MIDLEWARE PARA UM SISTEMA DE AGÊNCIA DE VIAGENS

MIDDLEWARE FOR A TRAVEL AGENCY SYSTEM

Avelino Francisco Zorzo*
Elder Macedo Rodrigues**
Filipi Dias Teixeira***
Jean Schmidt****
Leandro Teodoro Costa*****
Vilmar Consul******

* Doutor em Ciência da Computação pela Universidade de Newcastle upon Tyne, Inglaterra. Bolsista de produtividade em pesquisa do CNPq. avelino.zorzo@pucrs.br

** Doutorando em Ciência da Computação pelo Programa de Pós-graduação em Ciência da Computação na PUCRS.

*** Bacharel em Ciência da Computação pela PUCRS

**** CTO Travel Explorer

***** Doutorando em Ciência da Computação pelo Programa de Pós-graduação em Ciência da Computação na PUCRS.

****** CFO Travel Explorer.

Resumo

Este trabalho apresenta um modelo de arquitetura de um sistema para uma empresa de agência de viagens, a qual, com o aumento da demanda de clientes, apresentou dificuldades em escalar a capacidade de seu sistema, uma vez que utilizava um modelo de arquitetura tradicional, com todos os serviços integrados. As dificuldades estavam relacionadas, principalmente, ao consumo excessivo de *threads*, à escalabilidade cara do sistema, à sobrecarga na base de dados e à limitação de recursos por cliente. Todas essas restrições influenciavam nos tempos de resposta, afetando o desempenho global do sistema. O objetivo do emprego deste novo modelo foi sanar os problemas citados, através da separação dos serviços do sistema,. Desta forma, este artigo apresenta um conjunto de tecnologias, como gerenciadores de filas de mensagens, sistemas de bancos de dados chave-valor e sistemas de *cache* distribuído, que visam agregar valor e otimizar as funcionalidades do sistema. Baseado no estudo do conjunto de tecnologias analisadas, foi sugerido o uso de três ferramentas com base nas características do sistema proposto. Para se verificar a aplicabilidade das tecnologias escolhidas, foram realizados testes do novo modelo de arquitetura[1].

1 - A ordem dos autores é meramente alfabética

Palavras-chave: Gerenciadores de Filas de Mensagem. Sistemas de Bancos de Dados. Sistemas de *Cache* Distribuído.

Abstract

This paper shows a new architecture for a travel agency system. The current system has some scalability problems that the new proposed architecture intends to solve. These problems are mainly related to the excessive use of threads, cost system scalability, overload of the database and limitation of resources per customer. All these limitations have a direct influence on the response time and consequently the overall system performance is affected. For the employment of this new proposed architecture, which aims to solve the mentioned problems, we, initially, analyze different middleware technologies that could optimize the functionalities of the system. Following the analysis of these technologies, we chose three technologies (ActiveMQ, Memcached and Tokyo Cabinet) based on the characteristics of the system. These technologies were chosen after a thorough study, throughout a series of performance tests. Furthermore, to verify whether the chosen technologies would behave in accordance to the expected for the new architecture, a new series of tests that combined the three middleware technologies was performed.

Keywords: Message Broker. Distributed Store Systems. Distributed Cache Systems.

1 Introdução

Com a presente expansão tecnológica, muitas empresas então buscando agilizar o modo de conduzir seus negócios através de recursos computacionais inovadores. Um recurso bastante utilizado nos dias de hoje está diretamente relacionado às tecnologias para comunicação entre processos através de troca de mensagens (programação distribuída) (TANENBAUM, 2006), tais como, Chamada de Procedimento Remoto (*Remote Procedure Call* - RPC), Invocação de Método Remoto (*Remote Method Invocation* - RMI), *Web Services* (YU, 2008), entre outras. Atualmente, existem diversas aplicações desenvolvidas que utilizam o paradigma da programação distribuída; neste contexto são citados os sistemas bancários, os sistemas para gerenciamento de redes de telecomunicações, os sistemas de informação de grandes empresas, etc.

A utilização de sistemas distribuídos pode trazer inúmeras vantagens, entre as quais, o aumento na confiança no seu funcionamento, ou dependabilidade

(*dependability*) (ROMANOVSKY, 2003; ZORZO, 2003). A dependabilidade é composta por um conjunto de atributos de qualidade, como a disponibilidade, que caracteriza a probabilidade de um sistema funcionar corretamente em um determinado espaço de tempo. Isso é possível porque, com a carga de trabalho distribuída entre diversos computadores, caso uma das máquinas venha a falhar, o sistema continuará ativo. Outras vantagens na utilização de sistemas distribuídos podem ser encontradas em Tanenbaum (2006). Entretanto, existem problemas de difícil tratamento nesse tipo de sistema, tais como questões de segurança dos dados, sincronização de eventos de comunicação, congestionamento (LAMPORT 1978).

Outro problema na utilização deste paradigma acontece quando um número excessivo de conexões/requisições é realizado pelos clientes junto ao(s) servidor(es). A consequência é um alto tempo de processamento dessas requisições, influenciando diretamente nos tempos de resposta. Uma solução que aparentemente resolveria esse tipo de questão seria aumentar o "poder" computacional das máquinas ou aumentar o número de servidores que hospedam esses sistemas – escalabilidade horizontal (MICHAEL, 2007). Entretanto, esta alternativa nem sempre é viável, pois o acréscimo de máquinas pode representar um alto custo na manutenção do *hardware*, custos energéticos, entre outros, ou seja, os benefícios podem não compensar o investimento.

Neste trabalho, apresenta-se um projeto desenvolvido em colaboração com uma empresa que oferece serviços para agência de viagens. O objetivo principal é alterar a arquitetura do sistema existente, que oferece alguns problemas de escalabilidade, com o aumento no número de consultas e por consequente, a diminuição no tempo de resposta. O modelo da arquitetura do sistema utilizado para atender aos clientes (agências de viagens) possui camadas pertencentes ao mesmo projeto, ou seja, o sistema é monolítico. Assim, todo e qualquer processo de manutenção no sistema torna-se oneroso.

Com o intuito de contornar esses problemas e aproveitar melhor os recursos de *hardware* existentes – escalabilidade vertical (MICHAEL, 2007), este artigo propõe um modelo de arquitetura modular para sistemas de agências de viagens. Além disso, sugere um conjunto de tecnologias que permitam a modularização. Foram pesquisados os seguintes: gerenciadores de filas de mensagem, sistemas de bancos de dados chave-valor e sistemas de *cache* distribuído.

Este trabalho está estruturado da seguinte forma: a Seção 2 apresenta alguns dos gerenciadores de filas de mensagens existentes, características de dois sistemas de bancos de dados chave-valor, e ferramentas de sistemas de

cache distribuído; na Seção 3, é demostrado um exemplo de uso, relatando-se os problemas do sistema de agência de viagens existente; a Seção 4 traz uma proposta de modelo de arquitetura, em que as ferramentas citadas são utilizadas para otimizar o desempenho do sistema; a Seção 5 faz uma descrição do ambiente de teste e de análise dos resultados; por fim, a Seção 6 apresenta as conclusões referentes ao trabalho.

2 Tecnologias analisadas

Para melhor entendimento do problema e de sua solução, é importante antes familiarizar-se com alguns termos e conceitos. Por este motivo, nesta seção são descritas as características e funcionalidades de três diferentes tipos de tecnologias: gerenciadores de filas de mensagens, sistemas de bancos de dados chave-valor e sistemas de *cache* distribuído. Essas tecnologias foram utilizadas com o intuito de amenizar alguns dos problemas descritos anteriormente.

2.1. Gerenciadores de Filas de Mensagens

Um gerenciador de filas de mensagem (*message broker*) é um programa que traduz mensagens de um protocolo de um remetente (*sender*) para um recebedor (*receiver*) em uma rede, em outras palavras, faz o intermédio da comunicação entre aplicações. Um gerenciador de filas de mensagem pode ser visto como um conjunto de filas, no qual as mensagens são armazenadas e enviadas de acordo com a ordem de chegada (YAMAMOTO, 2009). O objetivo desse tipo de ferramenta em um sistema distribuído é realizar o balanceamento de carga entre os servidores através da gerência dos dados das filas, ou seja, uma vez que determinada fila, que envia mensagens para um servidor, está cheia devido a uma grande quantidade de mensagens que estão sendo processadas por esse servidor, o *message broker* pode ser configurado para enviar mensagens para as filas de um servidor que dispõe de maior quantidades de recursos ociosos e subutilizados, e, com isso, aproveitar de forma mais eficiente os recursos do ambiente. A seguir, são apresentados 3 gerenciadores:

- ActiveMQ (HENJES, 2007): é um sistema gerenciador de filas de mensagem (*message broker*). É uma ferramenta *open source* que implementa Serviços de Mensagem Java (*Java Message System* - JMS) permitindo comunicação entre processos com suporte para clientes em várias linguagens, entre elas Java (HORSTMANN, 2001), C++ (STROUSTRUP, 1993), Ruby

(FLANAGAN, 2008), Python (LUTZ, 2006). Além disso, o ActiveMQ tem muitas características avançadas, por exemplo, ele suporta JMS 1.1 e J2EE 1.4, fornece recursos como agrupamento (*clustering*) e armazenamento de múltiplas mensagens.

- FUSE Message Broker (FUSE, 2010): é uma evolução do ActiveMQ, que fornece escalabilidade e infraestrutura para conectar processos através de sistemas heterogêneos. O FUSE Message Broker é capaz de repassar grandes quantidades de dados de forma eficiente e confiável. Por ser baseado na implementação do ActiveMQ, o FUSE Message Broker possui funcionalidades muito similares a de seu predecessor, assim como o suporte aos mesmos clientes do ActiveMQ.
- RabbitMQ (RABBITMQ, 2010): é um sistema gerenciador de filas de mensagem (*Message Broker*). É uma implementação *open source* de serviços de comunicação (*messaging*) e para isso utiliza o protocolo *Advanced Message Queueing Protocol* (AMQP). O AMQP define um conjunto de normas e especificações para interoperabilidade de serviços de mensagens instantâneas (*messaging*). Além do RabbitMQ, existem outras duas implementações de aplicações que usam o AMQP, o QPID da Apache Foundation e o OpenAMQ (criado pelo grupo que definiu a especificação AMQP).

2.2. Sistemas de Bancos de Dados Chave-Valor

Base de dados chave-valor são sistemas eficientes de armazenamento e recuperação de dados de grande escala para aplicações *web*. Eles armazenam instâncias de uma entidade na forma de pares chave-valor onde uma chave corresponde à identificação de um valor e um valor consiste em um campo de dados (WANG, 2009). A seguir, são apresentados dois sistemas de BD chave-valor:

- MemcacheDB (HACKL, 2010): é um sistema distribuído de armazenamento que utiliza um mecanismo de persistência de dados baseado em chave-valor e possui um sistema para recuperação desses dados rápido e confiável. É compatível com o protocolo Memcached (veja Seção 2.3), portanto qualquer cliente Memcached pode conectar-se com ele. O MemcacheDB utiliza o Berkeley DB como suporte de armazenamento e uma grande quantidade de recursos como transação e replicação são suportados.
- Tokyo Cabinet (HACKL, 2010): é uma biblioteca de rotinas para gerenciar bases de dados (chave-valor) disponível para o sistema operacional Linux. A base

de dados é um arquivo simples que contém registros, ou seja, tuplas compostas de chave e valor. Cada chave e valor é uma sequência de bytes com tamanho variável. Tanto dados binários quanto *strings* de caracteres podem ser utilizados. Não há conceitos de tabelas, nem de tipos de dados. Os registros podem estar organizados em tabelas *hash*, árvores *B* ou em vetores de tamanho fixo.

2.3. Sistemas de Cache Distribuído

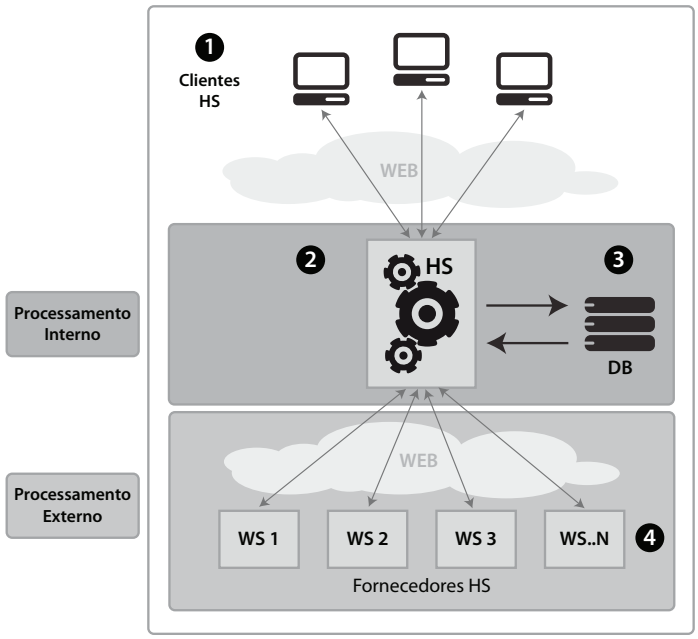
Sistemas de *cache* distribuído são ferramentas utilizadas para aliviar a carga em sistemas que fazem muitos acessos ao banco de dados. Em aplicações *web* que armazenam seus dados em um banco de dados, por exemplo, cada requisição de página requer um acesso à base de dados. Quando diversas requisições de páginas são efetuadas, muitos acessos ao banco de dados também são gerados. Desta forma, o banco pode se tornar o gargalo do sistema (TAY, 2000). Para evitar esse tipo de problema, a utilização de um mecanismo de *cache* pode se tornar uma alternativa muito interessante. Com isso, ao invés de armazenar os dados em disco (base de dados), os dados são armazenados em memória, tornando o processo de leitura muito mais rápido. A seguir, dois sistemas de *cache* distribuídos são apresentados:

- Memcached (LERNER, 2009): é um sistema de *cache* de objetos em memória concebido para aumentar a velocidade de aplicações dinâmicas, aliviando a carga no banco de dados. Ele permite que se armazene qualquer tipo de texto desde que esteja armazenado em um *array*. Além disso, o Memcached permite definir o local (uma chave) onde os dados serão armazenados e também por quanto tempo o conteúdo será armazenado.
- Shared Cache (SHARED, 2010): fornece topologias de *cache* distribuído e replicado, foi desenvolvido para aplicações Microsoft .NET (BROWN, 2004) que executam sobre *Server farms* (GANDHI, 2009). O objetivo é minimizar a carga de trabalho no banco de dados. A vantagem obtida com essa ferramenta é a capacidade de escalar as aplicações utilizando-se apenas uma quantidade maior de recursos de *hardware*, sem qualquer custo adicional de *software*. Além disso, o Shared Cache fornece um conjunto de topologias, o que permite a escolha entre as opções de armazenamento de *cache* que melhor se adapte às necessidades de uma arquitetura. Atualmente, o Shared Cache suporta três tipos de topologias, são elas: *Distributed Caching - partitioned*, *Replicated Caching* e *Single Instance Caching* (SHARED, 2010).

3 Modelo de Arquitetura Monolítico

O sistema de agência de viagens utilizado neste artigo é denominado Hotel Suite (HS) e tem como objetivo permitir que clientes possam efetuar a reserva em qualquer hotel que pertença à rede de hotéis fornecedores cadastrados no HS. A busca por esses hotéis, por parte dos clientes, pode ser realizada através da utilização de algum tipo de filtro como, por exemplo, localização do hotel, valor da diária, tipo de quartos, entre outros.

Figura 1: Visão geral da arquitetura HS



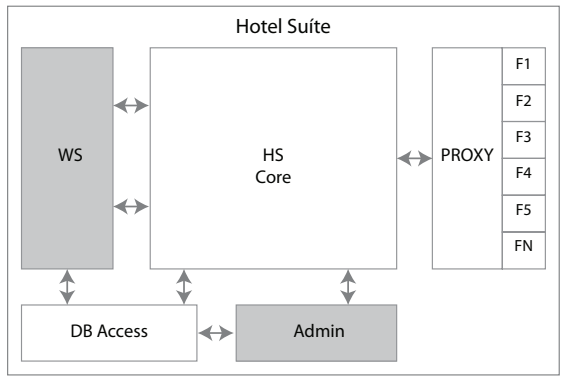
Fonte: autor Vilmar Consul

A realização desta busca/pesquisa ocorre de acordo com o que é descrito na Figura 1. Inicialmente, os Clientes$_{(1)}$ conectam-se ao HS$_{(2)}$ através de um navegador *web* qualquer e realizam uma pesquisa por hotéis que dispõem das características solicitadas pelos Clientes$_{(1)}$. Em seguida, o HS $_{(2)}$ verifica na base de dados (DB$_{(3)}$) os fornecedores contratados e dispara pesquisas XML. Os fornecedores$_{(4)}$ recebem e processam as pesquisas, retornando essas informações ao HS$_{(2)}$. O HS$_{(2)}$ sincroniza e persiste os resultados no DB$_{(3)}$ e por último, o HS$_{(2)}$ recupera os resultados persistidos no DB$_{(3)}$ e envia a resposta aos Clientes$_{(1)}$.

O HS, por possuir um conjunto de camadas unificadas (ver Figura 2), permite a criação de novas regras de maneira mais simples e fácil. Por outro

lado, qualquer manutenção no sistema requer a realização de *builds* e testes do fluxo completo de pesquisas e reservas, o que torna o processo caro e longo. Existem ainda, outras desvantagens neste modelo de arquitetura, por exemplo, não há controle de escala para solicitações que chegam ao HS. Um dos maiores problemas está relacionado ao pouco uso de *cache* de dados, ocasionando uma sobrecarga na base de dados.

Figura 2: Modelo de arquitetura monolítico do HS



Fonte: autor Vilmar Consul

As desvantagens e problemas encontrados para este modelo foram evidenciados após uma análise do perfil de utilização do sistema HS. Para melhor compreensão do perfil de utilização do sistema, na Tabela 1, apresenta-se uma estimativa de gasto do HS para o processamento de pesquisas dos destinos. É importante salientar que os dados apresentados foram obtidos com base em um volume mediano de 50 pesquisas por minuto.

Tabela 1. Perfil de utilização do sistema HS

			Threads por pesquisa	
Nome da Cidade	Pais	% Pesquisas Dia	# Threads Externas	# Threads Internas
Miami Beach	US	42,00%	323,4	231
Nova Iorque	US	10,00%	88	55
Orlando	US	8,00%	57,2	44
Miami	US	6,00%	39,6	33
Buenos Aires	AR	6,00%	59,4	33
Paris	FR	4,00%	68,2	22
Madrid	ES	4,00%	103,4	22
Las Vegas	US	3,00%	36,3	16,5
Roma	IT	3,00%	79,2	16,5
Orlando - Walt Disney World	US	2,00%	5,5	7,7

São Paulo	BR	2,00%	8,8	11
Barcelona	ES	2,00%	46,2	11
Punta Cana	DO	2,00%	16,5	11
Lisboa	PT	1,00%	10,45	5,5
Los Angeles	US	1,00%	13,2	5,5
Maceio	BR	1,00%	2,2	3,3
Londres	GB	1,00%	3,85	4,95
Santiago do Chile	CL	1,00%	3,85	4,95
Cancum	MX	1,00%	7,7	5,5
		Totais	972,95	543,4
		PPP	50	

Fonte: autor Elder M. Rodrigues

Como é possível observar, as *Threads* por pesquisa estão divididas em dois grupos, *Threads* Internas e *Threads* Externas. O primeiro grupo é composto por *threads* responsáveis pelo processamento interno do HS, ou seja, executar tarefas de pesquisas e persistência dos resultados. O segundo grupo de *threads* é responsável pelo processamento externo do HS, ou seja, durante a comunicação com os fornecedores. Como se pode observar, o consumo total de *threads* externas (64%) e internas (36%) é igual a 972,95 e 543,4 respectivamente, consumo este considerado excessivo para a execução do sistema.

Conforme foi apresentado, vários problemas são observados para este modelo de arquitetura. A seguir, são descritas as principais limitações verificadas para o sistema:

- Consumo de *threads* excessivo. Observa-se que o HS consome um número elevado de *threads* para processar as pesquisas de hotéis. Baseado nos monitoramentos, verificou-se que acima de 1800 *threads* começam a surgir problemas de contenção para o sistema operacional efetuar o processamento. Isso ocasiona tempos elevados nas pesquisas e *timeouts*;
- Escalabilidade cara. Implica na criação de servidores *web* e DB adicionais com custo de aluguel, licenciamento e manutenção dobrados, no caso de atualizações do sistema (publicação HS);
- Dependência de DB. A variação do volume de pesquisas por minuto torna o banco de dados um gargalo natural devido ao custo transacional de persistências de pesquisas, ocasionando *timeouts*;
- Limitação de recursos por cliente. Impossibilidade de se limitar ou dedicar recursos de infraestrutura por cliente via *software* (HS), em função de não ser possível restringir o volume de pesquisas por minuto individualmente, o que pode causar instabilidades.

Com o intuito de sanar os problemas desta arquitetura, foi proposto um novo modelo, no qual as camadas do sistema não são integradas e as tecnologias apresentadas anteriormente (Seção 2), são utilizadas para otimizar o desempenho do sistema. Na seção seguinte, apresenta-se de forma detalhada a descrição deste novo modelo.

4 Modelo de Arquitetura Proposto: Modular

Com base na descrição dos problemas apresentados na Seção 3, foi proposto um novo modelo de arquitetura modular para o sistema HS. A modularização será atingida com o uso de gerenciadores de filas de mensagens, sistemas de bancos de dados chave-valor e sistemas de *cache* distribuído. A seguir, verifica-se o ganho pretendido com a utilização de cada um desses sistemas:

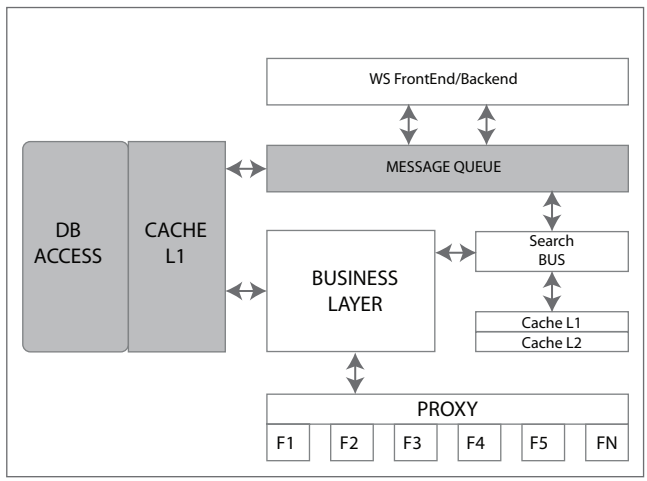
- Gerenciadores de filas de mensagens. Esses sistemas permitem o intercâmbio de informações assíncronas entre partes do sistema que podem estar distribuídas em uma ou mais máquinas. Com esse tipo de sistema, é possível a criação de filas de processamento para cada cliente ou fornecedor, por exemplo, com limitações no número de pesquisas por minuto, e distribuição de *threads* por várias máquinas. Com isso, ganha-se escala e gerenciamento de utilização de recursos;
- Sistemas de bancos de dados chave-valor. São sistemas de armazenamento de dados simplificados, sem estrutura de tabelas e sem administração, porém com bom desempenho, pois operações de gravação e recuperação de informações são realizadas de forma mais rápida. O objetivo principal é a gravação e o auxílio no processamento dos resultados de busca retornados pelos fornecedores e pelo HS;
- Sistemas de *cache* distribuído. São sistemas que possibilitam o armazenamento em *cache* das aplicações em máquinas distribuídas na rede. A leitura e recuperação de informações dessas aplicações ocorrem de maneira quase que instantânea. Esse sistema pode ser empregado na criação de aplicações *web* 2.0, utilizando Jquery (WELLMAN, 2009) e outros recursos para construção de páginas de processamento instantâneo. No caso do HS, será utilizado para efetuar o armazenamento em *cache* de informações dos hotéis mais utilizados pelos fornecedores, tornando mais rápido o processo de pesquisa e recuperação dos dados.

Como apresentado na Seção 2, para cada uma dessas tecnologias foi pesquisado e testado um conjunto de ferramentas. As ferramentas analisadas

foram: os gerenciadores de filas de mensagens ActiveMQ, Fuse Message Broker e RabbitMQ; os sistemas de bancos de dados chave valor MemcacheDB e Tokyo Cabinet; os sistemas de *cache* distribuído Memcached e Shared Cache. A escolha por essas ferramentas foi baseada no resultado de testes realizados, em que os fatores determinantes foram o tempo de resposta para operações de leitura, a escrita e a remoção de dados, assim como a compatibilidade de cada ferramenta com o sistema. O modelo de arquitetura proposto que agrega as tecnologias analisadas pode ser visto na Figura 3. Diferentemente do modelo atual do sistema, nesta nova versão da arquitetura, o HS não possui camadas unificadas e a utilização das ferramentas analisadas torna o processo de pesquisa e a persistência dos dados mais rápidos.

Para validar este modelo de arquitetura, um conjunto de testes foi realizado. A descrição dos testes e o ambiente de execução são apresentados de forma detalhada na Seção 5.

Figura 3: Modelo de arquitetura modular para o sistema HS



Fonte: autor Vilmar Consul

5 Ambiente de Teste

Esta etapa consistiu na execução de testes do sistema, utilizando-se APIs na linguagem de programação Python (LUTZ, 2006) para as ferramentas. Com o intuito de validar o modelo de arquitetura proposto, uma gama de testes foi executada. Para aumentar a confiabilidade dos testes, a carga de trabalho e o fluxo de dados no sistema foram os mesmos empregados no ambiente de produção.

Para a execução dos testes do sistema, duas máquinas foram utilizadas. Na primeira máquina, executou-se uma combinação dos serviços de gerenciamento de filas, *cache* distribuído e bancos de dados chave-valor. Na segunda máquina, executou-se o sistema HS. Para a simulação do ambiente, foram utilizadas mensagens de requisição de 1 KB e de 4 KB, contendo as informações referentes aos hotéis.

O objetivo dos testes foi simular usuários realizando diferentes tipos de pesquisas por hotéis e realizar uma análise sobre o tempo de resposta global do sistema e também sobre o tempo médio de resposta do sistema para atender a cada usuário. O ambiente foi testado, inicialmente, com 15 usuários realizando requisições simultâneas ao sistema. Em seguida, o mesmo teste foi realizado com 25, 35 e 45 usuários; para cada teste foram obtidos os respectivos tempos de resposta. É importante salientar que cada teste foi repetido 29 vezes com o intuito de aumentar a confiabilidade dos resultados.

A Tabela 2 e a Figura 4 apresentam resultados dos testes utilizando uma API em Python com os seguintes *midlewares*: ActiveMQ, Tokyo Cabinet e Memcached. Os resultados obtidos para os testes foram satisfatórios, uma vez que os tempos de resposta foram inferiores a 1 segundo para todos os testes executados. Este parâmetro foi definido pela empresa que desenvolve o HS.

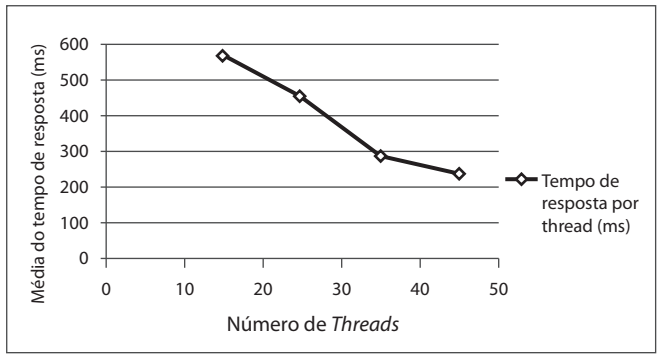
Tabela 2: Tempo de resposta médio por usuário *(thread)*

Nº de Threads	Média (ms)	Desvio Padrão	Mediana	Moda	Variância
15	567,48	206,5	582	n/a	42645,9
25	455,86	148,69	466	486	22108,91
35	294,93	59,87	288	259	3584,42
45	244,72	67,88	235	202	4608,21

Fonte: autor Elder M. Rodrigues

Na Tabela 2, é possível observar os valores obtidos da média dos vinte e nove testes realizados sobre a estrutura. Também foram realizados cálculos estatísticos que mostram com exatidão o comportamento dos resultados. Para melhor visualização da evolução do comportamento dos testes na estrutura, optou-se por mostrar graficamente alguns valores referentes aos testes (ver Figura 4).

Figura 4: Tempo de resposta por usuário *(thread)*



Fonte: autor Vilmar Consul

Como é possível observar na Figura 4, para um número maior de *threads*, os tempos de resposta tendem a diminuir. Isso ocorre pelo fato de que para uma configuração onde existam muitas *threads* ativas, há também uma grande parte que está processando informações, enquanto outra está em estado de espera aguardando o retorno das requisições de I/O, provenientes das chamadas à ferramenta Tokyo Cabinet. Inevitavelmente em algum momento os "papéis" irão se inverter, ou seja, *threads* que antes estavam em estado de espera aguardando o retorno das requisições de I/O irão começar a processar suas informações e vice-versa. Isso faz com o que o tempo médio para cada *thread* diminua, aumentando o desempenho global do sistema.

6 Conclusão

Este trabalho apresentou uma proposta de modelo de arquitetura modular de um sistema para agência de viagens. O modelo anterior, monolítico, apresentava um conjunto de problemas relacionados com o aumento no número de clientes e requisições. Estes problemas estavam relacionados, principalmente, ao consumo excessivo de *threads*, escalabilidade cara do sistema, sobrecarga na base de dados e limitação de recursos por cliente. Todas essas limitações influenciavam diretamente nos tempos de resposta e consequentemente o desempenho global do sistema. Para resolver esses problemas e buscar uma otimização no desempenho da aplicação, foi proposto um novo modelo de arquitetura. Esta nova arquitetura é modular e separa alguns dos serviços utilizados para realizar pesquisas por parte de clientes, *i.e.* gerenciadores de filas, base de dados chave-valor e *cache* distribuída.

Realizous-e uma análise de ferramentas que implementavam os serviços e, após, um conjunto de testes com uma combinação de uso destas ferramentas;

foram escolhidas três ferramentas (ActiveMQ, Memcached e Tokyo Cabinet) para serem utilizadas no novo sistema. Os resultados demonstraram um ganho significativo para o ambiente configurado com APIs Python. Uma das vantagens adquiridas com o novo modelo foi a facilidade de escalar o sistema e gerenciar a utilização dos recursos. No entanto, o ganho mais significativo ficou evidenciado com a utilização do sistema de armazenamento por chave-valor, o que contribuiu diretamente na diminuição dos tempos de resposta da aplicação. Desta forma, o novo modelo de arquitetura possibilitou melhoria significativa ao ambiente, proporcionando efetivamente um aumento no desempenho global do sistema.

7 Agradecimentos

Avelino F. Zorzo possui bolsa de produtividade CNPq. Elder M. Rodrigues possui bolsa de doutorado da CAPES/MEC associada ao Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos. Este trabalho foi realizado em cooperação com a TravelExplorer Software Ltda.

Referências

BROWN, Keith. The .NET Developer's Guide to Windows Security (Microsoft Net Development Series). *Addison Wesley Professional*, 2004.

FLANAGAN, David; MATSUMOTO, Yukihiro. The Ruby Programming Language. *O'Reilly Media*, 2008.

FUSE. Fuse Message Broker. Disponível em: <http://www.fusesource.com>. Acesso em: 12 dez. 2010.

GANDHI, Anshul; HARCHOL-BALTER, Mor; DAS, Rajarshi; LEFURGY, Charles. Optimal Power Allocation in Server Farms. *International Joint Conference on Measurement and Modeling of Computer Systems*, p. 157–168, 2009.

HACKL, Günter; PAUSCH, Wolfgang; SCHÖNHERR, Sebastian; SPECHT, Günter; THIEL, Günter. Synchronous Metadata Management of Large Storage Systems. *International Database Engineering & Applications Symposium*, p. 1–6, 2010.

HENJES, Robert; MENTH, Michael; HIMMLER, Valentin. Impact of Complex Filters on the Message Throughput of the ActiveMQ JMS Server. *Teletraffic Conference on Managing Traffic Performance in Converged Networks*, p. 192–203, 2007.

HORSTMANN, Cay. *Big Java:* Programming and Practice. John Wiley, 2001.

LAMPORT, Leslie. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, v. 21, p. 558–565, Jul. 1978.

LERNER, Reuven. At the Forge: Memcached Integration in Rails. *Linux Journal*, v. 177, p. 25–32, Jan. 2009.

LUTZ, Mark. Programming *Python*. *O'Reilly Media*, 2006.

MICHAEL, Maged; MOREIRA, José; SHILOACH, Doron; WISNIEWSKI, Robert. Scale-up x Scale-out: A Case Study Using Nutch/Lucene. *International IEEE Parallel and Distributed Processing Symposium*, p. 1–8, 2007.

RABBITMQ. RabbitMQ Messaging. Disponível em: <http://www.rabbitmq.com>. Acesso em: 07 de out. 2010.

ROMANOVSKY, Alexander; PERIORELLIS, Panos; ZORZO, Avelino Francisco. Structuring Integrated Web Applications for Fault Tolerance. *International Symposium on Autonomous Decentralized Systems*, p. 99–106, 2003.

SAWYER. et al. The role of laboratory experiments to test marketing strategies. *Journal of Marketing*, v. 43, n. 3, p. 60-67, Summer 1979.

SHARED. Shared Cache. Disponível em: < http://www.sharedcache.com>. Acesso em: 05 de out. 2010.

STROUSTRUP, Bjarne. A History of C++: 1979-1991. *International Conference on History of Programming Languages*, p. 271–297, 1993.

TANENBAUM, Andrew Stuart; STEEN, Maarten van. *Distributed Systems: Principles and Paradigms* (2nd Edition). Prentice Hall, 2006.

TAY, Teng Tiow; FENG, Yanjun; WIJESUNDARA, Malitha. A Distributed Internet Caching System. *IEEE Conference on Local Computer Networks*, p. 624–633, 2000.

WANG, Qiang; TANG, Feilong. A Highly Scalable Key-Value Storage System for Latency Sensitive Applications. *International Conference on Complex, Intelligent and Software Intensive Systems*, p. 537–542, 2009.

WELLMAN, Dan. *jQuery UI 1.7*: The User Interface Library for jQuery. Packt Publishing, 2009.

YAMAMOTO, Junichi; NAKAGAWA, Hiroyuki; NAKAYAMA, Ken; TAHARA, Yasuyuki; OHSUGA, Akihiko. A Context Sharing Message Broker Architecture to Enhance Interoperability in Changeable Environments. *International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, p. 31–39, 2009.

YU, Qi; LIU, Xumin; BOUGUETTAYA, Athman; MEDJAHED, Brahim. Deploying and Managing Web services: Issues, Solutions, and Directions. *The VLDB Journal*, v. 17, p. 537–572, May. 2008.

ZORZO, Avelino Francisco; PERIORELLIS, Panos; ROMANOVSKY, Alexander. Using Co-ordinated Atomic Actions for Building Complex Web Applications: a Learning Experience. *International Workshop on Object-Oriented Real-Time Dependable Systems*, p. 288-295, 2003.