e realizam boa parte do trabalho que antigamente era realizado manualmente pelos programadores. Tecnologias como CORBA do OMG, Java da Sun e DCOM e .NET da Microsoft escondem do programador uma série de detalhes complicados das camadas inferiores como estabelecimento de conexões de rede, chamadas de métodos remotos, seriação de argumentos, denominação de objetos, instanciação de serviços, etc.

No entanto, tecnologias de middleware tradicionais não estão preparadas para lidar com os ambientes altamente dinâmicos da infra-estrutura computacional do futuro. Cada vez mais a computação ubíqua com seus computadores móveis, sistemas embutidos e PDAs estará conectada à Internet global através de redes heterogêneas formadas por segmentos Ethernet, ATM, FDDI e redes sem fio de longo, médio e curto alcance. Neste ambiente, teremos sistemas distribuídos compostos por milhões de objetos sendo executados em milhares de máquinas de uma forma muito dinâmica. Objetos serão criados e destruídos a todo segundo. Sob a forma de agentes móveis, objetos migrarão de um ponto ao outro na rede a fim de procurar por recursos computacionais e por informações específicas de forma a cumprir as tarefas solicitadas pelos usuários. Estes últimos também passarão a ser tão dinâmicos quanto o resto do sistema pois terão acesso à rede a partir de sua casa e de seu escritório, telefone celular, automóvel, terminal de serviços públicos, etc.

Estes novos ambientes exigem uma nova forma de estruturação do middleware onde os seus serviços possam ser *configurados* para melhor atender às necessidades do momento e *reconfigurados* quando ocorrerem flutuações ou mudanças significativas no ambiente de execução das aplicações dos usuários.

1.6.3 Por que Middleware Reflexivo?

A maioria das aplicações se beneficiam fortemente da capacidade do middleware de esconder os detalhes das camadas inferiores (do middleware e do sistema operacional) e de oferecer uma interface de programação comum independente do sistema operacional e da arquitetura de hardware. Isso faz com que o desenvolvimento de aplicações e sistemas distribuídos seja facilitado enormemente e também faz com que o software desenvolvido seja muito mais facilmente portado de uma plataforma para outra.

Por outro lado, muitas aplicações também se beneficiam significativamente de ter acesso a informações internas das camadas inferiores como, por exemplo, a disponibilidade dos recursos de hardware e software e os algoritmos e parâmetros de configuração em uso pelo middleware e pelo sistema operacional. Por exemplo, uma aplicação de vídeo-conferência ou de transmissão de multimídia através da Internet pode melhorar a qualidade de serviço dramaticamente se selecionar um protocolo de transporte de rede que seja adequado para a infra-estrutura de rede em uso (rede local sem fio, rede local com fio, ligação de longa distância na Internet, etc.) e a largura de banda disponível naquele momento. Ela pode também se beneficiar em conhecer o contexto físico do local onde o usuário está presente, detectando a presença de um telão na parede e reconfigurando a aplicação para mostrar o vídeo na tela grande. Um sítio de comércio eletrônico pode melhorar o seu tempo de resposta aos clientes se examinar informações sobre a disponibi-

lidade de recursos como memória e processador e mudar dinamicamente a localização dos componentes do servidor, criando réplicas dos serviços mais procurados naquele momento ou mudando as políticas de escalonamento do middleware para priorizar as operações mais importantes para os clientes interativos. Um sofisticado calendário eletrônico para computação ubíqua poderia detectar em qual tipo de plataforma ele está sendo executado (PDA, relógio de pulso, computador de mesa ou grande tela de projeção) de forma que ele possa prover uma interface gráfica que seja otimizada para aquela plataforma.

Em outras palavras, a maioria das aplicações se beneficiam de middleware que esconde os detalhes das camadas inferiores, mas algumas aplicações importantes podem melhorar o seu desempenho significativamente através de interações com o estado dinâmico das camadas inferiores e através da "sintonia fina" do middleware para os requisitos das aplicações [3]. Portanto, um modelo ideal de middleware deve oferecer *transparência* para as aplicações que não estão interessadas nos detalhes e *translucidez* e controle fino para as aplicações que podem se beneficiar disso. Esta é a idéia central do modelo de *Middleware Reflexivo* [26].

1.6.4 O Modelo de Middleware Reflexivo

No modelo reflexivo, o middleware é implementado através de uma coleção de componentes que pode ser configurada pela aplicação. A interface do middleware em si permanece inalterada e pode acomodar aplicações desenvolvidas para middleware convencional. Além disso, o código das aplicações e do sistema podem inspecionar a configuração interna do middleware e, se necessário, reconfigurá-la para adaptá-la a mudanças no ambiente. Essas reconfigurações são feitas através de *meta-interfaces* particulares dos sistemas de middleware reflexivo. Desta forma, passa a ser possível selecionar protocolos de comunicação de rede, políticas de segurança, algoritmos de codificação e vários outros mecanismos para otimizar o desempenho do sistema em diferentes contextos e situações.

Em termos gerais, o modelo de middleware reflexivo se refere ao uso de uma *auto-representação casualmente conectada* de forma a prover inspeção e adaptação do middleware. A fim de obter esta *auto-representação*, o middleware mantém uma representação explícita da sua estrutura interna[8]. Esta auto-representação é *casualmente conectada* se mudanças na representação causam mudanças na implementação do middleware e se mudanças na implementação do middleware causam mudanças na representação.

Além das características mencionadas acima, uma arquitetura completa para middleware reflexivo deve também oferecer suporte para configuração dinâmica do comportamento do middleware (por exemplo, através de interceptadores como veremos a seguir) e para controle fino do gerenciamento de recursos através de meta-interfaces.

1.6.5 Reflexão Computacional

A idéia de Reflexão Computacional surgiu muito antes da existência do termo middleware. Os fundamentos de sistemas computacionais reflexivos foram originalmente introduzidos por Smith [52] e Maes [30] no contexto de Linguagens de Programação. Posteriormente, Kiczales combinou os conceitos de reflexão à programação orientada a objetos no trabalho intitulado *The Art of the Metaobject Protocol* [24]. Existem hoje uma série de linguagens

[8]Podemos dizer que, de certa forma, o middleware passa a "conhecer" a si mesmo e pode passar a "refletir" sobre si próprio, daí a expressão "middleware reflexivo".

de programação reflexivas, dentre elas CLOS [24], baseado em LISP, OpenC++ [11], baseado em C++, e Guaraná [38], baseado em Java.

Grosso modo, pode-se dizer que um sistema é reflexivo quanto ele é capaz de manipular e refletir sobre si próprio da mesma forma que faz sobre o seu domínio de aplicação. Como conseqüência deste processamento introspectivo, um sistema reflexivo oferece a possibilidade de inspeções e mudanças em si mesmo durante a sua execução. Desde a sua aplicação original no contexto de linguagens de programação, o conceito de reflexão obteve uma aceitação mais ampla em outras áreas como sistemas operacionais (veja por exemplo o sistema Apertos [58]) e sistemas distribuídos. Nos últimos cinco anos, a comunidade de middleware percebeu que o modelo reflexivo poderia também ser aplicado na construção de plataformas de middleware trazendo inúmeros benefícios como descrito no início desta seção. Descrevemos agora alguns conceitos fundamentais de sistemas reflexivos.

- **Reificação**: a ação de expor a representação interna de um sistema em termos de entidades programáveis que podem ser manipuladas em tempo de execução. O processo inverso, **absorção**, consiste em refletir as mudanças realizadas nas entidades reificadas no sistema real. Em conjunto, a reificação e a absorção implementam a **conexão causal** entre o sistema e a sua representação.

- **Arquitetura de meta-níveis**. Um sistema reflexivo possui arquitetura de meta-níveis quando ele é estruturado explicitamente em termos de um **nível base**, que lida diretamente com as funcionalidades do sistema, e um **meta-nível**, que lida com o processamento reflexivo.

- **Meta-Objeto e Protocolo de Meta-Objeto (MOP)**. Em sistemas reflexivos orientados a objetos, as entidades presentes no meta-nível são chamadas de meta-objetos. O protocolo de interação com os meta-objetos é chamado de protocolo dos meta-objetos (MOP) e é o que define o estilo de reflexão provido pelo sistema.

- **Reflexão Estrutural**: a habilidade de uma linguagem de programação em prover uma reificação completa do programa em execução, por exemplo, em termos de seus métodos e estado. No caso de middleware reflexivo, seria a habilidade do sistema em prover uma reificação completa da estrutura interna do middleware. Isto permite ao programador inspecionar e mudar a funcionalidade do middleware (ou programa, no caso de linguagens reflexivas) e o modo como ele interage com o seu domínio.

- **Reflexão Comportamental**: a habilidade de uma linguagem em prover uma representação completa de sua semântica em termos dos aspectos internos do seu ambiente de tempo de execução. Isto permite que o programador inspecione e mude o modo como o ambiente processa o programa, por exemplo, no que diz respeito a propriedades não funcionais e de gerenciamento de recursos. No caso de middleware reflexivo, reflexão comportamental é, em geral, associada a interceptadores e políticas de gerenciamento de recursos.

Estudaremos agora dois casos de sistemas de middleware reflexivo desenvolvidos, respectivamente, na Universidade de Illinois em Urbana-Champaign e na Universidade de Lancaster.

1.6.6 DynamicTAO

Em um grande projeto do grupo de pesquisa em sistemas da Universidade de Illinois em Urbana-Champaign, pesquisadores desenvolveram *2K*, um sistema operacional distribuído baseado em objetos CORBA [25]. O sistema era implementado como middleware em cima de núcleos de sistemas operacionais convencionais como Linux, Solaris e Windows. Além de vários serviços distribuídos que se somavam aos serviços já oferecidos por CORBA, uma das peças fundamentais de *2K* era uma camada de middleware reflexivo denominada dynamicTAO.

DynamicTAO é uma extensão de TAO, um ORB CORBA muito difundido que foi implementado em C++ utilizando uma série de padrões de desenho (*design*) de software [47]. As extensões implementadas em dynamicTAO permitem a reconfiguração em tempo de execução do funcionamento interno do ORB e das aplicações sendo executadas em cima dele. Em dynamicTAO, configuradores de componentes (`ComponentConfigurator`s) são usados para representar as dependências entre os componentes do ORB e entre os componentes do ORB e das aplicações. Um `ComponentConfigurator` é um objeto C++ que armazena essas dependências como listas de referências apontando para outros configuradores de componentes, criando assim um grafo dirigido de dependências entre componentes como mostra a Figura 1.10.

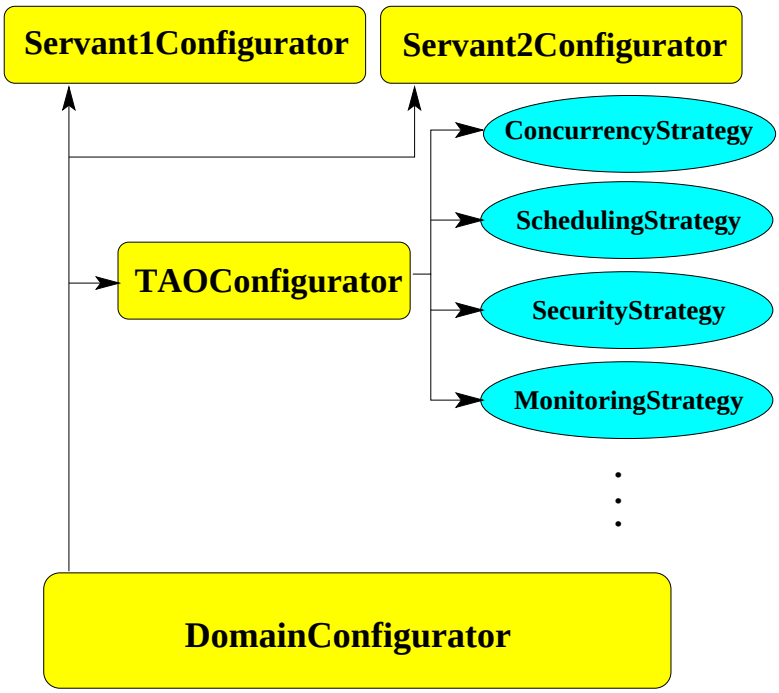

Figura 1.10: Representação explícita da estrutura interna do ORB dynamicTAO

Sempre que uma requisição para substituir um componente C é recebida, o middleware examina as dependências dinâmicas entre C e outros componentes usando o objeto `ComponentConfigurator` associado a C. Programadores podem estender a classe `ComponentConfigurator` usando herança a fim de moldar o configurador para lidar com tipos específicos de componentes ou oferecer alguma funcionalidade específica como, por exemplo, verificar a validade da reconfiguração ou enviar notificações para os componentes

que dependem do componente sendo reconfigurado. Assim, sempre que uma reconfiguração na estrutura do sistema de componentes é realizada, o configurador de componentes tem a oportunidade de entrar em ação e executar o código escrito pelo programador. Desenvolvedores de middleware usam este mecanismo para encapsular, nos configuradores, o código que toma as ações para garantir a consistência da estrutura interna do ORB na presença de reconfigurações dinâmicas. DynamicTAO permite a reconfiguração dinâmica dos componentes do middleware que implementam controle de concorrência, segurança e monitoramento.

DynamicTAO exporta uma meta-interface para carga e descarga de módulos no sistema de tempo de execução e para inspeção e mudanças na configuração do ORB. Esta meta-interface pode ser utilizada por programadores para depuração e testes, por administradores de sistemas para manutenção e configuração ou por outros componentes de software que podem inspecionar e reconfigurar as camadas internas no ORB baseados em informações coletadas de outras fontes (por exemplo, de monitores de utilização de recursos).

Além da meta-interface descrita acima, a fim de permitir a reconfiguração de uma grande coleção de ORBs distribuídas em uma rede como a Internet, dynamicTAO oferece uma meta-interface similar, mas destinada a agentes móveis [27]. Neste caso, administradores de sistemas usam uma interface gráfica para construir agentes móveis de reconfiguração e para injeta-los na rede. Os agentes viajam de um ORB para o outro, inspecionando e reconfigurando cada nó de acordo com as instruções programadas pelo administrador.

A fim de permitir a interposição de código do usuário do middleware no caminho das chamadas de métodos remotos, dynamicTAO utiliza interceptadores desde a sua versão inicial[9]. Desenvolvedores podem carregar dinamicamente e instalar interceptadores tanto do lado do cliente quanto do lado do servidor. Interceptadores podem ser de baixo nível (enxergando os bytes que compõem as mensagens transmitidas pela rede) e de alto nível (enxergando as requisições aos objetos CORBA).

Interceptadores foram utilizados em dynamicTAO para implementar monitoramento de chamadas a objetos distribuídos e para implementar um sofisticado sistema de segurança baseado em capacidades ativas [28]. Interceptadores poderiam também ser utilizados para implementar criptografia, compressão, qualidade de serviço, etc.

No sistema *2K*, onde dynamicTAO se insere, o gerenciamento de recursos fica sob a responsabilidade de um componente que não faz parte do ORB mas que pode ser carregado dinamicamente em seu espaço de endereçamento: o DSRT. O Escalonador Dinâmico de Tempo-Real Flexível (*Dynamic Soft Real-Time Scheduler*, ou DSRT) [36] é executado como um processo no nível do usuário em sistemas operacionais convencionais como Linux, Solaris e Windows. Ele usa a interface de tempo-real de baixo nível oferecida por esses sistemas para oferecer garantias de qualidade de serviço (QoS) para aplicação com requisitos de tempo-real flexível. Ele realiza controle de admissão, negociação de recursos, reserva de recursos e escalonamento de tempo-real.

Em dynamicTAO, reflexão estrutural é implementada através dos configuradores de componentes que auxiliam na reificação da estrutura do sistema. Reflexão comportamental é obtida através dos interceptadores e do uso do DSRT para gerenciamento de recursos. Desta forma, os mecanismos para reificação, inspeção e reconfiguração dos mecanismos internos do ORB que foram adicionados à implementação de TAO fazem de dynamicTAO

[9]Desde 2001, os interceptadores passaram a ser padrão na arquitetura CORBA

um exemplo de *ORB reflexivo*.

1.6.7 Open ORB

O projeto Open ORB [4], desenvolvido na Universidade de Lancaster (Reino Unido), é voltado para o desenho de plataformas de middleware altamente configuráveis e dinamicamente reconfiguráveis. O objetivo é prover um suporte de middleware flexível para aplicações com requisitos dinâmicos, tais como aquelas que envolvem multimídia distribuída e mobilidade.

Na arquitetura de Open ORB, componentes com interfaces bem definidas implementam os vários elements da funcionalidade do middleware. Instâncias personalizadas da plataforma podem então ser configuradas através da combinação dos componentes apropriados, seguindo um modelo de componentes que permite composição hierárquica (isto é, em um componente pode ser construído com base em componentes mais primitivos) e distribuição. É importante ressaltar que, em Open ORB, componentes são entidades presentes tanto durante o desenho da plataforma quanto em tempo de execução. Isto facilita a reconfiguração da plataforma pois permite a identificação, em tempo de execução, das partes da plataforma que necessitam ser alteradas.

Reconfiguração dinâmica é realizada através do uso extensivo de reflexão, com base em uma separação clara entre o nível-base da plataforma e seu meta-nível. Enquanto o *nível-base* consiste de componentes que implementam os serviços usuais de middleware, o meta-nível é composto pelos mecanismos que expõem a implementação da plataforma e permitem a sua inspeção e adaptação dinâmica. A estrutura do meta-nível segue o mesmo modelo de componentes utilizado para implementar o nível-base. Em particular, isto significa que os mecanismos reflexivos podem também ser aplicados para inspecionar e adaptar o próprio meta-nível. Componentes do meta-nível (também chamados de meta-objetos) encapsulam e gerenciam uma auto-representação causalmente conectada da implementação da plataforma. Mais precisamente, cada meta-objeto é encarregado de manter parte da representação reflexiva de um componente específico da plataforma. A completa auto-represetação de um dado componente, por sua vez, é particionada entre um conjunto de meta-objetos, os quais constituem o *meta-espaço* do componente.[10]

O uso do conceito de *meta-espaço* permite lidar com a elevada complexidade que é comum em arquiteturas reflexivas para middleware devido à multitude de aspectos que devem ser considerados. Open ORB permite particionar os diversos aspectos da auto-representação da plataforma entre os meta-objetos distintos que formam o meta-espaço de um componente. Para tanto, uma abordagem de reflexão baseada em múltiplos modelos de meta-espaço [37] é adotada. Segundo esta abordagem, o meta-espaço de um componente é particionado com base em um conjunto padronizado de modelos, cada um oferecendo uma visão diferente da implementação da plataforma e podendo ser reificado independentemente dos demais. A arquitetura de Open ORB define quatro modelos de meta-espaço, que são agrupados de acordo com a distinção entre reflexão estrutural e comportamental, discutida na seção 1.6.5. Estes modelos são ilustrados na Figura 1.11

Os modelos de meta-espaço denominados de *Interfaces* e *Architecture* representam o suporte oferecido para reflexão estrutural. O primeiro é utilizado para a representação

[10]Observe que o mesmo modelo de programação é também aplicado para o desenvolvimento de aplicações, o que significa que os mecanismos reflexivos de Open ORB podem também ser usados para a adaptação de componentes de aplicação.

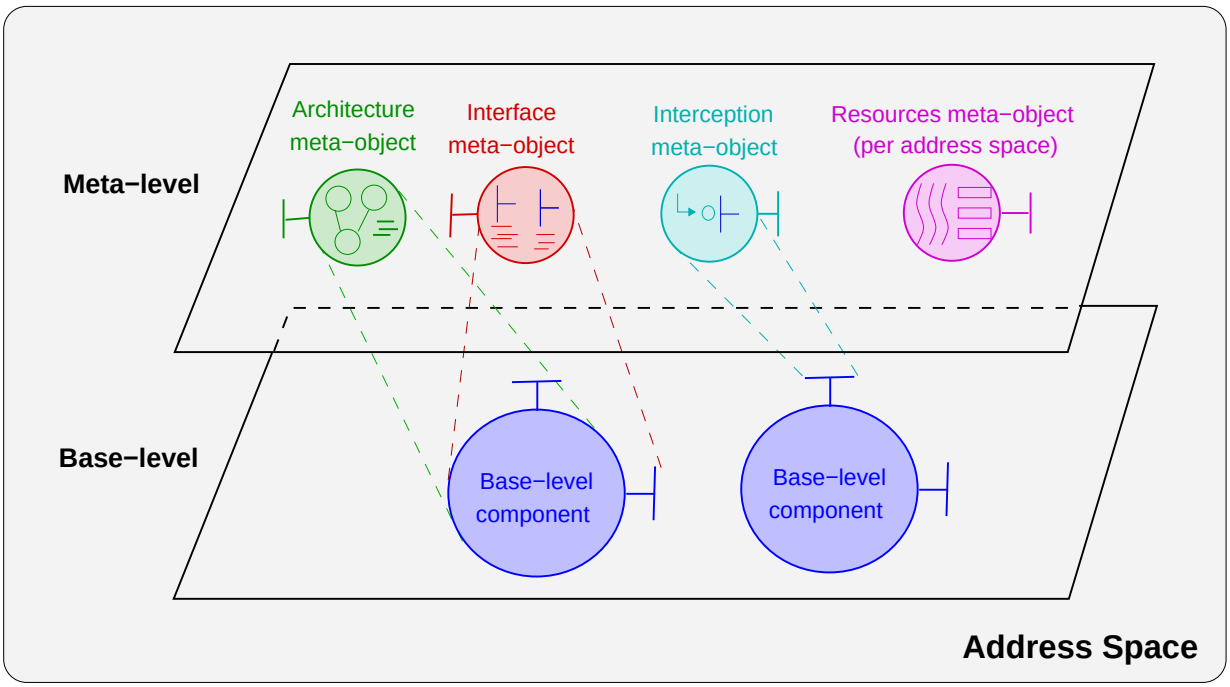

Figura 1.11: Estrutura do meta-espaço em Open ORB

externa de um componente, em termos do conjunto de interfaces que ele provê e requer. O protocolo de meta-objetos (MOP) associado a este modelo oferece serviços para enumerar e fazer a busca dos elementos que definem as interfaces do componente. Isto permite, por exemplo, a descoberta dinâmica dos serviços que um componente provê.

O modelo de arquitetura (*Architecture*), por sua vez, lida com a representação da implementação interna de um componente, em termos da *arquitetura de software* do componente. A auto-representação, neste caso, consiste de duas partes: um *grafo de componentes*, que representa as interconexões entre os componentes mais primitivos que constituem o componente reificado, e um conjunto de *restrições arquiteturais*, que definem as regras para validar configurações de componentes. O MOP associado a este modelo habilita a inspeção e adaptação dinâmica da arquitetura de software do componente reificado. Por exemplo, é possível adicionar, remover ou substituir componentes, bem como inspecionar e mudar as restrições arquiteturais. Desta forma, é possível conduzir mudanças na configuração da plataforma em tempo de execução, caso seja preciso adaptá-la a novas circunstâncias de operação.

Os dois outros modelos de meta-espaço, por outro lado, são dedicados a reflexão comportamental. O modelo de interceptadores *Interception* permite, através de seu MOP, a adaptação das propriedades não-funcionais associadas às interfaces de componentes. Isto é feito sob a forma de interceptadores que introduzem pré- e pós-processamento das interações emitidas e recebidas em uma interface. Desta forma, é possível adicionar (ou remover) tratamentos adicionais das interações, de forma a agregar propriedades como segurança e sincronização de tempo real às interações. Este modelo pode ser comparado ao uso de interceptadores em CORBA (ver seção 1.3), mas se diferencia pela sua flexibilidade, uma vez que o mecanismo de interceptação pode ser aplicado, em tempo de execução, em pontos arbitrários da implementação da plataforma.

Finalmente, o modelo de recursos *Resources* adota uma visão diferente de reflexão comportamental. Seu MOP oferece acesso estruturado aos recursos subjacentes da plataforma (tais como memória e processamento), bem como aos mecanismos para o gerenciamento destes recursos [15]. Em particular, o MOP permite a inspeção e a re-configuração dos recursos alocados a tarefas particulares no sistema (por exemplo, através do aumento ou diminuição dos recursos alocados, ou da mudança nos parâmetros e algoritmos para o seu gerenciamento). Isto facilita, por exemplo, o gerenciamento da qualidade de serviço da plataforma em ambientes cuja disponibilidade de recursos varia ao longo do tempo, tal como em ambientes móveis.

Vários protótipos da arquitetura Open ORB foram implementados, cada um enfocando um particular aspecto do projeto, em particular:

- *OpenCOM* [12], que provê uma implementação eficiente da arquitetura, em termos de um framework de componentes COM;

- *MetaORB* [13], que estende a arquitetura Open ORB com o gerenciamento consistente das estruturas de meta-informação que formam a auto-representação da plataforma, explorando também a relação entre este aspecto e os mecanismos reflexivos [14]; e

- *GOORB (Group Support for Open ORB)* [46], que é uma instância de OpenORB voltada para o desenvolvimento de serviços de grupo flexíveis.

Em cada caso, testes e avaliações foram realizados, os quais demonstram a adequabilidade da arquitetura para o suporte a aplicações de multimídia distribuída. O uso de OpenORB em cenários de aplicação reais, contudo, é um tema para trabalhos em andamento.

1.6.8 Comparação

Open ORB e dynamicTAO foram desenvolvidos independentemente em lados opostos do Atlântico por pesquisadores de diferentes escolas usando tecnologias distintas. Contudo, as suas motivações foram as mesmas e ambos os projetos levaram a soluções semelhantes baseadas em arquiteturas reflexivas.

Estes projetos ilustram duas abordagens opostas para o desenvolvimento de sistemas reflexivos e, mais especificamente, middleware reflexivo. O desenvolvimento de dynamicTAO se iniciou com TAO, um implementação completa de um ORB CORBA que era modular mas estática. Os desenvolvedores de dynamicTAO re-utilizaram dezenas de milhares de linhas de código que já estava em funcionamento e se concentraram em adicionar mecanismos reflexivos para tornar o sistema mais flexível, dinâmico e configurável.

Por outro lado, o desenvolvimento do código de Open ORB começou do zero, sendo que seus projetistas tiveram a oportunidade de planejar sua arquitetura desde o estágio inicial. Portanto, enquanto dynamicTAO se focou na re-utilização de código, Open ORB se focou no projeto de uma arquitetura inovadora para middleware onde todos os seus elementos são consistentes com os princípios da reflexão computacional.

Outros trabalhos importantes na área de middleware reflexivo são OpenCORBA [29], uma implementação de CORBA em NeoClasstalk, uma versão reflexiva da linguagem Smalltalk, e mChaRM (*multi-Channel Reification Model*) [8], uma plataforma que define canais de comunicação reconfiguráveis baseados em Java RMI.

1.6.9 O Futuro do Middleware Reflexivo

Nos últimos dois anos, implementações comerciais de middleware tradicional têm incorporado algumas das contribuições trazidas pela pesquisa em middleware reflexivo. Como vimos na seção 1.3, CORBA possui agora um padrão para interceptadores portáteis. Orbix2000, um ORB comercializado pela empresa Iona (ver www.iona.com), permite a especificação de diferentes políticas para o funcionamento do ORB e oferece suporte para carga dinâmica de novos componentes, chamados de *plug-ins*. Provavelmente, nos próximos anos, continuaremos a ver mais e mais os resultados da pesquisa em middleware reflexivo serem incorporados ao middleware padrão utilizado nos mais diversos tipos de sistemas.

Apesar da grande utilidade destes mecanismos, o grau de flexibilidade obtido por sistemas com arquitetura convencional é limitado. Isto acontece principalmente devido à natureza de caixa preta monolítica destas tecnologias que limita o quanto do sistema pode ser aberto e exposto ao programador. A reflexão computacional, por outro lado, oferece uma solução genérica para este problema através de uma abordagem coerente para o desenho de middleware que leva naturalmente à abertura do sistema. Finalmente, o uso de reflexão permite a manipulação e adaptação de aspectos da plataforma de middleware de formas que não foram nem previstas durante o projeto do sistema.

O que ainda falta para uma maior penetração das tecnologias de middleware reflexivo é um padrão internacional para definir estes sistemas de forma que eles possam inter-operar. Talvez este seja o nosso trabalho para a primeira década do século XXI.

1.7 Conclusão

Este texto ofereceu uma visão evolutiva do desenvolvimento da área de middleware para sistemas distribuídos nos últimos anos. Inicialmente, conduzimos um estudo geral de sistemas de processamento distribuído aberto (ODP), que fornecem a base para plataformas de middleware. Tecnologias hoje consolidadas para middleware orientado a objetos foram então dicutidas e avaliadas à luz do modelo de ODP. Isto levou à identificação das deficiências fundamentais destas tecnologias convencionais com respeito aos requisitos de aplicações emergentes de middleware. O texto então abordou recentes extensões introduzidas nestas tecnologias, as quais fornecem uma resposta, ainda que parcial, a estes novos requisitos.

Em uma avaliação do estado atual da área, podemos, com certeza, concluir que os recentes desenvolvimentos facilitaram em muito a vida do programador de aplicações distribuídas. De uma maneira geral, as plataformas de middleware hoje disponíveis oferecem serviços de alto nível, que permitem que o programador desenvolva aplicações sem precisar se preocupar com a maioria dos problemas gerados por ambientes distribuídos ou mesmo com os aspectos não-funcionais das aplicações. Isto tornou muito mais fácil desenvolver aplicações distribuídas, bem como portá-las de uma plataforma para outra e permitir a sua interoperabilidade em ambientes heterogêneos.

Entretanto, a introdução de novas categorias de aplicações distribuídas trouxe novos requisitos com relação ao suporte oferecido por plataformas middleware. Infelizmente, as tecnologias mais recentes de middleware não contemplam tais requisitos adequadamente, o que se deve principalmente à falta de flexibilidade de suas respectivas arquiteturas e implementações. Contudo, deve-se também observar que a área de middleware para

sistemas distribuídos ainda está em fase de amadurecimento, sendo que uma evolução significativa pode ser esperada para os próximos anos.

Neste texto (Seção 1.6), foram apresentados importantes princípios e paradigmas para o design e implementação de middleware, os quais consideramos altamente promissores para resolver as deficiências das tecnologias atuais. Em particular, as abordagens estudadas têm como foco a provisão de flexibilidade, de forma a permitir que uma plataforma de middleware possa ser adaptada, (ao invés de substituída) à medida em que novos requisitos surgirem. Desta forma, torna-se muito menos traumático o processo de evolução de uma tecnologia. Acreditamos que estas novas abordagens representam, senão uma solução completa, pelo menos uma contribuíção decisiva para a definição de padrões para a próxima geração de plataformas de middleware.

Referências

[1] Palmer W. Agnew and Anne S. Kellerman. *Distributed Multimedia: Technologies, Applications, and Opportunities in the Digital Information Industry: A Guide for Users and Providers*. ACM Press and Addison-Wesley, New York, NY USA, 1996.

[2] S. Allamaraju, C. Buest, J. Davies, T. Jewell, R. Johnson, A. Longshaw, R. Nagappan, P.G. Sarang, A. Toussaint, S. Tyagi, G. Watson, M. Wilcox, A. Williamson, and D. O'Connor. *Professional Java Server Programming: J2EE 1.3 Edition*. Programmer to Programmer. Wrox Press Ltd., 2001.

[3] Mark Astley, Daniel C. Sturman, and Gul Agha. Customizable Middleware for Modular Distributed Software. *Communications of the ACM*, 44(5):99–107, May 2001.

[4] Gordon S. Blair, Geoff Coulson, Anders Andersen, Michael Clarke, Fabio M. Costa, Hector A. Duran, Rui Moreira, Nikos Parlavantzas, and Katia B. Saikoski. The design and implementation of Open ORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.

[5] Gordon S. Blair and J-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.

[6] Bluetooth. The Official Bluetooth Website, 2001. http://www.bluetooth.com.

[7] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Middleware for mobile computing (a survey). Research Note UCL RN/30/01, University College of London, London, UK, July 2001.

[8] Walter Cazzola and Massimo Ancona. mChaRM: a reflective middleware for communication-based reflection. Technical Report DISI-TR-00-09, DISI, Universita degli Studi di Milano, May 2000.

[9] M. C. Chan and A. A. Lazar. Designing a CORBA-based High Performance Open Programmable Signalling System for ATM Switching Platforms. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[10] Zhigang Chen, See-Mong Tan, Roy H. Campbell, and Yongcheng Li. Real Time Video and Audio in the World Wide Web. In *Fourth International World Wide Web Conference*, Boston, December 1995. Also published in World Wide Web Journal, Volume 1 No 1, January 1996.

[11] Shigeru Chiba. A metaobject protocol for C++. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'95)*, pages 285–299, Austin, TX USA, October 1995.

[12] Mike Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proc. IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, Heidelberg, Germany, November 2001. Available at: http://www.lancs.ac.uk/postgrad/parlavan/publications/Middleware2001.pdf.

[13] Fabio M. Costa. *Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware*. Ph.d., University of Lancaster, 2001.

[14] Fabio M. Costa and Gordon S. Blair. Integrating reflection and meta-information management in middleware. In Pamela Drew and Robert Meersman, editors, *Proc. International Symposium on Distributed Objects and Applications (DOA'00)*, pages 133–143, Antwerp, Belgium, 2000. IEEE Computer Society Press.

[15] Hector A. Duran-Limon and Gordon S. Blair. The importance of resource management in engineering distributed objects. In W. Emmerich and S. Tai, editors, *Proc. 2nd International Workshop on Engineering Distributed Objects (EDO'2000)*, volume 1999 of *LNCS*, pages 44–60, Davis, CA USA, 2000. Springer.

[16] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA USA, 1999.

[17] ITU-T/ISO. *ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1: Overview*. ISO/ITU-T, 1995.

[18] ITU-T/ISO. *ITU-T X.903 | ISO/IEC 10746-3 Open Distributed Processing Reference Model Part 3: Architecture*. ISO/ITU-T, 1995.

[19] ITU-T/ISO. *ITU-T X.902 | ISO/IEC 10746-2 Open Distributed Processing - Reference Model Part 2: Foundations*. ISO/ITU-T, 1996.

[20] ITU-T/ISO. *ITU-T X.902 | ISO/IEC 10746-2 Open Distributed Processing - Reference Model Part 4: Architectural Semantics*. ISO/ITU-T, 1997.

[21] ITU-T/ISO. *ITU/T Draft Rec X950 - 1 | ISO/IEC IS 13235-1, ODP Trading Function Specification*. ISO/ITU-T, 1997.

[22] ITU-T/ISO. *ITU-T X.930 | ISO/IEC 14753 Open Distributed Processing - Interface References and Binding.* ISO/ITU-T, 1998.

[23] ITU-T/ISO. *ITU-T Draft Rec. X.960 | ISO/IEC FDIS 14769 - Information Technology - Open Distributed Processing - Type Repository Function.* ISO/ITU-T, 2000.

[24] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Presss, 1991.

[25] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[26] Fabio Kon, Fábio Costa, Roy Campbell, and Gordon Blair. The Case for Reflective Middleware. *Communications of the ACM*, 45(6), June 2002.

[27] Fabio Kon, Binny Gill, Manish Anand, Roy H. Campbell, and M. Dennis Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, pages 86–98, Zurich, September 2000.

[28] Fabio Kon, Manuel Román, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective orb. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, USA, April 2000. Springer-Verlag.

[29] Thomas Ledoux. OpenCORBA: a Reflective Open Broker. In *Proc. 2nd International Conference on Reflection and Meta-level Architectures (Reflection'99)*, volume 1616 of *LNCS*, pages 197–214, St. Malo, France, 1999. Springer.

[30] Paty Maes. Concepts and experiments in computational reflection. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL USA, 1987. ACM.

[31] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[32] Raphael Marvie and Philippe Merle. CORBA component model: Discussion and use with OpenCCM. *Informatica - An International Journal of Computing and Informatics, Special Issue on Component Based Software Development*, June 2001. Submitted for publication. Available at: http://corbaweb.lifl.fr/OpenCCM/docs/2001_06_Informatica.ps.

[33] Philippe Merle, Raphael Marvie, and Mathieu Vadet. The OpenCCM Platform. Technology website, 2001. http://corbaweb.lifl.fr/OpenCCM/.

[34] Microsoft. Microsoft COM Technologies - DCOM, 30/03/1998 2000.

[35] Microsoft. .NET Development, 2000.

[36] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networking, Special Issue on Multimedia Networking*, 7:227–255, 1998.

[37] Hidehaki Okamura, Y. Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proc. International Workshop on New Models for Software Architecture (IMSA'92)*, pages 36–47, Tokyo, Japan, 1992.

[38] Alexandre Oliva, Islene C. Garcia, and Luiz Eduardo Buzato. The reflective architecture of Guarana. Technical report, Institute for Computer Science, UNICAMP, September 1998.

[39] OMG. *Components FTF Edited Drafts of CORBA Core Chapters (CORBA 3.0).* Object Management Group, Needham, MA USA, document number ptc/99-10-03 edition, 1999.

[40] OMG. *Audio/Video Stream Specification, V1.0.* Object Management Group, Needham, MA USA, omg document formal/00-01-03 edition, 2000.

[41] OMG. *Discussion of the Object Management Architecture Guide.* Object Management Group, Needham, MA USA, 2000.

[42] OMG. *The Common Object Request Broker: Architecture and Specification - Revision 2.6 (CORBA v2.4.2).* Object Management Group, Needham, MA USA, December 2001.

[43] OMG. *CORBA 3.0 New Components Chapters (Components December 2000 FTF).* Object Management Group, Needham, MA USA, omg tc document ptc/2001-11-03 edition, November 2001.

[44] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal*, 2(5), July 2001. Avaliable at http://dsonline.computer.org/0105/features/rom0105_print.htm.

[45] Manuel Román, M. Mickunas, Fabio Kon, and Roy H. Campbell. LegORB and ubiquitous corba. In *Proc. IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM2000)*, pages 1–2, Palisades, NY USA, April 2000.

[46] K.B. Saikoski and G. Coulson. Experiences with OpenORB's compositional metamodel and groups of components. In *Proc. Workshop on Experience with Reflective Systems (WERS'01) (Held in conjunction with Reflection 2001)*, Kioto, Japan, September 2001. Available at: http://www.openjit.org/reflection2001/workshop-papers/saikoski.pdf.

[47] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 37(4):54–63, May 1999.

[48] Jon Siegel. *CORBA 3: Fundamentals and Programming.* John Wiley & Sons, Inc., 2nd edition, 2000.

[49] Jon Siegel. What's comming in CORBA 3. Release Information, July 2001. Available at: http://www.omg.org/technology/corba/corba3releaseinfo.htm.

[50] Ashish Singhai, Ahmond Sane, and Roy H. Campbell. Quarterware for middleware. In *Proc. 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 192–201, Amsterdam, The Netherlands, May 1998. IEEE.

[51] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A Reliable Transport Protocol for Wireless Wide-Area Networks. In *Proceedings of ACM Mobicom*, Seattle, WA, August 1999.

[52] Brian C. Smith. Reflection and semantics in Lisp. In *ACM Conference on Principles of Programming Languages (POPL)*, pages 23–35, Salt Lake City, USA, January 1984.

[53] Sun. Java Remote Method Invocation (RMI), 2000. http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.

[54] Sun. Java 2 Platform Enterprise Edition. Technology website, 2002. http://java.sun.com/j2ee/.

[55] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Harlow, England, 1998.

[56] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[57] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84, July 1993.

[58] Yasuhiro Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'92)*, pages 414–434, Vancouver, Canada, 1992.