**Capítulo 2**

**Conceitos Básicos**

Com o objetivo de firmar o embasamento teórico do projeto e do aprimoramento do InteropFrame, será elucidado neste capítulo os conceitos que foram utilizados no decorrer da pesquisa. Alguns deles foram utilizados no aprimoramento do *framework*, como também na sua construção. Conceitos como Modularização, Invocação Remota de Métodos, Geração Automática de Código, Componentes Distribuídos, entre outros. Por fim, um breve relacionamento entre esses conceitos e o aprimoramento do *framework* proposto nesse trabalho.

2.1 Modularização

Modularização significa a concepção de um sistema completo formado por módulos logicamente independentes {Hall2011}. Ela é capaz de reduzir a complexidade do problema, dividindo-o em subproblemas mais simples, ou seja, dividindo-o em módulos {Knuth1996}. Um módulo define um limite lógico executável. Dessa forma os detalhes internos dele não são visíveis a outros módulos ou sistemas. Os únicos detalhes visíveis são aqueles que ele expõe explicitamente, ou seja, a API pública {Hall2011}. Com isso é mais fácil de detectar problemas e resolvê-los, pois os módulos são, em princípio, independentes. Em sua característica um módulo é implementável, gerenciável, reutilizável, combinável e é uma unidade independente de software que provê interfaces a outros módulos ou sistemas {Kirk2012}.

É confuso o conceito de Modularização e Orientação a Objeto. As duas suportam a especialização, ou seja, elas quebram o sistema em partes pequenas dando a cada uma delas a sua devida responsabilidade. Entretanto elas atuam de formas diferentes. Com a Orientação a Objeto é possível modularizar de forma lógica, ou seja, essa modularização se referencia a visibilidade do código. Dessa forma a Orientação a Objeto utiliza parte da Modularização em seu contexto. Entretanto a Modularização abrange mais que isso. Ela pode ser utilizada tanto da forma física, onde é possível subdividir o código em vários arquivos, entretanto mantendo as mesmas dependências e comunicação entre eles, quanto da forma lógica, como na Orientação a objeto {Hall2011}.

A modularização ganhou a popularidade no início da década de 70. Entretanto é algo que ainda hoje não está tão presente nos requisitos não funcionais do desenvolvimento de software. Apesar disso é algo tão importante que trás grandes benefícios para a aplicação. Para isso é preciso aplicar os princípios de modularização para obter bons resultados. Princípios esses como alta coesão e baixo acoplamento. Com eles, uma das grandes vantagens que a modularização oferece é a reutilização. Isso se torna fácil quando um módulo é responsável por aquilo que realmente deve ser e não tem um grande número de dependências. A depender do *framework* utilizado, para a aplicar a modularização, a declaração das dependências pode ser feita de forma explícita como no OSGI. O que trás um ganho para a manutenção no código e para o melhor entendimento do mesmo {Hall2011}.

2.1.1 Componentes e Modelos de Componentes

A engenharia de software baseada em componentes surgiu como uma abordagem para softwares de desenvolvimento de sistemas com base no reúso de componentes de softwares {sommerville2011}. Não existe consenso sobre um componente ser uma unidade independente de software que pode ser composta com outros componentes {sommerville2011}. Segundo {councill2001 apud sommerville2011}, componente é um elemento de software que está de acordo com um modelo de componente padrão e pode ser independentemente implantado e composto de acordo com um padrão de composição. Entretanto, {szyperski2002 apud sommerville2011} menciona que um componente de software é uma unidade de composição de interfaces contratualmente especificadas e pode ser implantado de forma independente, além de estar sujeito a ser composto por parte de terceiros.

Um componente funciona como um provedor de um ou mais serviços. Dessa forma, quando um sistema precisa de um serviço, ele chama um componente para fornecer esse serviço sem se preocupar sobre onde esse componente está sendo executado, nem mesmo de características como linguagem de programação que o componente foi desenvolvido {sommerville2011}. Para isso, os componentes possuem duas interfaces relacionadas que refletem os serviços que o componente fornece (interface *provides*) e os serviços de que o componente necessita (interface *requires*), como mostra a Figura {figuras/001 – Interfaces de componentes.png} {sommerville2011}.

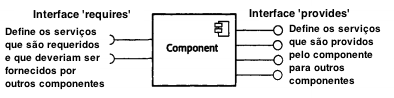


Figura {figuras/001 – Interfaces de componentes.png} – Interfaces de componentes {sommerville2011}

Um modelo de componente é uma definição de normas para implementação, documentação e implantação de componentes que garantem a interoperabilidade deles {sommerville2011}. Existem diversos modelos de componentes, entretanto neste trabalho serão abordados os modelos OpenCOM, Fractal e OSGI.

2.1.1.1 Modelo OpenCOM

2.1.1.2 Modelo Fractal

2.1.1.3 Modelo OSGI

Criadora da tecnologia OSGI, a OSGI Alliance foi criada por empresas como Eclipse, Apache, Red Hat, IBM, Oracle, entre diversas outras com o objetivo de criar especificações abertas que auxiliem na construção modular de um software. Dessa forma a tecnologia OSGI define um modelo de modularização para as aplicações Java. Ela facilita a modularização e garante a interoperabilidade de aplicações e serviços, como também facilita o gerenciamento remoto sobre uma ampla variedade de dispositivos. Além disso ela aumenta a produtividade do desenvolvimento de software e facilita a sua manutenção e evolução. Entretanto OSGI é apenas uma especificação. Existem diversas implementações dessa tecnologia, mas as mais conhecidas são: Equinox e Felix. A primeira é uma implementação da especificação OSGI desenvolvida pelo Eclipse. Ela é utilizada em diversas aplicações, inclusive no ambiente de desenvolvimento Eclipse. Já a outra implementação é desenvolvida pela Apache.

2.3 Sistemas Distribuídos

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens {Coulouris2007}. {Tanenbaum2007} definem um sistema distribuído como uma coleção de computadores independentes que aparece para o usuário como um único sistema. {Sommerville2011} menciona que os sistemas distribuídos são mais complexos que os sistemas centralizados, o que os torna mais difíceis de projetas, implementar e testar. Apesar dessa complexidade, praticamente todos os grandes sistemas computacionais são distribuídos.

{Sommerville2011 apud Coulouris2007} identifica vantagens da utilização de uma abordagem distribuída no desenvolvimento de sistemas:

* Compartilhamento de recursos: Um sistema distribuído permite o compartilhamento de recursos de hardware e software.
* Abertura: Os sistemas distribuídos são projetados para protocolos-padrão que permitem que os equipamentos e software de diferentes fornecedores sejam combinados.
* Concorrência: Em um sistema distribuído, vários processos podem operar simultaneamente em computadores separados na rede.
* Escalabilidade: Em princípio, os recursos de um sistema distribuído podem ser aumentados pela adição de novos recursos a depender da necessidade do sistema.
* Tolerância a defeitos: Um sistema distribuído pode ser tolerante a algumas falhas de hardware e software dispondo de vários computadores e replicando as informações importantes para o sistema.

2.4 Invocação Remota de Métodos (RMI)

A RMI é uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de um objeto que está em outro processo {Coulouris2007}. {Nascimento2013 apud Harold2004} menciona que a diferença entre objetos remotos e objetos locais é que os objetos remotos estão localizados em máquinas virtuais diferentes, assim como na Figura{figuras/006 – Invocação Remota de Métodos.png}. Devido à possibilidade de falhas independentes dos objetos invocadores e invocados , as RMIs têm semânticas diferentes das invocações a métodos locais, onde a transparência total não é necessariamente desejável {Coulouris2007}.

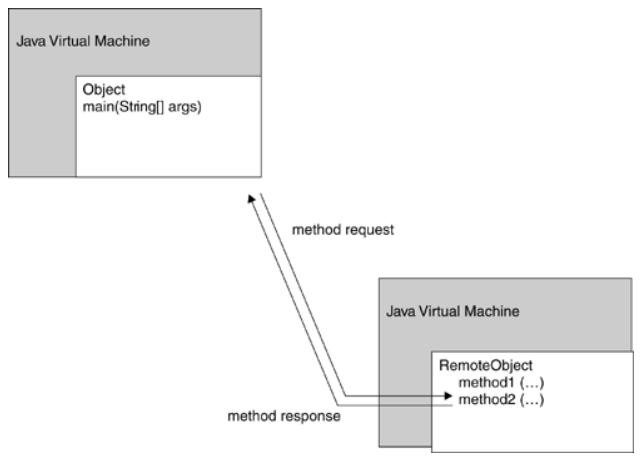


Figura {figuras/006 – Invocação Remota de Métodos.png} – Invocação Remota de métodos. {Reilly2002}

De acordo com {Reilly2002}, cada serviço RMI é definido por uma interface que descreve os métodos dos objetos que podem ser chamados remotamente. Segundo {Nascimento2013} e {Reilly2002}, essa interface deve ser compartilhada por todos os desenvolvedores uma vez que eles são incentivados a definir os métodos que podem ser chamados remotamente antes mesmo da implementação. {Nascimento2013} menciona que várias implementações da interface podem ser criadas, e os desenvolvedores não precisam estar cientes de que a implementação está sendo usada e nem aonde está localizada.

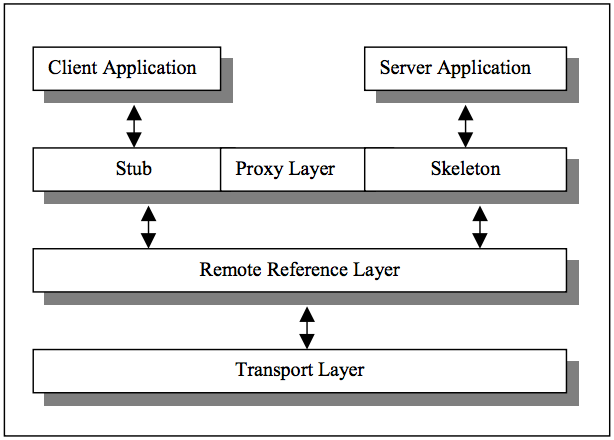


Figura {figuras/007 – Arquitetura em Camadas de RMI.png}: Arquitetura em Camadas da RMI. {Ruixian2000}

Segundo {Ruixian2000}, a arquitetura do RMI baseia-se em quatro camadas, assim como a Figura {figuras/007 – Arquitetura em Camadas de RMI.png}: Camada de Aplicação, Camada de Proxy, Camada de Referência Remota e Camada de Transporte.

A camada de mais alto nível, a de aplicação, é onde encontram-se as implementações das aplicações tanto do lado cliente quanto do lado servidor. A segunda delas, a camada de proxy, é responsável pelas chamadas aos objetos remotos. Nela é feito o empacotamento do parâmetros e o retorno do objeto. Para isso o lado cliente e o lado servidor assumem papéis diferentes. O primeiro é representado por Stub, já o segundo é representado por Skeleton. Em seguida vem a terceira camada, a de referência remota. Nela é feita a abstração entre a Camada de Proxy e a Camada de Transporte. Por fim vem a camada de transporte que define uma conexão entre as máquinas cliente e servidor {Ruixian2000}.

2.5 Serviços Web

Um serviço web (web service) fornece uma interface de serviço que permite aos clientes interagirem com servidores de uma maneira mais geral do que acontece com os navegadores web {Coulouris2007}. De acordo com {Deitel2010}, um serviço web é um componente de software armazenado em um computador que pode ser acessado por um aplicativo(ou outro componente de software) em outro computador por uma rede. Um serviço web possui uma interface descrita em um formato processável por máquina, especificamente a WSDL (Web Services Definition Language). Outros sistemas interagem com um *Web Service* utilizando mensagens de acordo com um padrão, tipicamente utilizando HTTP com uma serialização de XML, além de outros padrões relacionados a *Web*.

Os clientes acessam as operações na interface de um serviço web por meio de requisições e respostas formatadas em XML (Extensible Markup Language) e, normalmente, transmitidas por HTTP (HyperText Transfer Protocol) {W3c2004}. A XML é uma representação textual que, embora mais volumosa do que as representações alternativas, foi adotada por sua legibilidade e pela consequente facilidade de depuração {Coulouris2007} De acordo com {Deitel2010}, o serviço web pode ser apoiado em duas arquiteturas. A primeira é baseada no Simple Object Access Protocol (SOAP) e a segunda é baseada no Representational State Transfer (REST).

Segundo {Sommerville2011}, SOAP é um padrão de trocas de mensagem que oferece suporte à comunicação entre serviços. O SOAP é um protocolo independente de plataforma que utiliza a XML para fazer chamadas de procedimento remoto, geralmente sobre o HTTP {Deitel2010}. De acordo com {Coulouris2007}, originalmente o protocolo SOAP era baseado apenas em HTTP, mas a versão atual é projetada para usar uma variedade de protocolos de transporte, incluindo o SMTP, TCP ou UDP.

Segundo {Deitel2010}, o REST refere-se a um estilo arquitetônico de implementar serviços Web. Rest é uma estratégia com um estilo de operações muito restrito, no qual os clientes usam URLs e as operações HTTP, GET, PUT, DELETE e POST para manipular recursos representados em XML {Fielding2000}. Segundo {Deitel2010}, o REST também não está limitado a retornar dados no formato XML. Ele pode utilizar vários formatos, como XML, JSON, HTML, texto sem formatação e arquivos de mídia.

2.7 Geração Automática de Código

2.8 Discussão