# Proposta para Atualização de SGBDs para Aplicações com Demanda de Contínua Disponibilidade usando Suporte do Modelo de Componentes de *Software*

**Cleandro Flores De Gasperi, Marcia Pasin**[1]

[1]Programa de Pós-Graduação em Informática (PPGI)
Universidade Federal de Santa Maria (UFSM)
Av. Roraima 1.000 – 97.105-900
Cidade Universitária – Santa Maria – RS – Brasil

cleandro@cpd.ufsm.br, marcia@inf.ufsm.br

***Abstract.*** *Database management system (DBMS) is a critical part in enterprise software and its on-the-fly updating is not trivial due its inherent complexity and requirements such as continuous availability. Current solutions to enterprise software update are costly and involve human intervention. In this paper, we propose the use of component model as a support to achieve continuous availability in dynamic software update (DSU) in DBMS. The component model allows to encapsulate implementation details, which is an interesting property to software replacement. This solution provides as benefits the absence of additional hardware and reduced service unavailability. To validate our proposal, a prototype was developed using Fractal component model. The experimental evaluation confirms the effectiveness of our approach but, as expected, indicates that the component model itself is still a bottleneck.*

***Resumo.*** *Sistema de gerenciamento de banco de dados (SGBD) é parte fundamental do software corporativo e sua atualização dinâmica não é simples devido à sua complexidade. Frequentemente, atualização de SGBD é manualmente executada e está associada à indisponibilidade de serviço e uso de hardware adicional. Este artigo propõe o uso de modelo de componentes de software como alternativa para a aplicação de atualização dinâmica de software (ADS) em um SGBD. O modelo de componentes apresenta um nível de abstração que favorece a troca de componentes pois oculta detalhes de implementação. ADS apresenta como benefícios a ausência da necessidade de hardware adicional e da indisponibilização do serviço. Para validação da proposta foi desenvolvido um protótipo utilizando o modelo de componentes Fractal. Testes em ambiente controlado confirmam a viabilidade da solução mas indicam que o próprio modelo de componentes ainda é um gargalo.*

## 1. Introdução

A tarefa de atualização de *software* é necessária porque apesar do contínuo esforço no desenvolvimento e uso de novas ferramentas e técnicas de programação, o código está sempre sujeito ao envelhecimento e à existência de *bugs*. Para sobreviver, um *software* está em continuo desenvolvimento. Novas distribuições e versões são sistematicamente

disponibilizadas para corrigir eventuais *bugs* e implementar necessidades não cobertas pela implementação anterior.

Contudo, a experiência mostra que o processo de atualização do *software* pode sofrer problemas como falha durante a atualização (causando indisponibilidade parcial ou total do sistema), indisponibilização de serviço durante a atualização (ainda que em um cenário livre de falhas), inserção de novos *bugs* após atualização, etc. Na prática, esses problemas são tratados de forma restrita ou ignorados. O ideal seria que sistemas computacionais permitissem mecanismos automáticos, com a implementação de serviços para garantia de um conjunto amplo de propriedades específicas. Neste contexto, Atualização Dinâmica de *Software* (ADS) aparece como uma resposta.

ADS [Stoyle et al. 2007] é o mecanismo que possibilita substituição automática de programas mantendo a disponibilidade do serviço. Existem soluções para ADS que contemplam diferentes áreas da computação como sistemas operacionais (atualização automática do *Windows*, *Mac OS X*, distribuições de Linux), linguagens de programação (substituição de funções em linguagem C [Segal e Frieder 1993]) e engenharia de *software* (notadamente soluções com troca de componentes [Leger et al. 2007, Wang et al. 2006]). Entretanto, muitas soluções adotadas na prática, como aquelas aplicadas a sistemas operacionais, ainda necessitam interferência humana ou reinicializarão do serviço para disponibilizar novas atualizações.

O conjunto de propriedades que sistemas com ADS deveriam garantir foram apresentadas anteriormente em [Segal e Frieder 1993]. Estas propriedades contemplam $(i)$ transparência para ocultar ao usuário a ocorrência de uma atualização do sistema, $(ii)$ automação de forma a minimizar intervenção humana evitando erros e tornando o processo de atualização mais confortável para o usuário, $(iii)$ suporte para reestruturação de código possibilitando a inclusão e remoção de módulos, $(iv)$ descarte da necessidade de *hardware* adicional evitando maximização de custos, $(v)$ ausência de restrição de linguagem e ambiente, flexibilizando portabilidade para diferentes ambientes operacionais e linguagens, $(vi)$ facilidade de manutenção, pois a evolução de *software* ocorre de forma contínua, e $(vii)$ disponibilidade contínua pela troca de componentes ocorre sem interrupção total de serviço. A implementação dessas propriedades não é tarefa trivial e envolve diferentes aspectos em um projeto complexo. Por isso, sistemas com ADS focam em um conjunto restrito de propriedades.

Outro grande desafio, no contexto de ADS, é a ausência de soluções genéricas e automatizadas para ambientes mais amplos e complexos, como sistemas corporativos e $clouds$. Apesar de ADS não ser uma novidade (a pesquisa mais remota data da década de 1970 [Fabry 1976]), seu uso em aplicações empresariais é limitado. Servidores de aplicação e sistemas de gerenciamento de banco de dados (SGBD) integram parte fundamental do *software* de sistemas corporativos e empresariais e sua atualização não é trivial. Sistemas empresariais requerem soluções para tratar escalabilidade, elasticidade, dinamismo, heterogeneidade e, portanto, complexidade. Tipicamente, aplicações são implementadas em múltiplos *tiers* onde cada *tier* fornece um serviço específico. Tratar tais adversidades sem comprometer a disponiblilidade e desempenho é tarefa complicada. Nesses ambientes, manutenção e atualização são tarefas manuais ou parcialmente automatizadas.

Frequentemente soluções para atualização de *software* adotadas por instituições de

grande porte estão focadas no uso de *hardware* adicional (o serviço executa em um conjunto de máquinas auxiliares enquanto o sistema principal é atualizado) e na interferência humana. Esta estratégia impacta sensivelmente no custo do processo de substituição de *software* e envolve amplo planejamento de uma equipe para que o serviço não seja interrompido. Instituições de médio e pequeno porte tipicamente optam por mecanismos simples (parada total do serviço) mas comprometem a disponibilidade das aplicações. A solução mais adequada seria aplicada de forma automática, com baixo custo de execução e com possibilidade de realizar atualização de *software* de forma gradativa, sem interromper o serviço para os clientes e sem uso de *hardware* adicional. Adicionalmente, a reduzida interferêcia humana evitaria a inclusão de erros durante esse processo.

Neste sentido, este trabalho apresenta um passo à frente na atualização de *software* para ambiente empresarial e propõe o uso do modelo de componentes de *software* como suporte para aplicação de ADS em SGBD. Esta proposta é inédita. Um SGBD é um *software* complexo que possui muitos módulos executando tarefas importantes: otimização de consultas, gerenciamento transacional, provimento de persistência, controle de acesso concorrente, entre outros. No contexto de aplicações e sistemas empresariais, o modelo de componentes de *software* aparece como alternativa para tratar a complexidade inerente, mas, em contra-partida, adiciona o *overhead* da implementação da própria abstração de componentes. Um componente de *software* é uma unidade independente e encapsulada que possibilita ocultar detalhes de implementação. Destaca-se que, de fato, esta solução é interessante porque a ADS se beneficia amplamente do modelo de componentes de *software*. Como o sistema é representado através de um conjunto de componentes independentes, um componente pode ser substituído enquanto os demais continuam a operar, ainda que a disponibilidade do serviço como um todo seja reduzida e sem a necessidade de *hardware* adicional. Outro benefício é que esta solução é genérica e pode ser reaproveitada em diversas instalações.

Apesar da limitação tecnológica (não existe atualmente implementação de SGBD no modelo de componentes), o fato do SGBD ser modelado como um conjunto de componentes possibilitaria substituição gradual de *software* sem interrupção total do serviço. O modelo de componentes permite que seja associado ao sistema-alvo um mecanismo de descrição de arquitetura [Bass et al. 2003] favorecendo a análise das associações entre módulos do SGBD. Uma discussão sobre os benefícios do uso do modelo de componentes e da descrição de arquitetura para atualização de *software* é apresentada em [Oreizy et al. 1999]. Neste trabalho, o modelo de componentes Fractal [Bruneton et al. 2006] é usado para implementar a descrição da arquitetura.

Mais especificamente, este trabalho objetiva: $(i)$ propor um modelo genérico para representar um SGBD visando a arquitetura de *software* baseada no modelo de componentes de Fractal que permita a substituição destes, $(ii)$ apresentar uma lógica para a substituição gradativa de componentes sem parada total do serviço para os clientes, sem a necessidade de *hardware* adicional, e que não onere o desempenho esperado demasiadamente, $(iii)$ codificar um esboço de um protótipo para mostrar a viabilidade desta solução.

O restante do artigo está organizado como segue. Seção 2 discute trabalhos relacionados. Seção 3 apresenta a proposta de ADS para SGBD com suporte do modelo de componentes. Implementação de protótipo e o modelo de componentes Fractal são

descritos na seção 4. A avaliação experimental, conduzida através do protótipo implementado, e descrição dos resultados são apresentadas na seção 5. A seção 6 conclui o artigo apresentando considerações finais e perspectivas para trabalhos futuros.

## 2. Trabalhos Relacionados

De acordo com [Wahler et al. 2009], existem dois principais tipos de soluções para ADS: soluções baseadas em rotinas e soluções baseadas em componentes. Ainda pode ser acrescentado um terceiro tipo que agrega soluções baseadas em arquitetura de *software*. Este último tipo permite não apenas o controle dos componentes mas também das vinculações entre os componentes, o que representa um suporte interessante para implementação de ADS.

Sistemas para atualização de *software* baseados em rotinas são atrelados a detalhes de implementação, o que limita a portabilidade dessas soluções. Um exemplo de solução nesta linha é PODUS. PODUS [Segal e Frieder 1993] é uma ferramenta capaz de promover ADS na camada do sistema operacional. Uma rotina em execução somente é atualizada quando estiver inativa. A atualização inicia carregando em outro espaço de memória novas versões das rotinas a serem atualizadas. A seguir o programa é interrompido e a sua pilha de execução é analisada. Então, redirecionamentos são executados através de uma tabela de indireção. Na próxima vez que a rotina for chamada, será direcionada para a nova versão descrita na tabela. Quando todas as rotinas previstas na atualização estiverem na nova versão, o programa é considerado atualizado. [Lyu et al. 2001] usa solução análoga mas difere porque a modificação de rotina ocorre de forma direta, sem precisar de uma tabela de indireções. Em outras palavras, dentro do código da versão antiga da rotina estará o endereçamento para a nova versão. ADS é realizada por chamadas diretas do sistema operacional, sem necessidade de *software* adicional. Finalmente, PROTEUS [Stoyle et al. 2007] oportuniza ADS em linguagens *C-like*, e atualização pode ser aplicada para tipos de dados e funções. A nova versão é escrita com PROTEUS, onde são explicitados tipos e funções a serem substituídas através de sintaxe específica. As atualizações são realizadas em tempo de execução.

Contrastando com soluções baseadas em rotinas, soluções baseadas em componentes ocultam detalhes de implementação e são mais genéricas. Um exemplo de serviço para ADS baseado no modelo de componentes e servidores de aplicação é [Wang et al. 2006], focando em objetos (mas especificamente Enterprise JavaBeans) que executam em servidores de aplicações. Outra proposta interessante [Leger et al. 2007] descreve a substituição de componentes genéricos aplicando as propriedades transacionais ACID para garantir um conjunto de propriedades requeridas pela ADS (reportadas neste artigo no item anterior). A ideia é substituir componentes dentro de um contexto transacional. Atualizações não executadas com sucesso podem ser revertidas com o auxílio de informações armazenadas em um *log*.

Existem soluções para ADS que adotam um modelo de maior granularidade que os apresentados anteriormente, com foco na descrição de arquitetura do sistema [Oreizy et al. 1998]. O principal diferencial de soluções arquiteturais está na possibilidade de reutilização da solução em outras implementações, desde que o *software* esteja de acordo com a arquitetura. O mesmo se aplica a soluções que seguem o modelo de componentes. Uma vantagem do modelo com descrição de arquitetura, em relação ao modelo

de componentes, é o suporte para o controle das dependências entre os componentes. Este suporte, que facilita o mapeamento de dependências, é crucial para a atualização de *software* por três motivos: $(i)$ se um *software* depende de outro, a substituição deste código pode implicar na substituição de outro código, $(ii)$ pelo suporte para tratamento de dependências externas (substituição de um *software* sendo correntemente usado por outro componente), e por possibilitar a substituição de um componente (primitivo) por dois ou mais componentes (componente composto) e vice-versa.

Todas as soluções existentes, sejam elas baseadas em rotinas, no modelo de componentes ou na descrição da arquitetura, implementam um conjunto restrito das propriedades requeridas para atualização de *software*. Em linhas gerais, soluções atreladas a rotinas e detalhes de implementação são mais eficientes em questão de desempenho mas sua portabilidade para outras plataformas é limitada. Em contraste, soluções baseadas em componentes de *software* e descrição de arquitetura são mais genéricas, mais flexíveis e com maior possibilidade de reaproveitamento em outros contextos.

Linguagens de programação também apresentam características positivas para a implementação de ADS. Em Java existem dois mecanismos importantes: $(i)$ carga da classe (a partir do arquivo *.class*) e $(ii)$ reflexão computacional. Como a carga de classe pode ser feita dinamicamente, a atualização de *software* pode ser vista como o carregamento de uma nova versão da classe antiga. Reflexão computacional é uma técnica de programação onde um sistema pode atuar sobre sua própria computação e se adaptar em presença de condições de mudança. A substituição de um componente antigo por outro componente é um tipo de adaptação. Entretanto, mecanismos importantes como tratamento de dependências entre componentes e transferência de estado ainda precisam ser tratados de forma explícita pelo programador. Finalmente, na literatura ainda existem outras soluções interessantes para atualização de *software*, mas todas elas tem restrições. Por exemplo, a plataforma OSGi permite suporte para substituição de módulos Java porém, como acontece em Java puro, o programador precisa mapear as dependências entre os módulos.

## 3. Proposta para Atualização de SGBD com o Modelo de Componentes

Esta seção apresenta a proposta genérica para a substituição de componentes sem parada total do SGBD e sem uso de *hardware* adicional. O elemento-chave desta proposta é o Gerenciador de Atualizações (GA) que executa a substituição de módulos do SGBD. A seguir, primeiramente são apresentadas limitações que possibilitaram estabelecer esta proposta de trabalho na tecnologia atual e em espaço tão curto de tempo e com reduzida equipe de desenvolvimento, e depois, de forma mais específica, é descrita a arquitetura do $GA$.

### 3.1. Limitações da proposta na tecnologia atual

Conforme comentado anteriormente, para que componentes do SGBD possam ser substituídos com o suporte do modelo de componentes Fractal ou outro modelo de componentes, é necessária a construção de um SGBD de acordo com o modelo de componentes de *software*, um tipo de construção atualmente inexistente. Embora existam implementações construídas de acordo com o paradigma de orientação a objetos (um exemplo é o HyperSQL), SGBDs atuais, em sua maioria, são construídos seguindo o modelo orientado

a processos. Estas implementações, se conjugadas com *frameworks* para construção de sistemas com o conceito de componentes de *software*, permitem obter uma solução compatível à proposta deste trabalho. Para tanto, possivelmente, será necessária a reescrita do SGBD (trabalho para uma grande equipe de desenvolvimento), de tal forma que os objetos que implementam seus módulos serão encapsulados em componentes.

Entretanto, para avaliar a proposta aqui apresentada, esta limitação foi contornada através do mapeamento dos módulos tradicionais que compõem um SGBD para componentes de *software*. A estrutura básica de um SGBD é descrita na literatura de sistemas de bancos de dados (veja [Elmasri e Navathe 2005], [Ramakrishnam e Gehrke 2003]). Aplicar a abstração de componentes de *software* sobre a estrutura do SGBD, encapsulando cada módulo em um componente é um caminho natural e, muito provavelmente, implementações futuras de SGBDs comerciais sigam o modelo de componentes, desde que esta abstração está se tornando muito popular na indústria de *software*. Assim, uma vez que módulos estarão encapsulados em componentes, operações de controle definidas (por Fractal, por exemplo) podem ser usadas para substituir dinamicamente componentes de *software*.

Outras limitações desta proposta são $(i)$ execução de atualização de SGBD em um único nó (apenas um SGBD central é atualizado) e $(ii)$ obrigatoriedade de manutenção da interface dos componentes. Entretanto, a solução aqui apresentada pode ser ampliada para suportar atualização em um ambiente mais complexo, com múltiplas instâncias de SGBDs executando simultaneamente. Finalmente, a obrigatoriedade de manutenção da interface dos componentes não invalida esta proposta e também foi utilizada em trabalho anterior [Wang et al. 2006]. De fato, muitas atualizações são feitas para consertar pequenos *bugs* (preservando a interface de serviços), não impactando significativamente no serviço oferecido pelo módulo ou na estrutura interna do sistema.

### 3.2. Arquitetura e Serviços

O gerenciador de atualizações ($GA$) possibilita que a atualização de componentes do SGBD seja realizada de forma gradativa e automática. Uma visão simplificada da arquitetura do $GA$ é mostrada na Figura 1.

[Oreizy et al. 1999] destaca uma série de serviços importantes na atualização de *software*, que aqui são implementados pelo $GA$. Mais especificamente, o $GA$ controla a versão dos componentes, identifica o momento adequado para realizar substituições, garante integridade estrutural do sistema, controla a demanda das requisições dos clientes e realiza a substituição de componentes propriamente dita. Esses serviços, executados por diferentes módulos, são brevemente abordados na sequência deste texto.

#### 3.2.1. Serviço Gerenciador de Componentes

O controle de versão é uma tarefa fundamental para o funcionamento adequado da ADS e organiza as mudanças das informações dos componentes. No modelo aqui proposto, o $GA$ utiliza um *log* para registrar componentes (módulos do SGBD) que foram substituídos, que representam uma variação particionada de um repositório. Ou seja, o próprio SGBD representa a última revisão do repositório. Em conjunto com o *log*, o $GA$ usa uma tabela de versionamento. O $GA$ identifica unicamente cada componente para o adequado

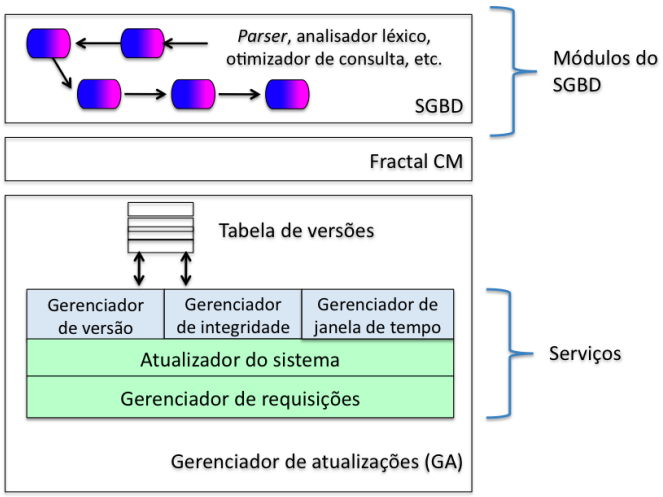


**Figura 1. Esboço da arquitetura do GA**

controle das versões, e armazena na tabela de versões informações sobre o a identificação do componente, o tipo de componente, e a versão que obedece uma ordem cronológica de finalização do componente, ou seja, o instante que o componente é liberado para a atualização no sistema. Internamente, o $GA$ também mantém informação sobre as respectivas interfaces de cada componente. A tabela de versões tem três aplicações principais: $(i)$ serve como base para recuperação do sistema, em caso de queda, $(ii)$ facilita o processo de atualização dos componentes, pois mantém a versão atual de cada componente, e $(iii)$ suporta um esquema de validação da interface implementada pelo novo componente.

### 3.2.2. Serviço Gerenciador de Janela de Tempo

O processo de atualização de componentes não deve onerar demasiadamente a atividade normal do sistema. Oportunidades para perceber o momento adequado para a realização desta tarefa devem ser continuamente buscadas. Se o serviço para os clientes estiver sofrendo grande demanda, a sobrecarga gerada pela atualização, mesmo que pequena, pode ser o acréscimo suficiente para comprometer a qualidade do serviço. Visando evitar sobrecarga, o $GA$ identifica uma janela de tempo adequada para realizar a substituição de um componente. Informações obtidas por sensores, como taxa de ocupação da CPU, número total de transações ativas e estimativa de custo de transações podem ser usadas para mensurar a carga do sistema. Comparando estas medidas ou uma combinação delas com um parâmetro previamente definido e acessível ao $GA$, poderá ser realizada a escolha de uma janela de tempo adequada para efetuar o processo de atualização.

### 3.2.3. Serviço Gerenciador de Integridade

Componentes de *software* possuem interfaces bem definidas. A interface é uma coleção de operações que define os serviços disponibilizados. Assim, para garantir a integridade estrutural do sistema, é suficiente e necessário que o novo componente tenha a mesma interface do componente que está sendo substituído. É importante, salientar que é uma

validade de integridade estrutural e não comportamental do componente. O novo componente oferece as mesmas operações do antigo mas não existem garantias quanto ao seu funcionamento, ou seja, quanto aos resultados que serão produzidos por suas operações. Na proposta atual, a manutenção da integridade do sistema é provida através da implementação de uma tabela de versões dos componentes. Nesta tabela, o $GA$ identifica quais interfaces o componente deve implementar. Então, pode ser assegurado que o novo componente implementa uma interface idêntica ou ao menos compatível com o antigo componente.

#### 3.2.4. Serviço Gerenciador de Requisições de Clientes

Para [Wang et al. 2006] o cumprimento da premissa de que garante disponibilidade contínua significa que nenhuma requisição de cliente pode ser recusada durante o processo de ADS. Dessa forma, o $GA$ intercepta e controla requisições encaminhadas ao sistema para assegurar a não interrupção de serviço para os clientes. Ao iniciar o processo de atualização de um componente, o $GA$ espera requisições ativas finalizarem e enfileira as novas requisições, em um procedimento similar a [Wang et al. 2006]. Então, a ADS é executada. Após finalizar a ADS, o $GA$ repassa a fila de requisições bloqueadas ao componente apropriado do SGBD.

## 4. Implementação

Para comprovação da viabilidade técnica desta proposta e serviços previamente descritos foi desenvolvido um esboço de um protótipo construído em linguagem *Java* 1.6 e a implementação de referência de Fractal *Julia* 2.5.1. O ambiente de desenvolvimento e compilação adotado foi o *IntelliJIDEA* 10.0.3 *Community Edition* da *JetBrains*. Uma breve descrição do modelo de componentes Fractal é apresentada no item 4.1. O item 4.2 descreve o mapeamento do modelo relacional adotado tipicamente por SGBDs para o modelo de componentes, uma simplificação para possibilitar a implementação do protótipo. O item 4.3 descreve a criação do ambiente do componente principal (chamada de $Control$), que implementa um esqueleto do $GA$ e seus respectivos serviços descritos anteriormente. Finalmente, os algoritmos para substituição de componentes propriamente ditos são descritos.

### 4.1. Modelo de Componentes Fractal

Fractal [Bruneton et al. 2006] é um modelo de componentes modular, extensível com suporte a várias linguagens de programação (Java e C, e de forma experimental para .NET, SmallTalk e Python). Fractal permite projetar, implementar, executar e reconfigurar dinamicamente sistemas e aplicações. Um componente Fractal é um elemento de execução encapsulado, com identificação única e que suporta uma ou mais interfaces. A interface é um ponto de acesso para um componente e que implementa uma interface da linguagem, que representa as operações suportadas.

Uma interface de um componente Fractal pode ser de dois tipos: interface cliente (ou requerida) e servidora (ou provida). Um componente usa a interface cliente para invocar operações implementadas por outros componentes. Um componente usa a interface servidora para invocar operações implementadas pelo próprio componente.

Genericamente, um componente Fractal possui duas camadas, a interna e a externa. A camada externa (ou membrana) possui as interfaces de controle que permitem introspecção e reconfiguração de componentes. A camada interna (ou conteúdo) consiste em um conjunto finito de outros (sub)componentes. As interfaces da membrana podem ser internas (acessíveis apenas pelos subcomponentes) e externas (aquelas acessíveis por outros componentes).

Em Fractal, componentes são conectados a outros componentes por interfaces de vinculação, que podem ser de dois tipos: primitivas ou compostas. Em uma vinculação primitiva, ou em um componente primitivo, uma chamada a uma interface cliente resulta diretamente na chamada da interface vinculada de servidor. Na vinculação composta, ou componente composto, a chamada envolve um ou mais componentes pois as interfaces de cliente e servidor não combinam (e é necessário executar algum tipo de conversão) ou porque os componentes conectados são hospedados em máquinas distintas.

Componentes tem controladores que são usados para acessar operações internas (do próprio componente) e operações externas (em outros componentes). Existem quatro tipos de controles em Fractal:

- **Ciclo de vida**: possibilita a reconfiguração dinâmica dos componentes, pois trata explicitamente da disponibilidade dos mesmos. Há duas operações disponíveis: *startFc* e *stopFc*. Basicamente, *startFc* ativa o componente e *stopFc* pára o componente.
- **Vinculação**: gerencia as dependências entre os componentes. Para realizar a remoção de um componente do sistema, necessariamente, o controle de vinculação deve ser utilizado a fim de promover a desconexão entre os componentes, e a futura religação entre eles.
- **Conteúdo**: realiza o controle de conteúdo para permitir da adição e remoção de subcomponentes. As operações disponíveis neste controle são *addFcSubComponent* e *removeFcSubComponent*.
- **Atributo**: controla atributos para configurar propriedades nos componentes. Tipicamente, atributos são tipos primitivos utilizados para configurar o estado de um componente.

Neste contexto, uma ADS pode ser vista como uma coleção de operações efetuadas pelos controladores de Fractal. Para que a ADS ocorra, um componente deve ser parado pelo controlador do ciclo de vida. Então, o controlador de vinculações trata dependências entre os componentes. Em seguida, o controlador de conteúdo remove a versão antiga e instala a nova versão do componente. Finalmente, o componente é reativado pelo controlador do ciclo de vida.

### 4.2. Mapeamento do Modelo Relacional para o Modelo de Componentes

Para realizar a substituição de componentes via Fractal, componentes do SGBD precisam ser descritos pelo modelo de componentes Fractal. Como a implementação de um SGBD de acordo com o modelo de componentes não existe, foi realizado um mapeamento dos módulos de um SGDB para componentes caixa-preta. A implementação sobrescreveu de forma mais fiel possível a estrutura básica de um SGBD. A classe raiz implementada pelo protótipo do SGBD é $Componente$, onde todos os subcomponentes do SGBD serão derivados. A classe $SgbdEmComponentesFractal$ funciona como porta

de entrada das requisições ao SGBD. As classes $Parser$, $Optimizer$, $ExecutionPlan$, $OperatorEvaluator$, $ExecutionEngine$, $TransactionManager$, $RecoveryManager$, $LockManager$, $FileAccessAndMethods$, $BufferManager$ e $StorageManager$ simulam os módulos componentes do SGBD.

Uma outra limitação deste procedimento sobre a tecnologia atual é a impossibilidade de um componente Fractal operar como servidor para mais de um componente. Para contornar esta restrição, a estrutura básica do SGBD sofreu uma simplificação do modelo, usando sequencialização das atividades, ou seja, um componente somente requisita serviço de um componente e provê serviço para um componente de cada vez.

### 4.3. Criação do Ambiente em Fractal

$Control$ é a implementação simplificada do $GA$. A classe $Control$ usa o método $getBootstrapComponent$ para criar o componente raiz que coordena todo o ambiente. A fábrica de tipos é criada pelo método $getTypeFactory$ e os tipos são criados com o método $createFcType$. A fábrica dos demais componentes é criada com o método $getGenericFactor$. Com o método $newFcInstance$ são criadas novas instâncias de componentes Fractal. A composição dos componentes (ou adição de um subcomponente a um componente composto) é realizada com o método $addFcSubComponent$. A vinculação dos subcomponentes entre si é possibilitada pelo método $bindFc$. O ambiente criado é ativado com o método $startFc$.

### 4.4. Algoritmos para Atualização de um Subcomponente

O processo de substituição de um (sub)componente é descrito por dois algoritmos. O Algoritmo 1 descreve o modo de operação do $GA$ e o Algoritmo 2 descreve a substituição de componentes propriamente dita.

A substituição de componentes é uma operação permanente, ou seja, enquanto o sistema estiver em operação, substituições de subcomponentes poderão ocorrer. Resumidamente, o Algoritmo 1 executado pelo $GA$ verifica continuamente a fila de novos componentes (linha 2). A partir da existência de um novo componente nesta fila, a substituição de fato do subcomponente é iniciada. O novo componente então é identificado e seu tipo é validado (linha 5): o novo componente deve ser mais recente que o atual (via **Serviço Gerenciador de Versão**) e deve implementar todas as operações da interface do antigo componente (via **Serviço Gerenciador de Integridade**, linha 9).

Na sequência ocorre a decisão sobre a janela de tempo adequada para ocorrer a atualização (via **Serviço Gerenciador de Janela de Tempo**, linha 11). Como já mencionado anteriormente, a substituição do subcomponente somente acontece se o impacto sobre o sistema não interferir demasiadamente na qualidade do serviço. Em períodos de maior carga do sistema, a operação de atualização deve ser evitada. Para avaliação da carga do sistema, o $GA$ utiliza informações como número de requisições ativas e o custo destas requisições para decidir pela substituição do subcomponente.

O ato de substituir um subcomponente começa pela parada do componente raiz, de acordo com Algoritmo 2 (linha 1). A partir deste instante, todos os subcomponentes não executarão mais as suas funções especificadas. Requisições recebidas após a parada de um componente são tratadas pelo **Serviço Gerenciador de Requisições** e executadas

**Algoritmo 1:** Serviços executados pelo GA

```
1: while true do
2:     Ler componente da fila_de_novos_componentes
3:     if Existe componente then
4:         Ler tipo do componente
5:         if Tipo do componente é valido then
6:             Envia componente para Serviço Gerenciador de Versão
7:             if O componente é mais novo que componente instalado then
8:                 Validar interface do componente via Serviço Gerenciador de Integridade
9:                 if O novo componente tem a interface válida then
10:                    Solicita janela para Serviço Gerenciador de Janela de Tempo
11:                    if Sistema pode ser atualizado then
12:                        Serviço Gerenciador de Requisição substitui componente
                           { executa Algoritmo 2 }
13:                    end if
14:                end if
15:            end if
16:        end if
17:    end if
18: end while
```

ao término deste processo. Segue-se então a desconexão de componentes e subcomponentes (linha 2). Então, o componente é removido (linha 3), ou seja, deixa de ser um subcomponente da arquitetura. O novo componente então é adicionado (linha 4). Neste instante ainda não está apto a funcionar, pois precisar ser informado sobre suas vinculações. Conectado o novo componente, então o componente raiz pode ser iniciado (linha 6). Faltam apenas operações de controle como a atualização da tabela de controle das versões (via **Serviço Gerenciador de Versão**) e a movimentação do componente para o *log* de componentes (linhas 7 e 8).

**Algoritmo 2:** Substituição de componente

```
1: stopFc root { parar componente raiz }
2: unbindFc component { desvincular subcomponente a ser substituído }
3: removeFcSubComponent { remover subcomponente }
4: addFcSubComponent { adicionar novo subcomponente }
5: bindFc component { vincular novo subcomponente dos demais }
6: startFc root { ativar componente raiz }
7: Mover componente antigo para log
8: Atualizar tabela de versões
```

## 5. Avaliação Experimental

Para avaliar a possibilidade da atualização de SGBD através do modelo de componentes de *software*, foi executado um conjunto de testes sobre um protótipo que simula o esqueleto do $GA$ envolvendo diferentes cenários e diferentes sobrecargas. A avaliação experimental objetivou estimar o custo da substituição de componentes em um protótipo que executa a substituição automática de componentes de um SGDB durante a execução do serviço para os clientes.

Testes foram realizados em uma máquina virtual *Oracle Virtual Box* 4.0.4, com sistema operacional *Linux Ubuntu* 10.10, sendo a memória base de 512 MB. Este ambi-

ente executa sobre uma máquina real com processador *Intel Core 2 Duo T8300*, 2 GB de memória RAM e sistema operacional *Windows XP Professional* 2002 SP 2.

A avaliação considerou três cenários distintos: $(i)$ **Sem Fractal**, ou seja, execução de protótipo de SGBD construído de forma tradicional (orientado a objetos), portanto sem Fractal (para avaliar o sistema base, isto é, o servidor de banco de dados sem atualização de componentes, $(ii)$ **Fractal sem ADS** , ou seja, execução de protótipo de SGBD construído na perspectiva de componentes de *software* baseado em Fractal, mas sem a realização de ADS, para avaliar o custo do serviço Fractal, e $(iii)$ **Fractal com ADS**, ou seja, execução de um protótipo de SGBD construído na perspectiva de componentes de *software* baseado em Fractal, mas com a realização de ADS, para avaliar o custo de Fractal, com a substituição de componentes.

Os testes realizados consistiram da execução de grupos de requisições de clientes e obtenção do número de transações processadas por segundo. Os grupos definidos para os testes foram 100, 500, 1.000 e 3.000 requisições emitidas, respectivamente. O ambiente de testes não suportou maior quantidade de requisições. Ocorre estouro de memória (restrição de *hardware*).

A Tabela 1 apresenta resultados obtidos com os experimentos indicando o número de requisições emitidas e processadas por segundo em cada um dos três cenários distintos. Para o cálculo dos valores apresentados na referida tabela, foi considerado o tempo médio de processamento das execuções.

| **Quantidade de requisições emitidas** | **Quantidade de requisições processadas** | | |
|---|---|---|---|
| | Sem Fractal | Fractal sem ADS | Fractal com ADS |
| 100 | 17,10 | 13,47 | 13,76 |
| 500 | 60,47 | 43,56 | 43,65 |
| 1.000 | 92,09 | 57,28 | 59,52 |
| 3.000 | 119,10 | 85,11 | 80,93 |

**Tabela 1. Comparativo do número de requisições processadas em 1s**

O que se pode ser observado na Tabela 1, com a execução desta avaliação experimental é que, como esperado, o próprio modelo de componentes de *software* ainda é um entrave para o desempenho de sistemas distribuídos: de acordo com restrições de implementação do modelo Fractal, os componentes são modelados em série, obedecendo uma ordem pré-estabelecida. Adicionalmente, substituição sem parada de componente não é permitida. A maior perda de desempenho não ocorreu durante a realização da atualização de componentes propriamente dita, mas do fato do SGDB ser modelado como diferentes componentes em série. Note que, na Tabela 1, os valores obtidos para as colunas **Fractal sem ADS** e **Fractal com ADS** (por exemplo 85,11 e 80,93 na última linha da tabela) são bem parecidos, enquanto que os valores obtidos na coluna 1 (**Sem Fractal**), sem a interferência do modelo de componentes apresenta valores mais altos (por exemplo, 119,10 na última linha da tabela), portanto indica o melhor desempenho (isto é, maior quatidade de transações sendo executadas na unidade de tempo).

## 6. Conclusões e Trabalhos Futuros

ADS não é um assunto novo, mas na prática soluções automáticas são restritas. Desde que a demanda de aplicações críticas que exigem disponibilidade contínua, principal-

mente aquelas que fazem uso de bancos de dados através da Internet, tem aumentado, é interessante que sistemas computacionais sejam providos de técnicas de baixo custo, robustas e transparentes para a atualização de *software* sem a parada do serviço.

Este trabalho apresentou uma arquitetura para construção de um SGBD, baseada em componentes de *software* capaz de permitir a sua atualização automática e transparente, sem a indisponibilização total do sistema e sem usar *hardware* redundante. Propriedades adicionais incluem a ausência de restrição de linguagem e ambiente, flexibilizando portabilidade para diferentes ambientes operacionais e linguagens, oportunizada pelo modelo de componentes, e facilidade de manutenção à medida que a evolução de *software* ocorre de forma contínua.

Foi realizada avaliação experimental através da implementação de um protótipo. A avaliação experimental demonstrou que a implementação da ADS em SGBDs com suporte do modelo de componentes é viável, embora existam restrições impostas pelo próprio modelo de componentes e pela tecnologia atual. A degradação média do desempenho obtida nos testes realizados em ambiente controlado foi de aproximadamente 28,78%, em relação ao protótipo construído no paradigma de orientação a objetos, demonstram que o *framework* Fractal provoca uma sobrecarga significativa no sistema. Comparando os resultados obtidos nos testes sem realização de ADS e realizando ADS, observa-se que a substituição de componentes não provoca quedas adicionais de desempenho. Isto significa que o custo de Fractal é único, independente de quanto se utiliza suas funcionalidades.

Trabalhos futuros incluem a implementação da solução aqui proposta com outros *frameworks* de suporte ao desenvolvimento baseado em componentes de *software*. Esta sugestão tem como objetivo verificar a possibilidade de minimizar a degradação média de desempenho aferida neste trabalho. Outro trabalho futuro é ampliar a arquitetura proposta para suportar SGBDs executados em múltiplos nós. Para sistemas nesta estrutura, a atualização deve acontecer em cada um dos nós de forma coordenada. O desafio consiste em gerenciar esta atualização, a fim de determinar quando todos os nós estarão atualizados, visando atualização completa de um $cluster$ ou de uma $cloud$. Outro desafio é a observação do comportamento da solução aqui proposta através de uma implementação (real) prática. A implementação real possibilitará refinar a solução proposta, identificando pontos críticos e apontando novas direções de pesquisa. É necessário observar também o grau de alterações que serão necessárias na conversão de um SGBD desenvolvido no paradigma de orientação a objetos para o modelo de componentes de *software*.

Finalmente, duas funcionalidades já implementadas pelo $GA$ podem ser melhoradas. Uma delas é o serviço de escolha de janela de tempo para realização da atualiação, que pode ser aprimorado com a adição de mecanismos de previsão de multidões [Baryshnikov et al. 2005], evitado assim, que atualizações sejam realizadas quando a base de dados estará sujeita a demandas excessivas. Outro módulo que pode ser melhorado é o esquema de controle de versão, que pode adicionar ao $GA$ a possibilidade de reverter uma atualização. Para isto, além de guardar a versão antiga do componentes em uma base de dados, um *log* precisará armazenar todas as operações realizadas, ou pelo menos a última operação realizada. A reversão consiste em aplicar o *log* de baixo para cima, fazendo com que a versão antiga de um componente sobrescreva uma versão atual.

## Referências

Baryshnikov, Y.; Coffman, E. G.; Pierre, G.; Rubenstein, D.; Squillante, M.; Yimwadsana, T. (2005). Predictability of Web-server traffic congestion, *10th International Workshop on Web Content Caching and Distribution (WCW 2005)*, Sophia Antipolis, France.

Bass, L.; Clements, P.; Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition.

Bruneton, E.; Coupaye, T.; Leclercq, M.; Quema, V.; Stefani, J. B. (2006). The Fractal component model and its support in Java: experiences with auto-adaptive and reconfigurable systems. *Software Pract. Exper.*, 36(11-12):1257–1284.

Elmasri, R.; Navathe, S. (2005). *Sistemas de banco de dados*. Pearson Addison Wesley, São Paulo, 4 edição.

Fabry, R. S. (1976). How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pp. 470–476, Los Alamitos, CA, USA, IEEE Computer Society Press.

Leger, M.; Ledoux, T.; Coupaye, T. (2007). Reliable dynamic reconfigurations in the Fractal component model. In *Proceedings of the 6th International Workshop on Adaptive and Reflective Middleware: held at the ACM/IFIP/USENIX International Middleware Conference*, ARM '07, pp. 3:1–3:6, New York, NY, USA, ACM.

Lyu, J.; Kim, Y.; Kim, Y.; Lee, I. (2001). A procedure-based dynamic software update. *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pp. 271–284, Washington, DC, USA. IEEE Computer Society.

Oreizy, P.; Medvidovic, N.; Taylor, R. N. (1998). Architecture-based runtime software evolution. *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pp. 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

Oreizy, P.; Gorlick, M. M.; Taylor, R. N.; Heimbigner, D.; Johnson, G.; Medvidovic, N.; Quilici, A.; Rosenblum, D. S.; Wolf, A. L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62.

Ramakrishnam, R.; Gehrke, J. (2003). *Database management systems*. McGraw-Hill Education, Singapura, 3 edition.

Segal, M.; Frieder, O. (1993). On-the-fly program modification: systems for dynamic updating. *Software, IEEE*, 10(2):53–65.

Stoyle, G.; Hicks, M.; Bierman, G.; Sewell, P. e Neamtiu, I. (2007). Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Transaction Program. Languages and Systems*, 29(4).

Wahler, M.; Richter, S.; Oriol, M. (2009). Dynamic software updates for real-time systems. *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pp. 2:1–2:6, New York, NY, USA. ACM.

Wang, Q.; Shen, J.; Wang, X.; Mei, H. (2006). A component-based approach to online software evolution: Research articles. *J. Software Maint. Evolution*, 18:181–205.