# 4
# Implementação

Nesse capítulo se descreve a implementação de um protótipo que implementa a arquitetura centralizada proposta no capítulo 3. O protótipo consiste de um sistema de auto-sintonia embutido em um SGBD relacional baseado nessa arquitetura. A idéia é mostrar que é factível a implementação deste tipo de sistema, assim como testar a influência do MAS sobre o desempenho do SGBD e pesquisar as dificuldades que apresenta a implementação.

Na próxima seção serão expostas questões relacionadas ao desenvolvimento do sistema de auto-sintonia. Finalmente será descrito um exemplo de aplicação desse sistema embutido no SGBD PostgreSQL e uma avaliação da factibilidade da implementação do protótipo, assim como as dificuldades que esta tarefa apresenta.

## 4.1
## Infra-estrutura

O sistema de auto-sintonia foi projetado seguindo a arquitetura centralizada descrita em 3.3. O sistema pode ser integrado com o SGBD seguindo a arquitetura em camadas ou embutida. Esse sistema deve permitir a inclusão de agentes componentes construídos seguindo um *template* oferecido pelo sistema [1] de forma a minimizar o esforço de programação de novos agentes componentes e a padronizar a interação entre eles. Os agentes podem trabalhar isoladamente (desde que não exista outro agente ativo no sistema) ou em conjunto. Isto permite uma integração progressiva de mecanismos de auto-sintonia no sistema de banco de dados. O *template* segue a arquitetura proposta em [33], que também foi usada em [11, 40, 51] e é apresentada na figura 4.1.

Algumas desvantagens dessa arquitetura colocadas em [19] são que alguns aspectos como a autonomia atravessam as diferentes camadas, e por

[1] devido aos requisitos de eficiência da aplicação, não foram usadas plataformas para a implementação dos agentes

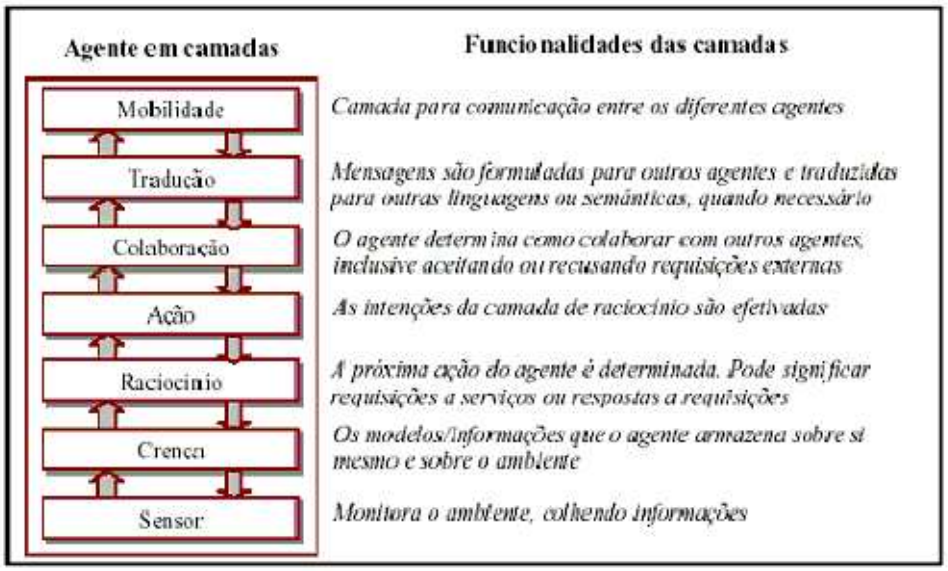


Figura 4.1: Framework para a construção de agentes [33]

estar muito interrelacionadas, não é trivial remover ou acrescentar uma camada. No entanto, ela foi escolhida para a implementação desse trabalho devido a sua relativa simplicidade e o fato de que já existia uma versão parcial disponível em C++ [51]. Nessa dissertação essa versão foi modificada para incluir a camada de Colaboração e a gerência de concorrência.

A implementação está baseada em padrões de desenho orientado a objetos como *Facade* e *Singleton* que podem ser consultados em [21].

### 4.1.1 Gerência de concorrência

Um programa concorrente contêm dois ou mais processos que trabalham em conjunto para executar uma tarefa. Visando garantir um controle contínuo da atividade do SGBD afetando no mínimo possível o seu funcionamento, todas as medições, assim como as ações tanto de sintonia como de colaboração deverão ser executados por processos concorrentes.

Esses processos podem ser *threads* ou processos de aplicação. Enquanto a criação de um processo de aplicação implica na duplicação do processo pai junto com todo seu contexto, as *threads* são processos leves(*lighweight*) que só precisam da criação de um novo contador de programa e pilha de execução [57]. O mecanismo para escrever aplicações *multithreaded* em C foi padronizado e criada a biblioteca *Pthreads* (POSIX API, standard 1003.1c-1995 da IEEE), amplamente suportada.

A linguagem C++ não possui gerência intrínseca de concorrência.

As funções relacionadas estão contidas na biblioteca *pthreads* [57]. Para encapsular o comportamento concorrente nesse trabalho foi criada uma classe *Thread* que faz chamadas às funções desta biblioteca. As informações no sistema são trafegadas entre os processos através de filas de mensagens (*Message Queues*) [57], que encapsula por sua vez detalhes de controle de concorrência.

### 4.1.2 Implementação dos agentes

As camadas dos agentes foram implementadas usando o padrão *Facade* [21], de forma a isolar as camadas do agente. Esse isolamento foi, no entanto, prejudicado pela ausência de reflexividade do C++. Assim, por exemplo, na implementação original desenvolvida em Java [33] os planos de execução são enviados à camada de ação como *strings*. As ações são instanciadas a partir destas variáveis utilizando facilidades oferecidas pela linguagem. Dado que o C++ não permite este tipo de operação, optamos aqui por criar fábricas de objetos cuja referência é trafegada de uma camada a outra segundo a classe a instanciar.

Nesse trabalho assume-se que todos os agentes utilizam uma ontologia comum e não se consideram aspectos relativos à mobilidade. Por essa causa, o *template* não inclui as camadas de Tradução e Mobilidade. As camadas restantes serão descritas a seguir:

#### Camada Sensor

A camada Sensor está constituída pela interface *ASensor* e pela classe *SensoryFacade*. A interface *ASensor* deve ser herdada pelas novas classes sensor. O *SensoryFacade* é o ponto de entrada na camada e, como as outras Façades, seu objetivo é manter o isolamento da camada.

O sensor executa em um processo independente do sistema, que é iniciado assim que são inicializadas as estruturas do agente e iniciada a execução da camada de Colaboração. Sua tarefa é a coleta de dados. Esta pode ser efetuada por amostragem ou por eventos (notificação). Os dados são repassados ao objeto crença subscrito como observador desse sensor, através do padrão *Observer* [21].

### Camada de Crença

A camada Crença é composta das classes *ABelief* e *BeliefFacade*. *ABelief* é a interface das classes Crença dos novos agentes componentes. O objeto Crença deve ser subscrito como observador de um ou vários sensores, de forma a ser notificado em caso de mudanças. Eventos importantes devem ser notificados por sua vez à camada de Raciocínio.

### Camada Raciocínio

Na camada de Raciocínio, o objeto observador da crença avalia a informação e pode gerar intenções que serão executadas pela camada de ação. Essas intenções podem ser de colaboração ou de ação. As intenções de ação contêm um plano com uma ou várias ações a executar, sendo que para manter o isolamento entre as camadas, nessa implementação são repassadas uma referência a uma fábrica do tipo de objeto indicado para executar a ação e seus parâmetros.

Um exemplo para um caso que precise da mudança de estratégia de substituição de páginas em *buffer* é mostrado na figura 4.2.

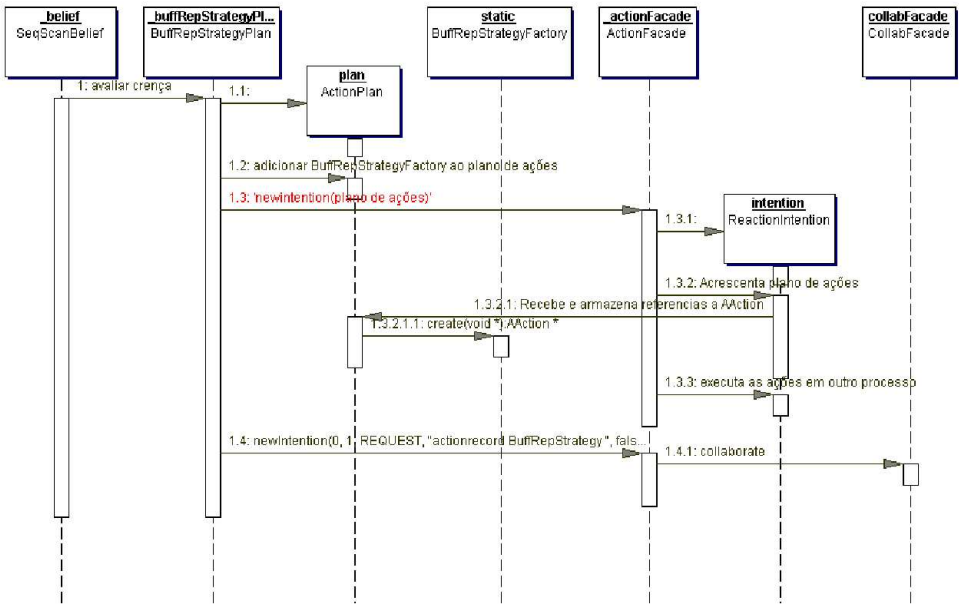


Figura 4.2: Processo executado pelas camadas de Crença, Raciocínio e Ação para o controle da estratégia de substituição de páginas em *buffer*.

Após ser notificado, o plano observador na camada de raciocínio (*buffReplStrategyPlan*) avalia as crenças. Como conseqüência dessa avaliação podem ser geradas intenções de ação e colaboração, que são transmitidas

à camada de ação. As intenções de ação indicam a execução de uma ou várias ações listadas dentro do plano de ação (*plan*) que é passado como parâmetro. Cada ação é definida por uma fábrica (BufRepStrategyFactory) e um grupo de parâmetros para a instanciação da ação. Já na camada de ação, uma intenção é criada que consulta o plano de ações e recebe as ações instanciadas. Essas ações serão executadas pela intenção seqüencialmente em um novo processo.

Por outro lado, a intenção de colaboração implica em uma chamada ao método *collaborate* da camada de colaboração repassando os parâmetros recebidos na camada de ação.

Além das classes *ActionPlan*, *APlan* e *ReasoningFacade*, a camada de Raciocínio está integrada também pela classe *ReasForwarder*, que recebe as mensagens transmitidas pela camada de Ação, e a classe *InterpretPlan*, que é observadora de *ReasForward* e é executada para processar essas mensagens, possivelmente invocando outros planos disponíveis no agente.



### Camada Ação

A camada de Ação executa as intenções de ação e repassa para a camada de Colaboração as intenções de colaboração. As ações são executadas seqüencialmente dentro de uma intenção, sendo que cada intenção é executada em um processo separado. A camada está formada pelas interfaces *AAction* e as classes *ActionFacade*, *AIntention*, *Forwarder* e *ReactionIntention*. A classe *Forwarder* recebe as mensagens da camada de Colaboração enviadas por outros agentes do sistema e as repassa para a camada de Raciocínio.

### Camada Colaboração

Essa camada está constituída pelas classes *CollabFacade*, *PostOffice*, *Acceptor*, *Connector* e *Message*. A classe *PostOffice* é um *Singleton*: somente existe um objeto *PostOffice* no sistema. Ele é um depósito de mensagens, de forma que os objetos *Connector* deixam as mensagens especificando o destinatário e a prioridade, e o *PostOffice* as coloca na fila de prioridade correspondente a esse destinatário. O destino pode ser *broadcast* (todos os agentes do sistema), um agente em particular ou um provedor de serviço. Nesse último caso o destinatário será primeiro registrado como provedor do serviço em particular no *PostOffice*. O processo de envio de uma mensagem é ilustrado na figura 4.3.

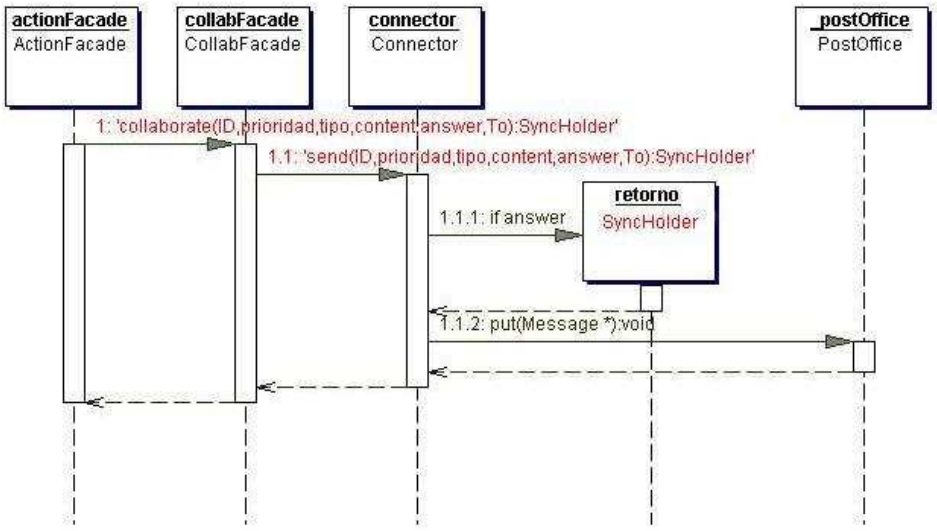


Figura 4.3: Envio de uma mensagem

Como mostra a figura, as mensagens enviadas à camada de Colaboração são repassadas ao *Connector*, que as envia ao *PostOffice* e retorna o controle ao agente. O *Connector* também cria e retorna um objeto para armazenar a resposta recebida no caso da camada de raciocínio esperar resposta.

O envio das mensagens pode ser síncrono ou assíncrono. O envio síncrono é utilizado quando um objeto precisa de uma informação para continuar seu processamento. Nesse caso ele envia uma mensagem e fica bloqueado até a resposta chegar. Esta é enviada diretamente do Acceptor ao objeto destino usando a referência contida no campo future.

No agente receptor, o *Acceptor* fica bloqueado esperando por mensagens na fila até ser notificado de que pode retirar a sua mensagem. Ela é repassada para a camada de ação, que a sua vez a transmite para a camada de Raciocínio, onde é analisada por um objeto da classe *InterpretPlan* que está registrado como observador dessa camada. Esse objeto pode criar planos que por sua vez gerem novas intenções de ação ou colaboração. Na figura 4.4 se mostra a seqüência de ações desde o *PostOffice* até a camada de Raciocínio durante o recebimento de uma mensagem.

A comunicação envolve as camadas:

1. transmissão: que descreve a forma em que a mensagem vai trafegar pelo sistema;

2. sintaxe: define o formato da informação;

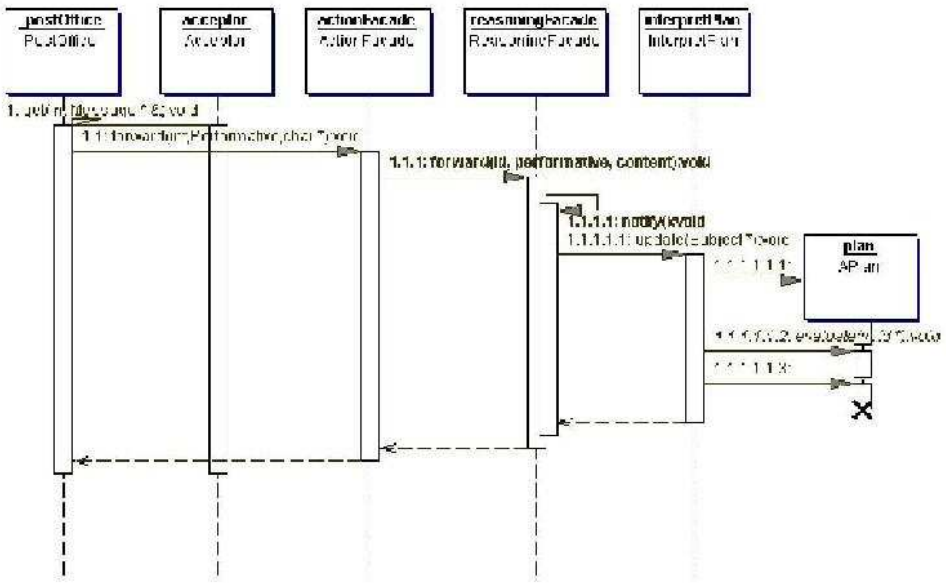


Figura 4.4: Recebimento de uma mensagem (comunicação assíncrona)

3. semântica: que define o tipo da mensagem e o seu conteúdo.

Estas três camadas são idealmente independentes, de forma que é possível substituir uma sem afetar as outras.

A comunicação pode ser tanto ponto a ponto como *broadcast*. As mensagens são encapsuladas para sua transmissão em objetos da classe *Message*. O protocolo da camada de sintaxe define os seguintes campos:

- MsgID
- SessionID
- Performative
- origem
- destino
- prioridade
- conteúdo
- future

O campo future guarda uma referência ao objeto ao que deverá ser entregue a informação retornada no caso de uma comunicação síncrona. Origem e destino são os identificadores únicos reservados a cada agente durante o processo de registro no sistema. O campo Performative contém o tipo de mensagem.

O conteúdo propriamente dito está estruturado como uma sucessão de cadeias de caracteres separadas por espaços, em que a primeira cadeia é o serviço requisitado no caso do tipo da mensagem (*performative*) ser um REQUEST. O restante da mensagem pode ser organizado livremente desde que a estrutura seja compartilhada por todos os agentes do sistema. Assim, por exemplo, uma intenção de colaboração expressada da seguinte forma:

```
newintention(0,1,REQUEST,"record BuffRepStrategy MRU=true",false,0);
```

sendo que esse método está definido como:

```
void ActionFacade::newIntention(int ID, int prioridade,
Performative performative, char *message, bool resposta, int Destino)
```

significa que o agente *BuffRepStrategy* solicita ao agente de histórico, cuja identificação não tem, o registro de uma ação como resultado da qual a política de substituição de páginas em *buffer* foi trocada para MRU. O agente não espera resposta.

Um problema dessa implementação está na mistura entre as camadas semântica e de sintaxe, pois é necessário "abrir"o conteúdo da mensagem para saber o serviço requisitado e com isso, o destinatário da mensagem no caso deste não estar especificado no campo destino. Uma solução pode ser acrescentar um campo ao envelope da mensagem com o nome do serviço. Essa mudança será efetuada em breve no sistema.

Na figura 4.5 se mostra um processo de comunicação. Nele estão envolvidos um processo do SGBD e dois agentes. Note que as requisições entre os agentes são trafegadas na forma de serviços. Aqui é possível perceber que um dos agentes possui somente as camadas de Crença, Raciocínio e Colaboração, como pode ser o caso do Agente Coordenador na Arquitetura Centralizada. Esse agente não precisaria da camada Sensor porque as noções que ele tem sobre o ambiente vem todas da camada de colaboração. A camada de ação não é necessária porque as ações a executar são requisições de serviços a outros agentes do sistema que serão enviadas através da camada de colaboração. No caso, o trabalho do agente consiste em decidir sobre a base das informações recebidas dos outros agentes quais devem ser as ações a tomar e mandar a executá-las na forma de requisições de serviço. Um exemplo de requisição de serviço poderia ser uma mensagem do agente de *BuffRepStrategy* ao agente de histórico pedindo para que seja armazenado o dado "MRU=true".

O tráfego de informação entre os processos pode obrigar ao uso de estruturas protegidas.

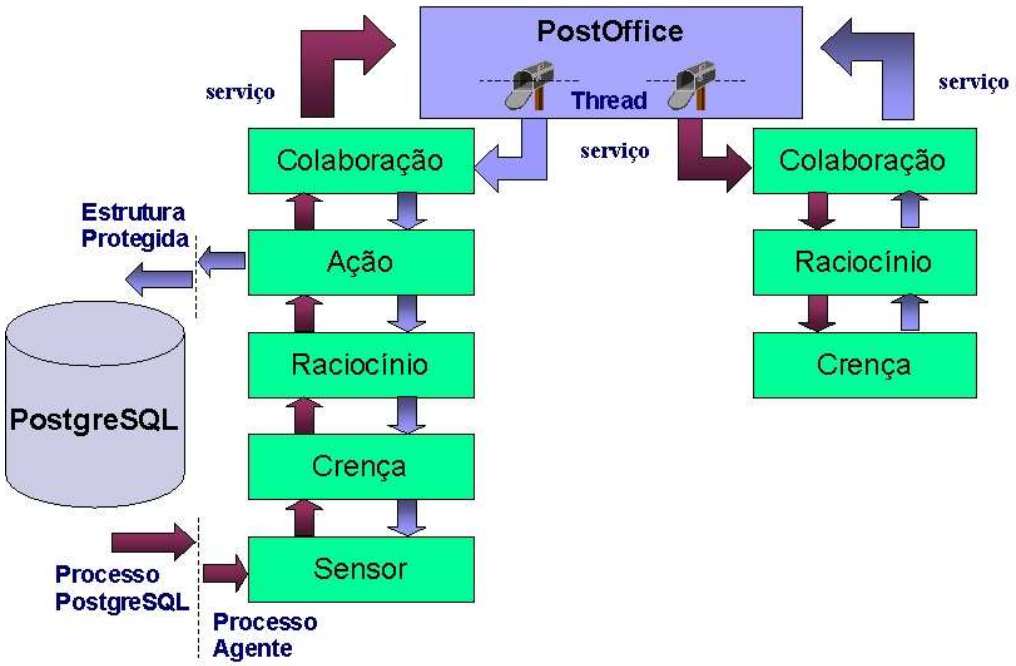


Figura 4.5: Interações no sistema de auto-sintonia

### Sistema de Agentes

A entrada no Sistema de Agentes é feita através do cadastro de cada agente na lista mantida no objeto *AgentSystem*, que é único no sistema. Ao mesmo tempo é inicializada a camada de Colaboração do agente e o agente se registra no *PostOffice*. A execução de tarefas no sistema de auto-sintonia é orientada a serviços. Esses serviços são especificados durante a configuração do agente e registrados no *PostOffice* para disponibilizá-los para os outros agentes do sistema. A saída acontece com o cancelamento do cadastro do agente com o *AgentSystem*.

## 4.1.3 Validação da gerência de concorrência

A função da sincronização é restringir o grupo de possíveis fluxos de execução a aqueles que são desejáveis. Nessa implementação a sincronização entre os processos do PostgreSQL e os sensores do agente é garantida pelas filas de mensagens. As filas de mensagens permitem implementar esperas por condição e não precisam do uso de semáforos para garantir acesso exclusivo. As ações do agente consistiram em escritas em variáveis compartilhadas protegidas por semáforos.

A correção de um programa concorrente implica na satisfação das propriedades *segurança(safety)* e *vitalidade(liveness)* [4], que podem ser

definidas como que nada "ruim"vai a acontecer durante a execução e que em algum momento algo de "bom"vai a acontecer, respectivamente. Para o tipo de aplicação que está sendo discutida aqui o termo "em algum momento"não é suficiente pois algumas decisões deverão ser tomadas em tempo real. Por esse motivo consideraremos que o sistema satisfaz a propriedade de *vitalidade* se o tempo em que ocorre "algo de bom"está dentro do admissível segundo o domínio da aplicação.

Dado que todas as comunicações serão efetuadas mediante filas de mensagens, o programa não deve entrar em um estado no qual um ou mais processos estejam em uma zona mutuamente crítica.

Outro requerimento para considerar o programa seguro é que esteja livre de *deadlock*. O *deadlock* acontece quando dois ou mais processos esperam por uma condição que nunca vai acontecer. As condições necessárias e suficientes para a ocorrência de *deadlock* são [4]:

- necessidade de que os processos envolvidos precisem compartilhar recursos que devam ser usados em exclusão mutua;
- que os processos mantenham os recursos já adquiridos enquanto esperam por outros;
- ausência do controle do sistema sobre a liberação dos recursos;
- existe uma cadeia circular de requisições e aquisições.

A política adotada é usar o recurso e liberá-lo imediatamente, isso implica que não existem possibilidades de *deadlock* no sistema.

O *livelock* é o estado no qual o processo não executa mais do que instruções inúteis. O *livelock* é uma violação da vitalidade. Uma situação em que o processo pode entrar em *livelock* é na espera por condição. Se a condição nunca acontece o processo vai esperar inutilmente. Novamente, situações de *livelock* podem ser codificadas como parte da programação dos agentes. Todas as operações de colaboração que impliquem espera devem incluir *timeouts*, protegendo-se de erros que possam eventualmente acontecer. Seguidas essas regras nosso sistema deverá ser um sistema seguro.

A inanição (*starvation*) é outra violação de vitalidade. Nessa situação o sistema consegue progredir, mas tem algum processo que não consegue acesso aos recursos porque sempre perde na concorrência, por exemplo, por ser de menor prioridade. Há dois candidatos potenciais a gerar situações de inanição que são as mensagens armazenadas nas filas do *PostOffice* e o escalonador de ações. Ações pouco prioritárias não serão jamais executadas se tiver sempre ações mais prioritárias esperando, da mesma forma

que as mensagens menos prioritárias não conseguirão sair da fila enquanto tiver mensagens de maior prioridade. Isso pode ser resolvido de duas formas: uma é codificar o escalonador seguindo princípios de justiça (*fairness*), garantindo que cada processo será eleito em algum momento, com independência da sua prioridade. A outra solução é assumir que as ações mais prioritárias são aquelas que devem ser executadas em tempo real, enquanto as menos prioritárias não têm restrições de tempo. Nesse caso o fato das mensagens menos prioritárias ficarem na fila não constitui um problema, desde que as prioridades sejam cuidadosamente alocadas.

## 4.2 Exemplo de aplicação

Para verificar a arquitetura proposta na prática o sistema proposto neste trabalho foi embutido dentro de um SGBD. Uma dificuldade que enfrentamos foi a escolha desse SGBD, que deveria expor tanto as variáveis a medir como os parâmetros ou funções de controle. Os SGBDs comerciais publicam um grupo de APIs que permitem o acesso às funções do sistema. Entretanto, não permitem modificar seu código. Outro problema é o fato de que já existe um grupo de funções de auto-sintonia implementada nesses sistemas comerciais que poderia afetar os resultados dos testes. Por último está a motivação deste trabalho ter sido conduzido em colaboração com outro [50] que seria aproveitado aqui que precisava de funções não publicamente disponíveis em sistemas comerciais.

Por essas razões foi escolhido um sistema de código aberto. Os sistemas analisados foram o MySQL e o PostgreSQL. O MySQL é um SGBD relacional de código aberto disponível em [43]. O PostgreSQL [47] é um sistema de gerência de bancos de dados objeto-relacional também de código aberto disponível nas plataformas Unix.

Para o desenvolvimento do trabalho seria necessário previamente implementar funções e criar novas variáveis dentro do SGBD que permitissem observar determinadas métricas e ajustar parâmetros de controle. O MySQL demostrou não ser uma boa escolha para este tipo de trabalho devido à falta de documentação e a desorganização do código. No PostgreSQL, por outro lado, são notáveis a limpeza e organização do código, assim como a abundância de documentação. Esses aspectos foram determinantes na escolha desse sistema para a nossa implementação.

## 4.3 Ambiente de desenvolvimento

A linguagem escolhida para a implementação desse trabalho foi C++. A razão dessa escolha se deve ao fato do PostgreSQL estar escrito nessa linguagem. O compilador de linha usado foi o gcc versão 3.2.2. A depuração foi uma das tarefas mais difíceis deste trabalho devido às dificuldades inerentes à depuração de código concorrente. Esta foi feita usando gdb [20] e *checkpoints* manuais.

A ferramenta de modelagem UML usada foi o Together 6.0 para Windows [60]. Essa é uma plataforma implementada na linguagem Java que permite modelar e documentar sistemas orientados a objetos. Usando a ferramenta é possível tanto a geração de código na linguagem C++ ou Java a partir dos diagramas UML como efetuar o processo inverso.

A plataforma foi um Athlon de 1.6GHz com 128Mb de RAM sob o sistema operacional Linux versão 8.0.



### 4.3.1 Interação com o SGBD

O código do PostgreSQL foi modificado de forma a incluir um parâmetro na linha de comandos para iniciar o sistema de auto-sintonia (arquivo postmaster/postmaster.c). No caso de ser habilitada a opção são instanciados os agentes e incluídos dentro do sistema de agentes.

O sistema de auto-sintonia foi implementado de forma a não intervir na atividade do SGBD. Assim, no caso do sistema de auto-sintonia estar desligado o SGBD seguirá funcionando. Os agentes estão embutidos [40] dentro do SGBD, consumindo recursos deste ainda quando executam em processos separados.

O motivo para não ter escolhido uma arquitetura integrada é o fato de não existir um sistema de banco de dados relacional baseado em agentes disponível para esta implementação. A arquitetura em camadas permite não requer modificações no sistema de banco de dados, mas depende das interfaces que este ofereça para observar e agir sobre o sistema controlado. Por outro lado, os agentes embutidos tem acesso a todos os componentes do banco, porém a sua construção se complica pelo fato de precisar compreender o funcionamento interno do sistema. Outras desvantagens contra a arquitetura embutida estão no fato de que somente pode ser implementada em SGBDs de código aberto, assim como em aspectos relacionados com a interação com o SGBD como é o caso do modelo de concorrência. Uma van-

tagem fundamental dessa arquitetura é que permite uma implementação mais otimizada que a versão em camadas. Diferentemente da arquitetura integrada, na arquitetura embutida pode-se utilizar o sistema ainda sem ativar os agentes.

O tema da inclusão de *threads* na implementação do PostgreSQL tem sido bastante levantada em listas públicas de discussão. No entanto, a equipe de desenvolvimento do PostgreSQL mantém a posição de que o isolamento que os processos oferecem garante uma segurança e uma portabilidade difíceis de alcançar com *threads*, além de que critérios de eficiência são muito dependentes da implementação particular de cada sistema operacional. Dessa forma, o PostgreSQL segue o esquema de um servidor concorrente que cria um processo toda vez recebe uma nova conexão. A comunicação entre os processos se efetua através de sinais, os dados são compartilhados usando uma estrutura chamada de Memória Compartilhada (*Shared Memory*) [57] e a sincronização através de semáforos.

A interação entre o PostgreSQL e o sistema de auto-sintonia se desenvolve através de uma interface (postgresInterface.cpp). Ela contém as funções que chamam os métodos de criação e destruição do sistema de agentes, assim como as funções que executam ações sobre o SGBD e aquelas invocadas por este para notificar um evento.

Ao ser iniciado o PostgreSQL com a opção -t, o *postmaster* faz um chamado à função ags_create da interface. Essa função cria os agentes, os inclui no sistema de agentes e inicializa as estruturas de comunicação entre o PostgreSQL e os agentes.

A detecção de eventos internos se efetua através de *checkpoints* ou *software probes*, que são instruções que se inserem no código de um programa para coletar e armazenar dados ou chamar a uma rotina de medição [18]. A interferência produzida pode ser intolerável se for introduzido em porções críticas do programa medido, ou se tempo de interferência for considerável. Levando em conta este problema, a política assumida é fazer com que o processo PostgreSQL somente precise colocar a informação e retornar pois o sensor correspondente está executando em outro processo concorrente.

O sistema de auto-sintonia implementado usa como repositórios de informação arquivos de texto simples. A informação contida nesses arquivos deve ser analisada para detectar violações das expectativas de desempenho e verificar a validade de uma ação que tenha sido executada antes em um contexto similar.

Com o objetivo de comprovar a factibilidade da implementação, assim como avaliar as dificuldades que esta apresenta, foram implementados

dois mecanismos de auto-sintonia referentes aos problemas de escolha da estratégia de substituição de páginas em memória (veja a seção A.1) e ao de *thrashing* e contenção de dados(veja a seção A.2). Os detalhes da implementação desses mecanismos se expõem a seguir.

### 4.3.2 Estratégia de substituição de páginas em memória

Foi acrescentada ao PostgreSQL uma função que implementa o método MRU de substituição de páginas em *buffer* (veja o anexo A.1). Esta função está disponível livremente na Internet, mas ainda não foi inclusa na distribuição oficial do PostgreSQL. A diferença da estratégia LRU, a MRU coloca as páginas no extremo MRU da fila, de forma que serão as primeiras candidatas a substituição. Isso faz sentido em casos em que a informação será usada somente uma vez, como pode ser o caso das varreduras seqüenciais, e pretende preservar a informação útil armazenada no *buffer*.

Ao detectar a geração de um plano que inclui varreduras seqüenciais, o sensor é notificado pelo processo PostgreSQL correspondente. O agente pode então avaliar a execução de uma ação consistente na troca da estratégia de substituição de páginas utilizando uma função incluída no PostgreSQL com esse propósito.

### 4.3.3 *Thrashing* de contenção de dados

A variável *razão de conflitos* é a razão entre o número total de bloqueios no sistema e quantos deles são possuídos por transações ativas. O registro desses valores é feito introduzindo *checkpoints* nos pontos de confirmação de bloqueios no arquivo lock.c do PostgreSQL que consistem em chamadas à função ags_notify_lock() da interface para notificar esses valores.

O final de cada transação também é notificado. Esse evento supõe a ativação do mecanismo de controle de admissão, que pode liberar transações da fila dependendo do valor da razão de conflitos (veja a seção A.2).

Finalmente, a figura 4.6 mostra o processo executado pelo sistema após um alarme indicando perigo de *thrashing* de contenção de dados no PostgreSQL.

Ao receber um dado do PostgreSQL, o agente de controle de MPL calcula o valor atual da razão de conflitos. Esse valor é enviado ao agente de

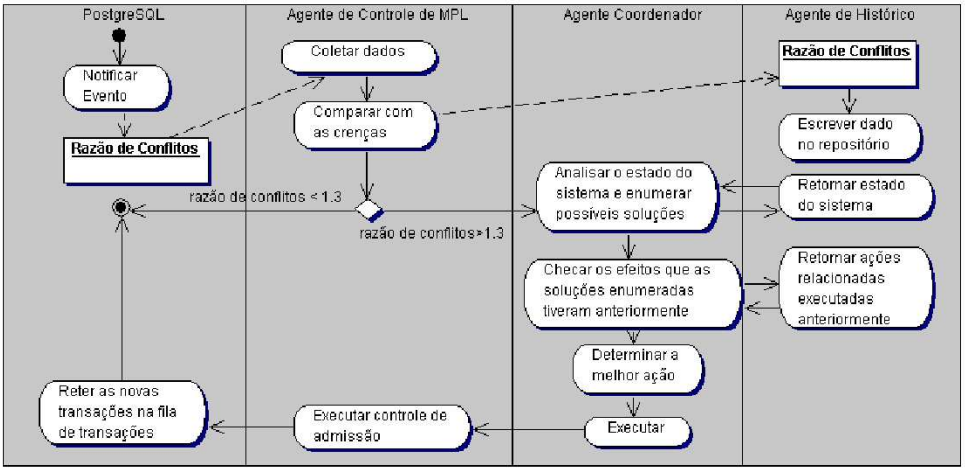


Figura 4.6: Atividade do sistema em um evento de violação de razão de conflitos

histórico para ser armazenado no repositório centralizado e também avaliado para detectar se existe perigo de *thrashing* de contenção de dados. Em caso de violação, o agente Coordenador é notificado. Ele inicia um processo de análise do estado do sistema objetivando diagnosticar as causas reais dos problemas e assim evitar situações como a comentada na seção 3.2, em que a reação do mecanismo de auto-sintonia a uma causa aparente piora mais ainda o desempenho do sistema. Posteriormente as possíveis soluções ao problema são enumeradas e avaliadas com a informação disponível sobre ações executadas. As ações finalmente escolhidas são repassadas aos agentes correspondentes na forma de requisições de serviço, cujo objetivo é agir sobre o PostgreSQL para resolver a situação.

## 4.4 Aspectos de implementação

O desenvolvimento do trabalho consistiu na implementação em C++ do *framework* de agentes discutido anteriormente (veja 4.1) e nas adaptações necessárias para embutir agentes instanciados a partir desse *framework* em um SGBD real, no caso no PostgreSQL. Para esse trabalho aproveitamos duas versões implementadas por pessoas do nosso grupo em C++ [40, 51]. Ambas implementações foram construídas pensando em um único agente. Partindo desse código, da documentação e o código original em Java disponível em [33], foi criada para essa dissertação uma versão que inclui a gerência de concorrência e a camada de colaboração.

A implementação em geral apresentou a dificuldade já discutida da

falta de suporte a concorrência em C++. O uso de *Message Queues* facilitou a implementação pois evitou a necessidade de utilizar semáforos para garantir o acesso exclusivo e por permitir implementar facilmente esperas por condição. Assim como em [33], grande parte da nossa implementação está baseada em padrões de projeto. A implementação da camada de colaboração foi dificultada pois em [33] a colaboração centralizada não está muito documentada e a implementação distribuída dessa camada não é completa. Para essa dissertação concluímos que a complexidade da variante distribuída não era necessária. A versão centralizada foi implementada seguindo a sugestão de usar o padrão *Mediator* [33], que é a base do *PostOffice*. Além de suportar o envio de mensagens assíncronas, inclui também o trafego síncrono de mensagens devido às necessidades do agente central, que não deveria ficar esperando indefinidamente por consultas necessárias para a tomada de decisões. Com esse objetivo foi acrescentado na mensagem o campo *future*, que contém uma referência à estrutura onde será armazenado o dado retornado. Essa referência é devolvida ao objeto que espera a resposta, o qual ficará esperando o dado para continuar o processamento.

Já construída a estrutura básica dos agentes basta estender as classes Sensor, *Belief*, *Reasoning* e *Action* (na realidade somente aquelas que o agente precise) para instanciar cada agente que formará parte do sistema. Para comprovar a comunicação entre os agentes e a sua inserção no SGBD, em nosso protótipo foram instanciados um agente de histórico, um agente de Controle de Carga, um outro agente para o controle de política de substituição de páginas no *buffer* e um agente central. Dado que os algoritmos usados por esses agentes influem na eficácia do sistema, eles devem ser melhorados para obter bons resultados.

A inserção do sistema de agentes no PostgreSQL representou um grande desafio. O PostgreSQL é um sistema robusto e complexo que precisou ser estudado a fundo devido à escolha de embutir o sistema de agentes dentro do SGBD. Diversas verificações são feitas pelo PostgreSQL durante a execução dos processos, de forma que o sistema deveria inserir-se da forma mais transparente possível para não gerar inconsistências. No protótipo os agentes executam em *threads* diferentes dentro do processo do postmaster. Uma solução melhor seria colocar o sistema de agentes dentro de um processo separado, de forma a garantir o isolamento do SGBD no caso de erros no sistema de agentes, facilitando também a depuração. No entanto, essa não é uma tarefa fácil, pois é preciso reproduzir o procedimento (não documentado) que o PostgreSQL executa a cada vez que um novo processo

é criado. Essa solução foi implementada posteriormente em [52].

## 4.5 Conclusões

A implementação por um lado mostrou a viabilidade da inserção de um sistema de agentes construído seguindo a arquitetura centralizada dentro de um SGBD real, assim como as dificuldades que isto implica. A problemática da integração do sistema de agentes com o SGBD usando uma arquitetura embutida fez aumentar consideravelmente a complexidade da implementação, impossibilitando no caso a apresentação de resultados quantitativos.

As dificuldades fundamentais enfrentadas durante a implementação foram:

- Integração com o PostgreSQL.
- Depuração em ambiente concorrente
- Gerência de concorrência.
- Implementação do framework de agentes sob as limitações do C++.