2
Trabalhos Relacionados

Como mencionamos anteriormente, a área de desenvolvimento de *softwares* baseados em componentes distribuídos está servida por diversos modelos e *middlewares*, sendo alguns destes bastante conhecidos e utilizados. Selecionamos para esta seção alguns dos que consideramos mais relevantes para o foco deste trabalho, por darem suporte à implantação e execução de componentes, e ao gerenciamento dinâmico de suas dependências de *software*.

Em sistemas não-componentizados sem capacidade de configuração dinâmica, os processos geralmente precisam ser interrompidos e reiniciados para que mudanças em suas dependências tenham efeito. Geralmente precisam ainda ser novamente configurados em tempo de execução, levando a problemas de manutenção, perda de tempo e disponibilidade reduzida.

Para prover a capacidade de configuração dinâmica, uma técnica comum é especificar interfaces providas e requeridas, que possam ser descobertas e conectadas em tempo de execução. Essas interfaces são contratos acertados em tempo de desenvolvimento. Mantendo esses contratos, componentes podem ser desconectados e substituídos por outros componentes que implementem as mesmas interfaces, sem encerrar a execução da aplicação.

Além disso, um *middleware* deve fornecer os serviços de instalação e execução, entre outros. Uma técnica comum é utilizar um serviço específico para representar um nó real da rede, onde este serviço pode exercer o trabalho de executor de outros serviços ou processos genéricos. Para aplicações baseadas em componentes geralmente utiliza-se também uma entidade "contêiner", que provê um ambiente protegido para a execução de componentes, oferece otimizações para a comunicação entre componentes internos e fornece serviços para os componentes nela hospedados. Existem outras necessidades, como formas de instalação e empacotamento de componentes, que também podem ser resolvidas em tais *middlewares*. Estas necessidades podem ser tratadas em

PUC-Rio - Certificação Digital Nº 0611895/CA

serviços que atuem como repositórios de componentes, por exemplo.

Nas próximas seções, apresentaremos alguns sistemas que consideramos representativos do estado da arte das tecnologias de *middleware*. Para evidenciar as contribuições mais marcantes de cada sistema, serão avaliados os seguintes critérios:

- *Modelo de Componentes*: O modelo de componentes é a forma contratual escolhida pelo *middleware* para especificar componentes de software. Serão abordadas também questões como independência de linguagem e quaisquer outras restrições ou benefícios derivados de tais modelos.
- *Serviços Básicos*: Por serviços básicos nos referimos ao conjunto de funcionalidades que permitam a implantação e a execução de aplicações baseadas em componentes.
- *Serviços Adicionais*: Cada *middleware* apresenta alguns serviços não-críticos mas que agregam valor à solução como um todo, para o conjunto de problemas que desejam resolver. Estes serviços geralmente são de mais alto nível, como sessão, segurança, transações, persistência, entre outros.

PUC-Rio - Certificação Digital Nº 0611895/CA

2.1 CCM

A OMG [30] adotou o Modelo de Componentes CORBA (CCM) [7, 31] para estender o Modelo de Objetos CORBA [32]. O CCM define características e serviços que possibilitam a implementação, gerência, configuração e implantação de componentes que integrem serviços CORBA comumente utilizados, como transação, segurança, persistência de estado e notificação de eventos.

Modelo de Componentes

O modelo é bastante parecido com as especificações *Enterprise Java Beans* [33], mas sem a limitação de linguagem por utilizar CORBA como base. São definidos alguns tipos básicos de mecanismos de comunicação chamados portas, que são:

- Facetas: Representam interfaces providas pelos componentes.
- Receptáculos: Representam interfaces requeridas pelos componentes.

- Fontes de Eventos: Emitem eventos de um tipo especificado para canais ou consumidores interessados.
- Destinos de Eventos: Recebem eventos de um tipo especificado.
- Atributos: Permitem a configuração de componentes de forma similar aos atributos CORBA originais, mas provendo maior flexibilidade, como por exemplo a possibilidade de lançamento de exceções.

O modelo utiliza "*homes*", uma palavra-chave da linguagem de definição de interfaces (mais especificamente, IDL3), para explicitar a estratégia de gerenciamento de ciclo de vida de cada componente. Esta interface pode ser utilizada por um cliente para criar e remover instâncias de um dado componente. Objetos *home* são publicados em "*HomeFinders*", similares ao Serviço de Nomes CORBA, para atuarem como pontos de entrada para se encontrar instâncias do componente especificado. *Homes* utilizam configuradores, chamados "*Standard Configurators*", para realizar configurações iniciais durante a instanciação, como quais componentes devem ser conectados e os relacionamentos entre eventos publicados e recebidos. Uma interface chamada "*Navigation*"provê facilidades de inspeção básicas aos componentes, fornecendo informações sobre todas as suas interfaces.

Além da linguagem de definição de interfaces, o CCM define também uma linguagem para a definição de componentes, chamada CIDL. Interfaces de componentes são compostas tanto por descrições básicas do componente, incluindo os tipos de serviços CORBA requeridos, como outras mais elaboradas como o estado de persistência dos componentes e de seus *homes*. Para efetuar a interpretação das CIDLs, é necessário um outro compilador.

Serviços Básicos

De forma simplificada, o processo de implantação e execução de componentes CCM é dado pelo empacotamento das implementações de componentes em um arquivo, normalmente uma DLL ou um JAR, e pelo uso de mecanismos de implantação providos por uma implementação CCM como [34, 35] para carregá-lo em um servidor de componentes. Estes mecanismos de implantação baseiam-se em descrições, chamadas de planos, para a implantação e execução de aplicações. Planos incluem muitos detalhes, sendo que os principais são informações sobre os nós que devem ser utilizados para a instanciação de cada componente, as conexões que devem ser feitas e repositórios de componentes a serem utilizados.

PUC-Rio - Certificação Digital Nº 0611895/CA

Servidores de componentes são específicos para uma dada plataforma e linguagem de programação, e carregam tanto o *home* de um componente como contêineres. Contêineres encapsulam implementações de componentes, fornecendo um ambiente de execução que permite a ativação e desativação destes componentes, atuam como *proxies* para os serviços básicos CORBA e fornecem uma camada de adaptação para o recebimento de eventos de interesse. Cada servidor de componentes tem seu próprio processo e gerencia apenas um tipo de implementação de componente, além de fornecer políticas de ciclo de vida para os *servants* através de *ServantLocators*. As políticas de ciclo de vida tratam da ativação e desativação de componentes, seja ativando-os a cada invocação de método e depois tornando-os passivos, ou delegando o tratamento aos contêineres ou aos próprios componentes.

Serviços Adicionais

O CIF, *Component Implementation Framework* ou Arcabouço de Implementações de Componentes, poupa os desenvolvedores de componentes de tarefas que podem ser generalizadas como implementações referentes a gerência de ciclo de vida e manutenção de estado dos componentes, provendo um conjunto de APIs com funcionalidades como a reconstituição do estado de um componente através do uso de um banco de dados.

2.2 Ice

Os desenvolvedores do *Internet Communications Engine* [2] tinham como objetivo inicial criar uma arquitetura similar a CORBA, mas que não partilhasse de certas limitações, nem sofresse os problemas derivados de uma gestão organizada por um comitê. Por isso, seu escopo é grande e complexo como o do CCM, pois propõe todo um novo modelo de comunicação distribuída.

A infra-estrutura de execução do Ice se baseia primariamente no IceGrid, que é um executor de serviços em processos próprios. No entanto, o Ice também é composto por alguns outros módulos, e dentre estes citamos como especial o IceBox, que atua como um contêiner para componentes de software e se assemelha mais ao nosso trabalho. Deste ponto em diante nos referiremos primariamente ao módulo IceBox.

PUC-Rio - Certificação Digital Nº 0611895/CA

Modelo de Componentes

Componentes são chamados de serviços IceBox. Estes devem implementar uma interface chamada *Service* que contém métodos para a inicialização e finalização de um componente / serviço. Esta interface é local, e só pode ser acessada pelo servidor IceBox. Este, por sua vez, tem uma interface que permite inicializar, parar ou desligar serviços. Interfaces devem ser definidas em uma linguagem de descrições semelhante à IDL CORBA, chamada SLICE. Existem mapeamentos SLICE para as linguagens de desenvolvimento C++, Java, C#, Visual Basic, Python, Ruby e PHP, embora nem todas tenham o mesmo grau de completude.

Um detalhe crucial é o fato de que não há apoio da infra-estrutura para a modificação das dependências externas dos serviços. Ao invés disso, tanto serviços IceBox quanto servidores IceGrid devem ou ter todas as informações necessárias para formar uma referência remota, ou conhecer o nome identificador que deve ser utilizado em conjunto a um serviço de registro que fornecerá a referência. Estes dados devem ser fornecidos na configuração do serviço ou servidor.

Serviços Básicos

Um servidor IceBox é composto por alguns sub-componentes, incluindo o chamado *ServiceManager* que atua como um componente administrador. Este é responsável por carregar e iniciar os serviços, que são carregados apenas na inicialização de um IceBox, não sendo possível haver carga posterior. Pode-se especificar uma ordem de carga, que será utilizada de forma inversa em sua finalização.

No Ice, o processo de implantação é visto como a publicação de um servidor IceGrid (que pode ser um IceBox) em um serviço de registro, através do qual se tornará disponível para outros servidores. Este serviço de registro é capaz de executar servidores, caso não exista nenhum registrado quando uma busca for feita.

PUC-Rio - Certificação Digital Nº 0611895/CA

Serviços Adicionais

A arquitetura Ice fornece também outros módulos, como o Freeze (persistência), IceSSL (segurança), Glacier2 (*firewall*), IceStorm (*publish/subscribe*) e IcePatch2 (replicação de árvore de diretórios).

2.3 JBoss

O projeto JBoss [14] surgiu como um servidor de aplicações Java estensível, reflexivo e dinamicamente reconfigurável. Seu objetivo era prover um conjunto de componentes que não só implementasse a especificação J2EE, mas também a estendesse. Ao fornecer tais funcionalidades, praticamente sem concorrência ao mesmo nível no mundo Java em seu lançamento, tornou-se um *middleware* bastante utilizado. Isto ocorreu especialmente fora do mundo acadêmico, principalmente devido às implementações J2EE.

Modelo de Componentes

O modelo de componentes utilizado no JBoss é baseado no modelo JMX, *Java Management eXtensions* ou Extensões de Gerenciamento para Java [20]. A implementação JBoss fornece um ambiente para a carga e atualização de componentes, suporta introspecção e também adaptação dinâmica, entre outras propriedades. No entanto, é limitado à linguagem Java.

O modelo JBoss estende o JMX, tratando de problemas não previstos como ciclo de vida, dependências, implantação, (re)configuração dinâmica e empacotamento.

Serviços Básicos

Como dito no item anterior, o JBoss trata dos problemas de implantação no próprio modelo de componentes, e não no *middleware*. São providos componentes que implementam todas as principais especificações J2EE, incluindo o suporte ao modelo EJB. Este modelo é constituído por quatro tipos de elementos: invocadores, contêineres, *proxies* dinâmicos e interceptadores.

PUC-Rio - Certificação Digital Nº 0611895/CA

Invocadores são componentes que provêem serviços de invocação de métodos remota através de diferentes protocolos. Contêineres JBoss são componentes que agregam conjuntos pré-definidos de aspectos [11, 12, 36] requeridos pelas classes de componentes da aplicação, como o uso de transações e segurança. *Proxies* dinâmicos fornecem funcionalidade similar para o lado cliente, e interceptadores implementam aspectos que devam ser aplicados de ambos os lados.

O gerenciamento de dependências é realizado por um controlador chamado *ServiceController*, que atua como repositório de componentes e mantém um registro das dependências dos componentes. No entanto, não há uma forma de se reconfigurar as dependências em tempo de execução. Caso as dependências de um componente não sejam mais válidas, uma interface de ciclo de vida é utilizada para desativar o componente, possivelmente levando à desativação de outros componentes em cascata.

PUC-Rio - Certificação Digital Nº 0611895/CA

Serviços Adicionais

Dentre as implementações das especificações J2EE, são mencionados suporte a serviço de nomes, gerenciamento de transações, serviço de segurança, suporte a *servlets JSP*, suporte a EJB, envio de mensagens assíncronas, *pooling* de conexões a bancos de dados, suporte ao protocolo IIOP, *clustering* e *fail-over*.

2.4 GridKit

O projeto GridKit [3, 37, 38] atua como uma evolução natural dos trabalhos anteriores criados por seu grupo, desde o modelo de componentes OpenCOM [39]. Com o amadurecimento deste modelo, diversos serviços e componentes de mais alto nível para o suporte à computação em *grids* foram sendo desenvolvidos, com o conjunto eventualmente se transformando no GridKit.

Modelo de Componentes

O modelo de componentes do GridKit é um trabalho bastante conhecido, chamado OpenCOM v2 [1, 40]. O modelo OpenCOM foi baseado no modelo COM [15], da Microsoft. Seus componentes são baseados em "interfaces"definidas através de uma versão estendida da IDL da OMG e receptáculos, que representam interfaces providas e requeridas, respectivamente. Interfaces podem ser conectadas (o termo original em inglês utilizado neste modelo é *binding*) a receptáculos, para resolver dependências. Além disso, componentes podem ser criados a partir de outros componentes em qualquer quantidade, atuando como envólucros.

Dentre outras funcionalidades, destacamos o suporte à reflexão computacional, que se traduz em ferramentas tanto de introspecção como de adaptação dinâmica. Destacamos também a noção de *Component Frameworks*, ou CFs, que são composições de componentes que aceitam componentes *plug-in* que modificam ou adicionam algo ao comportamento da composição.

Serviços Básicos

A implantação no GridKit também é derivada do modelo OpenCOM v2, que não se limita ao modelo de programação. São definidos três tipos de CFs básicos com *APIs* específicas, a serem desenvolvidos por programadores e aqui descritos de forma muito simplificada. O CF *caplet* serve para agregar componentes encapsuladores de particularidades do ambiente, como sistemas operacionais, máquinas virtuais, nós de rede, linguagens, etc. O CF *loader* agrega componentes que forneçam formas de se carregar outros componentes em diferentes *caplets*. E o CF *binder* agrega componentes que permitam a conexão entre interfaces e receptáculos, tanto entre componentes dentro de um mesmo *caplet*, como entre componentes de *caplets* diferentes.

Nesta arquitetura, assim como em muitas outras, desenvolvedores são separados em três tipos. Os chamados *deployment programmers*, ou programadores de implantação, criam *plugins* dos tipos *caplet*, *loader* e *binder*, que formam um conjunto de abstrações sobre os serviços-chave oferecidos por diferentes ambientes de implantação. *Systems programmers*, ou programadores de sistemas, utilizam o modelo OpenCOM básico para criar aplicações, isolados de particularidades dos diferentes ambientes de implementação. Por fim, *meta-systems programmers*, ou programadores de meta-sistemas, utilizam a

PUC-Rio - Certificação Digital Nº 0611895/CA

reflexão computacional para estruturar a reconfiguração do sistema em tempo de execução.

Serviços Adicionais

São fornecidos diversos serviços de alto nível, sendo que os mais importantes são:

- *Open Overlays*: Provê componentes que permitem o uso de diferentes tipos de redes de *overlay* [41]. Os outros serviços a seguir são construídos em cima deste.
- *Interaction Services*: Provê componentes que permitem o uso de diferentes tipos de interação, como *publish-subscribe*, espaço de tuplas, *peer-to-peer*, streaming, etc.
- *Resource Discovery*: Permite a descoberta de serviços e recursos genéricos, no domínio distribuído.
- *Resource Management*: Permite o gerenciamento tanto de recursos distribuídos de alta granularidade, como de baixa granularidade para uso em *QoS* (Qualidade de Serviço).
- *Grid Security*: Serviços que fornecem suporte à comunicação segura.

Serviços desenvolvidos por usuários GridKit geralmente são criados com base nestes serviços adicionais fornecidos.

2.5 Considerações Finais

O modelo CCM é um dos que mais nos influenciou ao definir tanto nosso modelo de componentes, como os serviços que em conjunto formam a infra-estrutura de execução. De fato, propiciou muitos exemplos positivos, mas também alguns negativos. Os positivos tornam-se óbvios ao avaliar nosso modelo de componentes, pois é fortemente baseado nos pontos fortes do CCM, como a utilização das portas facetas e receptáculos, facilidades de inspeção, definições de componentes, entre outros. Um detalhe importante de sua abordagem é o fato do conceito de contêineres ser fortemente associado ao conceito de componentes, pois não são capazes de carregar mais de um

PUC-Rio - Certificação Digital Nº 0611895/CA

componente. Isto o difere de outros trabalhos como o Ice, onde um contêiner IceBox é capaz de carregar diferentes componentes.

Dentre os aspectos negativos do CCM, ao estudar o modelo percebe-se imediatamente o quão complexo é. Portas específicas como as de eventos e atributos são exemplos disso, pois aumentam a complexidade do modelo e podem ser modeladas como facetas. Existem muitos elementos diferentes a serem estudados e entendidos - *homes*, contêineres, configuradores, políticas de *ServantLocators* - mesmo que se deseje apenas criar um simples componente "*HelloWorld*", quando algumas destas tarefas poderiam não ser obrigatórias ao implementador de componentes, pois muitas vezes não serão exploradas. O processo de implantação é ainda mais complexo, mas pode ser facilitado ao custo do desenvolvimento de ferramentas adicionais pelos provedores de implementações CCM. Pode ser fornecida, por exemplo, uma ferramenta gráfica que permita a criação e execução de planos de implementação, sem a necessidade de um conhecimento profundo das especificações de implantação por parte do desenvolvedor da aplicação. No entanto, esta alta complexidade apenas contribui para que tais ferramentas demorem a ser desenvolvidas, permitindo que outras tecnologias se estabeleçam mais rapidamente.

O projeto Ice também é bastante complexo, pois conta com uma arquitetura quase tão abrangente quanto a de CORBA. Algumas das maiores diferenças são o fato de contar com implementações de apenas um grupo, o que diminui as incompatibilidades e cria menos confusões, e o fato de que esta implementação teoricamente sempre se manterá completa, pois seus idealizadores primam por mantê-la enxuta e coerente.

Com relação ao módulo IceBox, este claramente não é o foco principal da arquitetura, parecendo bastante limitado. A impossibilidade de se instanciar componentes a qualquer momento em um mesmo IceBox, somada à limitação inerente do Ice em facilitar a descrição e descoberta de interfaces providas e dependências, o torna inferior nesse aspecto aos outros trabalhos listados aqui. Alguns outros pontos poderiam ser melhorados, como a inexistência de suporte a versionamento para serviços IceBox, ou a falta de tratamento de colisões de nome para *endpoints*, fazendo com que o programador deva se preocupar em adicionar números diferentes após os nomes. Notamos que isto não se aplica à arquitetura Ice de modo geral, que é muito bem especificada e conta com implementações robustas.

O *middleware* JBoss é um dos mais utilizados atualmente, devido à ascensão do uso da linguagem Java na Internet e em *intranets*. No entanto, sofre

PUC-Rio - Certificação Digital Nº 0611895/CA

justamente com o suporte único à esta linguagem para o desenvolvimento de seus componentes. É possível a comunicação com outras linguagens através de protocolos como o IIOP, mas isto não é encorajado nem muito utilizado, e toda a arquitetura é baseada em componentes na forma de objetos Java. Este fato leva também a perdas em desempenho, e a um aumento considerável na utilização de memória. Servidores JBoss geralmente são implantados em máquinas potentes, bem-equipadas. É também impossível reconfigurar as dependências externas de componentes em tempo de execução, sendo necessário desativá-los. Outros detalhes menores poderiam ser melhorados, como o fato dos interceptadores serem executados de forma encadeada, com cada interceptador sendo responsável por chamar o próximo. Isto aumenta a fragilidade do sistema, pois não só é necessário confiar no código de todos os interceptadores como um erro em algum destes pode levar a resultados inesperados.

O modelo OpenCOM v2, utilizado no GridKit, novamente cai no problema da alta complexidade. Apesar de fornecer facilidades para se abstrair propriedades do ambiente, como os CFs *caplet*, *loader* e *binder*, para componentes de baixo nível o sistema é bastante complexo, com certas partes de difícil entendimento e uso. Por outro lado, já existem serviços de alto nível de grande utilidade desenvolvidos, o que justificaria o uso para aplicações com requisitos de ambiente simples e que se baseiem fortemente nestes.

PUC-Rio - Certificação Digital Nº 0611895/CA