



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA
PARAÍBA**

Coordenação do Curso Superior de Tecnologia em Sistemas para Internet

Davi Cavalcanti Pinto Bandeira

RELATÓRIO DE ESTÁGIO

***CODI4In Cache Service: Um serviço de gerenciamento de *cache* para o plugin
CODI4In***

João Pessoa – PB

Março/2013

Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Unidade Acadêmica de Informação e Comunicação
Coordenação do CST em Sistemas para Internet

CODI4In Cache Service: Um serviço de gerenciamento de cache para o plugin CODI4In

Davi Cavalcanti Pinto Bandeira

Relatório de Estágio Supervisionado
apresentado à disciplina Estágio
Supervisionado da Coordenação do Curso
Superior em Sistemas para Internet do
Instituto Federal de Educação, Ciência e
Tecnologia da Paraíba como requisito parcial
para obtenção do grau de Tecnólogo em
Sistemas para Internet.

Orientador: Profa. Dra. Damires Yluska de Souza Fernandes

Supervisor: Profa. Dra. Damires Yluska de Souza Fernandes

Coordenador do Curso de Sistemas para Internet: Thiago José Marques Moura

Empresa: Unidade Acadêmica de Informática

Período: 20/11/2012 a 14/03/2013

João Pessoa
2013

Prof^a. Dra. Damires Yluska de Souza Fernandes
Orientadora

Prof^a. Dra. Damires Yluska de Souza Fernandes
Supervisora

Davi Cavalcanti Pinto Bandeira
Aluno de Sistemas para Internet

À minha família e em especial a minha
companheira Rosalía Ramírez.

.

AGRADECIMENTOS

Agradeço à minha família que sempre investiu na minha educação e me apoiou na escolha do curso - em especial quando optei por este em detrimento de outro -, sempre confiando no meu senso de julgamento.

Agradeço à minha companheira Rosalía que pacientemente esteve me apoiando e me dando forças para avançar com o desenvolvimento deste trabalho.

A meus professores do IFPB que, de forma exemplar, não somente foram fonte de conhecimento mas companheiros no processo de construção da aprendizagem. Em especial faço menção da professora Damires Yluska que me introduziu na pesquisa acadêmica e manteve um incentivo constante para que não desanimássemos.

Aos meus colegas de turma que fizeram as centenas de horas gastas nesta casa serem as mais proveitosas possíveis, trazendo, muitas vezes, riso quando o momento era de dor.

Acima de tudo e de todos, agradeço a Deus, por ter me conduzido a passos largos durante esta trajetória, dando-me mais do que pedi ou pensei.

RESUMO

A especificação e implementação de um serviço de gerenciamento de cache que faça representação do grafo contextual de usuários, cujo contexto é gerenciado pelo CODI4In, e que aumente a performance das aplicações que se beneficiam dele é tema deste relatório de pesquisa. O objetivo do CODI4In é gerenciar, de forma dinâmica e incremental, os elementos contextuais de usuários de aplicações diversas, transferindo para si esta responsabilidade, de maneira a permitir que as aplicações foquem nos serviços que se propõem a oferecer. Entretanto, vislumbra-se um aumento no tempo de latência no servidor do *plugin* gerado por este gerenciamento, afetando negativamente a experiência do usuário através da aplicação. Nesta problemática, apresentamos, neste trabalho, um serviço de gerenciamento de cache que, atuando como uma camada intermediária entre a aplicação e o CODI4In, possa diminuir o tempo de latência gerado entre a requisição feita pelo usuário e a resposta. Foram feitos testes no serviço de cache a fim de avaliar seu desempenho em diversos cenários. Tais resultados serão apresentados neste documento.

Palavras-chave: Gerenciamento de Cache, CODI4In, Contexto do Usuário, Grafo Contextual.

LISTA DE FIGURAS

Figura 1: Diagrama comparativo de personalização de consultas	06
Figura 2: Arquitetura do CODI4In	13
Figura 3: Fragmento da ontologia de contexto CODI-User	14
Figura 4: Exemplo de instanciação da CODI-User	15
Figura 5: Fluxograma das etapas percorridas pelo CODI4In para aquisição das preferencias e personalização de consultas	16
Figura 6: Arquitetura do CODI4In com o Serviço de Cache	19
Figura 7: Interação entre a aplicação, gerenciador de cache e <i>CODI4In</i>	21
Figura 8: Gerenciamento de elementos contextuais	22
Figura 9: Status do processo <i>memcached</i> após inicializado	24
Figura 10: Algoritmo de recuperação de elementos contextuais	25
Figura 11: Preferencias no formato JSON	26
Figura 12: Algoritmo em alto nível da requisição <i>POST</i> para elemento contextual	27
Figura 13: Resposta do serviço de cache a requisição <i>POST</i> para endereço	28
Figura 14: Tarefa cronometrada que realiza sincronismo entre o CODI4In e o serviço de cache	28
Figura 15: Tela inicial da aplicação IndSong	29
Figura 16: Persistência do elemento contextual <i>Address</i> através de um formulário Web, pela aplicação IndSong	30
Figura 17: Formulário com as caixas de seleções para definição dos gêneros musicais preferidos	31
Figura 18: Mensagem de indisponibilidade do serviço e do <i>plugin</i>	31
Figura 19: Lista de álbuns indicados de acordo com as preferencias do usuário	32
Figura 20: Gráfico comparativo dos resultados obtidos nos testes	34

LISTA DE SIGLAS

API	<i>Application Programming Interface</i>
DRAM	<i>Dynamic Random Access Memory</i>
IMDb	<i>Internet Movie Database</i>
HD	<i>Hard Drive</i>
HTTP	<i>Hypertext Transfer Protocol</i>
JSON	<i>JavaScript Object Notation</i>
LFU	<i>Least Frequently Used</i>
LRU	<i>Least Recently Used</i>
RAM	<i>Random Access Memory</i>
REST	<i>Representational State Transfer</i>
SRAM	<i>Static Random Access Memory</i>
TCP	<i>Transmission Control Protocol</i>
UDP	<i>User Datagram Protocol</i>
URI	<i>Uniform Resource Identifier</i>

SUMÁRIO

1. Introdução	1
1.1 Motivação	1
1.2 Objetivos.....	2
1.2.1 Objetivos específicos	2
1.3 Organização do Relatório	2
2. Embasamento Teórico	4
2.1 Contexto.....	4
2.2 Personalização de Consultas.....	5
2.3 Gerenciamento e uso de cache.....	7
2.4 Trabalhos relacionados	10
3. O CODI4In	13
3.1 O <i>Plugin</i> CODI4In.....	13
3.2 Ontologia de Contexto CODI-User	14
3.3 Estudo de Caso: A Aplicação MovieShow.....	15
4. O Serviço de Cache: Implementação e Resultados	18
4.1 Objetivo do Serviço de Cache	18
4.2 Requisitos do Gerenciador de Cache	19
4.3 A Aplicação IndSong.....	20
4.4 Especificação do Serviço de Gerenciamento de Cache para o CODI4In	20
4.5 Implementação.....	22
4.5.1 O Serviço de Cache.....	23
4.5.2 Estudo de Caso: A Aplicação IndSong	29
4.6 Testes	32
4.6.1 <i>Baseline</i> dos testes com <i>cache off</i>	32
4.6.2 Execução dos testes com <i>cache on</i>	33
5. Considerações Finais	35
5.1 Contribuições.....	35
5.2 Dificuldades encontradas.....	35
5.3 Trabalhos futuros	36
Referências	37
Apêndices	40

1. Introdução

Este relatório tece as atividades realizadas durante o estudo e implementação de um serviço de gerenciamento de cache para o *plugin* CODI4In – objeto de trabalho de projeto de pesquisa de Iniciação Científica ocorrido no IFPB – Campus João Pessoa. O projeto teve início em outubro de 2012 e foi concluído em março de 2013. O presente capítulo descreve os objetivos do projeto, introduz as atividades desenvolvidas e a organização de todo o relatório.

1.1 Motivação

No atual cenário da *Web*, em que a quantidade de informação a ser tratada representa um desafio, as aplicações têm requerido esforço e tempo para tirarem melhor proveito das especificidades de cada usuário, de maneira a prover conteúdo relevante. Nesta problemática criou-se uma ferramenta na forma de *plugin*, denominado CODI4In que, fazendo uso de informações que envolvem e compõem o cenário em que este usuário atua – ao que chamamos de contexto - armazena e recupera seus perfis [Tanca *et al.*, 2011; Levandoski e Khalefa, 2011; Vieira *et al.*, 2010; Souza *et al.*, 2008].

O objetivo do CODI4In é gerenciar, de forma dinâmica e incremental, os elementos contextuais de usuários de aplicações diversas, transferindo para si esta responsabilidade, de maneira a permitir que as aplicações foquem nos serviços que se propõem a oferecer.

Prevendo o aumento da quantidade de usuários acessando serviços que façam uso do *plugin* CODI4In [Freitas *et al.*, 2012] para armazenar e recuperar informações contextuais relevantes ao usuário em determinado ambiente, deslumbra-se, inevitavelmente, um aumento no tempo de latência no servidor do *plugin* gerado por estas operações e, portanto, o serviço que as geraram sofrerá, negativamente, o reflexo desta latência.

A problemática se encontra no fato de que o *plugin* deve recuperar o grafo contextual do usuário sempre que este fizer uma requisição. No grafo, conceitos são identificados como nós e os relacionamentos são identificados como arestas [Bowen, 2000], determinando assim um conjunto de elementos contextuais do usuário. Uma vez carregado, este grafo é transmitido para o serviço (por exemplo, uma aplicação web de consulta a dados) que fará o uso do mesmo. Imaginemos, então, o seguinte cenário: o usuário João acessa uma aplicação *Web* pela primeira vez ao realizar uma operação de *login*. Sendo validado com sucesso, o *plugin* fará sua primeira carga e replicará o grafo contextual gerado para a aplicação que fez a solicitação. João começa a interagir com a aplicação e, neste momento, altera alguns de seus dados pessoais (por exemplo, seu *expertise*) que podem ser identificados como elementos contextuais. O CODI4In pode, por

outro lado, também identificar outros elementos contextuais associados àquele usuário e expandir seu grafo contextual. Uma nova requisição é gerada para o *plugin* que, ao carregar o grafo contextual novamente, fará a devida alteração ou expansão. Este ciclo continua, por exemplo, por mais 5 ou 10 interações, seja acrescentando mais elementos contextuais, seja alterando ou removendo. A quantidade de requisições e o tempo de resposta delas pode proporcionar uma experiência frustrante ao usuário que está usando a aplicação. Este cenário é agravado ainda mais se imaginarmos uma quantidade significativa de requisições simultâneas sendo feita ao servidor do *plugin*, que, em algum momento pode parar de responder, tornando inviável a utilização da aplicação.

Neste panorama, este trabalho aborda a definição e implantação de um serviço de cache que faça representação e gerenciamento adequados do modelo de dados (o grafo contextual) provido pelo *plugin* CODI4In e que, trabalhando como uma camada intermediária entre a aplicação e o *plugin*, possa diminuir o tempo de latência entre a requisição feita pelo usuário e a resposta.

1.2 Objetivos

A ideia central deste trabalho foi especificar, desenvolver e testar o serviço de gerenciamento de cache que faça representação do grafo contextual gerado pelo CODI4In.

1.2.1 Objetivos específicos

Os objetivos específicos do presente projeto foram:

- Estudo de ferramentas de gerenciamento de cache;
- Análise das diferentes formas de implementação e de trabalhos relacionados;
- Especificação e desenvolvimento do serviço integrado ao *plugin* CODI4In;
- Realização de testes e documentação destes;
- Análise dos resultados obtidos, comparando estes com o cenário de não uso de cache.

1.3 Organização do Relatório

Além deste capítulo, este relatório está dividido como segue:

- No Capítulo 2, é descrito o referencial teórico, juntamente com alguns trabalhos relacionados a este.
- No Capítulo 3, é apresentado o *plugin* CODI4In, mostrando suas funcionalidades e uma aplicação desenvolvida antes do presente trabalho .

- No Capítulo 4, é apresentada a especificação e a implementação do serviço de cache e da aplicação *IndSong*, na qual elementos contextuais são capturados e enviados para o serviço, a fim de aumentar o desempenho do CODI4In. Neste capítulo também são mostrados os testes realizados.
- Finalmente, no Capítulo 5, as contribuições do trabalho são apontadas, dificuldades encontradas são relatadas e possíveis trabalhos futuros são indicados.

2. Embasamento Teórico

Este capítulo aborda o referencial teórico que embasou e que envolve o desenvolvimento de um serviço de cache para o *plugin* CODI4In. Serão apresentados conceitos sobre Contexto e Personalização de Consultas, seguidos de tópicos e conceitos pertinentes ao gerenciamento e uso de cache. Por fim, serão apresentados alguns trabalhos relacionados que utilizam alguma forma de gerenciamento de cache para diminuição de carga em servidores e aumento de desempenho.

2.1 Contexto

Contexto pode ser entendido como o que está por trás da habilidade de definir o que é ou não relevante em um dado momento [Souza *et al.*, 2008; Dey, 2001].

Tomando por base esta definição, imaginemos que um determinado usuário está usando um serviço de recomendação de restaurantes através de seu celular pela manhã. Sua *localização*, através do sistema embutido de *GPS* em seu telefone, é requisitada, seguida de suas preferências por tipos de pratos. Com estas informações computadas, a aplicação retorna uma lista de restaurantes próximos ao usuário. Este conjunto de informações (localização, tipo de dispositivo, preferências, etc.) pode ser compreendido como o *contexto do usuário* naquele momento.

A utilização do contexto do usuário de maneira eficaz pode ser, entretanto, uma dificuldade. Neste âmbito, motores de busca, por exemplo, têm tentado associar estas informações às suas consultas para fornecer resultados mais relevantes [Sagui *et al.*, 2006]. Serviços de recomendação, como os de produtos relacionados oferecidos pelo site Amazon¹, fazem uso de informações coletadas implicitamente para facilitar a utilização de seus serviços pelo usuário. Esta estratégia obtém êxito no momento em que uma consulta em uma base de dados sofre alterações de elementos contextuais para obtenção de resultados que sejam significativos para o usuário. Esta estratégia é normalmente denominada de personalização de consultas [Koutrika, 2004]. Assim, a ideia por trás da personalização da consulta é que diversos usuários, ao realizar uma consulta, possam obter resultados diferentes como resposta, de acordo com um modelo de usuário identificado [Koutrika, 2004].

Dessa maneira, todo o processo de personalização de consultas, incluindo a obtenção e tratamento de preferências, histórico de utilização, captura de informações dinâmicas como localização do usuário pode ser facilitado através do uso de contexto. Entretanto, gerenciar o

¹ <http://www.amazon.com>

contexto implica na implementação de tarefas como aquisição, representação, armazenamento, recuperação e uso do contexto.

Estas tarefas podem ser executadas fazendo uso de ferramentas e tecnologias já maduras e frequentemente usadas por diversos meios. A aquisição, por exemplo, é amplamente explorada em redes sociais, onde o usuário, através de formulários, votações, comentários, etc. explicitamente alimenta uma base de conhecimento com as informações inseridas. Além das técnicas citadas, informações contextuais podem também ser implicitamente capturadas, respeitando a privacidade exigida pelo usuário [Alves *et al.*, 2013].

De forma a padronizar a maneira como os diversos dados contextuais adquiridos devem ser organizados, pode-se utilizar ontologias [Vieira *et al.*, 2010; Souza *et al.*, 2008; Strang e Linnhoff-Popien, 2004]. De acordo com Gruber [1993], uma ontologia é uma especificação explícita de conceitos e relacionamentos que podem existir entre eles. O seu uso tende a facilitar a maneira como o contexto será armazenado, lido e aplicado, além de ajudar também a prover mecanismos de raciocínio.

Consideremos, por exemplo, duas aplicações feitas em idiomas diferentes. As duas utilizam bancos de dados com tabelas semelhantes e oferecem um mesmo serviço de aluguel de carros. Caso as duas resolvam fazer um intercambio de dados, será necessário que o seu modelo de dados seja adaptado e entendido para uso por ambas. Utilizando uma ontologia, neste caso relativa ao domínio das aplicações, ambas as aplicações poderiam adaptar seus modelos de dados para se adequar à representação proposta por ela. Desta forma, ao realizar o aluguel de um carro, a aplicação não teria que se preocupar se a tabela chama-se *aluguel* ou *rental*, pois o termo definido na ontologia seria utilizado para evitar a ambiguidade. Neste trabalho, uma ontologia é utilizada como modelo de representação de elementos contextuais do usuário, como será explicado na Seção 3.2.

2.2 Personalização de Consultas

O dicionário Priberam da Língua Portuguesa define personalização como o ato de tornar pessoal; dar caráter original a um objeto fabricado em série; adaptar às preferências ou necessidades do utilizador [Priberam, 2013]. Nos sistemas computacionais, o termo assume seu papel no momento em que um usuário atribui características suas ou de seu interesse em seu ciclo de atividades. Entretanto, a crescente quantidade de informações apreendidas, seja de maneira implícita ou explícita, pelos sistemas online, têm tornado a tarefa de exibir informações relevantes ao usuário algo complexo. Isto porque usuários diferentes inseridos em contextos diferentes esperam resultados personalizados para consultas iguais, os quais se adequem às suas preferências e objetivos. Koutrika [2004] define personalização de consultas como o processo de

melhorar dinamicamente uma consulta utilizando preferencias do usuário com o objetivo de se obter resultados personalizados. Espera-se, portanto, que estes resultados estejam o mais próximo possível do que o usuário buscava inicialmente e que, informações irrelevantes sejam, possivelmente, eliminadas dos resultados.

Tomemos como exemplo dois usuários chamados Caio e Fábio. Ambos fazem compras online de livros e realizam consultas a respeito destes. Mas, Caio prefere livros sobre culinária enquanto Fábio prefere literatura estrangeira. Baseando-se nas compras anteriores, de onde o sistema extraiu as preferências de cada um, o sistema retornará uma lista de livros relacionados à culinária para o primeiro e literatura estrangeira para o segundo, deixando os livros menos relevantes em últimos lugares ou mesmo removendo-os dos resultados.

A Figura 1 representa de maneira análoga o exemplo aqui citado. Nela, observam-se os dois cenários na utilização de consultas. Em uma consulta tradicional, o motor de busca retorna imediatamente os resultados sem haver processamento analítico durante a consulta. Na consulta personalizada, o motor de busca analisa as compras realizadas anteriormente, identifica as preferências e retorna os resultados que se aproximem mais do perfil de compras do usuário.

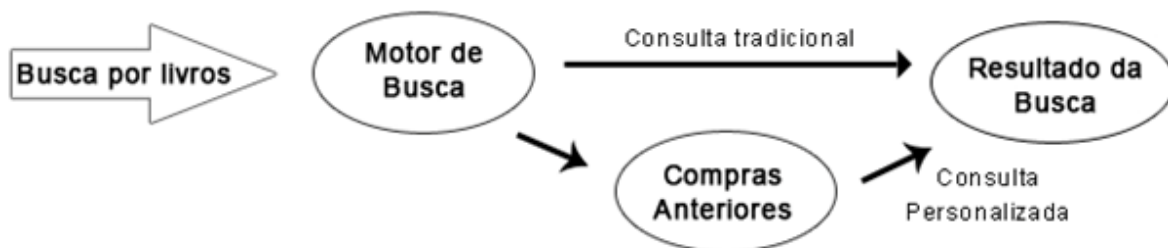


Figura 1. Diagrama comparativo de personalização de consultas.

Neste panorama, Diniz [2010] agrupa a personalização de consultas em três categorias: baseada em perfil, baseada em comportamento e baseada em colaboração. A primeira trata de dados do usuário adquiridos de forma explícita através de questionários e formulários ou de maneira implícita. A segunda faz uso do histórico de operações e resultados obtidos anteriormente. Em nosso exemplo, compras anteriores levariam o sistema a deduzir a preferência do usuário ao gerar novas recomendações. E por fim, a terceira categoria agrupa resultados de consultas que se utilizam de experiências relatadas por outros usuários por meio de votação, comentários, etc.

2.3 Gerenciamento e uso de cache

No início da década de 70, quando os gastos da indústria com software igualaram os gastos com hardware [Dantas, 2013], começou-se um processo de barateamento dos custos de produção e, conseqüentemente de consumo, de memória de longo prazo, os chamados *HDs*. Inicialmente, a quantidade de dados que estes podiam armazenar era significativamente inferior ao que encontramos hoje no mercado. Paralelamente a isto tínhamos processadores e memórias RAM que trabalhavam na faixa dos *Megahertz* e que, já naquela época, excedia em diversas ordens de grandeza o tempo de acesso ao dado dos HDs. Este problema perdura até os dias atuais e é agravado pelo fato de que a tecnologia empregada nos componentes eletrônicos de ambas as memórias continua evoluindo, mas o tempo de acesso entre elas continua aumentando.

Em 1946, VonNeumann, citado pelo professor Martin [2012], apresentou esta problemática da seguinte forma:

“Idealmente, alguém desejaria uma capacidade de memória infinitamente grande, capaz de tornar qualquer palavra imediatamente disponível [...] Nós somos forçados a reconhecer a possibilidade de construção de uma hierarquia de memórias, cada qual com uma capacidade maior que a precedente, mas menos velozmente acessível.”

Visando a diminuição do tempo de espera que um determinado dado é submetido do momento em que recebe o comando para ser lido até o momento em que ele está, de fato, disponível para leitura, foram desenvolvidos algoritmos que aumentassem a disponibilidade de dados [Martin, 2012]: i) que já foram acessados previamente e ii) serão, possivelmente, acessados no futuro. O primeiro é conhecido como princípio da Localidade Temporal, e o segundo, princípio da Localidade Espacial. Estes dois podem ser explicados utilizando o clássico exemplo da biblioteca, como segue.

Imaginemos um determinado aluno chamado Fábio. Ele precisa fazer uma pesquisa e resolve ir à biblioteca estudar livros. Desloca-se até o local após alguns minutos de trânsito e gasta algum tempo buscando entre os corredores e as prateleiras os materiais que lhe interessam. Para evitar que este gasto de tempo se repita num futuro próximo, Fábio resolve levá-los para sua casa e colocá-los em sua mesa. Ele coloca os livros que tem usado recentemente mais próximo de si (Localidade Temporal) e quando necessitar usar outros livros irá pegar todos os livros relacionados de uma única vez (Localidade Espacial).

No exemplo dado, podemos assumir que a biblioteca seja a memória de longo prazo, cujo acesso é mais lento, mas pode conter mais dados. As prateleiras seriam o cache, que tem uma capacidade moderada e tem acesso relativamente rápido, enquanto os livros são os dados sendo usados, com capacidade menor.

Estas duas propriedades aqui citadas, quando bem adotadas e implementadas, objetivam quatro aspectos principais [Smith, 1982]:

- 1) Aumentar a probabilidade de encontrar uma referencia de memória em cache (taxa de *hit*);
- 2) Minimizar o tempo de acesso à informação que já está em cache (tempo de acesso);
- 3) Minimizar o tempo de latência – atraso – devido a uma busca por uma referencia de memória não encontrada, chamada de *miss*; e
- 4) Minimizar o custo de atualização da memória principal, mantendo a consistência, etc.

Algumas questões são, então, levantadas: o que, de fato, deve ser mantido em memória? Por quanto tempo e quais devem ser os critérios para que os dados sejam substituídos por outros?

Para responder a primeira pergunta, devemos levar em consideração o que é relevante para o cenário que se deseja explorar. Retomando o exemplo da biblioteca, observa-se que os dados relevantes para Fábio são os livros que possuem assuntos relacionados à sua pesquisa. Entretanto, a biblioteca pode conter centenas de livros relacionados, tornando inviável o ato de levar todos para casa – seja por uma limitação da própria biblioteca ao permitir um número determinado de empréstimos, ou seja, pelo tamanho da mesa de sua casa (espaço insuficiente em memória). Desta forma, Fábio deve estabelecer um critério de decisão para escolher quais livros devem ser deixados para trás em detrimento de outros. Esta escolha torna-se importante no momento em que a possibilidade de Fábio voltar à biblioteca para trocar os livros representa um problema – devido à indisponibilidade, tempo ou qualquer outro fator. Analogamente, devem-se estabelecer regras que servirão como base na escolha dos dados que serão armazenados em cache. Tais regras refletirão, posteriormente, na velocidade do tempo de resposta de consultas, pois, tendo os dados mais provavelmente utilizáveis disponíveis, evita-se que uma requisição ao servidor de dados (a biblioteca, em nosso exemplo) seja feita.

A segunda questão envolve alguns conceitos que também tem relação com nosso exemplo.

Durante sua pesquisa, Fábio nota que dois livros não foram utilizados durante algum tempo e que um terceiro livro foi usado com pouca frequência. Ele percebe que precisará voltar à biblioteca para trocar alguns livros e, por isso, escolhe estes três livros como primeira opção.

A este conceito de substituição de dados denomina-se *cache replacement*, que abrange, entre diversas estratégias [Zhou *et al.*, 2004], duas aqui já exemplificadas:

1) *Least Recently Used* (LRU): neste modelo, os endereços de memória são organizados em formato de lista, na qual os itens referenciados mais *recentemente* estão no topo e o restante dos itens menos referenciados está organizado de forma decrescente [Smith, 1982]. Quando o cache está cheio, os dados menos recentemente acessados são substituídos. Este modelo faz uso do princípio da Localidade Temporal.

2) *Least Frequently Used* (LFU): de forma análoga ao modelo anterior, este modelo também utiliza uma lista, na qual o topo é composto pelos itens mais *frequentemente* referenciados enquanto no final da lista encontram-se os itens menos acessados, em termos de frequência. Quando o cache está cheio, os dados menos frequentemente acessados são substituídos.

Apesar das duas estratégias acima apresentadas definirem modelos de substituição de dados (*cache replacement*), deve-se ter em mente que o tamanho do cache está diretamente relacionado ao tamanho da lista que manterá os dados em memória, resultando, portanto, na frequência com que os dados terão que ser substituídos. Conforme afirma Smith [1982], quanto maior o tamanho do cache, maior a probabilidade de encontrar a informação necessária. Entretanto, consideram-se três aspectos no momento de estipular o tamanho do cache [Smith, 1982]: custo (geralmente o fator mais importante), tamanho físico (as memórias precisam caber dentro das máquinas) e tempo de acesso (quanto maior o cache, mais lento este pode se tornar).

Os tópicos levantados nesta seção até o momento tinham como foco a leitura do cache. Ao considerarmos a escrita de dados, tomaremos como base as políticas adotadas pela CPU para definir modelos de sincronização de dados com o cache e a memória de longo prazo. Entre elas, duas podem ser enumeradas: *write-through* e *copy/write-back* [Smith, 1982; Teoh, 2010]. A primeira caracteriza-se pelo fato de que, no recebimento do comando para escrita, o dado é escrito tanto na memória cache quanto na memória de longo prazo. No cenário de replicação de dados, este método pode ser comparado ao método *síncrono* [Oracle, 2011], que somente permite que um segundo comando seja executado quando ambas as partes tenham sido devidamente atualizadas [Fuller, 2007].

A segunda funciona da seguinte forma: a memória cache recebe um comando para escrita e a memória de longo prazo somente será atualizada quando este dado for substituído no cache. Este modelo pode ser analogamente comparado ao modelo *assíncrono* de replicação de dados [Oracle, 2011], que, apesar de possuir desempenho superior ao tipo síncrono, deve aceitar alguma inconsistência nos dados.

2.4 Trabalhos relacionados

Ao longo dos anos, o crescimento da indústria computacional propiciou o desenvolvimento de tecnologias de cache que auxiliassem no desempenho de aplicações ao fazerem uso das memórias SRAM e da memória principal (DRAM) para armazenamento de dados [Kuenning, 2010]. Diante disso, soluções para uso e gerenciamento de cache tem encontrado um mercado crescente. Neste cenário, os trabalhos relacionados ao tema deste documento podem ser classificados segundo dois ângulos: *Web Caching* e *Context caching*. A seguir, apresentamos alguns exemplos de cada um, iniciando-se com trabalhos associados à *Web Caching*.

Em 2003, Brad Fitzpatrick, fundador do serviço de *blogs* LiveJournal², desenvolveu uma arquitetura capaz de ser usada e acessada globalmente por diversas máquinas e processos simultaneamente. A esta lhe deu o nome de *memcached* [Fitzpatrick, 2004]. De maneira simplificada, o *memcached* funciona, internamente, como uma *HashTable*, que mantém seus nós ou registros na forma de chave-valor. Esta lista, posteriormente, é pesquisada para encontrar os nós desejados. Sua arquitetura será explicada em mais detalhes na Seção 4.5.1. A seguir, detalharemos alguns serviços que utilizam o *memcached* em larga escala como gerenciador de cache, direta ou indiretamente.

O serviço de micro mensagens Twitter³ foi um serviço que, tendo atingido mais de 200 milhões usuários com um saldo de mais de 140 milhões de *posts* diários [Picard, 2011], adotou o uso de cache para manter operante seu sistema que responde a dois trilhões de consultas por dia (aproximadamente 23 milhões de queries por segundo) [Aniszczyk, 2012]. As duas principais tarefas que o *Twemcache* – como é chamada a versão personalizada para o Twitter do *memcached* – exerce são: 1) otimizar operações em disco, e 2) otimizar os processos que envolvem a CPU, servindo como buffer.

O portal de vídeos Youtube⁴, inicialmente trabalhando com replicação MySQL, viu-se obrigado a adotar uma medida diferente no momento em que todos os seus servidores de bancos de dados já não suportavam mais dados, estando operando em capacidade máxima. Diversas tentativas de utilização de métodos de replicação *RAIDs* foram adotadas, mas o custo com a recuperação dos dados através de múltiplas máquinas impactou negativamente funcionalidades não relacionadas aos vídeos. A solução encontrada foi espalhar fragmentos de dados - *shards* - por vários bancos de dados em diferentes servidores, permitindo as operações de leitura e escrita fazerem melhor uso do serviço de cache oferecido pelo MySQL, reduzindo a carga do *hardware* em cerca de 30% [Cordes, 2007].

² <http://www.livejournal.com>

³ <http://www.twitter.com>

⁴ <http://www.youtube.com>

O serviço colaborativo Wikipedia⁵ funciona através de uma organização sem fins lucrativos chamada *WikiMedia Foundation*, que possui uma arquitetura de 400 servidores x86. Destes, 250 são servidores web e o restante rodam bancos de dados MySQL. De forma a servir de maneira eficiente seus milhões de páginas, que por trás rodam código na linguagem PHP, o qual compila os *scripts* em *bytecode* e descarta-os depois da execução, adotou-se a ferramenta de cache APC feita na própria linguagem. Desta forma, as páginas são compiladas uma única vez, reduzindo drasticamente o tempo de inicialização de aplicações grandes. Além desta medida, as páginas renderizadas são frequentemente armazenadas como dados ou objetos no *memcached* [Gupta, 2008].

Sob o aspecto de *Context caching*, o Facebook⁶, que apesar de se enquadrar nas duas categorias aqui citadas, possui uma atuação mais expressiva nesta devido a um de seus principais recursos – o *Social Graph* [Constine, 2013]. Sendo, possivelmente, o maior usuário de *memcached* da atualidade [Saab, 2008], teve que realizar otimizações ainda maiores em seu sistema de cache para conseguir dar vazão aos 800 servidores em uso com aproximadamente 28 *terabytes* de memória. Isto se tornou possível ao moverem as operações de recuperação (GET) para o protocolo UDP ao invés do protocolo TCP, permitindo um controle de fluxo em nível de aplicação para *multi-gets* (recuperação de centenas de nós em paralelo). Esta mudança propiciou que o escalonamento do *memcached* passasse a manipular, ao invés de 50 mil, 200 mil requisições UDP por segundo, com uma latência de 173 microssegundos.

Sob estas melhorias, foi possível manter disponível e sincronizado o *Facebook Social Graph*, que durante anos tem sido o recurso que tem distinguido esta de outras redes sociais [Constine, 2013].

O portal de compras Amazon⁷, através de seu *Context Links* [Amazon, 2007] utiliza fragmentos de texto em um determinado cenário para sugerir livros relacionados ao termo. Desta forma, ao visitar um anúncio, os histórico da navegação do usuário é salvo e posteriormente usado para exibir indicações de produtos semelhantes [Amazon, 2011].

Analisando os trabalhos aqui citados, percebe-se que a adoção de ferramentas de cache – em especial o *memcached*, que será abordado com mais profundidade no Capítulo 4 – tem sido a solução adotada pelos maiores servidores de conteúdo da *Web* na atualidade, mitigando problemas como indisponibilidade de serviços devido a grande quantidade de requisições simultâneas e lentidão na recuperação de dados.

No escopo deste trabalho, o serviço de cache do contexto do usuário é um mecanismo que permite armazenar e manter em memória elementos contextuais frequentemente acessados,

⁵ <http://www.wikipedia.com>

⁶ <http://www.facebook.com>

⁷ <http://www.amazon.com>

reduzindo acessos ao armazenamento secundário e com isso aumentando o desempenho das aplicações que fazem uso do CODI4In. Uma questão importante é determinar quando os elementos são atualizados ou removidos, de forma a garantir a sua consistência. Alguns podem permanecer estáveis por um longo tempo e de repente mudar, outros podem mudar constantemente. A ideia e o desenvolvimento desta solução serão apresentados no Capítulo 4.

3. O CODI4In

Este capítulo introduz o estado da arte do *plugin* CODI4In, bem como os conceitos utilizados para sua construção. Para tal, aborda a ontologia de contexto definida, a arquitetura do *plugin*, o status de sua implementação e seu acoplamento a uma aplicação *Web*.

3.1 O Plugin CODI4In

O CODI4In foi especificado como um *plugin* que pode ser acoplado a diversos tipos de aplicações que envolvam consultas a dados [Freitas *et al.*, 2012]. Ele tem como objetivo prover a obtenção e persistência de informações (elementos) contextuais, de forma dinâmica e incremental, a partir das interações dos usuários e de suas consultas. Em outras palavras, trabalha como um serviço *back-end* ampliando funções de aplicações *front-end*, a partir do uso das informações contextuais armazenadas na ontologia CODI-User (a ser explicada na Seção 3.2). A Figura 2 apresenta uma visão da arquitetura do CODI4In e de sua relação com aplicações *front-end*. Assim, quaisquer aplicações orientadas a dados – aplicações que gerenciam dados e os utilizam em consultas - podem ser exemplos de aplicações *front-end* que podem usar o serviço do CODI4In.

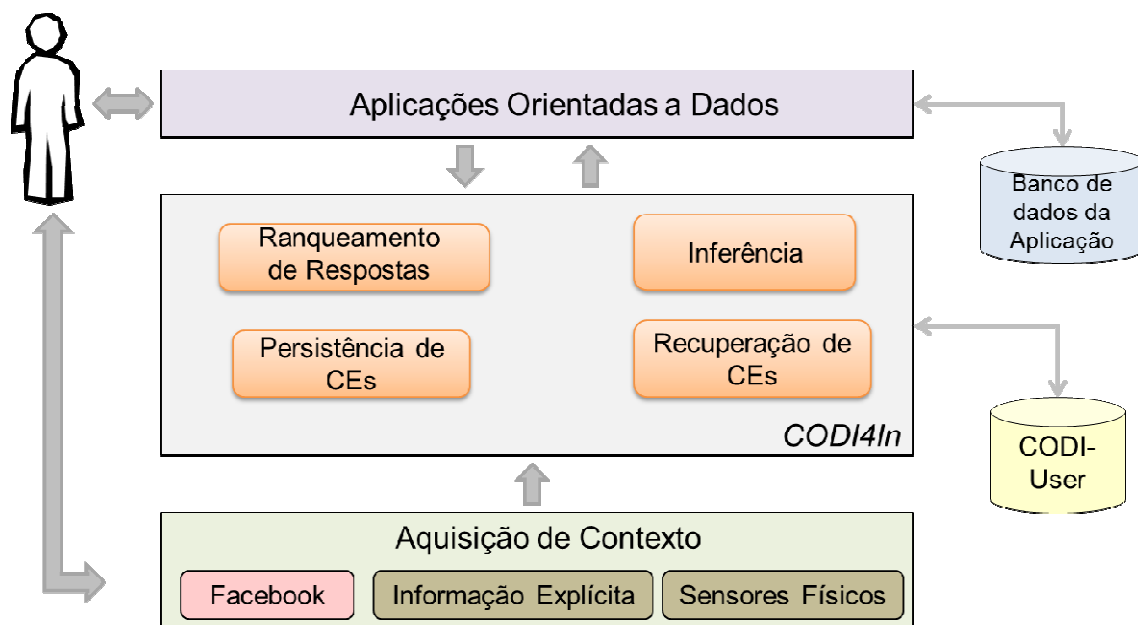


Figura 2 – Arquitetura do CODI4In.

Dessa maneira, no momento em que o usuário realiza uma consulta na aplicação, o serviço coleta as informações (elementos contextuais) da base da CODI-User e efetua a personalização dos dados (por exemplo, através de uma filtragem ou ranqueamento), valendo-se

não só dos parâmetros da consulta, mas também, dos elementos contextuais identificados e capturados pelo CODI4In. Como ilustração, suponha que um usuário submeta a seguinte consulta: “Mostre os pontos turísticos da cidade”. No momento da submissão, o CODI4In identifica que o usuário se encontra localizado no Rio de Janeiro. Considerando este elemento contextual (*localização*), o CODI4In efetua a personalização dos resultados, por meio de uma estratégia de ranqueamento, e exibe como primeiros resultados "visita ao Cristo Redentor" ou "bondinho do Pão de Açúcar".

3.2 Ontologia de Contexto CODI-User

A CODI (*Context Ontology for Data Integration*) é uma ontologia desenvolvida com base em requisitos da área de Integração de Dados e de gerenciamento de dados em ambientes P2P [Souza *et al.*, 2008]. Na CODI, são definidos dois conceitos principais: entidade de domínio (*Domain Entity*) e elemento contextual (*Contextual Element*). A ideia é que os elementos contextuais sejam construídos e/ou capturados sobre seis entidades de domínio: *user*, *environment*, *data*, *procedure*, *association* e *application*.

Com o propósito de criar uma ontologia de contexto de usuário independente de aplicações específicas, foi realizada uma extensão da CODI com foco na entidade de domínio *User*. Esta foi denominada de CODI-User [Freitas *et al.*, 2012]. Tendo em vista que a ontologia de contexto de usuário não deveria se ater a apenas preferências ou interesses, foram incorporados à CODI-User, outros elementos contextuais como, por exemplo, seu *expertise*, habilidades, fatores físicos. Elementos comuns a ambientes de consulta como localização, dispositivo, interface e tarefa também foram incluídos. Uma visão da entidade de domínio *User* juntamente com alguns de seus elementos contextuais é apresentada na Figura 3. Neste fragmento, são mostrados apenas metadados, referentes a relacionamentos da entidade *User* e alguns elementos contextuais (*Location*, *Expertise*, *PersonalInfo*, *Connection*, *Interest* e *Preference*).

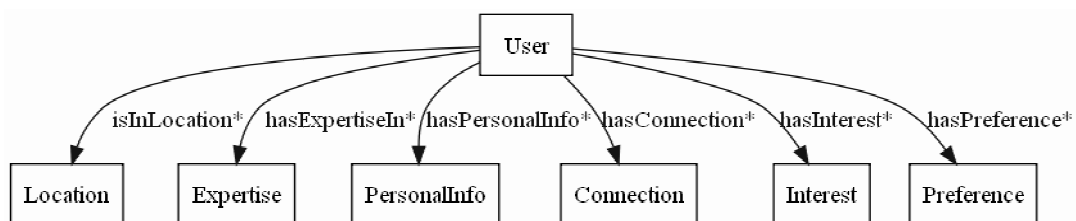


Figura 3. Fragmento da ontologia de contexto CODI-User

Na Figura 4 podemos visualizar um fragmento de uma instancia da ontologia, feita usando o software Protégé⁸. Nela, temos uma instancia da entidade de domínio *User*, identificada como “Maria”, com seus elementos contextuais associados em um dado instante. Na imagem podemos ver que Maria tem interesse em música, com preferencia em Rock e MPB e possui *expertise* em cantar.

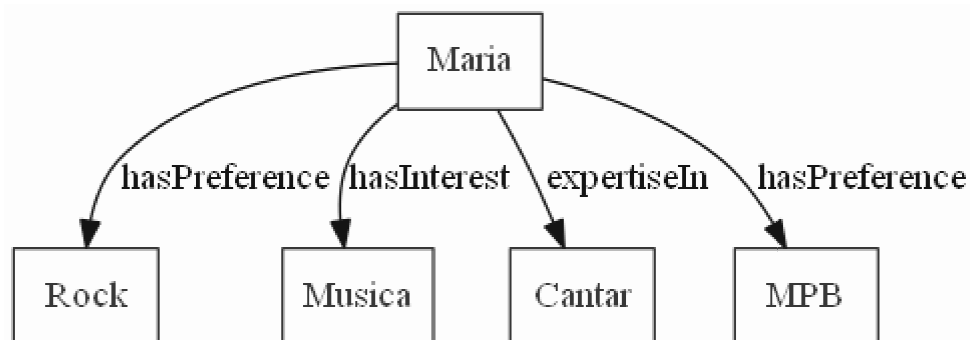


Figura 1. Exemplo de instancição da CODI-User

3.3 Estudo de Caso: A Aplicação MovieShow

Como primeiro estudo de caso, uma aplicação que gerenciase preferências por gêneros de filmes a fim de ranquear os resultados de pesquisas por filmes foi desenvolvida, a qual foi denominada MovieShow. Esta aplicação foi diretamente acoplada ao CODI4In de forma a, através de consultas a um banco de dados relacional populado com dados provenientes do IMDb⁹ - uma base de dados online de informações relacionadas a filmes -, exibir resultados de consultas submetidas a respeito de filmes.

Ao fazer o *login* no MovieShow, o usuário é questionado se deseja permitir que o sistema tenha acesso as suas informações pessoais no Facebook [Alves *et al.*, 2013]. Permitindo o acesso, o usuário é convidado a realizar o *login* na rede social. Em caso negativo, ele precisará efetuar seu cadastro e em seguida ingressar no sistema. Nesta seção, estaremos abordando a primeira opção, na qual o usuário permite que o CODI4In extraia informações de seu perfil do Facebook.

Uma vez que o usuário ingressou na aplicação e no Facebook, o CODI4In recupera as informações do seu perfil do Facebook no formato JSON¹⁰ - um padrão baseado em texto para troca de dados -, conforme o seguinte: (i) informações pessoais (*first_name*, *last_name*, *gender* e *email*) e (ii) informações de preferencias fornecidas pelas “curtidas” que o usuário fez em

⁸ <http://protege.stanford.edu/>

⁹ <http://www.imdb.com>

¹⁰ <http://www.json.org>

relação a filmes. O CODI4In recupera, então, o título dos filmes que o usuário curtiu e, através de um *Web Service* fornecido pelo IMDB, recupera o gênero desses filmes, persistindo-os na CODI-User como as preferencias de filmes do usuário. Além disto, o *plugin* gera uma lista ranqueada das preferencias. Entretanto, ao usuário é permitido o refinamento desta lista de maneira a reordená-la como lhe aprouver. Caso o usuário não queira fazer uso desta opção, o CODI4In utilizará a lista padrão (ordenada pelo método *Bubble sort*) para personalizar o resultado da consulta. Vale salientar que a consulta não é reformulada, apenas seu resultado. A Figura 5 demonstra o processo geral de aquisição do contexto utilizando o perfil do usuário no Facebook, o gerenciamento dos elementos contextuais, a personalização de consultas e a exibição dos resultados.

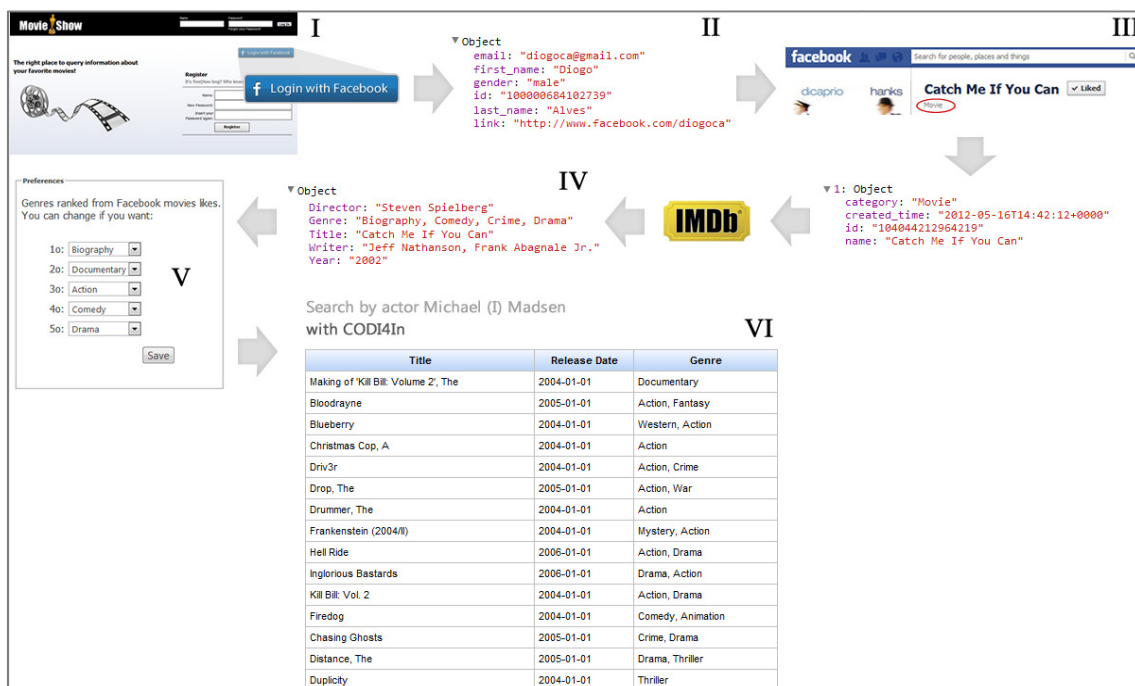


Figura 5. Fluxograma das etapas percorridas pelo CODI4In para aquisição das preferencias e personalização de consultas.

A Figura 5 exemplifica o seguinte cenário: o usuário Diogo faz o *login* no MovieShow permitindo que o sistema tenha acesso a seu perfil do Facebook (etapa I), autorizando, assim, que o CODI4In recupere suas informações pessoais (etapa II) e os filmes que Diogo curtiu - "Catch Me If You Can" (etapa III), neste caso. Com esta informação disponível, o CODI4In se comunica com o IMDB através de seu *Web Service* e recupera seus gêneros (*Biography*, *Comedy*, *Crime* e *Drama*) (etapa IV). No exemplo apenas um filme é exemplificado, mas, de fato, esta operação está sendo realizada sobre todos os filmes que o usuário em questão curtiu.

Os gêneros deste filme são então persistidos na ontologia CODI-User, ranqueados de acordo com a quantidade de ocorrências que cada um teve e exibidos através da interface do

MovieShow (etapa V). Finalmente, Diogo realiza uma consulta no banco de dados local por filmes estrelados pelo ator “Michael Madsen”. O esquema do banco é composto por quatro tabelas: *Movie* (*Id*, *Title*, *Release Year*, *Genre*, *Actor*, *Director*), *Actor* (*Id*, *Name*, *BirthDate*), *Director* (*Id*, *Name*, *BirthDate*) e *Genre* (*Id*, *Name*). A consulta é executada e os resultados são ranqueados de acordo com os elementos contextuais referentes às preferências de Diogo no que diz respeito a gênero de filmes (etapa VI). Nota-se que os filmes dos gêneros *Documentary* e *Action* são exibidos no topo da lista do resultado da consulta, de acordo com a ordem das preferências por gênero. Se algum filme não corresponder a alguma preferência do usuário, ele será colocado no final da lista.

Com base no funcionamento do MovieShow explicado nesta seção e, a partir da arquitetura do CODI4In, pudemos perceber que as preferências, que são constantemente utilizadas, tem de ser recuperadas diversas vezes diretamente no *plugin*, aumentando a quantidade de interações entre a aplicação e o CODI4In, resultando, além do seu alto nível de acoplamento, em requisições desnecessárias. Portanto, vemos a necessidade do desenvolvimento do gerenciador de cache, que será explicado a seguir.

4. O Serviço de Cache: Implementação e Resultados

Este capítulo apresenta a abordagem tema deste trabalho que envolveu a criação de um serviço de cache, sua comunicação com o *plugin* CODI4In e uma aplicação Web que consome os dados de ambos. Para tal, descreve o processo adotado, mostra a especificação realizada, pontuando e explanando os requisitos funcionais e características técnicas, a implementação, os resultados obtidos e alguns testes realizados.

4.1 Objetivo do Serviço de Cache

Conforme visto na Seção 1.1, o objetivo do CODI4In é gerenciar, de forma dinâmica e incremental, os elementos contextuais de usuários de aplicações diversas, transferindo para si esta responsabilidade, de maneira a permitir que as aplicações foquem nos serviços que se propõem a oferecer.

Prevendo o aumento da quantidade de requisições direcionadas ao *plugin* para armazenar e recuperar informações contextuais relevantes ao usuário em determinado ambiente, deslumbre-se, inevitavelmente, um aumento no tempo de latência no servidor do *plugin* gerado por estas operações e, portanto, o serviço que as geraram sofrerá, negativamente, o reflexo desta latência.

Com base nos diversos trabalhos relacionados apresentados na Seção 2.4, viu-se que o uso de gerenciadores de cache tem se tornado uma tendência entre diversas aplicações e serviços Web, uma vez que a quantidade de informações produzidas e consumidas por meio de consultas pode exceder a capacidade de resposta de um servidor de banco de dados em um determinado instante. Esta problemática engatilhou a proposta de desenvolvimento de um protótipo de um serviço de cache próprio para atender a possível demanda que o CODI4In venha a ter uma vez que esteja funcionando em sua plenitude, fornecendo elementos contextuais e reformulando consultas a partir de resultados obtidos pelo seu raciocinador, e provendo resultados mais próximos aos esperado pelo usuário. Neste cenário, podemos enumerar os seguintes objetivos principais para o serviço de cache:

- 1) Aliviar a carga imposta ao *plugin* devido à quantidade de requisições feitas e a quantidade de elementos contextuais que devem ser carregados e,
- 2) Diminuir o tempo de resposta em operações de recuperação, adição e edição de elementos contextuais.

Mais especificamente, o serviço de cache será responsável pela representação e gerenciamento do grafo contextual de cada usuário que, provido pelo *plugin* CODI4In, será utilizado para personalizar consultas advindas das aplicações *front-end*. Assim, o serviço de gerenciamento de cache trabalhará como uma camada intermediária entre a aplicação *front-end* e

o *plugin*, conforme visto na Figura 6, de modo a diminuir o tempo de latência entre as consultas feitas pelo usuário, sua execução e a apresentação da resposta.

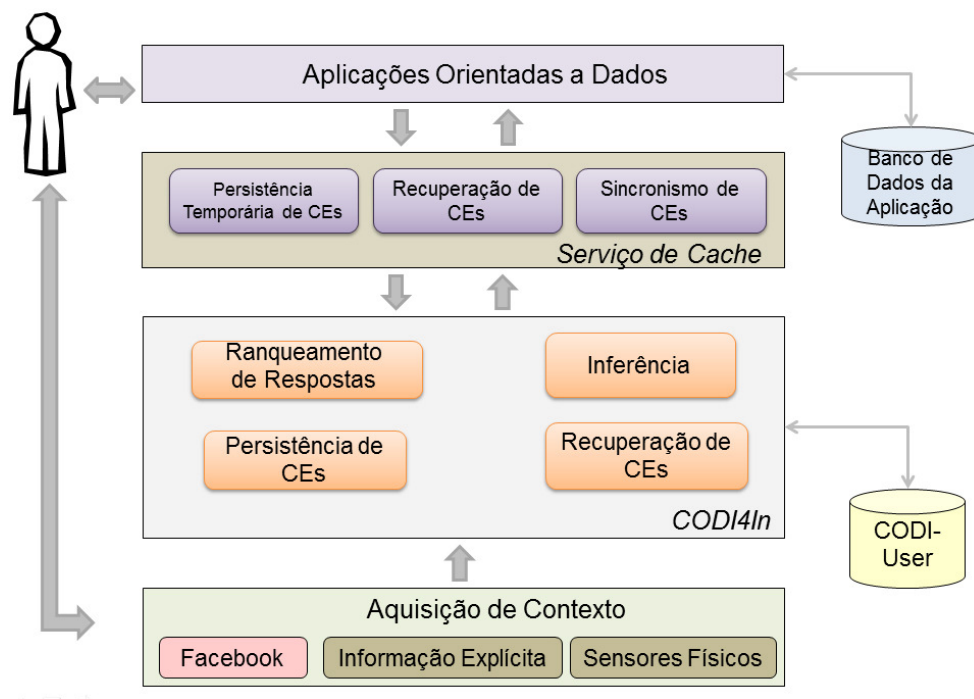


Figura 6. Arquitetura do CODI4In com o Serviço de Cache.

Como forma de validar o serviço de cache e sua interação com o CODI4In, foi desenvolvida também uma aplicação *Web*, chamada de *IndSong* – que será descrita na Seção 4.3.

4.2 Requisitos do Gerenciador de Cache

Ao trabalharmos com cache, invariavelmente encaramos o desafio de trabalhar com dados que estão temporariamente armazenados em memória e que após um determinado tempo ou evento serão descartados e substituídos por outros. Estes dados, por sua vez, também precisam estar representados de tal forma que sejam facilmente gerenciados por quaisquer aplicações que venham a fazer uso deles. Sendo assim, podemos definir os principais requisitos funcionais do serviço de cache como:

- Representar elementos contextuais que serão armazenados e recuperados pelo serviço;
- Criar uma tarefa cronometrada que faça o sincronismo entre os dados em cache e o mecanismo de persistência usado pelo *plugin*.

Como requisitos não funcionais, podemos definir que o serviço de cache deverá se comunicar para fora de sua fronteira utilizando *Web Services*, a fim de permitir sua interação com outras aplicações e serviços desenvolvidos em linguagens diferentes da sua.

4.3 A Aplicação IndSong

A fim de validar o serviço de cache e sua interação com o CODI4In, desenvolvemos uma aplicação *Web*, a qual denominou-se *IndSong* - alusão ao termo *indicação de trilhas sonoras* em inglês. Sua proposta é de que através da persistência e recuperação de elementos contextuais, neste caso as preferencias por gêneros musicais e o endereço do usuário, criarmos um sistema de sugestão de álbuns. Logo, as preferencias funcionam como filtro na consulta por indicações.

Semelhantemente ao MovieShow (descrito na Seção 3.3), o usuário, ao realizar um cadastro e se autenticar, tem direito a alterar suas preferencias. Indo mais além, a IndSong permite, também, que o endereço seja alterado, através de formulários que submetem os dados informados diretamente para o serviço de cache. Sendo assim, podemos elencar como requisitos funcionais da aplicação os seguintes:

- Cadastrar e tratar os dados referentes ao endereço do usuário e preferencias musicais;
- Recuperar uma lista de álbuns musicais e seus metadados a partir de um repositório confiável;
- Exibir a lista de álbuns filtrados pelo CODI4In em forma de indicações.

4.4 Especificação do Serviço de Gerenciamento de Cache para o CODI4In

De forma a manter a consistência dos dados esperada pela aplicação que faz uso do CODI4In, espera-se que o serviço de cache mantenha conexão com o *plugin* de tal forma que o primeiro reflita de maneira fiel os dados fornecidos pelo último. Portanto, podemos definir o principal requisito funcional como sendo sincronizar os elementos contextuais em cache com a ontologia CODI-User, a fim de que o CODI4In possa gerenciá-los.

Mantendo a funcionalidade do *plugin* de personalizar os resultados de consultas, foi desenvolvida uma seção na aplicação que consome dados através da API do *Free Music Archive*¹¹, *website* voltado para o domínio da música, e os reorganizam, de forma a utilizar as preferencias de gêneros musicais recuperados do gerenciador de cache para sugerir álbuns que mais se aproximem do gosto do usuário.

O diagrama de sequencia, mostrado na Figura 7, representa a forma em que a comunicação ocorre entre a aplicação, o gerenciador de cache e o CODI4In. Suas principais etapas são descritas a seguir:

¹¹ <http://www.freemusicarchive.org>

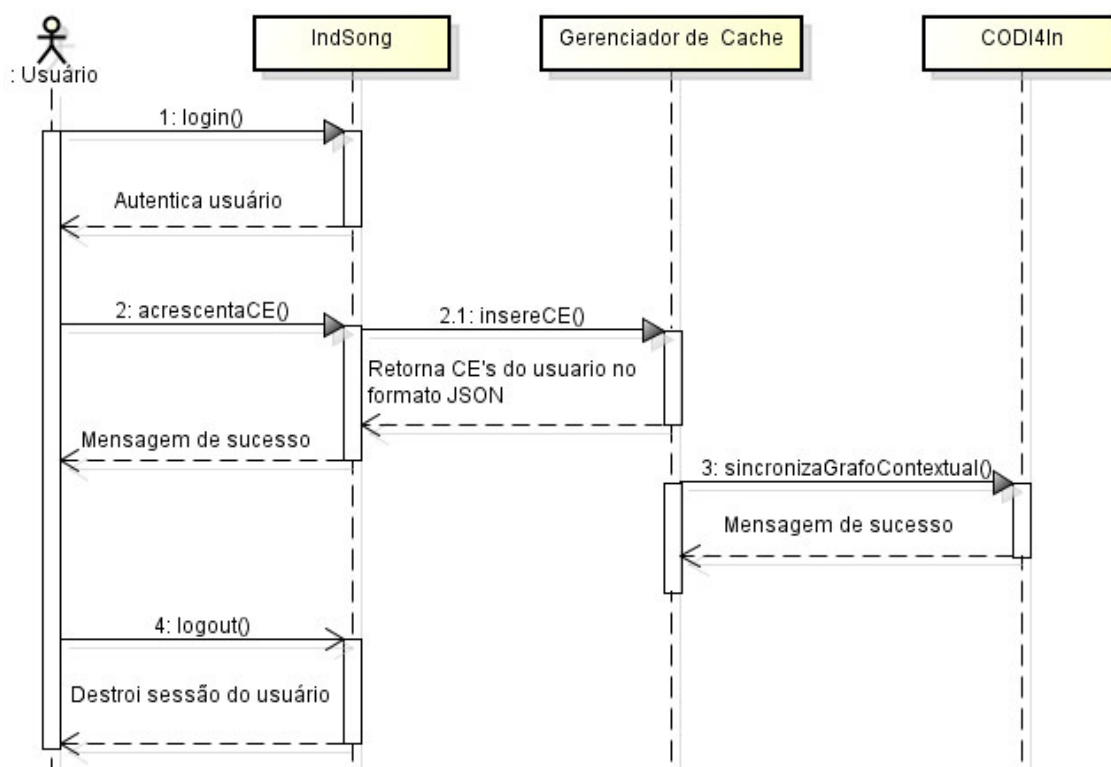


Figura 7. Interação entre a aplicação, gerenciador de cache e CODI4In.

1. Ao ser solicitado, o usuário entra com suas credenciais e realiza o *login* na aplicação. Uma nova sessão é, então, criada dando acesso às opções de gerenciamento dos elementos contextuais disponíveis na aplicação – preferencias e endereço.
2. Em uma seção específica é possível gerenciar as preferencias por gêneros musicais, que são recuperados do banco de dados da aplicação e alterar seu endereço, que é recuperado pelo CODI4In.
 - 2.1 Tomando-as como exemplo, as preferencias são enviadas pela aplicação por uma requisição POST e salvas no serviço de cache que retorna a nova lista no formato JSON¹². Esta lista é tratada pela aplicação e reexibida para o usuário.
3. Paralelamente ao gerenciador de cache uma tarefa cronometrada está em execução. A cada período de tempo previamente definido, uma rotina é executada assincronamente, varrendo os itens sendo gerenciados no serviço de cache e comparando-os com o estado em que se encontram seus equivalentes no CODI4In. Caso alguma alteração tenha sido realizada, esta rotina fará o sincronismo, atualizando o *plugin* com os elementos contextuais mais recentes.
4. Ao terminar o uso da aplicação, o usuário pode realizar o *logout*, destruindo suas credenciais da sessão, mas mantendo todos os seus elementos contextuais ativos no

¹² <http://www.json.org>

serviço de cache até que, por critério do algoritmo LRU apresentado na Seção 2.3, eles ou parte deles sejam substituídos por outros.

Para clarificar o entendimento do funcionamento do gerenciador de cache, a Figura 8 exibe um diagrama de atividades que ilustra a maneira em que os elementos contextuais são recuperados e armazenados. Todo o processo será explicado em detalhes na Seção 4.3.

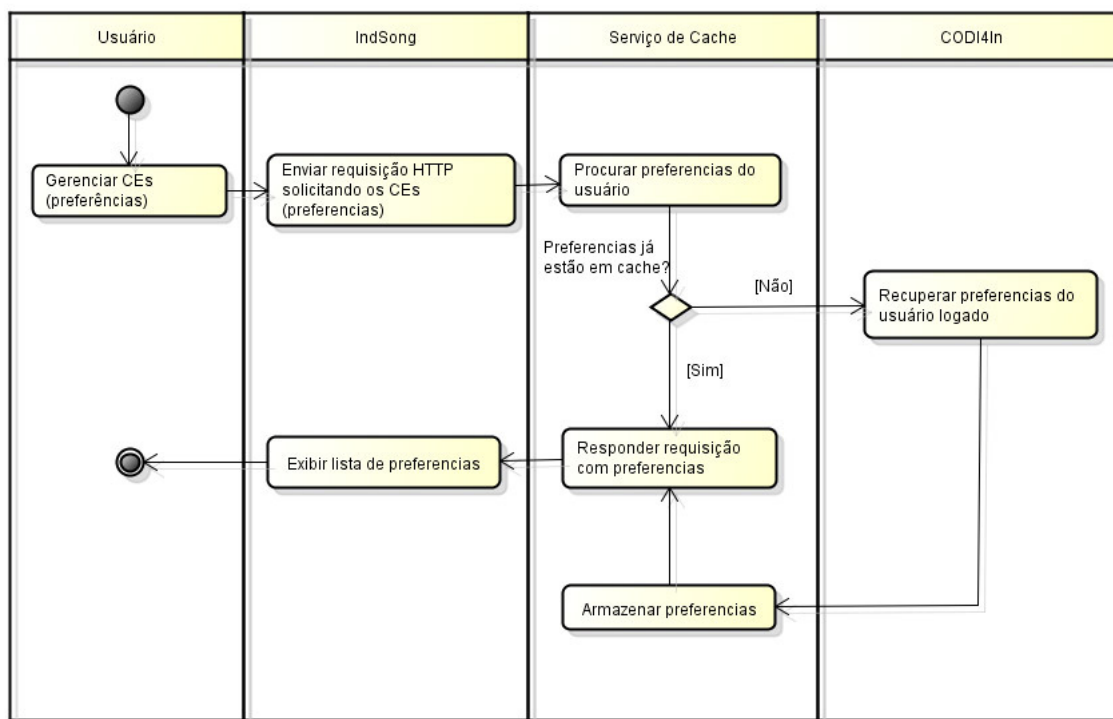


Figura 8. Gerenciamento de elementos contextuais

A Figura 8 mostra que, ao acessar suas preferências, o usuário logado dispara, através do IndSong, uma requisição *HTTP* ao serviço de cache. Este identifica o usuário em questão e busca por suas preferências. Caso as encontre, envia uma resposta à requisição feita com a lista de preferências, que será tratada e exibida pela aplicação. Caso elas não sejam encontradas, uma nova requisição é feita pelo gerenciador e enviada ao *CODI4In*, que tratará de recuperar as preferências persistidas e enviá-las como resposta, que será tratada e armazenada. O gerenciador de cache, então, responderá a requisição feita pela aplicação com a lista de preferências, que será exibida para o usuário.

4.5 Implementação

No desenvolvimento do serviço de cache, apresentado na seção seguinte, adotamos a linguagem Java para mantermos a padronização do que já havia sido feito no *CODI4In*. Para o modelo de comunicação remota do serviço de cache via protocolo *HTTP*, utilizamos a tecnologia

REST [Fielding, 2000], um estilo de arquitetura utilizado para aplicações em rede, através do protocolo HTTP, por meio dos pacotes Java Jersey¹³. A escolha desta tecnologia se deve a sua fácil implementação e interoperabilidade entre aplicações desenvolvidas em linguagens diferentes, por exemplo: uma aplicação Java se comunicando com uma aplicação desenvolvida em Ruby on Rails¹⁴. Realizamos todo o processo de desenvolvimento de forma iterativa e incremental.

Foi utilizada uma única API no desenvolvimento da aplicação *Web*: a *Free Music Archive* API, utilizando o formato JSON para transmissão de dados pela rede.

4.5.1 O Serviço de Cache

Para o desenvolvimento do serviço, a linguagem Java foi à escolhida para ser utilizada e a forma de comunicação para fora de sua fronteira foi definida como sendo *Web Services*. Uma preocupação que foi levada em conta ao implementar foi a transparência no acesso remoto que as aplicações enfrentariam. Por isso, resolveu-se adotar URIs iguais para o *plugin* e para o serviço. Por exemplo, para recuperar todas as preferencias de um determinado usuário, utiliza-se a seguinte URI para ambos: */user/{id do usuário}/preferences*.

Apesar de utilizarmos a linguagem Java para desenvolvimento dos métodos que manipulam os dados armazenados em cache, o núcleo do serviço encontra-se no uso da memória RAM como forma de armazenamento temporária dos elementos contextuais. Para este objetivo, utilizamos uma máquina com o sistema operacional Linux, rodando a distribuição Ubuntu¹⁵ versão 11.04. Logo, o serviço de cache é composto de duas partes: o servidor que executa o processo *memcached* e a interface de comunicação, desenvolvida em Java. Ambas serão explicadas detalhadamente a seguir.

- **O Servidor Cache**

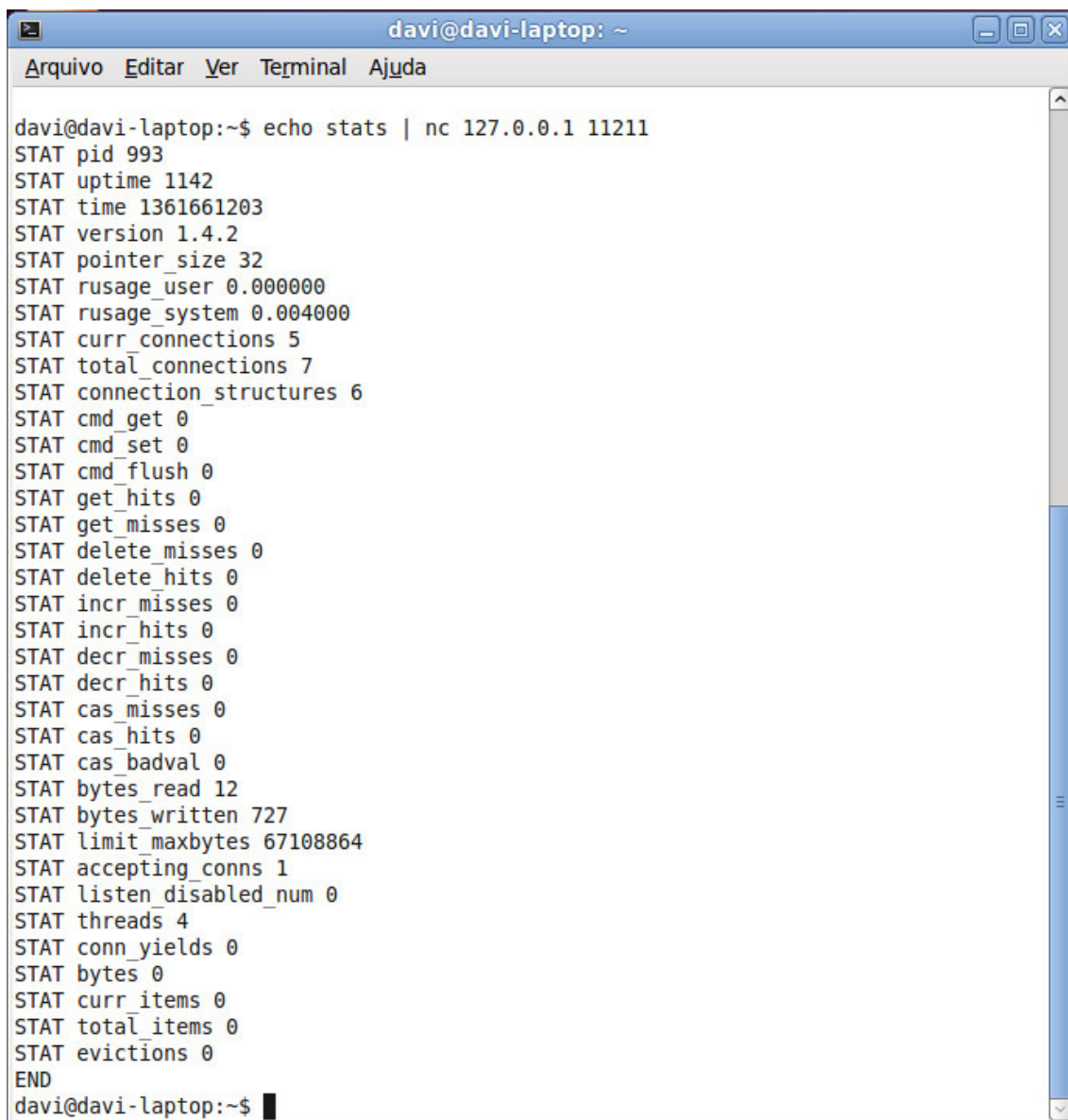
Conforme mencionado na Seção 2.3 e exemplificado na Seção 2.4, o *memcached*, devido a sua facilidade de implementação, escalabilidade e flexibilidade, tem sido utilizado largamente por diversos dos maiores *websites* produtores de conteúdo da Internet, entre eles o Facebook, que o utiliza para armazenar em cache informações contextuais por meio de seu *Social Graph*. Assim, resolvemos adotá-lo como nosso cliente de cache.

A Figura 9 exibe os status do processo *memcached* depois de inicializado, ao executarmos o comando *echo stats | nc 127.0.0.1 11211*.

¹³ <http://jersey.java.net/>

¹⁴ <http://rubyonrails.org/>

¹⁵ <http://www.ubuntu.com/>



```

davi@davi-laptop: ~
Arquivo  Editar  Ver  Terminal  Ajuda

davi@davi-laptop:~$ echo stats | nc 127.0.0.1 11211
STAT pid 993
STAT uptime 1142
STAT time 1361661203
STAT version 1.4.2
STAT pointer_size 32
STAT rusage_user 0.000000
STAT rusage_system 0.004000
STAT curr_connections 5
STAT total_connections 7
STAT connection_structures 6
STAT cmd_get 0
STAT cmd_set 0
STAT cmd_flush 0
STAT get_hits 0
STAT get_misses 0
STAT delete_misses 0
STAT delete_hits 0
STAT incr_misses 0
STAT incr_hits 0
STAT decr_misses 0
STAT decr_hits 0
STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT bytes_read 12
STAT bytes_written 727
STAT limit_maxbytes 67108864
STAT accepting_conns 1
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT bytes 0
STAT curr_items 0
STAT total_items 0
STAT evictions 0
END
davi@davi-laptop:~$ █

```

Figura 9. Status do processo *memcached* após iniciado.

Entre os diversos *stats* exibidos na Figura 9, podemos chamar à atenção aos *get_hits* e *get_misses*. Entre os conceitos abordados na Seção 2.3, *hits* e *misses* correspondem à taxa de acertos e erros/falhas ao tentar buscar por um dado em cache. Geralmente, qualquer tentativa de acesso a um dado não existente em um servidor cache resultará em *miss*. Em contrapartida, qualquer dado que acessado com sucesso resultará em *hit*.

De maneira simplificada, o *memcached* funciona, internamente, como uma *HashTable* [Fitzpatrick, 2004], que mantém seus nós ou registros na forma de chave-valor. Esta lista, posteriormente, é pesquisada para encontrar os nós desejados. Além disto, ela possui tamanho dinâmico, de forma que, não encontrando um determinado registro, é possível adicioná-lo na tabela. Outra característica de seu núcleo está em seu comportamento adaptativo: através de seu algoritmo LRU (*Least Recently Used*), explicado na Seção 2.3, cada instancia do *memcached*

elimina automaticamente os itens menos usados para criar espaço para novos. Esta e outras operações são armazenadas em forma de estatísticas pelo servidor, servindo de base para os *stats* mostrados na Figura 9. Na imagem, é possível notar que quase todos se encontram zerados. Isso se deve pelo fato do processo ter sido inicializado e não ter sofrido nenhuma alteração até o momento. Sua instalação e inicialização podem ser acompanhadas no Apêndice A.

- **A interface de comunicação**

Com o *memcached* em execução, começamos o desenvolvimento da interface que faz conexão entre o servidor e a aplicação. Na classe Java criada para este trabalho, podemos agregar os métodos em dois grupos: os métodos que aceitam requisições *GET*, usados para consumo de dados, e os métodos que recebem requisições *POST*, usados para persistência de dados. A Figura 10 apresenta um algoritmo em alto nível que resume o momento em que preferencias são recuperadas pelo usuário.

```

1  @GET
2  @Produces(MediaType.APPLICATION_JSON)
3  @Path("/user/{userId}/preferences")
4  getPreferences(int userId)
5      //Recuperação das preferencias do usuário direto no serviço cache
6      if (servidor cache está operante)
7          Future<Object> f = c.asyncGet("Preferences_" + userId)
8          preferencias = f.get(5, TimeUnit.SECONDS)
9          if (não encontrou preferencias em cache)
10             if (carregaPreferenciasPeloCODI4In(userId))
11                 c.set("Preferences_" + userId, 0, preferencias)
12                 return Response.ok(preferencias).build()
13             else
14                 return Response.status(Status.SERVICE_UNAVAILABLE).build()
15             end if
16         else
17             return Response.ok(preferencias).build()
18         end if
19     else
20         //Se o usuário for encontrado pelo CODI4In, retorne suas preferencias
21         if(carregaPreferenciasPeloCODI4In(userId))
22             return Response.ok(preferencias).build()
23         //Se não retorne erro 404 - Não Encontrado
24         else
25             return Response.status(Status.NOT_FOUND).build()
26         end if
27     end if
28 endGetPreferences()

```

Figura 10. Algoritmo de recuperação de elementos contextuais

Existem, basicamente, três fluxos que o algoritmo pode seguir, detalhados em seguida. Em todos eles temos em comum que, uma vez recebida a requisição GET, o ID do usuário é

capturado da URI (“/user/{userId}/preferences”) e guardado em uma variável (*userId*). Os fluxos são:

1) Elementos contextuais ainda não carregados em cache

- Iniciamos verificando se o servidor cache está operante. Em caso positivo, tentamos recuperar os elementos contextuais, que em nosso exemplo são as preferencias, através de sua chave (linhas 7 e 8), que é composta pelo nome “*Preferences_*” acrescido do ID do usuário. Para evitar que a aplicação fique inoperante aguardando que sua requisição seja respondida, definimos um tempo limite de 5 segundos para que as preferencias sejam encontradas e recuperadas.
- Se elas não foram encontradas no cache no tempo especificado, é necessário busca-las diretamente no CODI4In (linha 10), utilizando o ID do usuário como argumento. Quando encontradas, as preferencias são persistidas no cache já em formato JSON, através do comando da linha 11. Este comando recebe três parâmetros: o primeiro define o nome da chave; o segundo define o tempo de vida - em segundos - que este registro estará disponível em cache; e o terceiro define o valor do registro. Escolhemos que o nó (chave-valor) estará em cache por tempo indeterminado ao optarmos por 0 segundos. Fazendo isto, deixamos a cargo do *memcached*, através de seu algoritmo LRU, definir quais itens do cache devem ser substituídos por novos, à medida que o espaço em memória atinge seu limite.
- Finalmente, retornamos ao usuário por meio de uma resposta HTTP as preferencias no formato JSON, conforme podemos visualizar na Figura 11.

```
{
  "preferences": [
    "Romance",
    "Fantasy"
  ]
}
```

Figura 11. Preferencias no formato JSON

2) Preferencias já carregadas em cache

- Semelhantemente ao primeiro item do fluxo anterior, uma verificação para checar se o servidor de cache está online e operante é realizado.

- Averiguado seu funcionamento, tenta-se recuperar as preferencias através de sua chave, no formato definido anteriormente, dentro do tempo máximo de 5 segundos.
- Encontrada o registro em cache para a chave requisitada, uma resposta HTTP com *status* de sucesso é gerada para a aplicação, contendo as preferencias, conforme visto na Figura 11.

3) Servidor cache off-line

- Semelhantemente ao primeiro item do primeiro fluxo, uma verificação para checar se o servidor de cache está online e operante é realizado.
- Averiguando que o servidor cache encontra-se off-line ou indisponível, a requisição é imediatamente propagada para o *plugin* (linha 21) que retorna as preferencias do usuário se encontradas.

Além da requisição *GET*, o serviço de cache compreende, ainda, requisições *POST*, que objetivam a persistência temporária de elementos contextuais, e, posteriormente, no *plugin*, através de um algoritmo de sincronismo de dados. A Figura 12 ilustra usando uma linguagem de alto nível, o algoritmo do método de persistência. Neste exemplo, utilizamos o elemento contextual *Address*.

```

1  @POST
2  @Path("/user/address/save/")
3  @Produces("application/json")
4  @Consumes("application/json")
5  postAddress()
6      JSONObject enderecoJson = new JSONObject();
7      requisicao = recuperaDadosDaRequisicao()
8      enderecoJson.put("address", requisicao.endereco);
9      if (servidor cache está operante)
10         c.set("Address_" + requisicao.userId, 0, enderecoJson)
11     else
12         salvaEnderecoPeloCODI4In(requisicao.userId)
13         return Response.ok(enderecoJson).build()
14     end if
15 endPostAddress

```

Figura 12. Algoritmo em alto nível da requisição *POST* para elemento contextual

Ao receber uma requisição *POST* pela URI */user/address/save*, a interface do serviço armazenará os dados do endereço do usuário em uma variável, que tratará de ser transformada para o formato JSON. Caso o servidor de cache esteja operante, acrescenta-se uma nova entrada

na *HashTable* do processo *memcached*, com o nome de chave “*Address_*” seguido do ID do usuário. Caso o servidor esteja off-line, o endereço é salvo diretamente no CODI4In. Em ambos os casos, o endereço em formato JSON é retornado para a aplicação, tratado e reexibido para o usuário. A Figura 13 ilustra o a resposta gerada pelo serviço.

```
{
  "address": {
    "Neighborhood": "jardim oecania",
    "State": "paraiba",
    "Number": "601",
    "Complement": "apto 104-B",
    "street": "sebastiao interaminense",
    "Zip": "58037-770",
    "City": "jampa"
  }
}
```

Figura 13. Resposta do serviço de cache a requisição *POST* para endereço.

Podemos notar no algoritmo da Figura 12 que a persistência do elemento contextual endereço se dá, neste instante, unicamente no serviço de cache – se disponível. Esta prática foi adotada para diminuir o tempo de resposta que o usuário espera após realizar sua requisição. Entretanto, alguns problemas podem ser ocasionados a partir desta prática, como por exemplo, a incoerência dos dados registrados no serviço e no *plugin*. Para mitigar este problema sem afetar o desempenho desejado, criou-se uma tarefa cronometrada que funciona paralela e assincronamente às rotinas de recuperação e persistência dos dados em cache, a fim de manter a CODI-User sincronizado com o serviço de cache. A Figura 14 exhibe o funcionamento desta tarefa, usando uma linguagem de alto nível, sendo explicada em seguida.

```
1  syncData()
2      grafoContextual = carregarGrafoContextualEmCache()
3      for each CE in grafoContextual
4          CE_DoPlugin = CE_EquivalenteCODI4In(CE)
5          if(CE != CE_DoPlugin)
6              sincronizaCODI4In(CE)
7          end if
8      end for
9  endSyncData
```

Figura 14. Tarefa cronometrada que realiza sincronismo entre o CODI4In e o serviço de cache

O código exibido na Figura 14 é periodicamente executado por uma tarefa cronometrada. Ao ser executada, a rotina itera sobre todos os elementos contextuais que do grafo contextual e os compara com seus equivalentes que são gerenciados pelo CODI4In. Ao encontrar um elemento contextual que seja diferente, a rotina sincroniza-o a fim de manter a consistência dos dados em ambas as partes.

4.5.2 Estudo de Caso: A Aplicação IndSong

Desenvolvida usando a linguagem PHP, a aplicação IndSong manteve a mesma estrutura do MovieShow no que diz respeito aos requisitos para utilização de suas funcionalidades.

Para isso, o usuário precisa se cadastrar e se autenticar. A Figura 15 ilustra a tela inicial da aplicação, contendo o formulário de cadastro e de *login*.

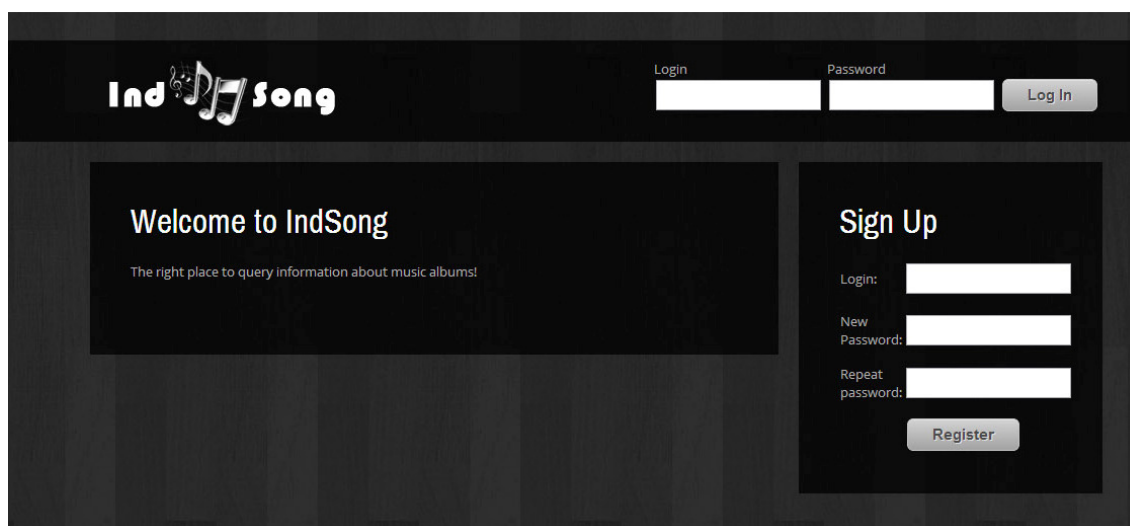
A imagem mostra a interface de usuário da aplicação IndSong. No topo, há uma barra de navegação com o logo "IndSong" à esquerda e campos de "Login" e "Password" à direita, com um botão "Log In". Abaixo, a página é dividida em duas seções principais. À esquerda, uma caixa de texto diz "Welcome to IndSong" com o subtítulo "The right place to query information about music albums!". À direita, há uma seção "Sign Up" com campos para "Login:", "New Password:" e "Repeat password:", e um botão "Register".

Figura 15. Tela inicial da aplicação IndSong.

Após realizar o cadastro e se autenticar, uma sessão é criada e ao usuário é permitida a interação com o sistema. Iniciamos o processo definindo o endereço do usuário. Preenchendo e submetendo o formulário ilustrado na Figura 16, geramos uma requisição *POST*, pelo protocolo HTTP, para o serviço de cache que tratará de persistir os dados, retornando-os para a aplicação. Estes, por sua vez, serão exibidos para o usuário. Para a transferência dos dados, o formato JSON foi adotado, devido ao seu pequeno tamanho em *Bytes*.

The screenshot shows the IndSong web application interface. At the top, there is a navigation bar with the logo 'IndSong' and links for HOME, REGISTER PREFERENCES, MY ADDRESS, and LOGOUT. The main content area is divided into two sections. The left section, titled 'My Address', contains a form with the instruction 'Please fill out the form below with your address information.' The form fields are: City (João Pessoa), State (Paraíba), Street (Sebastião Interaminense), Number (601), Zip (58037-770), Neighborhood (Jardim Oceania), and Complement (apto 104-B). A 'Save' button is at the bottom of the form. The right section, titled 'Album Indications', has two options: 'Traditional' and 'With CODi4in'. The 'Traditional' option is selected, and it says 'Get indications of albums provided by Free Music Archive'. The 'With CODi4in' option says 'Get indications of albums provided by Free Music Archive and powered by CODi4in'.

Figura 16. Persistência do elemento contextual *Address* através de um formulário Web, pela aplicação IndSong.

Na página *Register Preferences*, o usuário pode definir quantas e quais preferencias musicais lhe agradam. Por meio de um *script*, a lista de gêneros provenientes do *webservice* fornecido pelo *website Free Music Archive* foi persistida em um banco de dados relacional e trazida para o formulário de registro de preferencias. Para esta aplicação, utilizamos o banco de dados MySQL¹⁶ rodando localmente com uma única tabela: *Genres* (*genre_id*, *genre_title*, *genre_handle*).

Para utilizarmos os recursos que este *website* oferece, tivemos que realizar um cadastro para termos um *token* de acesso a sua API, que por meio de requisições REST nos permitiu consultar álbuns, gêneros, artistas e trilhas sonoras.

Uma vez as preferencias persistidas em nossa base de dados local, pudemos fornecer ao usuário dezenas de opções de gêneros musicais através de caixas de seleção em um formulário, conforme visto na Figura 17.

¹⁶ <http://dev.mysql.org>

Preferences

Please, set your preferred song genres: [Add Another Preference](#)

Preference #1: [Delete](#)

Preference #2: [Delete](#)

Preference #3: [Delete](#)

Figura 17. Formulário com as caixas de seleções para definição dos gêneros musicais preferidos.

Na Figura 17 podemos notar que dois gêneros carregados do serviço de cache já vêm selecionados. Entretanto, conforme dito anteriormente, todo o modelo de conexão remoto é transparente para o usuário. Caso o CODI4In esteja indisponível, todos os dados serão recuperados e persistidos no serviço de cache, que tratará de realizar o sincronismo quando o *plugin* volte a estar disponível. Se o serviço de cache estiver indisponível, a comunicação ocorrerá diretamente entre a aplicação e o *plugin*. Se ambos estiverem off-line, a aplicação ficará encarregada de tratar a indisponibilidade de ambos, conforme ilustrado na Figura 17.

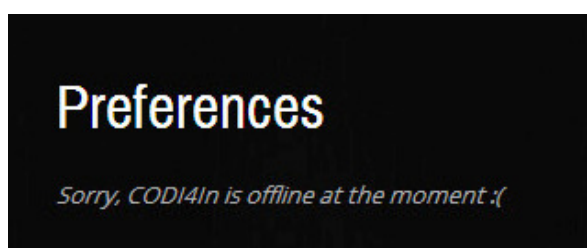



Figura 18. Mensagem de indisponibilidade do serviço e do *plugin*.

Conforme citado, a API do *Free Music Archive (FMA)* nos permitiu listar álbuns musicais de acordo com alguns parâmetros passados durante a requisição. Para criarmos nosso sistema de indicações, recuperamos as preferencias do usuário do serviço de cache, fizemos uma requisição por álbuns musicais ao *FMA* e passamos como parâmetros os gêneros preferidos,

retornando uma listagem filtrada. A Figura 19 exibe um álbum do gênero *Brazilian* definido na Figura 17, como exemplo de indicação.

Album Indications

With CODI4In



Chico Correa and Electronic Band

Chico Correa is producer, DJ, and musician Esmeraldo Marques of Paraíba, Brazil. His productions blend traditional Brazilian and new electronic rhythms. This 2006 album is a result of years experimenting fusion of the organic and the electronic. The result is a blend of public domain lyrics using well-known folk melodies from Brazillian states of Paraíba and Pernambuco, electronic music, and jamming with singers, saxophone players, an Afro-Brazilian percussion set, and a rock power trio.

When performing live, Chico Correa and Electronic Band are a six musician band accompanied by a VJ. The ensemble consists in Chico Correa (guitar, grooveboxes, synth), Stephan (sax), Larissa Montenegro (voice), João Cass (percussion), Vitor (drums), Orlando Freitas (bass and pifano flute), and VJ Carlos Dowling.

Artist: Chico Correa and Electronic Band
Tracks: 43

Figura 19. Lista de álbuns indicados de acordo com as preferencias do usuário.

4.6 Testes

Ao final do desenvolvimento dos trabalhos aqui apresentados, realizamos uma carga de testes para analisar o desempenho do serviço de cache e do CODI4In. Para isso, adaptamos um modelo de *benchmarking* apresentado pela IBM [Boyer, 2008], que, através de um ciclo de iterações, calcula a diferença entre o horário inicial e o horário final da rotina executada, considerando, em nosso caso, até a grandeza dos microssegundos. A seguir, detalharemos as etapas percorridas e apresentaremos os resultados obtidos nos dois cenários elencados a seguir.

4.6.1 *Baseline* dos testes com *cache off*

Para realização dos testes envolvendo somente o CODI4In, inicialmente tínhamos apenas um único usuário na CODI-User. Após definirmos suas preferencias, simulamos um cenário no qual 100 requisições *GET* fossem realizadas ordenadamente, para recuperação deste elemento contextual. Ao tentar recuperar a lista de preferencias, o *plugin* verifica, primeiramente, se o usuário encontra-se persistido na CODI-User. Uma varredura por todos os grafos contextuais é realizada em busca do nó do usuário que contenha o login e a senha passados como parâmetros

da requisição. Sendo encontrado o nó, suas preferencias são, então, carregadas e retornadas para o *script* de testes. Este roteiro foi repetido durante três vezes.

Analisando os resultados desta primeira rodada de testes, tivemos como resultado um *tempo médio de latência de 9 segundos*. Entretanto, imaginamos que um único usuário persistido na CODI-User poderia nos trazer resultados imprecisos, visto que o custo para buscar e encontrar um único registro é relativamente baixo. Com isto em mente, resolvemos persistir mais 99 usuários, para totalizarmos 100. Utilizamos a seguinte nomenclatura para o login dos usuários persistidos: “davi” acrescido de seu número, por exemplo: davi27. Também utilizamos o mesmo padrão para as senhas.

Com esta nova carga de usuários, repetimos o mesmo teste por mais três vezes, buscando o primeiro usuário que havia sido persistido, “davi”. Desta vez, notamos um aumento significativo no tempo médio de resposta: *101 segundos*. Duas novas cargas de testes foram realizadas, desta vez, para os usuários “davi49”, que se imaginarmos uma linearidade, encontra-se no meio dos nós, e “davi99”, que foi o último a ser cadastrado. *Para o primeiro, obtivemos um tempo médio de 9 segundos e para o segundo, 23 segundos*.

4.6.2 Execução dos testes com *cache on*

Com o processo *memcached* em execução e a interface de comunicação do serviço de cache em funcionamento, o mesmo modelo de testes do cenário anterior foi utilizado: *100 requisições GET para recuperação das preferencias do usuário*. Para termos um resultado mais preciso, focado exclusivamente na recuperação das preferencias que já se encontram em cache, realizamos uma carga prévia das preferencias dos três usuários - “davi”, “davi49” e “davi99” - para que, durante a primeira iteração, um gasto de tempo extra não fosse desprendido na recuperação dos dados do CODI4In para o serviço de cache.

Na carga de testes para o primeiro usuário, obtivemos um tempo médio de *2 segundos*. *Para “davi49”, 1 segundo e para o último, 2 segundos*. Fica evidente, portanto, que a quantidade de usuários na CODI-User não afeta tão drasticamente o tempo de busca e resposta de elementos contextuais em cache em relação ao uso do exclusivo *plugin*.

A Figura 20 ilustra um gráfico que resume a comparação dos resultados obtidos em ambos os cenários.

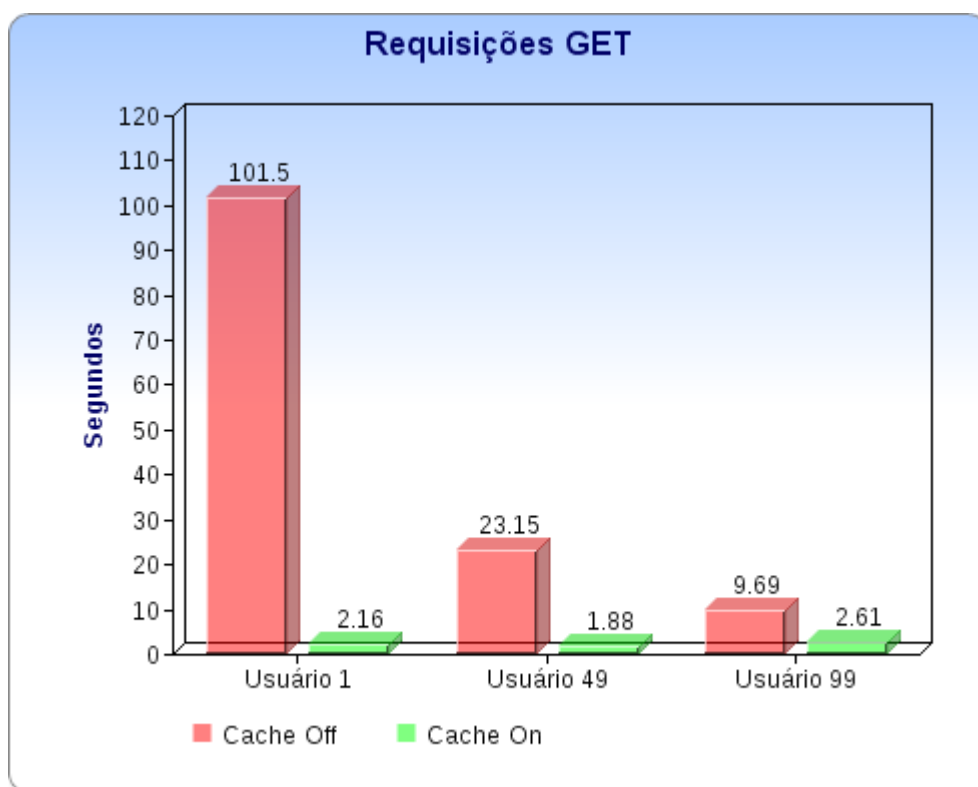


Figura 20. Gráfico comparativo dos resultados obtidos nos testes.

Pelo gráfico, podemos notar que o uso do serviço de cache em detrimento da adoção exclusiva do CODI4In pode, em determinados cenários, resultar em um aumento considerável na performance e, conseqüentemente, na diminuição do tempo de resposta.

5. Considerações Finais

Neste trabalho realizamos um estudo sobre contexto, personalização de consultas e gerenciamento de cache, focado em *Context caching*, utilizando o *memcached* como forma de alcançar este objetivo. O alvo principal foi a criação de um serviço de cache capaz de armazenar e fornecer os elementos contextuais (grafo contextual de usuários) recuperados do *plugin* CODI4In para serem usados por uma aplicação Web ou capturados por meio da aplicação e serem persistidos na CODI-User (ontologia que persiste os elementos contextuais). Também desenvolvemos uma aplicação Web, a qual chamamos IndSong, a fim de validar a interação entre a aplicação, o serviço de cache e o *plugin*.

5.1 Contribuições

As principais contribuições deste trabalho foram:

- Levantamento do estado da arte em relação ao *plugin* CODI4In.
- Identificação e descrição de trabalhos relacionados a este presente trabalho.
- Especificação da arquitetura de um serviço de cache que visa diminuir o tempo de latência entre a requisição feita por uma aplicação e a resposta gerada pelo *plugin*.
- Implementação de um serviço de cache que consome elementos contextuais por meio de requisições *POST* e os fornecem quando solicitados por requisições *GET*, no formato JSON.
- Implementação de uma tarefa cronometrada que mantém os elementos contextuais que estão no serviço de cache sincronizados com o *plugin*.
- Criação de uma aplicação Web capaz de se comunicar remotamente com o serviço de cache e o *plugin* por meio de requisições HTTP.
- Realização de testes de desempenho e de interação entre o serviço de cache e uma aplicação Web.

5.2 Dificuldades encontradas

Devido ao estágio em que se encontra o *plugin* CODI4In, que, ao usar a CODI-User consegue administrar, até o momento, os elementos contextuais *Preference* e *Address*, não pudemos desacoplar totalmente todos os seus recursos, a fim de torna-lo independente do MovieShow. Assim, ao inicializarmos o processo *memcached* e popularmos sua *HashTable* com os dados citados, não conseguimos visualizar seu uso em sua totalidade, armazenando todos os elementos contextuais da ontologia. A pequena quantidade de usuários registrados também se

apresentou como uma dificuldade, visto que a quantidade de preferencias e o endereço estão diretamente relacionados a eles.

5.3 Trabalhos futuros

Paralelamente a este trabalho, está sendo expandido o uso do CODI4In no que diz respeito a aplicação dos elementos contextuais restantes da CODI-User. Também já está sendo desenvolvida uma aplicação que funcionará em um domínio diferente do MovieShow, validando o uso do *plugin* em outros domínios.

O serviço de cache crescerá à medida em que novos recursos forem incrementados ao CODI4In e aumentará sua performance à medida em que novos algoritmos de busca por chaves e armazenamento de objetos sejam implementados.

Referências

Alves, D., Freitas, M., Moura, T., Souza, D. (2013) **“Using Social Network Information to Identify User Contexts for Query Personalization”**, In Proceedings of the The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications. Sevilla, Spain.

Amazon. (2007) **“Context Links (beta): You Create the Content. We'll Link It.”**. Disponível em http://affiliate-blog.amazon.co.uk/2007/06/context_links_b.html. Acessado em 16/02/2013.

Amazon. (2011) **“Docs: Product Advertising API”**. Disponível em <http://aws.amazon.com/archives/Product-Advertising-API/8967000559514506>. Acessado em 16/02/2013.

Aniszczyk, C. (2012) **“Caching with Twemcache”**. Disponível em <http://engineering.twitter.com/2012/07/caching-with-twemcache.html>. Acessado em 28/01/2013.

Bowen, L. (2000) **“Introduction to Contemporary Mathematics – Euler Paths and Circuits”**. Disponível em <http://www.ctl.ua.edu/math103/euler/quick.htm>. Acessado em 05/02/2013.

Boyer, B. (2008) **“Robust Java benchmarking, Part 1: Issues”** Disponível em <http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html>. Acessado em 27/02/2013.

Constine, J. (2013) **“Facebook Is Done Giving Its Precious Social Graph To Competitors”**. Disponível em <http://techcrunch.com/2013/01/24/my-precious-social-graph/>. Acessado em 16/02/2013.

Cordes, K. (2007) **“YouTube Scalability Talk”**. Disponível em <http://kylecordes.com/2007/youtube-scalability>. Acessado em 28/01/2013.

Dantas, J. (2013) **“Evolução do Software”**. IFPB, João Pessoa, Paraíba. 52 slides, PDF. Disponível em <https://academico.ifpb.edu.br/qacademico/index.asp?t=2061>.

Fielding, T. (2000) **“Architectural styles and the design of network-based software architectures”** PhD Thesis, University of California, Irvine, 2000.

Fizpatrick, B. (2004) **“Distributed Caching with Memcached”**. Disponível em <http://www.linuxjournal.com/article/7451>. Acessado em 16/02/2013.

Freitas M., Silva J., Bandeira D., Mendonça A., Souza D. (2012a) **“CODI4In – Um Plugin para a Persistência de Informações Contextuais do Usuário em Ambientes de Consultas”**, Em Anais do Connepi 2012, Palmas.

Fuller, C. (2007) **“Synchronous versus Asynchronous”**. Disponível em http://fuller.mit.edu/tech/sync_asynchronous.html. Acessado em 16/02/2013.

Gruber, T. (1993) **“A Translation Approach to Portable Ontologies”**, In: Knowledge Acquisition, v. 5, n. 2, pp. 199-220.

Gupta, A. (2008) **“Scaling Wikipedia with LAMP: 7 billion page views per month”**. Disponível em https://blogs.oracle.com/WebScale/entry/scaling_wikipedia_with_lamp_7. Acessado em 28/01/2013.

Koutrika G., Ioannidis Y. (2004), **“Personalization of Queries Based on User Preferences”**. Preferences 2004, Dagstuhl, Germany.

Kuenning, G. (2010) **“Cache Memories”**. Disponível em <http://www.cs.hmc.edu/~geoff/classes/>. Acessado em 16/02/2013.

Levandowski, J., Khalefa, M.M. (2011) **“The CareDB Context and Preference-Aware Database System”**, In Proceedings of the International Workshop on Personalized Access, Profile Management, and Context Awareness in Databases. PersDB, Seattle.

Martin, M., Roth, A. (2012) **“Unit 6: Caches”** https://www.cis.upenn.edu/~cis501/lectures/06_caches.pdf. Acessado em 26/01/2013.

Oracle. (2011) **“The Art of Data Replication: An Oracle Technical Paper”**. Disponível em <http://www.oracle.com/technetwork/articles/systems-hardware-architecture/o11-080-art-data-replication-487868.pdf>. Acessado em 15/01/2013.

Picard, A. (2011) **“The history of Twitter, 140 characters at time”**. Disponível em <http://www.theglobeandmail.com/technology/digital-culture/social-web/the-history-of-twitter-140-characters-at-a-time/article573416/>. Acessado em 27/01/2013.

Priberam. (2013). Disponível em <http://www.priberam.pt/dlpo/default.aspx?pal=personaliza%C3%A7%C3%A3o>. Acessado em 26/01/2013.

Saab, P. (2008) **“Scaling memcached at Facebook”**. Disponível em https://www.facebook.com/note.php?note_id=39391378919. Acessado em 01/12/2012.

Sagui, F., Chesñevar, C., Lorenzetti, C., Maguitman, A. e Guillermo R. Simari. (2006) **“Exploiting User Context and Preferences for Intelligent Web Search”**. Proceedings of the Workshop de Investigadores en Ciencias de la Computación (WICC 2006).

Smith, A. (1982) **“Cache Memories”**, Universidade da California, Berkley, California.

Souza, D., Belian, R., Salgado, A. C. and Tedesco P. (2008) **“Towards a Context Ontology to Enhance Data Integration Processes”**, In Proceedings of the 4th Workshop on Ontologies-based Techniques for Databases in Information Systems and Knowledge Systems (ODBIS), VLDB, p. 24-30, New Zealand, August.

Strang, T., Linnhoff-Popien, C. (2004) **“A Context Modelling Survey”**. In: First International Workshop on Advanced Context Modeling, Reasoning and Management. Nottingham, England.

Tanca, L, Bolchini C., Quitarelli E., Schreiber F. (2011) **“Problems and Opportunities in Context-Based Personalization”**, In Proceedings of PersDL, VLDB Workshops, Seattle, WA, EUA.

Teoh, S. (2010) “**Cache policies**”. San José State University, San José, California. 14 slides, PPT. Disponível em http://www.cs.sjsu.edu/~teoh/teaching/previous/cs147_sm10/lectures/lecture12b_slides.ppt.

Vieira, V., Tedesco, P., e Salgado, A. (2010) “**Designing Context-Sensitive Systems: An Integrated Approach**”. *Expert Systems with Applications*, vol. 38(2), p.1119-1138.

Zhou, Y., Chen, Z. e Li, K., (2004) “**Second-Level Buffer Cache Management**”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 15(6), p.505-519.

Apêndices

Apêndice A – Instalação do processo *memcached*

O pacote *memcached* pode ser instalado no ambiente Linux utilizando a seguinte linha de comando: `apt-get install memcached`. Uma vez instalado, o processo é inicializado com o seguinte comando: `memcached -d -m 512 -u root`. Esta linha de comando recebe alguns parâmetros, dos quais salientamos o `-m 512`. Por padrão, o *memcached* roda na porta 11211, utilizando 64 MB de memória RAM. Este parâmetro, entretanto, permite que definamos explicitamente a quantidade de memória alocada.