
Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação


# Testes de *Midleware* para Sistema de Agência de Viagens

Leandro Costa, Elder Rodrigues, Filipi Teixeira
Vilmar Consul, Jean Schmidt, Avelino Zorzo

## Relatório Técnico N$^{o}$ 063

Porto Alegre, Novembro de 2010

**Abstract**

This technical report 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 resorces 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, analyse different middleware technologies that could optimize the functionalities of the system. Following the analysis of these technologies, three technologies (ActiveMQ, Memcached and Tokyo Cabinet) were chosen based on the characteristics of the system. These technologies were chosen after a thorough study, throughout a series of performance tests. 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 were performed. In our tests, we used two different APIs written in Python and C#.

# 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) [17], como por exemplo, Chamada de Procedimento Remoto (*Remote Procedure Call* - RPC) [18], Invocação de Método Remoto (*Remote Method Invocation* - RMI) [18], *Web Services* [23], entre outras. Atualmente, existem diversas aplicações desenvolvidas utilizando o paradigma da programação distribuída, neste contexto podem ser citados os sistemas bancários, os sistemas para gerenciamento de redes de telecomunicações, sistemas de informação de grandes empresas, etc.

São muitas as vantagens de se utilizar sistemas distribuídos, uma delas está relacionada à dependabilidade (*dependability*) [15] [24], esta que constitui um conjunto de atributos de qualidade, como por exemplo, disponibilidade, que caracteriza a probabilidade de um sistema funcionar corretamente em um determinado espaço de tempo. Pois 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 [18]. Entretanto, existem problemas de difícil tratamento nesse tipo de sistema, como por exemplo, questões de segurança dos dados, sincronização de eventos de comunicação, congestionamento, estes que são descritos em [10] com um maior nível de detalhes.

Outro problema na utilização deste paradigma verifica-se quando um número excessivo de conexões/requisições são realizadas pelos clientes junto ao(s) servidor(es). A consequência é um alto tempo de processamento dessas requisições, influenciando diretamente no tempo de resposta. Uma solução que aparentemente resolveria esse tipo de questão é aumentar o "poder" computacional das máquinas ou aumentar o número de servidores que hospedam esses sistemas (escalabilidade horizontal (*scale out*) [13]), porém esta alternativa nem sempre é viável, pois o acréscimo de máquinas pode representar um alto custo envolvendo manutenção do *hardware*, custos energéticos, entre outros, ou seja, os benefícios podem não compensar o investimento.

Neste trabalho será apresentado um projeto desenvolvido pela Facin/PUCRS em colaboração com uma empresa de agência de viagens, esta que enfrentava os problemas dos altos tempos de resposta descritos anteriormente, além de limitações de escalabilidade do sistema. Atualmente, o modelo da arquitetura do sis-

tema utilizado por eles possui camadas pertencentes ao mesmo projeto, por este motivo, todo e qualquer processo de manutenção no sistema torna-se oneroso.

Com o intuito de contornar estes problemas e aproveitar melhor os recursos de *hardware* existentes (escalabilidade vertical (*scale up*) [13]), é proposto um novo modelo de arquitetura que tem por objetivo contornar estas questões. Para isso, foram pesquisadas ferramentas com tecnologias para agregar e otimizar o desempenho global do sistema. As tecnologias pesquisadas para solucionar esses problemas foram as seguintes: Gerenciadores de filas de Mensagem (*Message Broker*), Sistemas de banco de dados chave valor (*Distributed Key-value store systems*) e Sistemas de *cache* distribuído (*Distributed cache systems*).

Para melhor compreensão do problema e das melhorias propostas, o trabalho está estruturado da seguinte forma. A Seção 2 apresenta três dos gerenciadores de filas de mensagens existentes no mercado, serão descritas suas funcionalidades e aplicabilidade em um ambiente distribuído. Na Seção 3 são descritas as funcionalidades e conceitos de dois sistemas de banco de dados chave valor, em seguida, são apresentadas as características dos sistemas de *cache* distribuído na Seção 4, para esta última duas ferramentas foram analisadas. Na Seção 5 é apresentado um exemplo de uso, relatando os problemas deste sistema de agência de viagens. A seção seguinte apresenta este mesmo exemplo de uso, porém com uma proposta de modelo de arquitetura onde as ferramentas citadas são utilizadas para otimizar o desempenho do sistema. A Seção 7 apresenta a descrição do ambiente de teste onde as tecnologias analisadas para sanar estas questões são utilizadas. Na Seção 8 são apresentados os resultados seguidos das conclusões realizadas na Seção 9.

# 2 Gerenciadores de filas de mensagens

Um gerenciador de filas de mensagem (*message broker*) é um programa que traduz mensagens de um protocolo formal de um *sender* para um protocolo formal de um *receiver* em uma rede, em outras palavras ele faz o intermédio da comunicação entre aplicações. Um gerenciador de filas de mensagem pode ser visto como um conjunto de filas (FIFO) onde as mensagens são armazenadas nessas filas e enviadas de acordo com a ordem de chegada [22]. O objetivo deste 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 que

endereçam um servidor que dispõe de maior quantidades de recursos ociosos e subutilizados e com isso, aproveitar de forma mais eficiente os recursos do ambiente.

De posse do conhecimento deste tipo de ferramenta foi realizado, inicialmente, uma pesquisa por várias dessas ferramentas, com base em características específicas da arquitetura. Após este estudo, foram definidas três ferramentas *message brokers* para serem analisadas e testadas. As características e funcionalidades de cada uma dessas ferramentas são descritas a seguir.

- **ActiveMQ:** O ActiveMQ [8] 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). Ele é um sistema de comunicação entre processos que suporta clientes em várias linguagens, entre elas Java [9], C++ [16], Ruby [5], Python [12]. Além disso, o ActiveMQ tem muitas *features* avançadas, ele suporta JMS 1.1 e J2EE 1.4, fornece recursos como *clustering* e armazenamento de múltiplas mensagens;

- **Fuse Message Broker:** O FUSE Message Broker [1] é uma plataforma *open source* para comunicação de processos através de Serviços de Mensagem Java (*Java Message System* - JMS). Ele fornece escalabilidade e uma infra-estrutura 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:** O RabbitMQ [3] é um sistema gerenciador de filas de Mensagem (*Message Broker*). Ele é uma implementação *open source* de serviços de comunicação (*messaging*) e para isso utiliza o protocolo *Advanced Message Queueing Protocol* (AMQP). O protocolo AMQP foi projetado ao longo de dois anos e terminado no segundo semestre de 2006. O protocolo define um conjunto de normas e especificações para interoperabilidade de serviços de *messaging* (mensagem instantânea). 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) servem como exemplos.

# 3 Sistemas de banco de dados chave valor

Base de dados chave valor são os únicos sistemas eficientes em 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 a identificação de um valor e um valor consiste em um campo de dados [20].

De posse do conhecimento deste tipo de ferramenta foi realizado, inicialmente, uma pesquisa por várias dessas ferramentas, com base em características específicas da arquitetura. Após este estudo, foram definidos dois sistemas de banco de dados chave valor para serem analisados e testados. As características e funcionalidades de cada uma dessas ferramentas são descritas a seguir.

- **MemcacheDB:** O MemcacheDB [7] 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, 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:** O Tokyo Cabinet [7] é uma biblioteca de rotinas para gerenciar bases de dados (chave valor) disponível para 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 *hash tables*, *B+ tree* ou em *arrays* de tamanho fixo. No endereço do desenvolvedor, http://1978th.net/, estão disponíveis as versões mais atuais do Tokyo Cabinet, inclusive ferramentas adicionais como o Tokyo Tyrant que prove suporte distribuído em rede do Tokyo Cabinet.

# 4 Sistemas de *cache* distribuído

Sistemas de *cache* distribuído são ferramentas utilizadas para aliviar a carga em sistemas que fazem muito acesso 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 [19].

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.

De posse do conhecimento deste tipo de ferramenta foi realizado, inicialmente, uma pesquisa por várias dessas ferramentas, com base em características específicas da arquitetura.

Após este estudo, foram definidos dois sistemas de *cache* distribuído para serem analisados e testados. As características e funcionalidades de cada uma dessas ferramentas são descritas a seguir.

- **Memcached:** O Memcached [11] 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 esse texto 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. Assim como em um banco de dados, é necessário informar ao Memcached o endereço de armazenamento, por padrão é definido o endereço "localhost:1121", mas este parâmetro pode ser configurado.

- **Shared Cache:** O Shared Cache [2] fornece topologias de *cache* distribuído e replicado, foi desenvolvido para aplicações Microsoft .NET [4] que executam sobre *Server farms* [6], onde 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 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* [2].

# 5 Modelo de arquitetura atual

Nesta Seção será apresentado um exemplo de uso da arquitetura de um sistema de agência de viagens, pertencente a uma empresa situada no Parque Tecnológico da PUCRS. O sistema utilizado por esta empresa é denominado Hotel Suite (HS) e tem como objetivo permitir que clientes possam efetuar a reserva em qualquer hotel que pertença a 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.

A realização desta busca/pesquisa ocorre de acordo com o que é descrito na Figura 1 a seguir, onde inicialmente, os Clientes$_{(1)}$ conectam-se ao HS$_{(2)}$ através de um navegador *web* qualquer e realizam uma pesquisa por hotéis que disponibilizem das características informadas 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 estas 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$_{(1)}$.

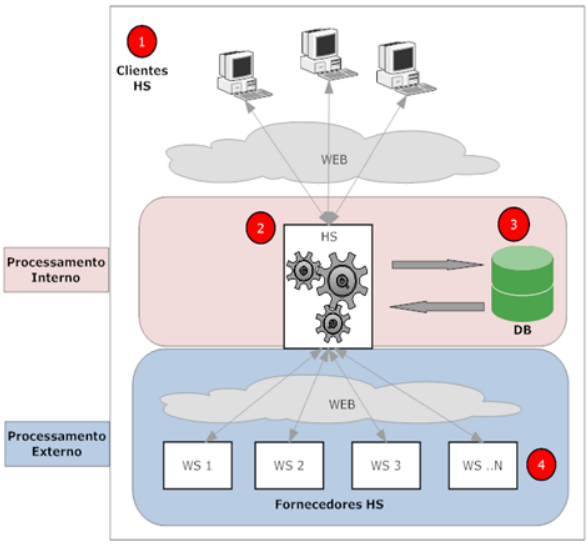


Figure 1: Visão geral da arquitetura HS

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 *flow* completo de pesquisas e reservas. Existem ainda, outras desvantagens neste modelo de arquitetura, por exemplo, não existe controle de escala para solicitações entrantes no HS. Também não há medidores internos de desempenho entre as camadas do sistema e um dos maiores problemas está relacionado ao pouco uso de *cache* de dados, a consequência é uma sobrecarga na base de dados.

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, é apresentado na Tabela 1, 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 (PPP).

Table 1: Perfil de utilização do sistema HS

| | | | Threads por pesquisa | |
|---|---|---|---|---|
| NomeCidade | 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 |
| Cancun | MX | 1,00% | 7,7 | 5,5 |
| | | Totais | 972,95 | 543,4 |
| | | PPP | 50 | |

Como é possível observar, as *Threads* por pesquisa estão divididas em dois grupos, *Threads* Internas e *Threads* Externas. O primeiro grupo trata-se das *threads* responsáveis pelo processamento interno do HS, este que executa tarefas

de pesquisas e persistência dos resultados. O segundo grupo de *threads* é responsável pelo processamento externo do HS, executado durante a comunicação com os fornecedores. Como pode-se observar, o consumo total de *threads* externas e internas é igual a 972,95 e 543,4 respectivamente, consumo este considerado excessivo para a execução do sistema. Para efeito de análise, é apresentado na Figura 3 um gráfico dos percentuais de *threads* internas e externas do sistema HS.

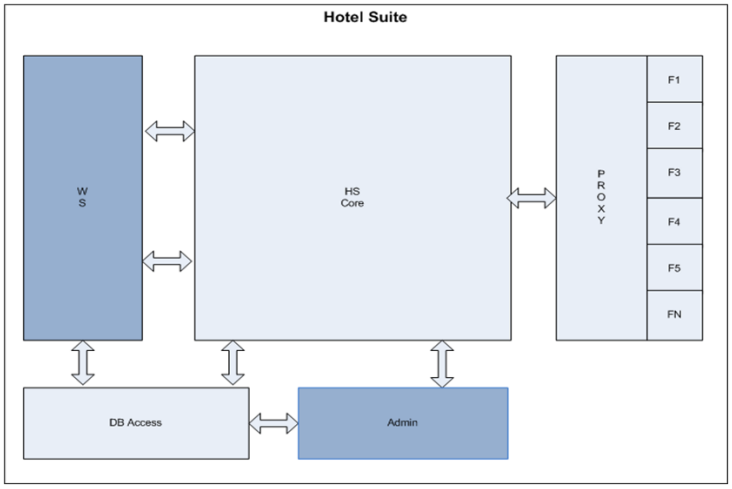


Figure 2: Primeiro modelo de arquitetura do HS

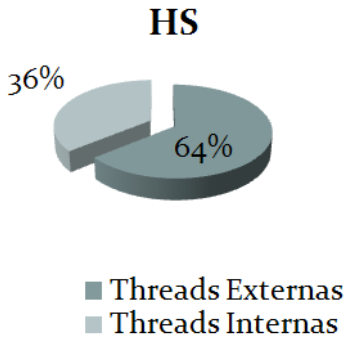


Figure 3: Percentual de *threads* internas e externas do sistema HS

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:** Como pôde-se observar, 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 além de manutenção dobrada no caso de atualizações do sistema (publicação HS).

- **Dependência de DB:** A variação do volume PPP, torna o banco de dados um gargalo natural devido o custo transacional de persistências de pesquisas, ocasionando *timeouts*.

- **Limitação de recursos por cliente:** Impossibilidade de limitar ou dedicar recursos de infra-estrutura por cliente via *software* (HS), pois não é possível restringir o volume PPP individualmente, o que pode causar instabilidades.

Com o intuito de sanar os problemas desta arquitetura foi proposto um novo modelo onde as camadas do sistema não são integradas e as tecnologias apresentadas anteriormente (Seções 2, 3 e 4) são utilizadas para otimizar o desempenho do sistema. Na seção seguinte é apresentada de forma detalhada a descrição deste novo modelo.

# 6 Modelo de arquitetura proposto

Baseado na descrição dos problemas apresentados anteriormente foi proposto um novo modelo de arquitetura do sistema HS. Para esta nova versão optou-se por utilizar algumas tecnologias com o intuito de sanar os problemas encontrados no modelo de arquitetura atual. As tecnologias para contornar os problemas enfrentados são as seguintes: Gerenciadores de filas de mensagens, Sistemas de banco de dados chave valor e Sistemas de *cache* distribuído. A seguir é apresentado o ganho pretendido com a utilização de cada um desses sistemas:

- **Gerenciadores de filas de mensagens:** Estes 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 criar filas de processamento para cada cliente ou fornecedor, por exemplo, com limitações de processamentos simultâneos (PPP) e distribuição de

*threads* por várias máquinas. Com isso, ganha-se escala e gerenciamento de utilização de recursos.

- **Sistemas de banco de dados chave valor:** São sistemas de armazenamento de dados simplificados, sem estrutura de tabelas e sem administração, porém muito performáticos, pois operações de gravação e recuperação de informações são realizadas de forma mais rápida. O Objetivo principal é a gravação e auxílio no processamento dos resultados de busca retornados pelos fornecedores e pelo HS. Com isso, o DB (SqlServer) deixa de ser utilizado para esse objetivo, ganhando desempenho e diminuindo os tempos de resposta da aplicação.

- **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 instantânea. Esse sistema pode ser empregado na criação de aplicações *web* 2.0, utilizando Jquery [21] 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 de todos os fornecedores, tornando mais rápido o processo de pesquisa e recuperação dos dados.

Como apresentado nas seções anteriores, para cada uma dessas tecnologias foi pesquisado e testado um conjunto de ferramentas. Sendo que, as ferramentas analisadas foram: Os gerenciadores de filas de mensagens ActiveMQ, Fuse Message Broker e RabbitMQ; Os sistemas de banco de dados chave valor MemcacheDB e Tokyo Cabinet; Os sistemas de *cache* distribuído Memcached e Shared Cache.

Após um processo de análise e uma série de testes executados, três ferramentas foram escolhidas para otimizar o sistema, são elas: ActiveMQ, Tokyo Cabinet e Memcached. Desta forma, o modelo de arquitetura proposto agregando as tecnologias analisadas pode ser visto na Figura 4.

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 persistência dos dados mais rápido.

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 seguinte.

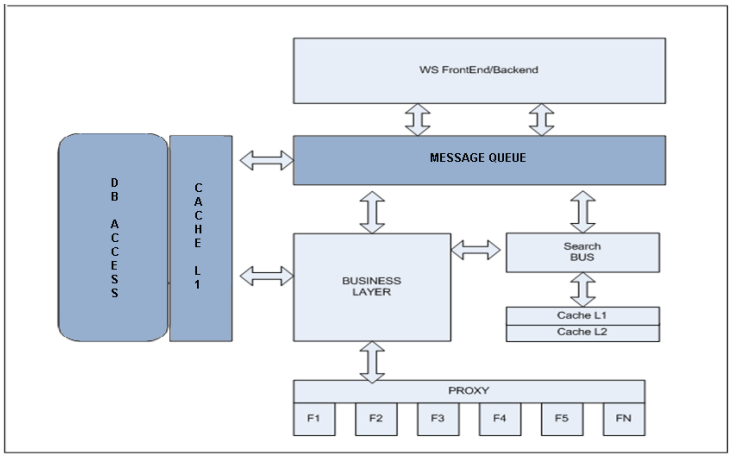


Figure 4: Modelo de arquitetura proposto para o sistema HS

# 7 Etapa de testes

Nesta etapa, dois tipos de testes foram executados. O primeiro consistiu na execução de testes do sistema utilizando APIs na linguagem de programação Python [12] para as ferramentas. O segundo tipo de teste foi semelhante ao primeiro, para este a diferença foi a utilização de APIs em C# [14].

## 7.1 Ambiente configurado utilizando a API Python

Para execução dos testes do sistema duas máquinas foram utilizadas (ver Figura 5). Como é possível observar na máquina A é onde são executados os serviços do ActiveMQ, Memcached e Tokyo Cabinet. Na máquina B é executado o sistema HS. Para a simulação do ambiente foram utilizadas mensagens de requisição de 1 KB e mensagens de 4 KB contendo as informações referentes aos hotéis.

O objetivo do teste é simular usuários realizando diferentes tipos de pesquisas por hotéis e analisar o tempo de resposta global do sistema, ou seja, medir o tempo médio de resposta do sistema para atender cada usuário. O ambiente foi testado, inicialmente, com 15 usuários fazendo 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. Os resultados dos testes para o ambiente são apresentados na seção seguinte.

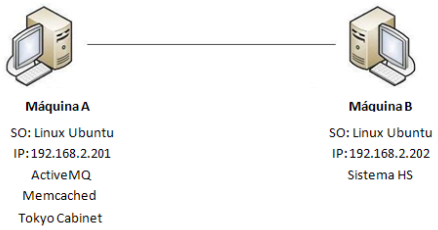


Figure 5: Informações do ambiente de teste do sistema

## 7.2 Testes do ambiente utilizando a API C#

Diferentemente do ambiente anterior, três máquinas foram utilizadas para a execução dos testes do sistema (ver Figura 6). Como é possível observar na máquina A o sistema HS é executado, na máquina B é executado o serviço do ActiveMQ e na máquina C são executados os serviços do Memcached e Tokyo Cabinet. Para a simulação do ambiente foram utilizadas mensagens de requisição de 11 KB e mensagens de 16 KB contendo as informações referentes aos hotéis.

O objetivo do teste é simular usuários realizando diferentes tipos de pesquisas por hotéis e analisar o tempo de resposta global do sistema, ou seja, medir o tempo médio de resposta do sistema para atender cada usuário. O ambiente foi testado, inicialmente, com 5 usuários fazendo requisições simultâneas ao sistema. Em seguida, o mesmo teste foi realizado com 15, 25 e 35 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. Os resultados dos testes para o ambiente são apresentados na seção seguinte.

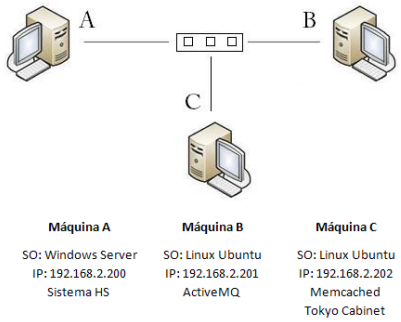


Figure 6: Informações do ambiente de teste do sistema

# 8 Análise dos resultados

Nesta seção são apresentados os resultados dos testes utilizando as duas APIs descritas anteriormente, com os seguintes *midlewares*: ActiveMQ, Tokyo Cabinet e Memcached.

## 8.1 Resultados dos testes utilizando a API Python

Os resultados obtidos para os testes utilizando APIs Python para as ferramentas foram satisfatórios, uma vez que os tempos de resposta foram inferiores a 1 segundo para todos os testes executados (ver Tabela 2).

Table 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 |

Na Tabela 2 é possível observar os valores obtidos da média dos vinte e nove testes realizados sobre a estrutura. Também foram feitos cálculos estatísticos que mostram com exatidão o comportamento dos resultados. Para uma melhor visualização da evolução do comportamento dos testes na estrutura, optou-se por mostrar graficamente alguns valores referentes aos testes (ver Figura 7).

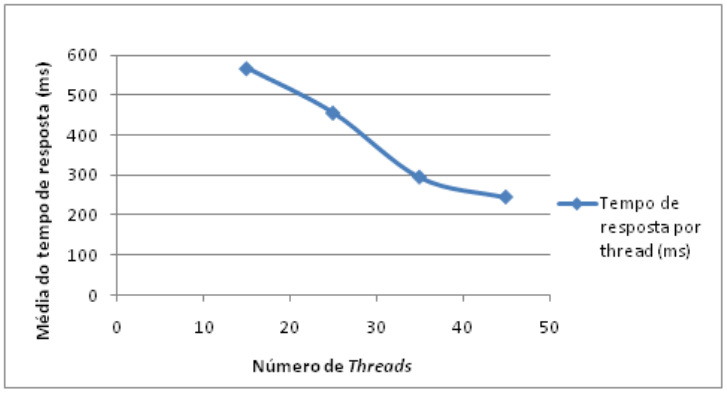


Figure 7: Tempo de resposta por usuário (*thread*)

Como é possível observar na Figura 7, para um número maior de *threads*, os tempos de resposta tendem a diminuir. Isso ocorre pelo fato de que para uma configuração onde existem muitas *threads* ativas, há também uma grande parte que está processando informações enquanto uma outra está em estado de espera aguardando o retorno das requisições de I/O, provenientes das chamadas à ferramenta de chave valor. 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.

## 8.2 Resultados dos testes utilizando a API C#

Os resultados dos testes utilizando APIs C# para as ferramentas não foram bons, se comparado com os testes realizados para o ambiente anterior em Python. Isso se deve ao fato de que a configuração/programação do ActiveMQ utilizando esta API não foi realizada de maneira eficaz, o que influenciou diretamente nos resultados e desempenho do sistema. Isso se comprova analisando os tempos de resposta na Tabela 3, onde os tempos subiram para cada teste, sendo superior a 3 segundos para o último.

Table 3: Tempo de resposta médio por usuário (*thread*)

| Nº de *Threads* | Média (ms) | Desvio Padrão | Mediana | Moda | Variância |
|---|---|---|---|---|---|
| 5 | 609 | 53,1 | 593 | 578 | 2827,9 |
| 15 | 1427,8 | 214,2 | 1359 | 1359 | 45902,7 |
| 25 | 2575,5 | 900,1 | 2375 | 2281 | 810262,3 |
| 35 | 3607,4 | 469,7 | 3609 | 3203 | 220657,4 |

Na Tabela 3 é possível observar os valores obtidos da média dos vinte e nove testes realizados sobre a estrutura. Também foram feitos cálculos estatísticos que mostram com exatidão o comportamento dos resultados. Para uma melhor visualização da evolução do comportamento dos testes na estrutura, optou-se por mostrar graficamente alguns valores referentes aos testes (ver Figura 8).

Como é possível observar na Figura 8, foi obtido um comportamento praticamente linear para os testes, diferente do comportamento obtido nos testes anteriores no ambiente em Python, onde o tempo diminuia conforme se aumentava o número de *threads*. Isso se deve aos problemas encontrados na API do ActiveMQ.

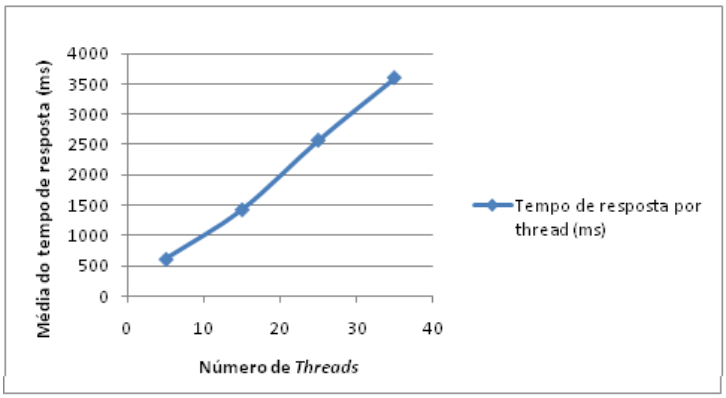


Figure 8: Tempo de resposta por usuário (*thread*)

Embora, o ambiente tenha sido configurado para ser executado em *multithread*, os problemas com a API do ActiveMQ fazem com que este teste deva ser visto como se fosse atendido um cliente por vez, ou seja não houve paralelismo durante a execução dos testes, por isso os tempos de resposta foram elevados.

Para finalizar o trabalho, será apresentada na seção seguinte uma síntese com uma análise conclusiva do desenvolvimento e ganhos obtidos com o modelo proposto para o sistema.

# 9 Conclusão

Este trabalho apresentou uma proposta de modelo da arquitetura de um sistema para uma empresa de agência de viagens, esta que enfrentava muitos problemas para a execução de seu sistema, utilizando o modelo de arquitetura atual. Estes problemas são relacionados, principalmente, ao consumo de *threads* excessivo, escalabilidade cara do sistema, sobrecarga na base de dados e limitação de recursos por cliente. Todas essas limitações influenciam diretamente nos tempos de resposta e consequentemente o desempenho global do sistema é afetado. Para o emprego da proposta deste novo modelo, cujo objetivo é sanar as questões apresentadas, destacam-se 3 etapas como sendo fundamentais para a efetivação do trabalho. A primeira consistiu na busca e pesquisa por tecnologias para agregar e otimizar as funcionalidades do sistema HS. Na segunda etapa foi realizada uma análise destas tecnologias e em seguida três ferramentas (ActiveMQ, Memcached e Tokyo Cabinet) foram escolhidas com base nas características do sistema. A

terceira foi a etapa de testes do novo modelo de arquitetura, onde o objetivo foi validar o modelo proposto. Para os testes dois tipos de APIs foram utilizadas, APIs Python e C#.

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 foi evidenciado com a utilização do sistema de armazenamento por chave valor, este que contribuiu diretamente na diminuição dos tempos de resposta da aplicação. Apesar dos resultados dos testes para o ambiente configurado com APIs C# não terem sido satisfatórios se comparado com o ambiente configurado em Python, eles serviram como base para experimentos futuros, uma vez que a empresa tem preferência pelo ambiente utilizando esta API.

# 10 Agradecimentos

Avelino F. Zorzo possui bolsa de produtividade CNPQ/Brasil e CAPES/PROCAD. Elder M. Rodrigues possui bolsa CAPES e é pesquisador integrante do INCT-SEC. Este trabalho foi parcialmente financiado pela Travel Explorer Web Software Ltda.

# References

[1] Fuse message broker, disponível em http://www.fusesource.com/products/enterprise-activemq/#documentation, acessado em 05/10/2010.

[2] Shared cache, disponível em http://www.sharedcache.com/cms/, acessado em 05/10/2010.

[3] Rabbitmq, disponível em http://www.rabbitmq.com/documentation.html, acessado em 07/10/2010.

[4] K. Brown. *The .NET Developer's Guide to Windows Security (Microsoft Net Development Series)*. Addison-Wesley Professional, 2004.

[5] D. Flanagan and Y. Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[6] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[7] G. Hackl, W. Pausch, S. Schönherr, G. Specht, and G. Thiel. Synchronous metadata management of large storage systems. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, IDEAS '10, pages 1–6, New York, NY, USA, 2010. ACM.

[8] R. Henjes, M. Menth, and V. Himmler. Impact of complex filters on the message throughput of the activemq jms server. In *ITC20'07: Proceedings of the 20th international teletraffic conference on Managing traffic performance in converged networks*, pages 192–203, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] C. S. Horstmann. *Big Java: Programming and Practice*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[11] R. M. Lerner. At the forge: memcached integration in rails. *Linux J.*, 2009, January 2009.

[12] M. Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[13] M. M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *IPDPS*, pages 1–8. IEEE, 2007.

[14] T. Nash. *Accelerated C# 2010*. Apress, Berkely, CA, USA, 1st edition, 2010.

[15] A. Romanovsky, P. Periorellis, and A. Zorzo. Structuring integrated web applications for fault tolerance. In *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*, pages 99 – 106, 2003.

[16] B. Stroustrup. A history of c++: 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, New York, NY, USA, 1993. ACM.

[17] A. Tanenbaum. *Distributed operating systems*. Prentice Hall, 1995.

[18] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[19] T. T. Tay, Y. Feng, and M. N. Wijeysundera. A distributed internet caching system. In *Proceedings of the 25th Annual IEEE Conference on Local Computer Networks*, LCN '00, pages 624–, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Q. Wang and F. Tang. A highly scalable key-value storage system for latency sensitive applications. *Complex, Intelligent and Software Intensive Systems, International Conference*, 0:537–542, 2009.

[21] D. Wellman. *jQuery UI 1.7: The User Interface Library for jQuery*. Packt Publishing, 2009.

[22] J. Yamamoto, H. Nakagawa, K. Nakayama, Y. Tahara, and A. Ohsuga. A context sharing message broker architecture to enhance interoperability in changeable environments. *Mobile Ubiquitous Computing, Systems, Services and Technologies, International Conference on*, 0:31–39, 2009.

[23] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17(3):537–572, 2008.

[24] A. Zorzo, P. Periorellis, and A. Romanovsky. Using co-ordinated atomic actions for building complex web applications: A learning experience. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:288, 2003.