PROJETO E IMPLEMENTAÇÃO DE CHAT USANDO REMOTE METHOD INVOCATION (RMI)

Diogo J. Cardoso[1], Jéssica A. Bonini[1]

[1]Departamento de Computação Aplicada – Universidade Federal de Santa Maria (UFSM)
Caixa Postal 5082 – 97195-000 – Santa Maria – RS – Brasil

{dcardoso, jbonini}@inf.ufsm.br

Abstract. *Centralized computer systems predominated at the beginning of the computer age. The advancement of technology has brought with it the need for more efficient systems such as distributed systems. These systems require the use of messaging mechanisms, since processes can be on different machines. One of these mechanisms is called Java RMI, a middleware that provides infrastructure for communication between objects. Thus, this article presents the construction of a Chat that uses RMI and uses a client-client architecture where the server acts only as a mediator.*

Resumo. *Sistemas computacionais centralizados predominaram no início da era da informática. O avanço da tecnologia trouxe consigo a necessidade de sistemas mais eficientes, como os sistemas distribuídos. Estes sistemas requerem o uso de mecanismos de troca de mensagens, já que os processos podem estar em máquinas diferentes. Um destes mecanismos é o chamado Java RMI, um middleware que fornece infraestrutura pra a comunicação entre objetos. Assim, este artigo apresenta a construção de um Chat que utiliza o RMI e usa uma arquitetura cliente-cliente onde o servidor age apenas como um mediador.*

1. Introdução

No século XIX, Charles Babbage foi responsável por criar o primeiro equipamento considerado um computador [Alves 2014]. Desde então a informática vem crescendo de maneira surpreendente. No início da era da informática, os sistemas computacionais eram fortemente centralizados [Tanenbaum 2003], ou seja, são executados em uma coleção de computadores e cada um possui recursos individuas. Existe ainda, um computador chamado de servidor que é responsável pela concentração das informações. A partir dos problemas apresentados por este tipo de sistema e da necessidade de sistemas mais eficientes os sistemas computacionais passam por uma descentralização e surgem os sistemas distribuídos.

Sistemas distribuídos são formados por computadores autônomos que se interligam por meio de uma rede e que podem compartilhar recursos do sistema como hardware, software e dados [Coulouris et. al 2006]. Um sistema distribuído é ainda transparente ao usuário, ou seja, se apresenta como um sistema único e coerente [Tanenbaum and Steen 2007]. Estes sistemas diferenciam-se dos centralizados, principalmente, pela maneira como a comunicação é feita. Levando em consideração que nos sistemas distribuídos os processos podem estar em máquinas diferentes, surge a necessidade de mecanismos de trocas de mensagem, como o *middleware*.

Um dos *middlewares* usados para fornecer uma infraestrutura de comunicação

entre processos é o Java RMI ou *Remote Method Invocation*. Sua arquitetura básica é cliente-servidor, mas pode-se ainda implementar comunicação com arquitetura cliente-cliente usando o servidor apenas como mediador [Tanenbaum and Steen 2007]. Dessa forma, este artigo apresenta a construção de um Chat com arquitetura cliente-cliente e que usa o RMI como método de comunicação.

2. Fundamentação Teórica

2.1. Sistemas Centralizados e Distribuídos

As duas primeiras décadas da informática foram marcadas por sistemas computacionais altamente centralizados [Tanenbaum 2003]. Sistemas centralizados são caracterizados pela execução em um conjunto de máquinas. Neste conjunto existe um computador chamado servidor, que concentra toda a informação e inúmeros outros provedores de dados, que utilizam recursos individuais [Stair 1998]. A Figura 1 apresenta um sistema centralizado.

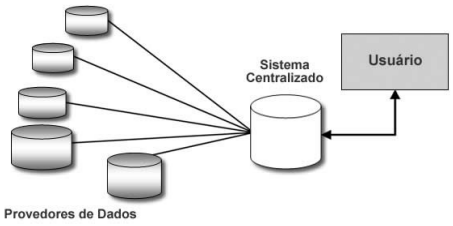

Figura 1. Sistema Centralizado.

Apesar de vantagens como segurança e baixos requisitos de infraestrutura, os sistemas centralizados possuem problemas quanto a disponibilidade dos dados e a ocorrência de gargalo, no caso de muitos acessos simultâneos [Stair 1998]. Dessa forma, existe a necessidade de sistemas mais eficientes, como os sistemas distribuídos.

Nos sistemas computacionais distribuídos os computadores são autônomos, interligados por meio de uma rede e podem compartilhar recursos do sistema (como impressoras e arquivos) através de um software para compartilhamento [Coulouris et. al 2006]. Pode-se dizer, que essa coleção de máquinas é transparente ao usuário, ou seja, se apresenta como um sistema único e coerente [Tanenbaum and Steen 2007]. Sistemas distribuídos possuem grande tolerância a falhas e um sistema de segurança baseado na confiabilidade, integridade e disponibilidade. Além disso, o problema do gargalo das consultas é minimizado, já que essas são feitas de forma distribuída [Stair 1998]. A Figura 2 mostra um exemplo destes sistemas,

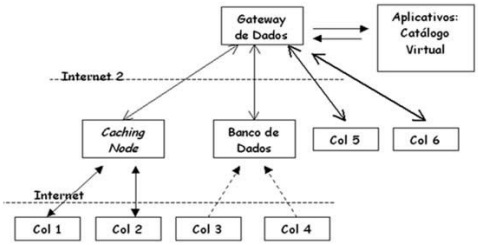

Figura 2. Sistema Distribuído.

A forma como ocorre a comunicação e a sincronização entre os processos caracteriza a principal diferença entre um sistema distribuído e um sistema centralizado.

No primeiro, a comunicação é feita através de uma memória compartilhada, já no segundo os processos podem estar em máquinas diferentes o que torna necessária a troca de mensagens. Para esta troca de mensagem são utilizados mecanismos de comunicação, entre eles está o *middleware*.

2.2. Middleware

A camada de software, Figura 3, executada entre o sistema operacional e uma aplicação é chamada de *middleware*, e é responsável pela implementação do modelo de sistema distribuído [Tanenbaum 2003]. O principal objetivo de um *middleware* é oferecer abstrações e recursos para o programador, de forma que esse não tenha real conhecimento dos detalhes de programação em rede [Valente 2014].

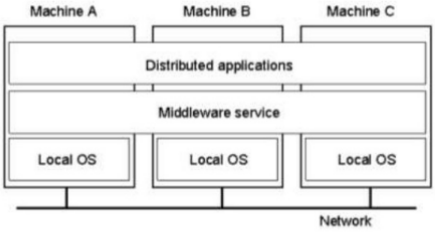

Figure 3. *Middleware*.

Podemos classificar os *middlewares* de acordo com o nível e o tipo de abstração provida: *host infrastructure middleware*, *common based middleware* e *communication middleware*. O primeiro refere-se ao *middleware* que fornece infraestrutura para a execução de aplicações e o segundo aos que fornecem diversos serviços como transações e autenticação. Já o último refere-se aos *middlewares* que fornecem infraestrutura de comunicação, ponto central dos sistemas distribuídos e deste artigo, um exemplo é o Java RMI [Valente 2014].

2.3. *Remote Method Invocation* (RMI)

A necessidade de permitir que linguagens de programação procedurais pudessem chamar funções localizadas em outros computadores e essas executassem como se estivessem na máquina local deu origem a *Remote Procedure Call* (RPC). A tecnologia de RPC apresentou dois problemas: a necessidade de lidar com dados complexos da programação orientada a objetos e a necessidade de conhecer a IDL (*Interface Definition Language*) utilizada para implementar as funções e para gerar os stubs do cliente, automaticamente, no servidor. Para lidar com estes problemas foi criado o *Remote Method Invocation*, desenvolvido especificamente para a linguagem de programação Java [Castro et. al 2007].

O RMI permite a comunicação entre objetos, que estão executando no mesmo computador ou em computadores diferentes, por meio de chamadas de métodos remotos, como se fossem métodos locais. Embora estas invocações sejam realizadas da mesma forma que para métodos locais, existe a possibilidade de falha e por isso existe a necessidade de tratá-las através de exceções. O modelo de execução do RMI é cliente-servidor, porém existe também a arquitetura cliente-cliente que pode ser mediada por um servidor.

2.4. Arquitetura Cliente-Servidor e Cliente-Cliente

Em uma arquitetura cliente-servidor, existe o processo servidor, responsável por

fornecer serviços ou recursos, e os processos clientes, os quais requisitam estes serviços. No serviço de Chat cliente-servidor, toda a comunicação ocorre entre estes dois tipos de processos, sendo que toda a informação da comunicação armazenada no servidor. Na execução do Chat, o *stub* cliente envia uma requisição ao *stub* do servidor em um número de porta e endereço IP especificado nos parâmetros de configuração do RMI. No lado do servidor, o *stub* (esqueleto) é responsável por receber a requisição, desempacotar os argumentos e ativar o método requerido. Após a execução do método, o servidor empacota a mensagem e envia ao cliente (objeto) que a solicitou através de uma invocação remota. A Figura 4 apresenta uma arquitetura cliente-servidor RMI e uma arquitetura cliente-cliente RMI, respectivamente.

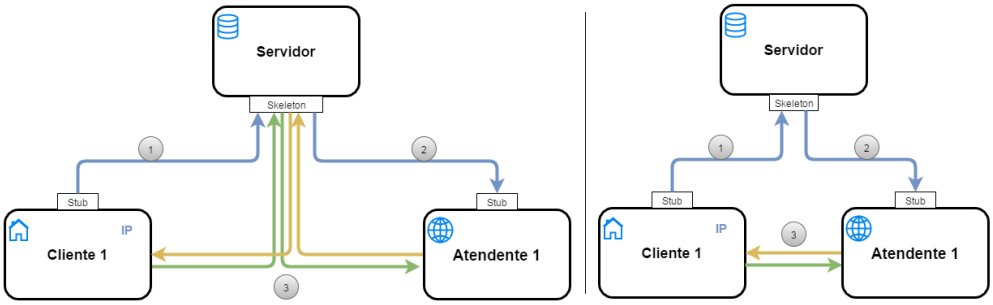

Figura 4. Arquitetura Cliente-Servidor RMI e Arquiteura Cliente-Cliente (Em azul as requisições e em amarelo e verde as mensagens).

Como apresentado na figura anterior, a principal diferença na comunicação está na forma como as mensagens são trocadas (círculo 3). Na arquitetura cliente-servidor todas as mensagens passam primeiramente pelo servidor, que é responsável por receber e transmiti-las entre os clientes. Já na cliente-cliente o servidor recebe apenas as requisições de conexão e após esta ser estabelecida a troca de mensagens ocorre diretamente entre clientes. Dessa forma, em uma arquitetura cliente-cliente a comunicação ocorre entre clientes e o servidor age como mediador, ou seja, somente como uma "conexão" entre os dois clientes. Essa última foi a usada na implementação do Chat desse artigo e será melhor explicada na seção 4.

3. Trabalhos Relacionados

3.1. Tecnologias: RPC, SOAP e CORBA

Em uma chamada de procedimento remoto (*Remote Procedure Call* – RPC) existem dois processos onde um (processo cliente) envia uma mensagem de requisição, contendo parâmetros para a execução de determinado método, a outro (servidor) e este último deve enviar uma mensagem resposta contendo os resultados da execução. O método de comunicação utilizando RPC é de alto nível, permitindo que o programador não precise se preocupar com detalhes de implementação da chamada remota, similarmente ao que acontece em uma chamada local.

Um componente da arquitetura de serviços Web é o SOAP (*Simple Object Access Protocol*). Este é um protocolo para a troca de mensagens em um ambiente distribuído. Baseia-se na linguagem XML e em outros protocolos da camada de aplicação como RPC e HTTP. Seus principais componentes são: um envelope que contém as informações das mensagens e como esta deve ser executada; um conjunto de

regras para explicar instâncias dos tipos de dados definidos na aplicação; e uma convenção que representa chamadas de procedimentos e respostas. Esse protocolo contém as chamadas e retornos dos *Web Services* encapsuladas em sua estrutura.

A tecnologia CORBA (*Common Object Request Broker Architecture*) surgiu para facilitar e estabelecer a troca de dados entre sistemas distribuídos heterogêneos de forma transparente, ou seja, desconsiderando a linguagem de programação, a localização e o sistema operacional (ou plataforma) no qual esteja sendo executada. A comunicação entre cliente e servo é feita através de *Object Request Brokers* (ORBs), estes utilizam o protocolo IIOP (*Internet Inter-ORB Protocol*) para estabelecer a comunicação.

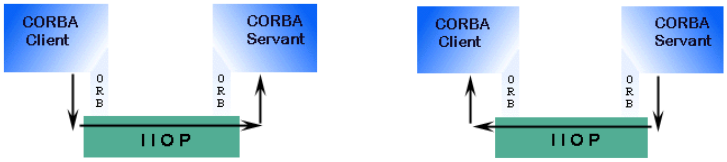

Figura 5. Cliente CORBA e servo CORBA, respectivamente.

Na Figura 5 O cliente CORBA envia sua requisição através de seu ORB local para um servo de um ORB remoto e esse servo retorna uma resposta ao ORB remoto do cliente.

3.2. RPC e CORBA vs RMI

A técnica RPC possui a vantagem de ocultar detalhes de soquetes ao programador e possuir ligação dinâmica com portas dos servidores. Porém existem problemas, aplicações do RPC pode ser executadas sobre protocolos TCP ou UDP, o que leva a implementações com diferentes níveis de confiabilidade. Além disso, RPCs não suportam variáveis globais.

CORBA possui vantagens quanto a possibilidade de escrever serviços em diversas linguagens de programação e executá-los em diferentes plataformas. Utiliza uma IDL que permite a separação clara entre interface e implementação, oferecendo aos programadores a possibilidade de criar diversas implementações com base na mesma interface. Além disso, é ideal para sistemas legados e para assegurar que as aplicações escritas no presente possam ser acessíveis no futuro. Suas desvantagens estão baseadas na obrigatoriedade e uma linguagem de definição de interface (IDL), na impossibilidade de transferir dados e código e na incerteza do futuro, já que não pode-se garantir que aplicações escritas hoje serão utilizadas ao longo dos anos.

O RMI possui a capacidade de enviar novos objetos (dados e códigos) através de uma rede e também para JVMs estrangeiros. Como essa tecnologia está presente desde a JDK 1.02 possui maior possibilidade de familiaridade por parte dos desenvolvedores. Possibilita que objetos remotos implementem uma ou mais interfaces e a definição de métodos dentro destas. Suas maiores desvantagens são as ameças de segurança com a execução remota de código e a limitação a linguagem orienta a objetos JAVA. Apesar disso, como todas as aplicações (cliente, atendente e servidor) seriam implementadas utilizando JAVA optou-se por utilizar o RMI como forma de comunicação.

4. Desenvolvimento[1]

Para esse artigo foi escolhida a arquitetura cliente-cliente devido ao desempenho

1 Código disponível no GitHub: https://github.com/diogocrds/ChatJavaRMI

desejado do sistema. Em um chat de atendimento espera-se que não haja atraso nas mensagens, ou seja, que estas cheguem no tempo correto de envio. Além disso, a arquitetura escolhida evita a sobrecarga do servidor o que torna menos provável falhas que afetam todo o sistema, como, por exemplo, lentidão ou quedas do serviço. As mensagens são enviadas diretamente entre clientes, reduzindo o controle do servidor sobre os dados transferidos na rede e aumentando a privacidade entre os clientes.

4.1. Chat: Arquitetura Cliente-Cliente

No *middleware* implementado para este artigo, existem três processos diferentes: cliente, servidor e atendente. O processo cliente é responsável por requisitar, ao servidor, uma conexão com algum atendente livre; o processo servidor recebe esta requisição e passa uma referência do primeiro atende livre que ele encontrar para o cliente; e por último, o processo atendente fica aguardando até receber uma conexão com um cliente desejando ser atendido.

No momento que o atende e o cliente estabelecem a conexão, o servidor não tem mais influência sobre eles e fica na espera de novas requisições. Este processo pode receber dois tipos de requisições: requisição do atendente e requisição do cliente. A primeira, é a requisição feita por um novo atendente que deseja cadastrar-se no sistema para atender os clientes. Após receber esta requisição, o servidor cadastra o atendente no sistema e põe uma referência a ele na lista de atendentes livres. A segunda, como citado anteriormente, acontece quando um cliente requisita a conexão com um atendente. O servidor consulta sua lista de referências a atendentes livres e envia a primeira para o cliente. Caso a lista de atendentes esteja vazia, o servidor guarda a referência do cliente em uma fila de espera. Assim que um atendente ficar livre, o servidor envia sua referência ao primeiro cliente da fila.

4.2. Chat: Interfaces e Métodos

O servidor (*Class ServerChat*) implementa a interface remota *IServer* que receberá as requisições de atendentes e clientes. Os clientes (*Classe ClientChat*) e atendentes (*Classe AtendentChat*) implementam a interface remota *IChat* que será responsável no caso dos clientes por enviar a requisição ao servidor e a mensagem com a referência ao atendente; e no caso dos atendes por enviar a requisição ao servidor e por receber a mensagem com pedido de conexão do cliente. Esta interface será responsável pela troca de mensagens através do método *void deliver(IMessage msg)*.

Ao final de cada atendimento, o servidor testa se existe algum cliente na fila de espera e aloca o atendente livre a ele, senão o atendente entra na lista de atendentes livres. A ação de encerrar o atendimento pode ser executada tanto pelo atendente quanto pelo cliente. O cliente só tem a opção de fechar a janela, terminando o processo, pois deve entrar na fila de novo caso queira outro atendimento. Já o atendente tem a opção de somente “encerrar” o atendimento, sendo alocado a um novo cliente assim que possível.

Se o Chat for encerrado pelo atendente, esse deve invocar o método *freeAtendent(IChat chat_atendent)* do servidor e assim, informar que está livre para novo atendimento. Caso queira terminar o processo (opção “Fechar”), o atendente deve invocar o método *requesteLeave(IChat chat_atendent)* do servidor, indicando que não está mais disponível para atendimentos.

O contato do cliente ao servidor deve ser feito através da invocação remota do método *requestAtendent(IChat chat_client)* e o contato entre atendente servidor dever ser feita pela invocação remota ao método *requestJoin(IChat chat_Atendent)*. Dessa

forma, o servidor age como indexador entre clientes e atendentes. As mensagens (*Class Message*) que trafegam entre cliente e atendente devem implementar a interface *IMessage*.

5. Resultados

5.1. Chat: Telas

A Figura 6 mostra um atendente recém-cadastrado no servidor, ele é colocado na fila de atendentes livres pois no momento não existe nenhum cliente aguardando conexão.

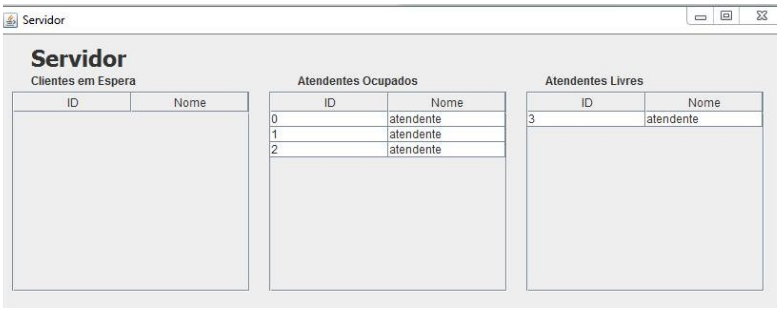

Figura 6. Atendente Livre na Fila de Espera.

Na Figura 6 ainda é possível observar as três listas que o servidor mantém. A primeira contém os clientes que solicitaram um atendente e aguardam a referência, já que não existem atendentes livres. A segunda e a terceira contêm os atendentes ocupados e livres, respectivamente, até o momento.

Figura 7. Tela do cliente e atendente após conexão.

O cliente requisita um atendente ao servidor e está aguardando a referência, caso exista algum atendente livre. Se existir atendente livre, o servidor envia a referência para o cliente e esse conecta-se diretamente ao atendente referenciado. Na Figura 7, estão representadas as telas do cliente e do atendente, respectivamente, após a conexão e o início da troca de mensagens. Se não existir atendente livre, o cliente vai para uma fila de espera. Se o atendente quiser encerrar o processo deve clicar no botão “Fechar”, assim o servidor recebe uma requisição de encerramento e fica ciente que aquele atendente não existe mais.

6. Conclusão

Sistemas distribuídos têm sua principal característica baseada na premissa de comunicação entre processos, estejam eles na mesma máquina ou em máquinas diferentes. Este fator traz consigo a necessidade da invocação de métodos de forma remota para tornar possível a troca de mensagens. Existem diversos mecanismos que possibilitam esta troca, entre eles está o *middleware*, uma camada de software localizada entre o sistema operacional e a aplicação.

No ponto de vista de sistemas, o software implementado é classificado como um midleware de comunicação, onde suas principais características são: a transparência ao usuário e a possibilidade de funcionar tendo processos em máquinas diferentes. A utilização do RMI, *middleware* da linguagem Java, traz como vantagem o fato de que na implementação não é necessário se preocupar com lógica relacionada à rede. Essa funcionalidade, que garante a comunicação confiável no canal de rede, é movida para a camada do *stub* e do *skeleton*.

Com este trabalho, também foi possível perceber que a escolha da arquitetura do RMI a ser utilizada influencia no desempenho do sistema. A escolha deve ser feita dependendo do objetivo a ser atingido. No caso de um Chat de atendimento, deseja-se que as mensagens sempre cheguem no tempo correto e o servidor não sofra sobrecarga, por isso a arquitetura Cliente-Cliente é a mais ideal. Dessa maneira o cliente e atendente enviam as mensagens diretamente um ao outro, poupando o servidor de ter que manejar todos os dados sendo transferidos na rede.

Referências Bibliográficas

Alves, P. (2014), “Dia da Informática: confira a história do computador e sua evolução”. TechTudo, Informática – Computadores. Disponível em <http://www.techtudo.com.br/noticias/noticia/2014/08/dia-da-informatica-confira-historia-do-computador-e-sua-evolucao.html>, Acesso: Maio de 2016.

Coulouris, G., Dollimore, J. and Kindberg, T. (2006) “Distributed Systems: Concepts and Design”. Addison-Wesley, 4ª edição.

Castro, M., Raeder, M. and Nunes, T. (2007) “RMI: Uma Visão Conceitual”. Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), RS. Disponível em <http://www.inf.pucrs.br/~gustavo/disciplinas/sd/material/Artigo_RMI_Conceitual.pdf>, Acesso: Maio de 2016.

Stair, M. (1998) “Princípios de sistemas de informação”. LCT, 2 edição. Rio de Janeiro, Brasil.

Tanenbaum, A. (2003) “Computer Networks”. Prentice Hall PTR, 4 edição. Amsterdã, Holanda.

Tanenbaum, A. and Steen, M. (2007) “Distributed Systems: Principles and Paradigms”. Prentice Hall PTR, 2 edição. Amsterdã, Holanda.

Valente, M. (2014) “Arquitetura de Software para Aplicações Distribuídas”. Universidade Federal de Minas Gerais, MG Disponível em <www.dcc.ufmg.br/~mtov/arqsw/middleware_2014.pdf>, Acesso: Maio de 2016.