Pós-Graduação em Ciência da Computação

"DBSitter-AS: um Framework Orientado a Agentes para Construção de Componentes de Gerenciamento Autônomo para SGBD"

Por

Paulo Roberto Moreira Maciel

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife, Agosto/2007

Paulo Roberto Moreira Maciel

"DBSitter-*AS*: um Framework Orientado a Agentes para Construção de Componentes de Gerenciamento Autônomo para SGBD"

DISSERTAÇÃO APRESENTADA AO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO, COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADORA: *PATRÍCIA AZEVEDO TEDESCO*

CO-ORIENTADORA: *ANA CAROLINA SALGADO*

Recife, agosto de 2007

Agradecimentos

Agradeço a Deus, por minha saúde e por ter permitido que eu concluísse mais este trabalho.

Agradeço à minha esposa Luciana por todo amor, compreensão e apoio.

Agradeço à minha família, por me dar o suporte necessário para que pudesse chegar onde estou.

Agradeço também aos amigos e companheiros de pesquisa Adriana Carneiro, Davi Anabuki, Jocelia Silva e Ana Elizabeth que me ajudaram durante esta caminhada.

Agradeço aos amigos e colegas do Centro de Informática da UFPE, Carlos Eduardo, Vaninha, Rosalie e Rosa Cândida pelo apoio e troca de idéias.

Agradeço às minhas amigas e orientadoras, Patrícia e Ana Carolina, por todo o auxílio, paciência e confiança, indispensáveis, durante este trajeto.

De forma geral meu muitíssimo obrigado a todos que conviveram comigo nestes dois últimos anos pela paciência, orações e incentivo.

Maciel, Paulo Roberto Moreira

DBSitter-AS: um Framework Orientado a Agentes para Construção de Componentes de Gerenciamento Autônomo para SGBD / Paulo Roberto Moreira – Recife: O autor, 2007.

viii, 105 p. : il., fig., quadros.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2007.

Inclui bibliografia e apêndice.

I. Inteligência Artificial. 2. Computação Autônoma. 3. SGBD. 4. Sistemas Multiagentes. I. Título.

006.31 CDD (22.ed) MEI2007-074

Resumo

A Computação Autônoma é uma área de pesquisa que busca o desenvolvimento de software capaz de autoconfiguração, auto-otimização, autoproteção, auto-reparação, autoconhecimento e antecipação de necessidades, utilizando-se de padrões abertos. No caso particular de Sistemas de Gerenciamento de Bancos de Dados (SGBD), podemos resumir esses princípios como características para autogerenciamento. Uma das formas de implementar características de autogerenciamento é através do desenvolvimento de Sistemas Multiagentes (SMA) que realizem a monitoração, prevenção ou solução de falhas e aperfeiçoamento de um sistema alvo. Não há ainda muitas especificações formais de como implementar autonomia para SGBD, em especial se desenvolvida via SMA.

No contexto acima, o objetivo deste trabalho é a especificação de um framework arquitetural que guie o desenvolvimento de componentes de SMA que possam prover SGBD de capacidade de autogerenciamento, auxiliando os Administradores de Bancos de Dados nas suas atividades diárias.

O Framework DBSitter-AS foi concebido utilizando a metodologia Tropos para desenvolvimento de SMA e especifica como construir uma sociedade de agentes capaz de realizar ações coordenadas de monitoria, prevenção e resolução de falhas em SGBD de mercado. A sociedade de agentes concebida está configurada em uma camada externa ao SGBD, possui uma estrutura de persistência própria e permite registro de regras e políticas organizacionais a serem obedecidas. O DBSitter-AS é uma especificação flexível no que tange a permitir o cadastro de sintomas de falhas e ações de resolução configuráveis para tipos diferentes de SGBD. Para mostrar como a especificação pode ser adaptada para casos reais, mostramos um exemplo de implementação para um caso típico de falha.

Palavras-chave

Computação Autônoma; SGBD; Sistemas Multiagentes; Metodologia de desenvolvimento orientado a agentes.

Abstract

Autonomic Computing is a research area that aims at developing software capable of self-configuring, self-optimization, self-protecting, self-healing, self-knowledge and the anticipation of needs, through the use of open patterns. In the specific case of Data Base Management Systems, these features can be summarized in self-management characteristics. One of the means of implementing self-management features is by developing Multiagent Systems (MAS) which monitor, prevent and solve faults. To the best of our knowledge, there are not many formal specifications of how to implement autonomic characteristics for DBMS, in particular if they are developed as MAS.

In this light, the goal of this research work is to specify an architectural framework to guide the development of MAS components that can provide DBMS with self-management functionalities, thus helping the DBA in their daily activities.

The DBSitter-AS framework was designed with the Tropos methodology. It specifies how to develop an agent society that is capable of performing coordinated actions of monitoring, fault prevention and resolution in commercial DBMS. The developed agent society is set up as a layer external to the DBMS. It counts with a persistence structure and allows for the recording of organizational rules and politics that must be followed by the agent society. The DBSitter-AS allows for the addition of new fault symptoms and fault-solving actions for different DBMS. In order to show how the specification can be adapted to different situations, we present an implementation example for a typical fault case.

Keywords

Autonomic Computing; DBMS; Multiagent Systems; Agent Oriented Development Methodology.

Índice

Lista de Quadros

Lista de Figuras

1 Introdução

Os Sistemas de Gerenciamento de Bancos de Dados (SGBD) estão presentes na base de todos os Sistemas Computacionais, dos mais variados negócios e diversos portes. Todos esses sistemas têm como requisito um gerenciamento de dados seguro e que satisfaça às demandas de armazenagem e manipulação do seu bem maior, as informações. Os SGBD atuais são usados em uma grande diversidade de aplicações (e.g. *Data Warehousing*, *e-commerce*, aplicações multimídia e bancos de dados distribuídos) com as mais variadas características. A diversidade de aplicações tornou os SGBD sistemas dotados de grande número de características que precisam ser ajustadas e monitoradas com acuidade.

Com o crescente aumento da complexidade dos sistemas computacionais atuais, impulsionados por sua utilização através da Internet e necessidades de integração com outros sistemas, também a administração dos SGBD tornou-se uma tarefa complexa, que exige garantias de bom desempenho e alta disponibilidade de serviços. A complexidade da administração também se reflete nos profissionais que administram os SGBD. Os Administradores de Bancos de Dados (ABD) são cada vez mais exigidos para que fatores como segurança, desempenho, continuidade de serviços e integridade dos dados seja mantida a todo custo. Essas exigências demandam dos ABD alto grau de especialização e os tornam profissionais caros, muitas vezes sobrecarregados de atividades.

Também atualmente, tanto clientes como fabricantes de SGBD, começaram a dar mais ênfase na redução do custo total da propriedade do software (software, hardware para executá-lo e pessoal para administrá-lo) e a despeito da redução de preço de hardware e software, o custo total de propriedade de SGBD tem aumentado, predominantemente por causa do custo humano, em especial o do ABD [Elnaffar et al. 2003].

Neste contexto, os Sistemas de Computação Autônoma [Horn 2001] são vislumbrados como uma estratégia para solucionar os problemas gerados pela crescente complexidade na administração de sistemas computacionais e, por conseguinte, para diminuir os custos envolvidos. A Computação Autônoma [Horn 2001] prega que os software sejam desenvolvidos com as seguintes capacidades: 1) conhecer a si próprio, ou seja conhecer seus componentes, estados, funções e interações com o meio externo; 2) poder se configurar e reconfigurar; 3) realizar constante auto-otimização; 4) ser capaz de se reparar; 5) ser capaz de se proteger de falhas ou ações externas; 6) Usar auto-adaptação para melhor interagir com o meio-ambiente e sistemas vizinhos. Além destes requisitos, estes sistemas também

devem poder antecipar suas necessidades, manter sua complexidade oculta e utilizar tecnologias não-proprietárias e baseadas em padrões abertos.

Iniciativas de tornarem a administração dos SGBD mais autônoma já podem ser percebidas através de características de autodiagnóstico e auto-aperfeiçoamento que aos poucos são introduzidas, a cada versão lançada no mercado, nos SGBD mais usados comercialmente (e.g. Monitor Automático de Diagnóstico do Banco de Dados, do Oracle 10g e o *Index Tunning Wizard*, do Microsoft SQL Server). Pesquisas acadêmicas também têm sido realizadas com o intuito de desenvolver sistemas que dotem SGBD de capacidades de autogerenciamento. Dentre as pesquisas acadêmicas realizadas na área de computação autônoma aplicada a SGBD, podemos destacar o Sistema ADAM [Ramanujam e Capretz 2005] e o Sistema DBSitter [Carneiro et al. 2004].

O Sistema ADAM, da Universidade de Western Ontário, Canadá, foi concebido como um invólucro de auto-administração para SGBD relacionais, desenvolvido em arquitetura multiagentes e prototipado para resolução de alguns tipos de falhas para o SGBD Oracle 8.1.7.

O Sistema DBSitter é um sistema projetado na Universidade Federal de Pernambuco, Brasil, concebido como uma sociedade de agentes inteligentes que utiliza um repositório de problemas mais comuns de SGBD e suas soluções e tem como objetivos oferecer funcionalidades de detecção, prevenção e correção automática desses problemas ou apresentar sugestões de solução, além de permitir a funcionalidade de aprendizado do sistema. O Sistema DBSitter utiliza tecnologias de código aberto que garantam a possibilidade de extensão do produto.

Embora existam iniciativas de desenvolvimento de funcionalidades de autogerenciamento para SGBD, ainda há uma carência de especificações formais de arquiteturas que orientem a construção de componentes que implementem essas funcionalidades nos SGBD.

1.1 Objetivos

Nesta dissertação, conduzimos nossos esforços para a aplicação dos princípios da Computação Autônoma à área de Bancos de Dados. O objetivo do presente trabalho é propor o *Framework* DBSitter-AS, uma especificação arquitetural formal que define como construir componentes de software que forneçam autonomia de gerenciamento para SGBD.

Inspirado no Sistema DBSitter [Carneiro et al. 2004], o DBSitter-AS (DBSitter é uma analogia com a palavra inglesa para babá, babysitter, e AS, abreviatura do

Inglês *Architectural Specification*) foi criado procurando respeitar os princípios originais do DBSitter e também os princípios da Computação Autônoma.

Os objetivos específicos do DBSitter-AS são:

- Modelar uma sociedade de agentes que detectem, previnam e corrijam falhas de SGBD, de forma autônoma;
- Definir os papéis e funcionalidades a serem desempenhados por cada agente e a forma de comunicação entre eles;
- Definir a forma de coordenação de agentes empregada;
- Estabelecer um modelo de persistência de informações que permita maior robustez à arquitetura;
- Permitir execução de atividades orientadas por políticas organizacionais e regras de negócio.

1.2 Metodologia

Para alcançar os objetivos, realizaremos as atividades listadas abaixo: A primeira será investigar os fundamentos, princípios e modelos empregados em Computação Autônoma. Depois, tendo em vista que o DBSitter-AS será modelado como uma sociedade de agentes, realizaremos estudos de metodologias específicas para desenvolvimento de Sistemas Multiagentes e selecionaremos a mais indicada ao sistema que pretendemos modelar.

Em seguida será realizado um levantamento das características dos tipos falhas de SGBD, as atividades que os ABD realizam para solucioná-las e requisitos dessas atividades. Esta atividade tem o intuito de respaldar o conhecimento sobre formas de automatização dessas atividades de correção e impactos gerados sobre o SGBD, sobre o usuário e sobre a empresa.

O próximo passo será definir a especificação de um conjunto de modelos, recomendações e definições arquiteturais, que formará o *framework* DBSitter-AS. Para exemplificar a criação de um componente que siga as especificações propostas, será realizado experimento de desenvolvimento e sua implementação.

1.3 Organização do Documento

Esta dissertação está estruturada da seguinte forma:

Capítulo 2: A Computação Autônoma

Neste capítulo apresentamos uma visão geral de Computação Autônoma. Descrevemos as principais arquiteturas utilizadas nesta área de pesquisa e fazemos uma revisão do estado da arte das características de gerenciamento autônomo aplicadas aos principais Sistemas de Gerenciamento de Bancos de Dados de mercado e iniciativas acadêmicas de pesquisa. No final do capítulo, discutimos vários exemplos de aplicação das características fundamentais da Computação Autônoma aos SGBD.

Capítulo 3: Sistemas Multiagentes e Sistemas Autônomos

Neste capítulo apresentamos as principais técnicas e conceitos de Inteligência Artificial usados na elaboração do *Framework* DBSitter-AS, a saber: agentes inteligentes e Sistemas Multiagentes (SMA) e as metodologias de desenvolvimento de SMA. Para cada um deles, apresentamos as definições, vantagens e características mais importantes que justificaram suas escolhas como fundamentação teórica para modelagem do DBSitter-AS. Também são discutidas as arquiteturas de integração entre SMA e SGBD.

Capítulo 4: O *Framewok* DBSitter-AS

Este capítulo descreve o framework desenvolvido neste trabalho. Ao longo do relato, apresentamos as principais características do sistema, a evolução do raciocínio usado na sua concepção e sua arquitetura.

Capítulo 5: Uma implementação do DBSiter-AS

Neste capítulo relatamos a implementação de um experimento de desenvolvimento de componente autônomo provedor de autonomia para SGBD, as tecnologias utilizadas e descrevemos os resultados obtidos.

Capítulo 6: Conclusões e Trabalhos Futuros

Este último capítulo apresenta as considerações finais sobre o trabalho desenvolvido, suas principais contribuições e algumas propostas de trabalhos futuros.

2 A Computação Autônoma

A crescente diversidade e abrangência das soluções empregadas na indústria da Tecnologia da Informação (TI) tornam a administração dos sistemas computacionais atividade cada vez mais complexa. A demanda cada vez maior por pessoal extremamente especializado e a velocidade da evolução tecnológica podem levar a indústria de TI a uma crise de gerenciamento – esse é o mote decisivo que dá à Computação Autônoma a importância e atenção das comunidades de pesquisa e desenvolvimento comerciais e acadêmicas atuais.

A Computação Autônoma propõe a utilização da tecnologia para gerenciar tecnologia [IBM Autonomic Blueprint 2006], como uma forma essencial de lidar com o problema da complexidade inerente aos sistemas computacionais atuais, onde atividades de gerenciamento, hoje realizadas por administradores humanos, são delegadas para processos automáticos. Essa abordagem prega que componentes de software sejam projetados de maneira a incorporar mecanismos de autogerenciamento. Esse novo paradigma da Ciência da Computação é uma possibilidade real para evitar que o acelerado aumento de complexidade de gerenciamento, necessidade de integração e diversidade de tecnologias venha a tornar caótico o ambiente de Tecnologia da Informação mundial.

Neste capítulo, inicialmente apresentaremos uma visão geral do paradigma de Computação Autônoma, suas características, aplicação, benefícios e desafios a serem superados durante seu processo de evolução. Em seguida, serão abordadas questões relativas às arquiteturas propostas para implementação da Computação Autônoma e aplicação da Computação Autônoma na área de Banco de Dados.

2.1 Histórico

Em 2001, Paul Horn, em seu manifesto "*Autonomic Computing: IBM's Perspective on The State of Information Technology*" [Horn 2001] chamava apropriadamente a atenção da comunidade de TI para os aspectos complexos do gerenciamento das tecnologias atuais. Em especial, Horn argumentou que é impossível gerar recursos de acompanhamento e intervenção para os sistemas implantados, na mesma velocidade em que estes são criados. A velocidade das inovações das linguagens de programação, técnicas e arquiteturas de interconectividade não são acompanhadas por preocupações de gerenciamento individuais dos componentes de software que compõem os sistemas e ambientes computacionais inteiros. Essa situação tem o potencial de causar uma crise de

enorme complexidade. Uma conclusão torna-se óbvia: ou se muda radicalmente o modo como software e sistemas são concebidos e desenvolvidos ou não haverá como gerar recursos para administrá-los. A forma como isso pode ser alcançado depende fundamentalmente de quão os sistemas de TI poderão ter comportamento autônomo ou minimamente regulado pela intervenção humana.

Segundo Paul Horn [Horn 2001], é chegado o tempo de projetarmos e construirmos sistemas capazes de executarem a si mesmos, ajustando-se a circunstâncias variadas e preparando seus recursos para lidar mais eficientemente com as cargas que impusermos a eles. Estes sistemas autônomos devem antecipar necessidades e permitir que usuários possam se concentrar no que querem atingir, em vez de configurar como o sistema deve se ajustar para levá-los onde desejem.

O alerta dado foi absorvido pela comunidade de Tecnologia da Informação mundial [Autonomic Computing 2007]. Nos últimos anos temos presenciado a incorporação de características para autonomia nos sistemas que estão sendo desenvolvidos e também em cada versão de software lançada (e.g. software e hardware "*plug-ang-play*" e "*wizards*" de configuração). Os grandes fabricantes mundiais de software têm montado divisões e laboratórios específicos para pesquisa da computação autônoma, assim como na comunidade acadêmica se percebe o número crescente de eventos acadêmicos e trabalhos publicados nessa área.

A Computação Autônoma pode ser aplicada em praticamente todos os campos da computação onde recursos de autogerenciamento são necessários e em áreas que necessitem auto-adaptação a condições diversas de contexto operacional ou de demanda de recursos em tempo real. Como exemplo, temos software de missão crítica e alta disponibilidade de informação (e.g. recuperação de falhas, controle de tráfego ou correção de rota), aplicativos para resposta dinâmica a prioridades de negócio (e.g. *e-commerce*), software de análise massiva de transmissão de dados em tempo real e aplicações para detecção de anomalias e diagnóstico automático em diversas áreas. O gerenciamento inteligente de recursos computacionais distribuídos, replicação de dados e aplicações para auditoria e segurança (e.g. detecção e tratamento de intrusão eletrônica) também são áreas nas quais o desenvolvimento de componentes autônomos vem sendo percebido. Nas Seções 2.3 e 2.4 são descritas algumas das iniciativas de pesquisa desenvolvidas em Computação Autônoma.

2.2 Significado e Benefícios da Computação Autônoma

Paul Horn [Horn 2001], ao definir Computação Autônoma, nos remete ao sistema mais perfeito conhecido – o corpo humano e seu sistema nervoso autônomo; onde cada órgão de maneira harmônica com os demais gerencia a si próprio, sem a

necessidade da consciência cerebral emitir vozes de comando a todo instante para todas as funções vitais ou de baixo nível.

Por Computação Autônoma, define-se [Horn 2001] o conjunto de características que permitem que sistemas computacionais possam gerenciar a si mesmos resultando em objetivos de alto nível definidos por seus administradores. Esses objetivos traduzem-se em características automáticas que permitam instalação, configuração, otimização, manutenção, integração e gerência de conflitos. Essas características são cada vez mais desejadas, já que em breve não mais será possível para especialistas humanos realizarem tais tarefas, dado não apenas à complexidade, mas também ao volume de tarefas necessárias para administração dos sistemas computacionais usados.

Oito aspectos caracterizam os Sistemas de Computação Autônoma [Horn 2001] e ditam que **todo Sistema Autônomo deve**:

1. ***Conhecer a si próprio**, ou seja, ter uma identidade de sistema, conhecer seus componentes, estados, funções e interações com o meio externo;*
2. *Possuir **capacidade de se configurar e reconfigurar**;*
3. ***Realizar constante auto-otimização**;*
4. *Ser **capaz de se reparar**;*
5. *Ser **capaz de se proteger**;*
6. *Usar **auto-adaptação** para melhor interagir com o meio-ambiente e sistemas vizinhos;*
7. *Ser uma **solução não-proprietária**, aberta e baseada em padrões;*
8. ***Prever e antecipar recursos** otimizados necessários, enquanto mantém a **complexidade oculta**.*

Enxergando a analogia das cadeias biológicas, percebemos que em todas as escalas (e.g células, órgãos, sistemas, organismos) existem unidades que se combinam em subsistemas que também possuem regras e mecanismos de auto-regulação próprios. De forma similar, a essência do autogerenciamento das cadeias de TI serão as pequenas unidades autônomas. Essas unidades pouparão a grande massa de gerenciamento e permitirão pensar regulação focada em cenários mais estratégicos de integração e em termos de negociação de serviços de TI.

A incorporação de características de computação autônoma em produtos de Tecnologia da Informação (TI) é inerente à sua evolução. Através dessa evolução, fabricantes de software buscam auxiliar na produtividade dos profissionais que serão usuários, redução de custos e, em ultima análise, colaborar com a efetividade do negócio.

Podemos citar como benefícios trazidos pelo desenvolvimento de mecanismos de computação autônoma [IBM Autonomic Blueprint 2006]:

- Melhoria do tempo de resposta e qualidade de serviços (ou QOS, *Quality of Services*);
- Redução do custo total de propriedade (ou TOC, *Total Ownership Cost*). O custo total de propriedade é a soma dos custos e benefícios diretos e indiretos relacionados à aquisição de um componente de TI (e.g. custo do software, implantação, configuração, capacitação de pessoal para efetiva utilização, adaptação ambiental e cultural, redução de gastos);
- Diminuição do tempo gasto até atingir a utilização eficiente do recurso. Este benefício decorre da automação seletiva de processos e redução de requisitos de tempo e habilidades profissionais necessárias;
- Diminuição no tempo de retorno do investimento (ou ROI, *Return of Investiment*).

2.2.1 Características e Desafios da Computação Autônoma

Para que sistemas de computação autônoma possam incorporar as oito características citadas em sistemas definidos como Sistemas Autogerenciáveis, quatro aspectos são fundamentais: autoconfiguração, auto-otimização, auto-reparação e autoproteção [Ganek e Corbi 2003] [Kephart e Chess 2003]. Os sentidos desses quatro aspectos devem evoluir na indústria de desenvolvimento de software, para um significado ampliado, quais sejam:

- **Autoconfiguração**, onde software e hardware se ajustem automaticamente ao meio-ambiente e sem adaptações forçadas. Esse ajuste deve ocorrer na adição de recursos em uma instalação sem interrupção de serviços e com mínima necessidade de intervenção humana. Capacidades "*plug-and-play*", "*wizards*" de configuração e conexões sem fio devem ser cada vez mais usadas. Mas o verdadeiro desafio para autoconfiguração de serviços ocorre no nível empresarial, onde a adição de recursos deve se ajustar da melhor forma à infra-estrutura existente (e.g serviços de comércio eletrônico) e políticas organizacionais vigentes.
- **Auto-otimização**, em termos de componentes que permanentemente busquem no ambiente, aspectos ou mudanças que possam melhorar sua própria eficiência e maximize a utilização de recursos, de modo a se adaptarem às necessidades dos usuários, sem necessidade da intervenção humana. Esse aspecto também

deve ser estendido a múltiplos sistemas e possivelmente heterogêneos para melhor balanceamento e distribuição de cargas de trabalho.

- **Auto-reparação**, em termos de detecção, diagnóstico e reparo de problemas em software e hardware. O sistema deve ser capar de prever falhas e tomar ações que previnam a descontinuidade de serviços, buscando a confiabilidade e disponibilidade dos sistemas aplicativos da empresa.
- **Autoproteção**, em termos de prevenção a acessos não autorizados ou falhas que comprometam a segurança do sistema ou dos recursos computacionais da empresa como um todo. Os sistemas devem realizar detecção de intrusão, emissão de alertas e prever identificação e autorização de usuários em contextos de acesso diferentes, o que reduzirá a sobrecarga de administradores humanos.

A estrutura básica de um sistema autônomo deve ter habilidades que permitam sua existência em ambiente distribuído e orientado a serviços. A combinação de subelementos autônomos que através de protocolos de comunicação que negociam serviços e recursos com outros subelementos e juntos atingem objetivos maiores ou lidam com situações de exceção, sempre respeitando as políticas de alto nível, formam as bases da arquitetura da Computação Autônoma. Outra característica importante dos sistemas autônomos é que eles devem aderir a padrões abertos.

As bases teóricas da Computação Autônoma residem em grande parte nas técnicas de Inteligência Artificial [Russell e Norvig 2003], que permitam utilização de recursos de diagnóstico de problema ou utilização de agentes inteligentes [Wooldridge 2001].

2.2.2 Desafios e Considerações em Pesquisa Sobre Computação Autônoma

Para que as organizações possam atingir um nível de maturidade suficiente, onde as estruturas das operações internas e de negócio estejam suficientemente alinhadas ao uso de computação autônoma, é necessário que processos, ferramentas e competências atingidas, evoluam de uma maneira gradual.

Grandes desafios devem ser superados pela indústria de desenvolvimento de software atual no seu caminho para a produção de elementos autônomos que aos poucos possam ser incorporados nas tecnologias usadas pelas organizações comerciais. Esses desafios dizem respeito à pesquisa e seus aspectos conceituais, arquiteturais, de *midlleware* e de aplicação [Parashar 2005].

Desafios Conceituais

São relativos à pesquisa e definição de modelos e abstrações para:

- Especificação, compreensão, controle e implementação de comportamentos autônomos.
- Adaptação de teorias e modelos clássicos para aprendizado de máquina, otimização e controle de sistemas multiagentes e dinâmicos;
- Elaboração de modelos de negociação multilateral entre elementos autônomos;
- Concepção de modelos estatísticos para sistemas em redes de larga escala que permitam predição de problemas, baseados em sensores individuais.

Desafios Arquiteturais

Dizem respeito à construção de aplicações e sistemas através de elementos autônomos, que gerenciem seus comportamentos internos e relacionamentos com outros elementos de acordo com políticas estabelecidas por humanos e onde comportamentos locais e globais possam ser especificados, implementados e controlados de maneira previsível e robusta.

Especificamente em relação ao Elemento Autônomo, são encontradas adversidades em todas as etapas do seu ciclo de vida. Este ciclo de vida [Kephart e Chess 2003] caracteriza-se pelas etapas de projeto e implementação, instalação, configuração, otimização, atualização, monitoração, determinação de problemas e recuperação até desinstalação ou substituição.

Em cenários mais complexos podemos pensar em sistemas autônomos como uma grande rede de fornecimento de serviços, onde elementos autônomos podem escolher fornecedores, clientes e trocá-los, em caso de baixa qualidade de serviços. Os relacionamentos entre elementos autônomos também carecem de padronizações em seus requisitos, comportamentos e protocolos, em todas as fases do processo de interação entre elementos autônomos: especificação, localização (e contextualização), negociação, provisão, operação e conclusão de acordo.

Desafios de *Middeware*

Temos como desafio, o fornecimento de serviços básicos com comportamento autônomo robustos, disponíveis e escaláveis, a despeito do dinamismo do sistema e da aplicação. Isto é possível através da inclusão no *midleware* de fatores como a descoberta de outros serviços, troca de mensagens, segurança, privacidade, confiança, dentre os principais.

Desafios de Aplicação

Estes desafios traduzem-se na concepção e desenvolvimento de aplicações capazes de autogerenciamento (autoconfiguração, auto-adaptação, auto-otimização, autoproteção e auto-reparação) e na concepção de modelos de programação, *frameworks* e serviços de *middleware* que dêem suporte à definição de elementos autônomos e composição de elementos autônomos. Esses modelos e serviços devem suportar definições de conteúdo, sensibilidade ao contexto, execução, orientação a políticas e gerenciamento das aplicações.

2.3 Arquiteturas em Computação Autônoma

Padrões arquiteturais e padrões de projeto ajudam a permitir que programas desenvolvidos em plataformas diferentes e em especial por fabricantes diferentes possam trabalhar em conjunto. Em face à atual complexidade dos Sistemas TI, integração de sistemas e processamento de informação em larga-escala, preocupações em relação à padronização no desenvolvimento de sistemas mostram-se legítimas e pertinentes, tanto pela comunidade acadêmica quanto pelas corporações comerciais.

Especialmente para computação autônoma, que busca a reduzir custos e ocultar a complexidade dos sistemas de TI, faz-se necessário que modelos de desenvolvimento, modelos de projeto e modelos de interação entre sistemas sejam definidos e adotados pela comunidade de TI.

Considerando a computação autônoma como estruturada em sistemas compostos por elementos autônomos, podemos conceber esses sistemas como sistemas multiagentes (SMA) [Wooldridge 2001], o que torna claro a importância de arquiteturas orientadas a agente de software [Kephart e Chess 2003].

Algumas arquiteturas genéricas de construção de sistemas de software autônomos (e.g. arquitetura de referência da IBM [IBM Autonomic Blueprint 2006]) já se encontram propostas e disponíveis tanto nos meios acadêmicos como comerciais, sendo grande parte dessas arquiteturas passíveis de serem implementadas utilizando-se o paradigma dos sistemas multiagentes.

Em relação às arquiteturas devemos fazer uma distinção entre arquitetura do elemento autônomo e arquitetura de relacionamento entre elementos autônomos. Nas seções a seguir, essas arquiteturas são apresentadas em maior detalhe.

2.3.1 Os Elementos Autônomos

As arquiteturas de elementos autônomos definem as responsabilidades de autogerenciamento de componentes individuais guiadas por políticas. Um elemento autônomo é definido como a menor unidade de uma aplicação autônoma ou sistema [Parashar 2005]. Pode ser uma base de dados, um subsistema de armazenamento em disco, um aplicativo, ou outro software ou módulo de sistema autocontido, que possua interfaces de entrada e saída bem definidas e dependências de contexto[1] bem caracterizadas.

Como requisitos para elementos autônomos, temos [White et al. 2004]:

- Deve ter autogerenciamento embutido (configuração, reparo, otimização, proteção) e resolver problemas localmente;
- Deve ser capaz de estabelecer e manter relacionamentos com outros elementos autônomos. Também deve ser capaz de publicar descrição precisa dos serviços oferecidos, de modo a ser compreendido por outros elementos, além de negociar e estabelecer acordos e compromissos;
- Deve gerenciar seu comportamento para o cumprimento de obrigações firmadas, através de autoconfiguração e auto-ajustes de parâmetros internos. Deve honrar os termos dos compromissos e receber e acatar políticas. Da mesma forma deve ser capaz de recusar relacionamentos que violem as políticas estabelecidas ou fazer contrapropostas. Para desempenhar esses requisitos, o elemento autônomo deve ser dotado de capacidade analítica que dê suporte a essas funções. Uma vez que um elemento receba orientações conflitantes de dois elementos gerentes ou orientação que viole política, ele mesmo deve ser responsável por resolver o conflito, mesmo que invocando outro elemento autônomo para fazê-lo.

White e colegas [White et al. 2004], recomendam fortemente que elementos autônomos devam possuir adicionalmente os seguintes comportamentos:

- Questionar requisitos factíveis quando ao solicitar serviços;
- Oferecer, associado a seu serviço, características de desempenho, tolerância a falha, disponibilidade e segurança;
- Uniformizar requisitos dos seus serviços disponibilizados com os requisitos solicitados a outros elementos;
- Proteger a si próprio contra serviços inapropriados (solicitações e respostas), autenticar todas as solicitações e respostas, bem como se salvaguardar através de acordos ou contratos pertinentes.

[1] Consciência do ambiente de execução e capacidade de reagir a mudanças nesse ambiente.

A Arquitetura de Elemento Autônomo da IBM

Segundo Kephart e colegas [Kephart e Chess 2003] [White et al. 2004], uma arquitetura básica de um elemento autônomo é composta por exatamente um gerente de autonomia e zero ou mais elementos gerenciados, conforme representado na Figura 2-1, e interfaces sensor e atuador.

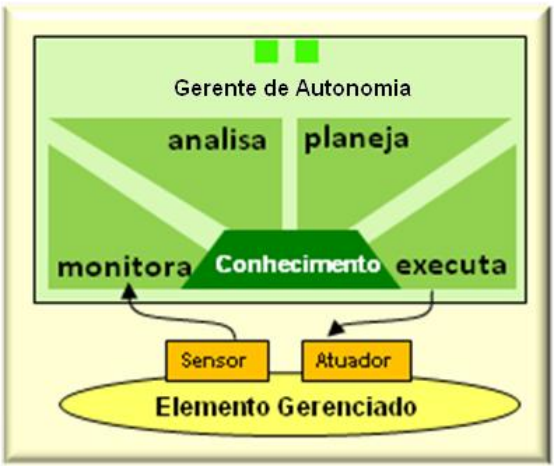

Figura 2-1: Esquema de um Elemento Autônomo, adaptado de [Kephart e Chess 2003]

Essas estruturas implementam um Laço de Controle, formado pelas atividades de monitoração, análise, planejamento e execução. Possuem estruturas internas de armazenamento de conhecimento, capacidade de planejar sua execução em sintonia com políticas e interagem com outros elementos para prover ou consumir serviços computacionais, sendo, portanto, uma arquitetura orientada a serviços.

Agentes de software [Wooldridge 2001] são considerados apropriados para implementações desses elementos autônomos.

2.3.2 Relacionamentos Entre Elementos Autônomos

Arquiteturas de relacionamento são baseadas nos acordos estabelecidos e mantidos pelos elementos autônomos. São governadas por políticas e promovem a tolerância a falha, robustez e autogerenciamento de um sistema.

Políticas de relacionamento são responsáveis pelo provimento dos serviços, gerenciamento em conformidade com políticas globais e interação entre os elementos autônomos. As Políticas facilitam especificações de alto nível para os objetivos e propósitos que um sistema ou organização busca atingir, em consonância com seus requisitos [Bahati et al. 2006]. Isso potencialmente se reflete nos sistemas desenvolvidos, além de servir para aumentar a expressividade da informação do comportamento desejado e reduzir a complexidade do gerenciamento, através de diretivas que permeiem todos os níveis do produto.

Arquiteturas de relacionamentos entre elementos autônomos regem as interações entre esses elementos. As arquiteturas definem o padrão de relacionamentos entre os elementos, que pode ser as mais variadas possíveis (e.g. dinâmicos, efêmeros, oportunistas e colaborativos). Relacionamentos são regidos por regras e restrições e formados através de acordos negociados ou pré-estabelecidos e são determinados pelas políticas.

Estruturas mais complexas necessitam de suporte de conversação definidas por "coreografias" de interações multinível. Sistemas Multiagentes (SMA) são considerados implementações típicas para sistemas de relacionamentos entre elementos autônomos.

Definimos Sistema, então, em uma visão ampla da computação autônoma, como sendo um conjunto de elementos que disponibiliza muito mais automação que a soma de suas partes autogerenciadas. Um exemplo de Sistema Autônomo [Kephart 2002], seus componentes arquiteturais e relacionamentos, é representado na Figura 2-2 por um *Data Center* Autônomo. No exemplo podemos entender o *Data Center* como um Sistema Autônomo composto por elementos autônomos que desempenham funções específicas, fornecem e requisitam serviços de outros elementos. Por sua vez, aplicações internas ao *Data Center* também são sistemas autônomos, compostas por elementos próprios. Percebe-se a existência de elementos gerentes, árbitros de recursos compartilhados, repositórios de políticas e elementos servidores de serviços específicos (e.g. roteador, servidor, *storage*, Banco de Dados (BD)).

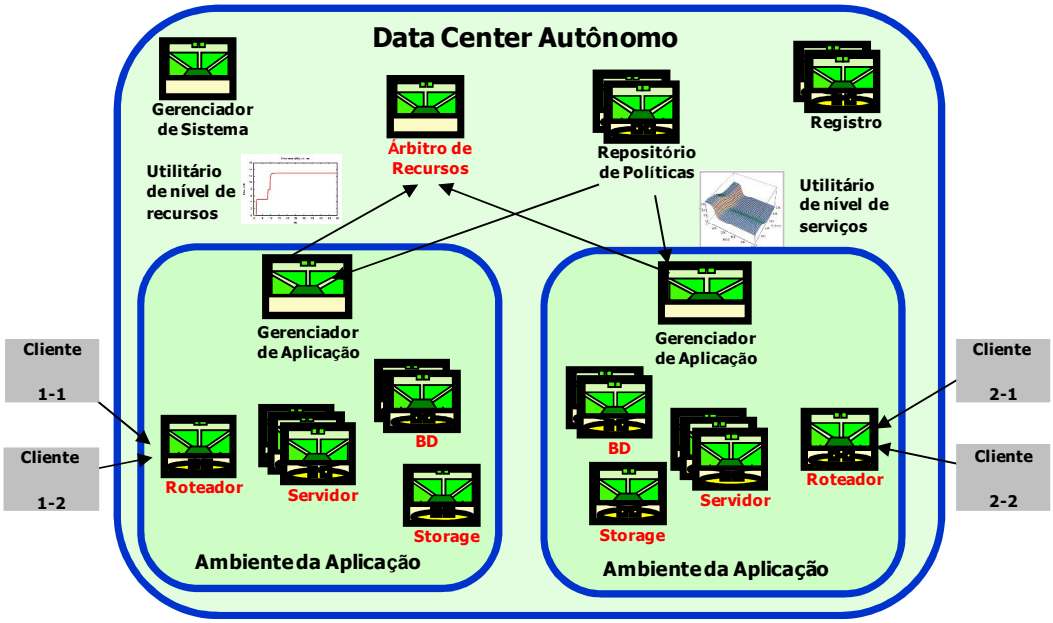

Figura 2-2: Esquema de um Data Center Autônomo, adaptado de [Kephart 2002]

A Arquitetura de Referência da IBM

Através dos esforços dos laboratórios de pesquisa da IBM, muito se tem avançado na busca da definição de padrões arquiteturais para computação autônoma. A seguir são descritos os fundamentos dos padrões propostos.

Esta recomendação arquitetural [IBM Autonomic Blueprint 2006] [White et al. 2004] forma a base do modelo adotado pela IBM para o desenvolvimento de aplicativos com capacidades de autogerenciamento, dentro de uma arquitetura de sistema abertos e para ambientes heterogêneos. A arquitetura da IBM prega que autonomia em sistemas de software deva ser concebida respeitando uma arquitetura de referência, dividida em blocos de construção (*building blocks*). Esses blocos atuam em conjunto em um sistema autônomo para prover capacidades de autogerenciamento.

A integração entre os blocos é feita através de *Enterprise Bus Service Patterns*. Esses blocos podem colaborar entre si através de técnicas padronizadas de comunicação, tais como *Web Services* [Web Services 2007]. Os blocos componentes dessa arquitetura, conforme ilustrado na Figura 2-3 são:

Gerente Manual (*Manual Manager*): Implementação de interface com o usuário que permite que profissionais de TI possam realizar atividades manuais de gerenciamento ou mesmo colaborar com gerentes não-humanos (Orquestradores) para delegação de funções de gerenciamento.

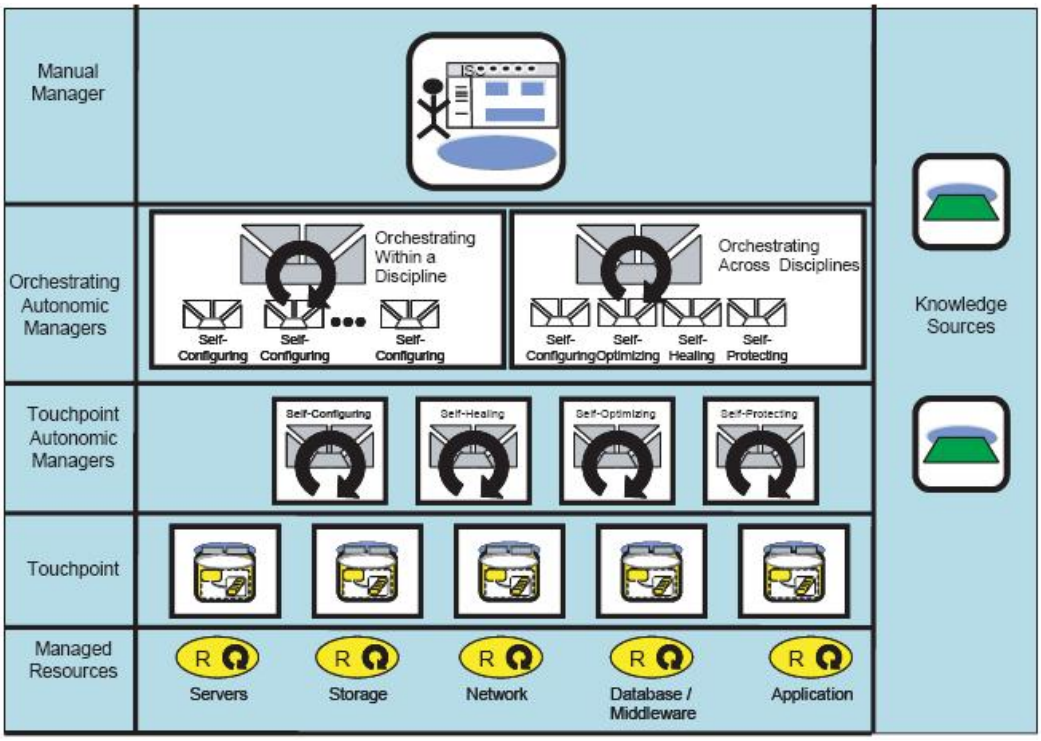

Figura 2-3: Arquitetura de referência da IBM para computação autônoma [IBM Autonomic Blueprint 2006]

Gerentes de Autonomia (Autonomic Managers): São implementações em software responsáveis por automatizar funções de gerenciamento e transmitir seu resultado de acordo com o padrão de comportamento definido pelas interfaces de gerenciamento. Essas estruturas implementam o **Laço de Controle** do elemento autônomo de referência, formado pelas atividades de *monitoração, análise, planejamento* e *execução*. São divididos em duas categorias: **Orquestradores de Autonomia** (Orchestrating Autonomic Managers) e **Gerentes de Pontos de Contato** (Touchpoint Autonomic Managers). O esquema dessa estrutura é exibido na Figura 2-4.

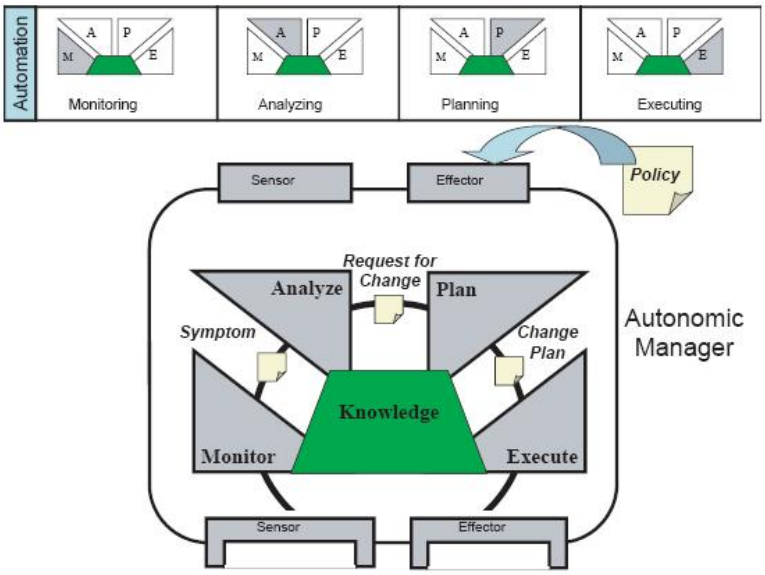

Figura 2-4: Arquitetura detalhada do Gerente de Autonomia [IBM Autonomic Blueprint 2006]

Para realizar todas as atividades do laço de controle, os Gerentes de Autonomia devem ser capazes de perceber o ambiente (*sensor*) e também de atuar em atividades (*effector*). As atuações devem ser regidas por políticas de alto-nível. Políticas de alto-nível são conjuntos de considerações que são projetadas para guiar as decisões que afetam o comportamento de um recurso gerenciado e dizem respeito a objetivos e restrições globais da organização (e.g. sempre realizar verificação de privilégio do solicitante antes de executar solicitação).

Os Orquestradores regem os Gerentes de Pontos de Contato e são responsáveis pelo gerenciamento de autonomia em níveis mais altos da infra-estrutura de TI. Eles podem ser dispostos por categorias isoladas de autonomia (e.g. autoconfiguração) ou englobando várias categorias. Os Gerentes de Pontos de contato, por sua vez, responsabilizam-se pelo gerenciamento específico de uma categoria de autonomia em uma determinada área de atuação e para um recurso específico do sistema (e.g. servidor, rede ou determinada funcionalidade).

Pontos de Contato *(Touchpoints):* São componentes do sistema que exibem o estado (percepções a serem coletadas) e realizam a interface das operações de gerenciamento a serem executadas em relação a um determinado recurso ou conjunto de recursos correlatos. Um exemplo de ponto de contato de recursos de um servidor de banco de dados exporia para acesso os bancos de dados existentes e as tabelas neles contidas. Os pontos de contato podem ser desenvolvidos seguindo padrões de interfaces de gerenciabilidade, como o *Web Services Distributed Management* (WSDM), abordados na Seção 2.3.3.

Fontes de Conhecimento (*Knowledge Sources*): São repositórios de informação (e.g. bancos de dados, registros de log, dicionários) que fornecem conhecimento ao sistema em conformidade com as interfaces previstas. Informações típicas tais como sintomas percebidos, políticas, requisições de mudança ou alteração em planejamentos estão contidas nessas fontes.

A Arquitetura de Referência da IBM não prega qualquer implementação em particular. Tem como objetivo a integração de sistemas heterogêneos, embora a IBM incentive a utilização do IBM Autonomic Computing Tookit [IBM Autonomic Toolkit 2007], um conjunto de ferramentas para análise e construção de componentes autogerenciáveis.

Arquitetura Baseada em Redes de Autômatos

Outro grupo de pesquisa da IBM, dos laboratórios de Zurich, Suíça, baseado nas oito características básicas dos sistemas autônomos (Seção 2.2), propõe uma arquitetura genérica implementada através de redes de autômatos de comunicação [Koehler et al. 2003].

Essa arquitetura propõe que a interação entre o sistema de computação autônoma e seu meio-ambiente ocorra através de três componentes que são responsáveis por todas as interações entre o sistema e o meio-ambiente (Negociação, Execução e Observação) e dois processos: Deliberação e Recuperação de Falhas e uma Base de Conhecimento Compartilhada (Figura 2-5), descritos a seguir:

- ***Componente de negociação (negotiation)***: responsável por receber requisições do meio-ambiente, negociar requisitos de atendimento, tornar-se conhecido por outros sistemas e dar ciência de seus próprios requisitos para outros sistemas. Seu objetivo básico é a construção da especificação de um *comportamento alvo*, que também é armazenada na *Base de Conhecimento Compartilhada*.
- ***Base de Conhecimento Compartilhada (Shared knowledge)***: Quando a especificação de um *comportamento alvo* é registrada na base de conhecimento, é disparado para o processo de Deliberação, um novo comportamento.

- ***Processo de Deliberação (Deliberation)***: Encaminha os novos comportamentos ao Componente de Negociação, tendo a incumbência de atender aos requisitos de auto-otimização e reconfiguração.
- ***Componente de execução (Execution)***: tem apenas uma via de interação de saída para o meio-ambiente, para executar o *comportamento* definido no *Processo de Deliberação*.
- ***Componente de observação (Observation)***: tem apenas uma via de interação de entrada para receber informações sobre o estado do meio-ambiente. Com essa informação, o componente observa o efeito de determinado comportamento e *registra na base de conhecimento*. A composição de registros de comportamentos e efeitos permite que processo de *Recuperação de Falhas* possa analisar se os comportamentos foram executados de maneira correta.
- ***Processo de Recuperação de Falhas (Failure Recovery)***: Responsabiliza-se por atender aos requisitos de auto-reparação e autoproteção, de maneira indireta, via interação com os componentes de execução e observação. Após análise dos possíveis desvios entre as mudanças esperadas e as efetivamente ocorridas no meio-ambiente, envia *comportamento de recuperação* para o componente de execução, caso seja necessário.

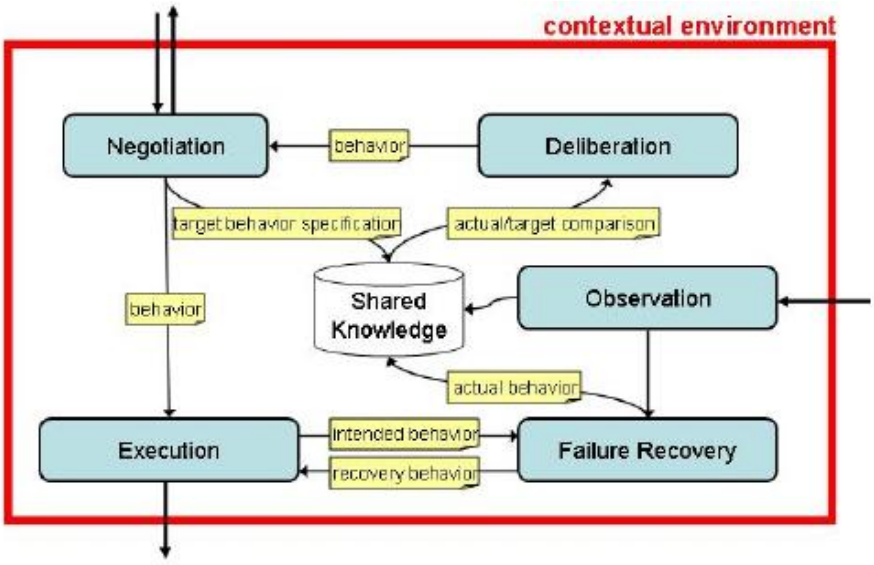

Figura 2-5: Arquitetura genérica baseada em redes de autômatos [Koehler et al. 2003]

Através de poucos componentes a arquitetura busca atender aos requisitos de autoproteção e ocultamento de complexidade. Como modelo computacional é sugerido o uso de sistemas baseados em agentes, já que cada agente pode encapsular métodos de raciocínio que implementem um dos componentes do sistema e também forneçam os mecanismos de comunicação e interação entre eles.

Além de arquiteturas genéricas para elementos e sistemas autônomos, destacam-se esforços para construção de padrões de tecnologias de interoperabilidade, modelos e metodologias, além esforços para incorporação de características de autonomia em novas versões de projetos e sistemas já desenvolvidos. Na próxima seção citaremos algumas dessas iniciativas que contribuem para o desenvolvimento da pesquisa sobre Computação Autônoma.

2.3.3 Outras Arquiteturas e Padrões para Computação Autônoma

Baseados nos conceitos fundamentais da computação autônoma de ser de tecnologia aberta e devido à grande heterogeneidade da indústria de TI, podemos perceber que a tendência natural de mercado é que as tecnologias de autogerenciamento requeiram que recursos, fontes de conhecimento, gerentes e demais componentes sejam desenvolvidos e apoiados por uma diversidade grande de empresas fornecedoras. Isso é especialmente verdade no que diz respeito à integração de tecnologias avançadas, tais como: *Open Grid Computing*[2], *Web Services* [Web Services 2007], tecnologias Multiagentes [Wooldridge 2001] e robótica inteligente ou autônoma.

Este fato nos leva a reconhecer a importância que arquiteturas e padrões adotados devem ter para o desenvolvimento do paradigma da computação autônoma. Dentre as iniciativas da indústria de TI e da academia, podemos dividir os projetos de pesquisa, baseado nas suas características, em:

- Consórcios para o desenvolvimento de padrões abertos para definição de mecanismo de interoperabilidade e sistemas em ambiente heterogêneo. Como exemplo, podemos citar:
    - OASIS[3] (*Organization for the Advencement of Structured Information Standards*), que recentemente aprovou um novo padrão chamado WSDM (*Web Services Distributed Management*) compatível com a arquitetura de referência da IBM para computação autônoma;
    - *Java Community Process*[4], que desenvolve extensões de gerenciamento, especificação de API (*Application Program Interface*) para registros de *log*, serviços para Agentes de software e especificação de *Portlets* para a linguagem Java;
    - IETF[5] (*Internet Engineering Task Force*) que busca desenvolver linguagem de definição de políticas e protocolo de gerenciamento de rede. A

[2] http://www.opengridcomputing.com
[3] http://www.oasis-open.org/
[4] http://jcp.org/
[5] http://www.ietf.org/

representação e padronização de políticas têm papel importante na definição de comportamento global de Sistemas Autônomos.

- DMTF[6] (Distributed Management Task Force) que desenvolve padrões para aplicações e gerenciamento de computação em grupos de trabalho (Working Group Computing) e modelo de comunicação para *Web Services*;

- Projetos que buscam incorporar dentro dos seus sistemas as características da computação autônoma, tais como:
  - AutoAdmin [Autoadmin 2007]: Projeto da Microsoft que busca a redução do custo total de propriedade do software através de pesquisas de auto-ajuste e auto-administração no banco da dados SQL Server. Esse projeto será abordado em maiores detalhes na seção 2.4.1.
  - Océano [Océano 2007]: Pesquisa para desenvolvimento de gerenciamento de recursos computacionais em *Software Farms*, conjuntos de servidores massivamente conectados em paralelo por redes locais de alta velocidade e densamente acoplados. Busca o constante monitoramento de componentes e distribuição da demanda por autonomia, auto-otimização e autoconsciência através da virtualização de recursos de hardware, o que permite hospedagem de aplicações de grande número de usuários.
  - Storage Tank [Menon et al. 2003]: Projeto para desenvolvimento de um dispositivo para gerenciamento do armazenamento de dados, universalmente acessível e multiplataforma. Procura desenvolver características de auto-otimização, auto-reparação, armazenamento e gerenciamento de dados baseado em políticas, redirecionamento de servidor e recuperação de dados baseado em *log*;
  - Smart DB2 [Lohman e Lightstone 2002]: Projeto específico da IBM para redução da intervenção humana e redução de custos no SGBD DB2. Visa implementação de características de auto-otimização e autoconfiguração. Esse projeto será abordado em maiores detalhes na seção 2.4.1;
  - Sabio [Pool 2002]: Classificador autônomo de grandes volumes de documentos, baseado no uso de palavras e frases. Procura o desenvolvimento de características de auto-organização e autoconsciência;
  - OceanStore [Oceanstore 2007]: Prejeto para desenvolvimento de dispositivo de armazenamento de dados persistente, global, consistente e de alta disponibilidade. Dotado de características de auto-reparo, auto-otimização, autoconfiguração e autoproteção, possui funções de memória *cache* baseado em políticas, replicação autônoma, e monitoração, teste e reparo contínuos;

[6] http://www.dmtf.org/

- Tivoli Autonomic Engine [Ganek 2003]: Componente da ferramenta de monitoração de ambientes computacionais Tivoli que fornece correlação entre servidores de múltiplos sistemas de TI para auxílio em análise de causas da falha e ações corretivas automatizadas.

- Pesquisas que buscam desenvolver modelos, ambientes de desenvolvimento e programação e metodologias que apóiem o desenvolvimento de aplicativos computacionalmente autônomos. Podemos citar como exemplo:
  - Unity [Chess et al. 2004]: Projeto de pesquisa que visa o desenvolvimento de comportamentos para elementos autônomos. Os componentes foram implementados em linguagem Java[7] e se comunicam através de *Web Services* [Web Services 2007], em conjunto com interface gráfica de gerenciamento, habilitando comportamentos de autoconfiguração, auto-otimização, autoproteção e auto-reparo em aplicações de software.
  - Q-Fabric [Poellabauer 2002]: Pesquisa que busca o fornecimento de suporte ao gerenciamento *on-line* contínuo de sistemas através de auto-organização. As características de Qualidade de Serviço *(QoS - Quality of Services)* são gerenciadas através de adaptações à realidade de cada sistema gerenciado.
  - vGrid [Khargharia et al. 2003]: Inspirado nos gerentes de memória virtual implementados nos processadores, o vGrid usa Processamento em Grade para compartilhar recursos escassos com um grande número de aplicações, liberando-as das tarefas de gerenciamento e execução desses recursos.
  - QADPZ (Quite Advanced Distributed Parallel Zystem) [Constantinescu 2003]: Projeto que propõe uma arquitetura que busca a utilização de computadores com ambientes heterogêneos para execução distribuída de aplicações, utilizando processamento em tempo ocioso. Para conseguir seus objetivos, características de autoconhecimento, autoconfiguração, auto-otimização e auto-reparação foram desenvolvidas.
  - Astrolabe [Birman et al. 2003]: Concebido para construção aplicações de computação distribuída de larga escala, tais como *Data*, o Astrolabe monitora mudanças em estados de uma coleção de recursos distribuídos e reporta sumários para seus usuários. É implementado em protocolo *peer-to-peer*, o que o faz operar sem necessidade de servidor central. Tem capacidade de executar Mineração de Dados e Fusão de Dados. A partir das características mencionadas, o Astrolabe implementa autogerenciamento, autoconfiguração e auto-reparação.

[7] http://java.sun.com/

- Legacy SYstems [Fuad e Oudshoorn 2006]: Tem como objetivo a transição para o provimento de características de autonomia para sistemas legados, de uma maneira fácil para o programador da aplicação. Se utiliza da injeção de *wrappers* no código legado através de um sistema próprio que opera em tempo real.
- Autonomia [Dong et al. 2003]: Um modelo baseado em *midleware* que fornece infra-estrutura para habilitação características de autonomia para o gerenciamento e controle de aplicações de paralelismo em larga escala e distribuídas. Implementa autoconfiguração, auto-estruturação e auto-reparo e busca desenvolver, em novas versões, auto-otimização e autoproteção.
- eModel [Crawford e Dan 2003]: Se propõe a mapear requisitos de computação autônoma em tempo de execução: medição de carga em tempo real, análise e predição. É possível utilizar o eModel como uma ferramenta de Planejamento de Capacidade e também como forma de aumentar o gerenciamento autônomo de sistemas.
- IBM Autonomic Computing Toolkit [IBM Autonomic Toolkit 2007]: Coleção de componentes, ferramentas, cenários e documentação projetados para usuários que queiram aprender, adaptar ou desenvolver comportamentos autônomos em seus produtos ou sistemas.

2.4 Computação Autônoma em Bancos de Dados

Os SGBD atuais são usados para uma grande diversidade de aplicações (e.g. Data Warehousing, e-commerce, aplicações multimídia e Bancos de Dados distribuídos) com as mais variadas características. A diversidade de aplicação torna os SGBD sistemas dotados de grande número de características que precisam ser ajustadas e monitoradas com acuidade. Isso faz com que os SGBD atuais sejam ferramentas complexas e caras em termos de administração.

Por outro lado, sem SGBD, Sistemas de Informação Computacionais não teriam como controlar de maneira eficiente o armazenamento e manipulação de informação. SGBD como Oracle, DB2 ou SQL Server possuem centenas de parâmetros de configuração que regem seus funcionamentos de maneira adequada ou de maneira aperfeiçoada em função de determinada condição; a despeito desse fato, poucos profissionais embora bastante especializados, detêm o conhecimento de como realizar sintonia para desempenho ou ajustes finos.

A complexidade de administração enfrentada pelos ABD e projetistas de BD inclui [Lightstone et al. 2003] a realização de tarefas como:

- Decisão de *hardware* e *software* para compra;
- Projeto lógico do BD (e.g. tabelas, visões, particionamento, cluster);
- Definição da topologia de armazenamento;
- Definição de políticas de segurança;
- Definição de esquemas distribuídos;
- Planejamento de recuperação e alta disponibilidade;
- Manutenção cotidiana (e.g. estatísticas, desfragmentações de dados, dados em cluster, replicações);
- Realização de cargas de dados;
- Ajustes finos em função de sobrecarga (e.g. alocação de memória, parâmetros de configuração);
- Provisão de recursos (hardware, software, banda de comunicação);
- Evolução contínua de esquemas de dados e projeto

Podemos completar essa lista com a otimização de consultas SQL. A referida complexidade também é evidenciada no aumento da necessidade de recursos de infra-estrutura, como CPU, disco, servidores em *cluster*, discos de memória em RAID, pelo uso de ambientes heterogêneos e pela adoção de modelos não-estruturados de dados (e.g. vídeo, som, imagem).

Todos esses fatores de importância tornam os administradores humanos de SGBD mais susceptíveis à falha e encaixam os SGBD na categoria dos sistemas com forte apelo ao desenvolvimento de características de autonomia.

Características que proporcionem algum grau de independência da intervenção humana têm aumentado versão após versão. Historicamente os principais SGBD de mercado já fizeram avanços na área de automatização de tarefas desde o final da década de 1970.

Em relação aos esforços que a indústria de Bancos de Dados tem desenvolvido, podemos perceber as seguintes tendências em relação aos esforços de desenvolvimento de características de autonomia [Lightstone et al. 2003]:

- Mudança para sistemas que realizem auto-ajuste fino contínuo e adaptação a cargas de trabalho em detrimento de sistema ajustados estaticamente ou por intervenção humana;
- Desenvolvimento de sistemas que suportem análises em tempo real de dados em constante alteração;
- Criação de novas classes de funções que suportem integração de informação de sistemas distribuídos e heterogêneos;
- Criação de tecnologias em *Grid* para bancos de dados;

- Substituição gradual de armazenamento local de dados por armazenamento atrelado à rede e disponibilizado para múltiplos ambientes;
- Reconhecimento que paralisações não são aceitáveis e disponibilidade e continuidade de serviço são funções básicas de qualquer SGBD;
- Desenvolvimento de sistemas auto-reparáveis;
- Reconhecimento renovado da importância da segurança de dados em SGBD.

Em geral componentes de hardware e software para gerenciamento de dados vêm evoluindo no sentido de adquirir um comportamento mais autônomo [Ganek e Corbi 2003], como coleta, análise e utilização de estatísticas para aprendizado, identificação de potenciais "gargalos" de desempenho, otimização automática de consultas baseado em análises históricas e tempos de respostas.

Segundo Elnaffar [Elnaffar et al. 2003], duas estratégias vêm sendo empregadas para atingir níveis maiores de autonomia em SGBD: a reescrita do código do sistema com inclusão de técnicas que o torne autônomo e a incorporação gradual de mecanismos de autonomia nos sistemas atuais. A tendência observada pela indústria é que se utilize a segunda estratégia.

A seguir serão detalhadas, um pouco mais, algumas das iniciativas citadas de criação de SGBD autônomos.

2.4.1 As Iniciativas de Mercado

Além da concorrência tecnológica, o grande motivador para que os fabricantes de SGBD incorporem características de autonomia nos seus produtos é a diminuição do custo total do software para o consumidor final, o que, em função direta, gera penetração em novos mercados. Para isso é necessário que uma inovação tecnológica aumente a eficiência do produto e não aumente o custo da administração realizada por técnicos especializados.

A seguir são descritos os avanços proporcionados pelos três maiores fabricantes de SGDB e os projetos de pesquisa para seus produtos: IBM (DB2), Microsoft (SQL Server) e Oracle (Oracle 10g) e uma breve descrição das características de ferramentas de análise de desempenho para SGBD de fornecedores independentes.

O Projeto SMART da IBM

O projeto SMART [Lohman e Lightstone 2002] é a vertente para bancos de dados da iniciativa de computação autônoma dos laboratórios de pesquisa da IBM. Iniciado em 2000 e contando com a participação de dois centros de pesquisa da IBM,

dois laboratórios de desenvolvimento, além de universidades consorciadas, o projeto busca desenvolver e implementar nas versões do SGBD DB2 [Zilio et al. 2000], características que incluam os seguintes componentes principais, de computação autônoma:

- Configuração "*UP and Running*" - um pacote inicial de planejamento, instalação, configuração e desenvolvimento que permite a escolha automática de configurações baseadas em *benchmarks* catalogados, realizados por especialistas.
- Projeto - Desde a versão 6 do DB2 há a ferramenta *Index Advisor*, que recomenda utilização de índices baseados em cenários. A iniciativa SMART ampliou o conceito para recomendação de índices candidatos, visões e tabelas particionadas.
- Manutenção – o *Maintenance Advisor* utiliza estatísticas para informar quais objetos do banco de dados necessitam de que tipo de manutenção. Para a plataforma IBM z/OS foi introduzido o conceito de estatística em tempo real, que indica qual objeto foi modificado e as necessidades de manutenção. O objetivo é que essa estratégia seja expandida para outras plataformas como Linux, UNIX e Windows.

  Destaca-se, também, o DB2's LEarning Optimizer (LEO) [Stillger et al. 2001]. Otimizadores de consulta de BD atuais já realizam seleção de estratégias de leitura de forma autônoma em tempo de execução, baseado em algoritmos complexos. A proposta do DB2 LEO é complementar a seleção da estratégia, através da realização do armazenamento de *feedback* sobre consultas anteriores que tenham predicados semelhantes e do auto-ajuste em parâmetros e suposições estabelecidas. Essa otimização permite reduzir substancialmente a necessidade de sintonia fina em consultas com problemas.
- Detecção e correção de problemas – a ferramenta *Health Center* permite monitoração constante do SGBD e emissão de alertas para os ABD por e-mail, pager ou celular e também registrando em arquivos de log, problemas e possíveis soluções. O próximo desafio dessa facilidade será manter uma base que aprenda com as soluções dadas pelos ABD humanos aos problemas encontrados.
- Disponibilidade e Recuperação de Falhas – através do *Recovery Expert,* uma ferramenta que busca identificar automaticamente a melhor maneira de recuperar tabelas até de um dado período especifico no tempo, baseado nos logs de erro.

O Projeto Microsoft AutoAdmin

O projeto AutoAdmin [Autoadmin 2007], da Microsoft vai ao encontro das estratégias de mercado em usar sistemas auto-administráveis e auto-aperfeiçoáveis

para diminuir os altos custos da aquisição e manutenção de SGBD, presentes em aplicações computacionais de porte.

O objetivo principal desse projeto é dotar as bases de dados de características que possam rastrear sua utilização e adaptá-las aos requisitos dos aplicativos que as utilizam. Isso permite que as bases de dados possam se auto-aperfeiçoar.

Os frutos do grupo de pesquisas da Microsoft para bancos de dados [Lomet et al.] vêm sendo incorporados ao o SQL Server e no momento, o foco principal é a identificação de problemas em índices e visões, para que esses se adaptem à carga submetida. O *Index Tunning Wizard*, incorporada ao SQL Server desde a versão 7.0, recomenda índices apropriados e utilização de visões.

Em adição aos esforços do projeto AutoAdmin, uma série de outras facilidades também foram lançadas nas duas últimas versões do SQL Server (2000 e 2005) [Microsoft SQL Server] , tais como:

- Atualização automática de estatísticas;
- Gerência automática do tamanho do banco de dados;
- Detecção automática de colunas de tabelas que necessitem de índices ou estão com estatísticas vencidas;
- Ajuste automático de parâmetros, tais como alocação dinâmica de memória para o SGBD como todo ou usuário em particular;
- Criação e gerenciamento on-line de índices;
- Configuração de alarmes de desempenho baseados em limites pré-determinados;
- Ferramenta (*Wizard*) para reorganização de partições e organização de dados tridimensionais (cubos);
- Ferramentas conselheiras para índices e particionamentos.

As Inovações de Autogerenciamento do Oracle 10g

Apresentado pela Oracle como sendo o primeiro SGBD realmente auto-gerenciável, o SGBD Oracle 10g incorpora características de design que buscam otimização de desempenho, segurança e disponibilidade.

A arquitetura implementada [Kumar 2004] permite a captura de informações do uso do SGBD em todas as camadas por uma estrutura chamada Repositório Automático de Carga de Trabalho (*Automatic Workload Repository – AWR*). Essa estrutura armazena um histórico global de desempenho do BD, que permite o diagnóstico e a tomada de ações corretivas pró-ativas.

Também integrado, o Monitor Automático de Diagnóstico do Banco de Dados (*Automatic Database Diagnostic Monitor – ADDM*) permite o diagnóstico integral de problemas com o BD através do cruzamento das informações coletadas pelo AWR

com informações registradas de especialistas em desempenho. Através da busca de gargalos e potenciais causas, é possível fazer uma análise de impacto e oferecer recomendações para o ABD.

Outra estrutura importante, o Gerenciamento Automático de Memória (*Automatic Memory Management – AMM*) realiza a alocação automática de memória RAM em função da carga de trabalho corrente, para usuário ou para o SGBD como todo.

Em conjunto com novas funcionalidades de agendamento (*scheduler*), contidas na console de gerenciamento do Oracle, o *Oracle Enterprise Manager* (OEM), é possível implementação automática de correções baseadas nos registros das estruturas citadas e em ações registradas como corretivas.

Melhorias sensíveis na instalação e configuração do SGBD também merecem destaque, com vários parâmetros de ambiente sendo "percebidos" pelos instaladores automáticos, com menor intervenção do usuário.

2.4.2 Iniciativas de Código-fonte Aberto ou Acadêmicas

As pesquisas realizadas em iniciativas acadêmicas, consorciadas ou não com os grandes fabricantes de SGBD também têm contribuído para o desenvolvimento das técnicas de computação autônoma. Ferramentas, arquiteturas e *Frameworks* que aumentem o potencial de automação para SGBD vêm sendo debatidas e prototipadas, indicando que em breve as inovações propostas, e em parte já desenvolvidas pelos grandes fabricantes, poderão ser partes integrantes de soluções multiplataforma, de código aberto ou de software livre. A seguir são destacados alguns estudos nas referidas áreas.

O Sistema DBSitter - UFPE

O DBSitter [Carneiro et al. 2004] é uma ferramenta desenvolvida na Universidade Federal de Pernambuco (UFPE) que, em fase de protótipo, se propõe a monitorar e gerenciar de forma genérica um SGBD. Construído em plataformas abertas e na linguagem Java, usa técnicas de RMI (Remote Method Invocation), Raciocínio Baseado em Casos (RBC) e Agentes Inteligentes.

A combinação das técnicas citadas permitiu a construção de uma ferramenta distribuída, onde um conjunto de agentes sensores e agentes atuadores monitora o ambiente operacional e o SGBD, de forma coordenada. Isso permite que sinais vitais, tanto do SGBD como do sistema operacional, sejam repassados para o ABD ou que ações pré-determinadas sejam disparadas automaticamente, baseadas na similaridade com eventos já cadastrados. Adicionalmente, a base de conhecimento do

DBSitter é enriquecida com os eventos que acontecem e medidas de eficiência das ações corretivas são coletadas. Outro ponto de destaque é a previsão de eventos, baseada nas estatísticas coletadas. Isso possibilita, por exemplo, que fragmentações em tabelas possam ser corrigidas antes que um ponto crítico seja atingido.

Os requisitos de ser desenvolvido em plataforma aberta, ser configurável para vários tipos de SGBD e utilizar técnica multiagentes são diferenciais importantes do projeto Dbsitter.

O Sistema ADAM – Werstern Ontário University

O sistema ADAM (Autonomous Database Administration and Maintenance) [Ramanujam e Capretz 2005] foi desenvolvido na Universidade de Western Ontario, Canadá, com o objetivo de ser um invólucro de auto-administração para SGBD relacionais.

Concebido em uma arquitetura multiagentes, o sistema foi projetado utilizando a metodologia Gaia [Zambonelli et al. 2003] para desenvolvimento de SMA e prototipado para resolução de alguns tipos de falhas para o SGBD Oracle 8.1.7.

A versão piloto do sistema foi desenvolvida usando o *framework* JADE 2.5 (Java Agent Development Framework) [JADE 2007] e implementou quatro tipos de agentes: agente monitor do BD, agente monitor de objetos de BD e duas instâncias de agentes de *backup* e *recovery* que realizam procedimentos de *backup* e desfragmentação de tabelas. As funcionalidades principais testadas foram monitorações pró-ativas e reativas, atualização de conhecimento e interação entre agentes.

Arquitetura Global para Bancos de Dados Auto-ajustáveis – PUC Rio

Este trabalho de pesquisa desenvolvido na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) propõe duas arquiteturas [Milanés 2004] [Milanés e Lifschitz 2005] que utilizam agentes para realizar auto-sintonia de parâmetros para SGBD. A auto-sintonia é tratada de forma global, já que alterações de parâmetros em SGBD podem ter reflexos no desempenho de outras operações.

A primeira arquitetura apresenta-se mono agente, onde todos os passos envolvidos na escolha e criação de índices são automatizados. A escolha é baseada em uma heurística onde o cálculo dos benefícios de chaves candidatas é avaliado e guiará a criação e remoção automática de índices reais. De forma semelhante, a segunda arquitetura com múltiplos agentes é direcionada a SGBD cuja carga de trabalho se apresente com diferentes sazonalidades e perfis.

Baseado nas arquiteturas propostas e utilizando-se de uma estratégia de reutilizar componentes já existentes no SGBD, um estudo de caso foi desenvolvido

para o SGBD de código aberto PostgreSql [PostgreSql 2007], onde dois mecanismos de auto-ajuste foram implementados: Substituição da estratégia de paginação em cache e ajuste do *Data Contention Trashing*[8].

Da mesma forma, outros trabalhos vêm sendo realizados seguindo os direcionamentos deste grupo de pesquisadores da PUC-Rio [Costa et al. 2003], a exemplo da implementação da criação autônoma de índices em bancos de dados, proposta por Salles [Salles 2004].

Múltiplos Buffer Pools e Buffer Pools Dinâmicos no PostgreSQL – Queen's University

Trabalho realizado na Universidade Queen's em Kingston, Ontario, Canadá, que busca introduzir características de auto-ajuste [Ogeer 2004] no SGBD PostgreSQL [PostgreSql 2007].

A área de *buffer* é a principal área de gerenciamento de memória de um SGBD. A divisão das áreas de *buffer* em várias e também o redimensionamento dinâmico do tamanho dos *buffers* pode propiciar melhor desempenho de leitura e manipulação de dados.

O estudo feito estendeu a versão 7.3.2 do SGBD PostgreSQL para suportar múltiplas áreas de *buffers* e automaticamente redimensioná-las de acordo com a carga de trabalho exigida. Testes rodando o *benchmark* do TPC – B [TPC] comprovaram bons resultados para a abordagem empregada.

2.5 Aspectos Fundamentais da Computação Autônoma Aplicados a SGBD

Além dos já citados aspectos fundamentais da computação autônoma (autoconfiguração, auto-otimização, auto-reparação e autoproteção), há outros dois aspectos que merecem destaque em relação a SGBD [Elnaffar et al. 2003], que são:

- **Auto-organização**: capacidade de reorganizar dinamicamente a disposição de armazenamento das estruturas de dados e auxiliares;
- **Auto-inspeção**: capacidade de coletar, armazenar e analisar informações sobre desempenho e carga de trabalho.

Outro paradigma, também em pesquisa atualmente é a Computação Pró-ativa [Tennenhouse 2000] [Want et al. 2003], que procura ampliar os horizontes da Computação através do reconhecimento da necessidade de monitorar e modelar o

[8] Sobrecarga do SGBD, provocada pela excessiva concorrência a recursos, que eleva drasticamente o tempo de resposta.

mundo físico, especialmente em atividades humanas que estejam, hoje em dia, limitadas pelo grau de envolvimento humano requerido (e.g. aplicações que exijam monitoração constante e análise estatística em tempo real). Os projetos de sistemas pró-ativos são guiados por sete princípios essenciais: conexão com o mundo real, conexão massiva em rede, tratamento da incerteza, antecipação, laço de controle fechado e personalização de sistemas.

2.6 Considerações

Com o grande crescimento na complexidade dos sistemas informatizados atuais e suas demandas por integração, a Computação Autônoma surgiu como uma solução atrativa. Grandes desafios serão encontrados até que ambientes de Tecnologia da Informação (TI) atinjam patamares de autonomia que supram as necessidades advindas da complexidade imposta à administração humana. Estes desafios envolvem tornar os sistemas individuais autônomos e atingir comportamento autônomo para sistemas corporativos em escala global.

O desenvolvimento de características de autonomia tem um forte apelo como solução para administração de produtos complexos e de alto custo total de propriedade, o que justifica sua aplicabilidade aos SGBD.

Do ponto de vista do gerenciamento de SGBD, as características de autonomia são desenvolvidas para substituir parte da força de trabalho dos administradores de banco de dados e que tipicamente podem ser enquadradas em uma das categorias [Elnaffar et al. 2003]:

- *Plug-and-Play SGBD*: características que suporte instalação e operação inicial. Planejamento de carga inicial, configuração e transferência de dados;
- Projeto Lógico e Físico: Dimensionamento e distribuição de áreas de armazenamento, seleção de índices, visões e particionamentos apropriados;
- Manutenção Preventiva: Manter o sistema ativo e trabalhando satisfatoriamente. Desfragmentação e recriação de estrutura, *backups*, geração de estatísticas, gerenciamento de espaços, tabelas, índices e manutenção de objetos de BD.
- Diagnóstico e Correção de Problemas: Identificação de anomalias, determinação de causas, notificação e tomada de ação de correção ou ajuste fino.
- Disponibilidade e Recuperação de Desastres: Ações que visam retornar o SGBD a um estado estável ou recuperá-lo de desastres. Inclui

recuperação de *logs* para *backups* incrementais, e sincronização de bases distribuídas.

Nesse contexto, a Computação Autônoma tem impulsionado os avanços nas últimas versões dos SGBD de mercado e embora muitos desenvolvimentos tenham sido alcançados, ainda muito há para evoluir em direção do Sistema Autônomo de Gerenciamento de Banco de Dados – SAGBD. Características como previsão, inferência e aprendizagem ainda continuam como desafios dos mais difíceis a serem superados.

Dentre as pesquisas que buscam a implementação de características de autonomia em SGBD, podemos identificar duas vertentes: as que focam em resolução e prevenção de falhas em SGBD (e.g. correção de corrupção de tabelas ou arquivos físicos) e as que focam em ajustes de sintonia em processos de SGBD (e.g. melhoria em estratégias de leitura através de seleção de índices, melhoria de utilização de memória ou de *buffers* de leitura).

As evoluções advindas do mercado são marcadas pelo desenvolvimento incremental de características de autonomia, lançadas em conjunto com novas versões dos SGBD e desenvolvidas nos laboratórios do fabricante ou em parcerias comerciais. As instituições acadêmicas têm focado inovações para SGBD livres e de código aberto ou proposto abordagens em camadas externas ao SGBD a ser gerenciado de forma autônoma. Diversas das características inerentes a sistemas autônomos ainda encontram-se bastante incipientes nas alternativas de software aberto, que encontram em iniciativas acadêmicas as poucas oportunidades de reduzir a disparidade em relação aos SGDB de mercado mais utilizados.

Apesar de todas as dificuldades técnicas e de integração, percebe-se que o caminho para o SAGBD começa a ser trilhado e que as próximas versões cada vez mais serão dotadas de características de autonomia, o que mudará radicalmente o cenário de gerenciamento de bancos de dados e da atuação dos profissionais que os utilizam.

A despeito dos recentes avanços, o surgimento de um SGBD realmente autônomo ainda necessitará de grande esforço de pesquisa. Uma das vertentes de pesquisa que buscam o desenvolvimento do SAGBD busca implementar características de autonomia de gerenciamento através do uso de agentes inteligentes de software.

No capítulo 3 apresentaremos os fundamentos dos Sistemas Multiagentes, suas correlações com Sistemas Autônomos e arquiteturas que visam implementação de gerenciamento autônomo em SGBD.

3 Sistemas Multiagentes e Sistemas Autônomos

Conforme visto no Capítulo 2, os elementos autônomos são os componentes basilares dos Sistemas autônomos. Segundo Kephart e Chess [Kephart e Chess 2003], os elementos autônomos possuem ciclos de vida complexos, lidando continuamente com atividades em múltiplos *threads* e examinando e respondendo ao ambiente nos quais estão inseridos. De fato, as características desses elementos autônomos, que são autonomia, pró-atividade e interação baseada em objetivos também são características básicas dos agentes de software.

Assim, podemos entender elementos autônomos como agentes de software e Sistemas Autônomos como Sistemas Multiagentes (SMA). Isso indica a importância que os conceitos de arquitetura orientada a agentes podem ter em computação autônoma.

Sistemas autônomos, em larga escala, serão compostos por agentes de tipos variados (e.g. diretórios de serviços, propagadores, agregadores de dados, agenciadores de serviço, gerentes de dependência, sentinelas, analistas de segurança, monitores de desempenho) que perceberão o ambiente, registrarão e publicarão informação relevante para o bom funcionamento da sociedade, agenciarão serviços e zelarão pela saúde do sistema como um todo.

Assim, neste capítulo detalharemos os agentes autônomos e os Sistemas Multiagentes. Na Seção 3.1 falaremos sobre os Agentes Inteligentes, os Sistemas Multiagentes e as Metodologias de Desenvolvimento SMA. Na Seção 3.2 discutiremos as abordagens arquiteturais para implementação de SGBD baseados em agentes e posteriormente discutiremos as correlações entre estes e os SMA.

3.1 Os Agentes Inteligentes

Sistemas multiagentes são sistemas nos quais vários agentes inteligentes interagem em busca de um conjunto de objetivos ou executam algum conjunto de tarefas [Weiss 1999]. Nesse contexto, agentes são entidades computacionais autônomas, que percebem o ambiente através de sensores e atuam sobre ele através de atuadores [Russell e Norvig 2003]. Além disso, os agentes perseguem seus objetivos e executam tarefas que visam a otimização de alguma medida de desempenho. Enquanto parte de uma sociedade, os agentes podem ser influenciados por outros agentes ou talvez por humanos na busca pelos seus objetivos e durante a execução de suas tarefas.

Wooldridge [Wooldridge 2001] define agente apontando as seguintes características:

1. São entidades de resolução de problemas claramente identificáveis, com limites e interfaces bem-definidas;
2. São situados em um ambiente particular – agentes recebem entradas de acordo com o estado de seus ambientes através de sensores, e atuam no ambiente através de atuadores;
3. São projetados de modo a executar um objetivo específico – agentes têm objetivos particulares para atingir;
4. São autônomos – ou seja, eles têm controle tanto dos seus estados internos como de seu próprio comportamento;
5. São capazes de exibir um comportamento flexível de resolução de problemas, na busca pelos seus objetivos de projeto.

Agentes podem ser reativos ou cognitivos. Os agentes reativos têm um comportamento muito simples: escolhem suas ações baseados unicamente nas percepções que têm do ambiente, embora não tenham representação do ambiente (objetos e outros agentes). Não possuem memória, portanto não têm história dos fatos que aconteceram e das ações que executou. Também não têm controle deliberativo (planejado) de suas ações [Hübner e Sichman 2003].

Nos modelos cognitivos, normalmente se considera que os agentes possuem um estado mental e raciocinam para construir um plano de ações que os leva a um objetivo pretendido. Podem alterar seu funcionamento a fim de adaptarem-se melhor ao seu ambiente (autonomia funcional), são sociáveis (possuem a capacidade de comunicação), são capazes de raciocinar sobre que ações devem realizar e retêm história. Podemos afirmar que agentes cognitivos possuem adequação natural para a construção de sistemas autônomos.

3.1.1 Sistemas Multiagentes

A área de SMA estuda o comportamento de grupos organizados de agentes autônomos que cooperam para a resolução de problemas que estão além das capacidades de resolução individuais. Agentes de software existem e atuam em ambientes que podem ser classificados em acessíveis (permite ao agente obter informação precisa, completa e atualizada) ou inacessíveis; determinísticos (onde a uma ação corresponde um único efeito conhecido) ou não-determinísticos; discretos (onde há um número finito de ações e percepções) ou contínuos; estáticos (não variam com o tempo, exceto pela ação do agente) ou dinâmicos. A mera presença de

outros agentes torna o ambiente dinâmico, do ponto de vista de cada agente [Russell e Norvig 2003]. O mundo real e a Internet são exemplos de ambientes dinâmicos.

Da mesma forma que, no mundo real, as pessoas necessitam de outras para a realização conjunta de atividades, os agentes necessitam interagir e cooperar para que possam solucionar problemas mais complexos ou de natureza distribuída. Com a grande necessidade de interconexão dos recursos computacionais atuais, a situação usual é que agentes interajam com outros, ou seja, atuem em sociedade e compartilhem experiências e conhecimentos para atingirem seus objetivos. Estas sociedades ou comunidades de agentes configuram os Sistemas Multiagentes (SMA).

As propriedades desejadas dos SMA advêm do equilíbrio entre esse aparente paradoxo entre a autonomia existencial dos agentes e o estabelecimento de restrições aos comportamentos individuais em benefício do grupo. Isto significa que um agente não precisa de outro para existir, mesmo que precise da ajuda de outros agentes para conseguir seus objetivos [Hübner e Sichman 2003].

A pesquisa de SMA tem crescido bastante nos últimos anos, em especial nas áreas de Engenharia de Software e IA, na busca de uma metodologia que torne possível a adoção desta tecnologia em massa. Weiss [Weiss 1999] define como razões para o crescente interesse em SMA, as necessidades tecnológicas e de aplicação, pois SMA oferecem uma maneira de entender, gerenciar e usar sistemas distribuídos de larga escala, dinâmicos e abertos.

Além disso, SMA potencialmente têm a capacidade de oferecer as seguintes propriedades [Weiss 1999]:

- Aumento de velocidade e eficiência;
- Aumento de robustez e confiança;
- Aumento de escala e flexibilidade;
- Diminuição de custos;
- Aumento na velocidade de desenvolvimento e reuso.

Sistemas Multiagentes são considerados apropriados para casos onde haja ambientes abertos ou ao menos complexos ou com alto grau de dinamismo ou incertezas; onde o uso de agentes seja uma metáfora mais natural (maioria das organizações, comércio ou ambientes onde haja competição); onde haja distribuição de dados, de controle ou de conhecimento (por exemplo, bancos de dados distribuídos) ou onde haja sistemas legados e seja necessário construir um invólucro através de uma camada de agentes que permita comunicação e cooperação [Wooldridge 2001].

Hübner e Sichman [Hübner e Sichman 2003] enumeram as seguintes vantagens de se usar SMA:

- Viabilizam sistemas adaptativos e evolutivos: o SMA tem capacidade de adaptação a novas situações, tanto pela eliminação e/ou inclusão de novos agentes ao sistema quanto pela mudança da sua organização.
- É uma *metáfora natural* para a modelagem de sistemas complexos e distribuídos: em muitas situações o conhecimento está distribuído, o controle é distribuído, os recursos estão distribuídos. E, quanto à modelagem do sistema, a decomposição de um problema e a atribuição dos subproblemas a agentes permite um alto nível de abstração e independência entre as partes do sistema.
- Toma proveito de ambientes heterogêneos e distribuídos: agentes com arquiteturas diferentes, que funcionam em plataformas diferentes, distribuídas em uma rede de computadores, podem cooperar na resolução de problemas. Isto permite o uso das potencialidades particulares de cada arquitetura e, pela distribuição, melhora o desempenho do sistema.
- Permite conceber sistemas abertos: os agentes podem migrar entre sociedades, isto é, agentes podem sair e entrar em sociedades, mesmo que desenvolvidos por projetistas e objetivos distintos.

A Inteligência Artificial Distribuída (IAD) é o ramo da IA que lida com o estudo, construção e aplicação de SMA. A IAD utiliza o conceito de **inteligência distribuída** [Ferber 1999] cuja metáfora aplicada é o **comportamento social**. O comportamento social é o conjunto de funcionalidades que permitem que os agentes possam coordenar seus conhecimentos, objetivos, habilidades e planos individuais de uma forma conjunta, em favor da execução de uma ação ou da resolução de algum problema onde se faça necessária a cooperação [Zambonelli et al. 2000].

Na IAD, quatro conceitos são fundamentais para a formação e caracterização de uma comunidade de agentes inteligentes: Organização, Comunicação, Negociação e Coordenação, discutidas a seguir.

Organização

Organização é o conjunto de compromissos globais, crenças e intenções comuns aos agentes que querem atingir um objetivo comum [Jennings 1996]. Esses compromissos definem as diretrizes que regem as ações individuais e coletivas de uma comunidade e também uma política de interações entre seus membros e da comunidade com o meio-ambiente. Como exemplos de organizações, temos as famílias, as empresas, agremiações, universidades. Abstrações organizacionais de SMA são comumente representadas como na Figura 3-1.

Metáforas organizacionais são consideradas úteis para o estudo de comunidades de agentes, porque permitem uma abordagem analítica, composta por [Ferber 1999]:

1. *Análise Funcional* que descreve as funções de um SMA nas suas diferentes dimensões[9];
2. *Análise Estrutural*, que distingue entre várias possíveis formas de organização e identifica alguns parâmetros estruturais;
3. *Concretização de Parâmetros*, que lida com a transição de uma estrutura organizacional para uma organização concreta e aborda a dificuldade da efetiva realização de uma organização multiagentes.

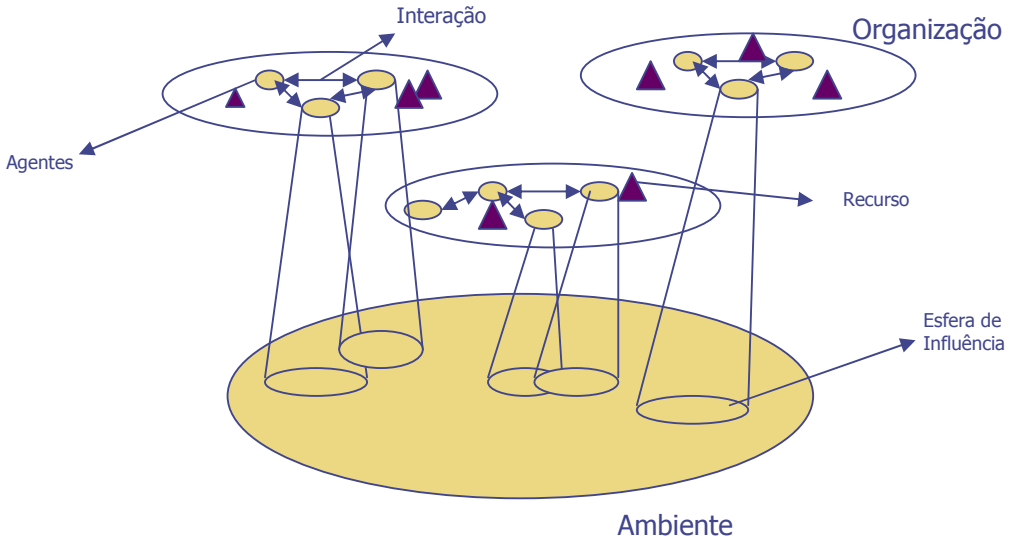

Figura 3-1: Abstração Organizacional de um SMA, adaptado de [Wooldridge 2001]

Comunicação

Em seu aspecto fundamental é a habilidade do agente se comunicar, ou seja, trocar informação com outros agentes. A comunicação de divide em duas partes, percepção (capacidade de receber mensagens) e ação (envio de mensagens).

Através da comunicação é possível a coordenação de ações e comportamentos para melhor atingir os objetivos dos agentes, bem como da sociedade na qual eles existem [Huhns e Stephens 1999].

[9] Dimensões representam os diversos pontos de vista de um SMA: agente individual, comportamento coletivo ou interações com meio-ambiente.

Tipicamente a infra-estrutura de um SMA deve fornecer protocolos para os agentes trocar e compreender mensagens e capacitem os agentes a manterem conversações, que são trocas estruturadas de mensagens.

Protocolos de comunicação podem ser binários, que envolvem um único remetente e um único destinatário, ou n-ários, que envolvem um único remetente e vários destinatários (por exemplo, *multicast* e *broadcast*). Maneiras como o *blackboard*, onde mensagens são registradas em repositórios compartilhados, para leitura posterior, também são possíveis.

Em relação aos participantes em uma comunicação, estes podem ser ativo, passivo ou ambos, à medida que eles enviam solicitações, apenas recebem e executam solicitações ou assumem ambos os papéis de iniciante (quem envia a mensagem) ou respondente (quem executa a solicitação) no diálogo.

Negociação

Negociação pode ser entendida como o esforço para solução de conflitos e cooperação, ou mais especificamente é o processo, através do qual dois ou mais agentes que tenham interesses próprios conseguem chegar a uma decisão conjunta.

O processo de negociação necessita dos seguintes elementos: uma ***Linguagem*** comum, entendida pelos agentes, um ***Protocolo*** que reja a negociação e um ***Processo de Decisão***, que determina posições, concessões, e critérios utilizados para os acordos[Huhns e Stephens 1999].

Em uma negociação podemos ter basicamente três situações: ou há um **conflito** e a negociação é nula, ou há um **compromisso** no qual os agentes preferem estar isolados, mas já que não estão, concordam em negociar e a **cooperação**, onde a negociação é sempre preferida pelos agentes participantes [Weiss 1999].

Entre as principais formas de negociação, existem as abordagens baseadas em Mercado [Ferber 1999]:

- Um vendedor e um comprador negociam diretamente (negociação bilateral);
- Muitos compradores e um vendedor participam de um leilão clássico (negociação multilateral 1-N);
- Muitos vendedores e um comprador participam de um leilão invertido (negociação multilateral N-1);
- Muitos compradores e muitos vendedores formam um mercado (negociação multilateral M-N).

Coordenação

Sempre que agentes possuam ações interdependentes, faz-se necessário a existência de mecanismos que gerenciem essas ações. A Coordenação é o conjunto de tarefas suplementares que devem ser executadas em um SMA, as quais não seriam necessárias caso existisse apenas um único agente [Ferber 1999].

Coordenação é o fator fundamental que torna possível qualquer trabalho em conjunto. Por exemplo, quando duas pessoas tentam sair pela mesma porta ao mesmo tempo. Em relação a SMA, esse aspecto é tido como pressuposto nos casos que nenhum dos agentes individuais possa resolver o problema sozinho ou informações e resultados estejam disponíveis apenas através de outros agentes, recursos sejam limitados (e.g. tempo, espaço, dinheiro), seja necessário otimizar custos (e.g. eliminação de tarefas redundantes) ou agentes com objetivos inter-dependentes tenham que cooperar.

O processo de coordenação visa garantir que todas as partes necessárias existam na sociedade e haja interação suficiente e harmoniosa que possibilite a execução das atividades. Também procura garantir que todos os agentes atuem consistentemente e que tudo seja feito com os recursos disponíveis e de maneira otimizada e respeitando as restrições globais do SMA [Ferber 1999].

Ações, quando executadas simultaneamente, precisam ser coordenadas já que o relacionamento entre elas pode ser positivo por uma se beneficiar da outra (melhor desempenho que se executadas sozinhas) ou negativo quando impossibilitam a execução de ações (conflito por limitação de recursos ou incompatibilidade de objetivos).

Existem algumas formas de coordenação adotadas em SMA, dentre as quais podemos destacar: Coordenação via Sincronização – a mais simples e limitada, descreve precisamente a seqüência das ações concorrentes. Coordenação via Planejamento – possui fases de elaboração de planos, seleção de planos e execução. É limitada em relação a situações não previstas ou complexas. Coordenação Reativa – não há planejamento. Usa apenas o ciclo percepção-ação do agente e é mais adequada a situações difíceis de prever.

Como exemplo de protocolo de coordenação bastante estudado, temos as Redes de Contrato (Contract-Nets), inspiradas nos processos de contratação existentes em organizações humanas. Nesse processo, agentes coordenam suas ações através de contratos para cumprir seus objetivos específicos, onde existe um agente que atua como gerente, decompondo seus contratos em subcontratos a serem realizados por outros potenciais agentes empreiteiros. É composto por quatro fases: anuncio da tarefa, encaminhamento de proposta, análise das propostas e emissão do "contrato".

3.1.2 Metodologias para Desenvolvimento de SMA

Na seção anterior, comentamos sobre a necessidade da existência de um esforço na área de Engenharia de Software e da IA para que a adoção do paradigma orientado a agentes pudesse ganhar maior aceitação. Nesta seção, iremos descrever alguns resultados dos esforços das pesquisas existentes, refletidas nas metodologias de desenvolvimento.

Quando se fala em SMA, muitos são os desafios para projetos de software, que advêm da sua natureza distribuída e autônoma dos seus componentes e da necessidade de adaptabilidade e robustez, que não estão presentes na maioria dos sistemas atuais tratados pela Engenharia de Software tradicional. Essa mudança de objetivos e a investigação de maneiras de estender os modelos tradicionais são as maiores preocupações das metodologias de desenvolvimento de SMA atuais. O critério maturidade, por outro lado, ainda é o quesito mais questionado, mesmo nas principais metodologias formuladas.

As metodologias de desenvolvimento baseadas em SMA aplicam a decomposição do problema em termos de agentes autônomos, suas percepções, ações, objetivos e ambiente. Isso oferece uma abstração que auxilia na correlação entre as noções de subsistemas e as organizações de agentes.

De acordo com Zambonelli e colegas [Zambonelli et al. 2003], não existe uma metodologia "melhor" do que outra, mas simplesmente metodologias mais adequadas para cada caso. Podemos citar como exemplo, a existência de metodologias derivadas da extensão de diagramas de UML [Bauer et al. 2001] [Odell et al. 2004] que podem ser adequadas ao desenvolvimento de sistemas em equipes que tenham experiência no desenvolvimento com uso de UML [Booch et al. 1999], o que pode facilitar o processo de transição.

Existem metodologias, como a Gaia [Zambonelli et al. 2003], que se baseiam na visão dos agentes como uma comunidade e enfatizam a comunicação entre os agentes. Outras atuam de maneira a resolver os problemas com foco nos agentes individuais, sendo o comportamento do grupo uma conseqüência do comportamento dos agentes individuais. Um exemplo dessas é a metodologia Agent-Oriented-Relationship (AOR) [Wagner 2003].

Dentre as metodologias consideradas mais utilizadas e, portanto com maior grau de maturidade, podemos destacar as seguintes: Gaia [Zambonelli et al. 2003], Tropos [Castro et al. 2002] [Bresciani et al. 2004], AOR [Wagner 2003], Mase [DeLoach et al. 2001] e Prometheus [Padgham e Winikoff 2002]. Algumas dessas metodologias são abordadas em maiores detalhes, a seguir.

A Metodologia Gaia

Gaia [Zambonelli et al. 2003] é uma metodologia que tenta definir um método especificamente moldado para a análise e o projeto de SMA. Ela dá relevância tanto para a estrutura do agente individual como para a sociedade de agentes. De acordo com essa metodologia, SMA são vistos como um número de agentes autônomos interativos que vivem em uma sociedade organizada onde cada agente representa um ou mais papéis específicos, sendo a estrutura do SMA definida em termos de um *Modelo de Papéis*. Esse modelo identifica os diferentes papéis que os agentes devem representar dentro do SMA e os protocolos de interação entre eles.

A Metodologia Gaia tem seu processo dividido em três fases distintas: *Análise*, *Projeto de Arquitetura* e *Projeto Detalhado*. Como fase inicial, os autores recomendam uma etapa de *Levantamento de Requisitos*, que a exemplo da fase de *Implementação* é deixada em aberto para utilização de técnicas outras.

A fase de *Análise* tem como objetivo coletar e organizar as especificações que formam a base para o projeto da organização computacional. Essa fase inclui:

- A identificação dos objetivos das organizações que constituem o sistema como um todo e seu comportamento global esperado;
- O modelo de Ambiente (uma abstração computacional que representa o ambiente onde o SMA está situado e recursos disponíveis);
- O modelo de Papéis preliminar (descrições abstratas de uma função pretendida por uma entidade);
- O Modelo de Interações preliminar (definições de protocolos para cada tipo de interação entre papéis);
- As regras que a organização deve respeitar e enfatizar no seu comportamento global.

O resultado da fase de análise consiste em um *Modelo de Ambiente*, um *Modelo de Papéis Preliminar*, um *Modelo de Interações Preliminar* e um *Conjunto de Regras Organizacionais*, que são utilizadas na fase de projeto. A fase de *Projeto de Arquitetura*, por sua vez, inclui:

- A definição da estrutura organizacional do sistema em termos de sua topologia e regime de coordenação;
- A derivação de padrões organizacionais;
- A finalização dos modelos preliminares de papéis e de interação.

Uma vez definidos esses artefatos, a fase de *Projeto Detalhado* pode ser iniciada, englobando:

- A definição do *Modelo de Agentes* (definição dos tipos de agentes que farão parte do Sistema);
- A definição do *Modelo de Serviços* (serviços atribuídos a cada agente).

A Metodologia Gaia não se compromete explicitamente com modelos de representação definidos. Há apenas sugestões de diagramas e modelos a serem usados, mas que deixam livre a escolha do formalismo de modelagem e diagramação para os desenvolvedores e analistas.

Um ponto negativo da metodologia Gaia é não cobrir as fases de Análise de requisitos, Implementação e Verificação e Testes além de ter uma abordagem estritamente seqüencial, que é mais apropriada para projetos com um conjunto de requisitos estável e dimensões limitadas.

A Metodologia Tropos

Esta metodologia [Castro et al. 2002] [Bresciani et al. 2004] tem como característica mais marcante seu foco na análise de requisitos, onde os *stakeholders* do domínio do sistema no mundo real e sua intenções são identificadas e analisadas.

A metodologia propõe cinco fases de desenvolvimento:

- ***Early Requirements*** é o entendimento do problema a partir da análise e estudo da configuração organizacional onde o problema está inserido.
- ***Late Requirements*** é onde o sistema é descrito dentro de seu ambiente operacional, a partir de requisitos funcionais relevantes e requisitos de qualidade ou não funcionais.
- **Projeto Arquitetural** é a definição global da arquitetura do software em termos de subsistemas e suas dependências.
- No **Projeto Detalhado** é especificado em minúcias o comportamento de cada componente arquitetural.

A fase de Pré-requisitos é influenciada pelo *framework* i* [Yu 1995], um ambiente de modelagem que oferece conceitos tais como:

- **Ator**: modela entidades que têm objetivos estratégicos e intenções dentro do sistema ou da configuração organizacional. Um ator representa um agente de software, agente físico ou agente social, assim como um papel ou uma posição. Um papel é uma abstração do comportamento de um ator social e uma posição é um conjunto de papéis assumidos por um agente.
- **Meta**: Representa interesses estratégicos dos atores. São classificados em *hardgoal*s (requisitos funcionais) e *softgoals (requisitos não-funcionais).*
- **Plano**: É um meio para satisfazer um *hardgoal* ou *softgoal*.
- **Dependência entre atores**: Indica que um ator depende de outro por algum motivo (satisfazer objetivos, realizar planos ou liberar recursos).

Outros conceitos do metamodelo do Tropos são:

- **Capacidade**: É a habilidade de um ator de definir, escolher e executar um plano.
- **Crença**: É o conhecimento do ator do mundo.

Na fase de Implementação, os desenvolvedores realizam o mapeamento de cada conceito definido no Projeto Detalhado para o modelo BDI (Beliefs, Desires and Intentions) [Bratman et al. 1987]. Os autores do Tropos sugerem o ambiente de desenvolvimento JACK Intelligent Agents [Busetta et al. 2004] como plataforma capaz de suportar esta fase, visto que o JACK estende classes em linguagem Java para aderência ao BDI.

O processo incremental da metodologia Tropos é bastante adequado para desenvolvimento de SMA que tratem de requisitos mais instáveis ou difíceis de serem identificados. Por outro lado não é apropriada para sistemas que exijam desenvolvimento rápido de protótipos e entrega ágil de produtos.

A metodologia Tropos será mais detalhada nos Capítulos 4 e 5.

A Metodologia AOR

A metodologia AOR – Agent-Object Relationship [Wagner 2003] é inspirada em 19 princípios ontológicos, dentre os quais a modelagem Entidade-Relacionamento (ER). É bastante diferente das outras metodologias orientadas a agentes no que diz respeito a modelos externos (modelos de análise) e modelos internos (modelos de projeto). A metodologia cobre as fases de análise, projeto, codificação e implementação em um estilo em cascata.

A abstração central da AOR é a "entidade" (e.g. agentes, eventos, ações, compromissos e objetos comuns) e relacionamentos definidos entre eles que vão além das associações tradicionais, agregação/composição e generalização/especialização dos modelos UML. Os modelos de análise buscam capturar a visão do observador externo para modelar o domínio do sistema, gerando diagramas de padrões, interações e seqüências para um ou mais agentes. Os modelos internos então, refinam cada agente em modelos de implementação para a linguagem e plataforma desejadas.

A linguagem de modelagem AORML (AOR Modeling Language) fornece os recursos necessários para as atividades de modelagem. O processo completo consiste em cinco etapas: 1) Análise do domínio do sistema (modelo externo AOR); 2) Transformação do modelo externo em modelos internos para cada agente; 3) Transformação dos modelos internos em modelos de projeto de banco de dados ou definições de estruturas de dados; 4) Refino dos modelos de projeto em modelos de implementação; 5) Geração de código na linguagem alvo.

A metodologia AOR inova ao considerar modelos internos, externos e mapear regras de reação, por outro lado não inclui o conceito de objetivo (ou meta), fundamental em outras metodologias e nem modela o comportamento pró-ativo de agentes. Versões futuras prometem adicionar metaconceitos para atividades, objetivos e engenharia de requisitos. A versão atual é mais apropriada para SMA fechados, o que em parte, torna limitada sua aplicabilidade.

Considerações

As metodologias de desenvolvimento SMA são relativamente novas e, como dito, não podemos afirmar que há uma melhor que outra. Há metodologias que contemplam mais fases, indo do levantamento de requisitos à implementação, outras que se apropriam melhor a determinadas equipes de desenvolvimento que dominem técnicas de engenharia de software semelhantes (e.g. metodologias baseadas em UML, como a AUML, usada com equipes que estejam habituadas a especificação UML em sistemas tradicionais orientados a objetos) ou ainda metodologias se adaptem melhor ao ritmo empregado (e.g. desenvolvimento rápido, cascata).

Para o trabalho apresentado nesta dissertação foram realizados dois pequenos protótipos, um em Gaia outro em Tropos para seleção de uma metodologia a ser adotada. Procurou-se analisar cada fase componente das duas metodologias, os artefatos gerados em cada fase e a adequação a desenvolvimento de sistemas com perfil semelhante ao DBSitter-AS.

A metodologia Gaia apresentou aspectos positivos como ênfase na Organização e no Papel que ajuda bastante para compreensão de uma sociedade de agentes, mas a ausência de uma fase explícita de Análise de Requisitos e da adoçãode formalismos de representação (e.g. diagramas e modelos) pesou contra a sua escolha. Por se tratar de um sistema com foco mais computacional que informacional, o DBSitter-AS necessita de uma análise mais consistente de requisitos não-funcionais, tais como (e.g. segurança, confiabilidade, facilidade de manutenção) e do impacto desses requisitos no comportamento global do sistema. Foi aproveitado do estudo Gaia, a análise do negócio baseada em Organização e Suborganizações e o registro documental acerca de papéis e comunicações entre papéis.

A metodologia Tropos foi selecionada como ferramenta de análise para desenvolvimento do DBSitter-AS principalmente por enfatizar a etapa de Levantamento de Requisitos, cobrir todas as etapas de desenvolvimento de software e ser mais madura em relação à adoção de modelos e artefatos de documentação. No Capítulo 4 apresentaremos os trabalhos gerados através da utilização da metodologia Tropos para modelagem formal do *Framework*.

3.2 Abordagens Arquiteturais para Integração de SGBD com Agentes

Direcionando nossa análise sobre Sistemas Multiagentes para a realidade computacional presente na monitoração e gerenciamento de SGBD, podemos identificar várias características que nos levam a considerar a construção de um SMA como uma alternativa viável de ferramenta atuante nas funções de prevenção e correção de falhas. Agentes de software podem atuar em uma comunidade cujo objetivo seria a prevenção e a resolução dos problemas ocorridos em um ou mais SGBD ou no ambiente operacional onde estejam inseridos. Tarefas de monitoria, gerenciamento de recursos, atuação preventiva para correção de parâmetros de desempenho ou correção de falhas podem ser configuradas para serem desempenhadas por uma comunidade de agentes inteligentes.

Para se construir um SMA é necessário uma arquitetura que estabeleça como um problema deve ser decomposto em subproblemas, ou seja, como o SMA pode ser visto como um conjunto de módulos e como esses módulos devem interagir. Lifschitz [Lifschitz e Macêdo 2004] propõe três arquiteturas para integração entre SGBD e agentes: Em Camada (*Layered),* Integrada (Integrated) e Embutida (Built-in), mostradas na Figura 3-2.

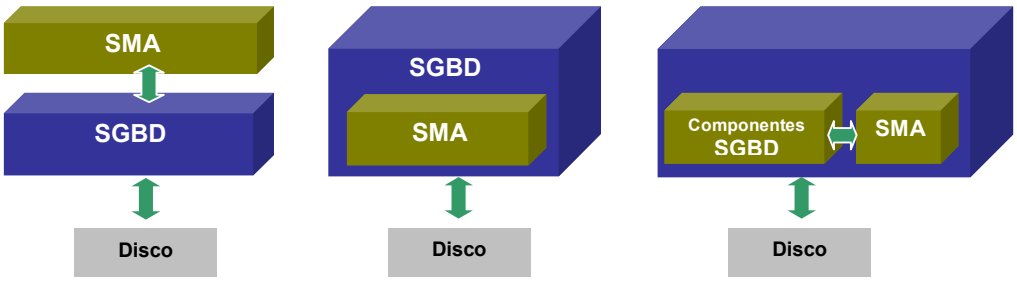

Figura 3-2: Arquiteturas para Integração de Sistemas de Agentes e SGBD, adaptada de [Lifschitz e Macêdo 2004]

A arquitetura *Em Camada* (a) prega a implementação da camada de agentes de software no topo, externa ao SGBD, sem que haja necessidade de alterações neste. Essa camada atua como mediadora recebendo, manipulando e interpretando requisições ao SGBD. Já a arquitetura Integrada (b) é o oposto da implementação *Em Camada* e prega a substituição de componentes originais do SGBD por subsistemas de agentes. Na arquitetura Embutida (c) os agentes são integrados ao SGBD utilizando-se de serviços já providos pelos componentes do SGBD, enquanto

incorporam novas funcionalidades. Os agentes podem interferir no comportamento dos componentes do SGBD, alterando os comportamentos deles ou coletando informações sobre suas operações e processos.

O Quadro 3-1, apresenta um estudo comparativo entre as principais características [Lifschitz e Macêdo 2004] de cada abordagem apresentada.

Quadro 3-1: Características das arquiteturas para SGBD baseadas em Agentes

Característica	Em Camada	Integrada	Embutida
Alteração no código do SGBD	*não há*	*muita*	*pouca*
Permite novas funcionalidades	*sim*	*sim*	*sim*
Complexidade de implementação	*média*	*muito grande*	*grande*
Acoplamento ao SGBD	*pouco*	*muito grande*	*grande*
Aplicada a múltiplos SGBD	*sim*	*não*	*não*

Neste contexto, podemos classificar as implementações de SGBDs autônomos discutidas na Seção 2.4.2 da seguinte maneira: a arquitetura Integrada normalmente é usada apenas pelos fabricantes de SGBD nos seus produtos, visto que necessitam de solução especializada e com grande acoplamento aos seus produtos; as arquiteturas Em Camada têm sido usadas em pesquisas acadêmicas de sistemas que atuam em SGBD comerciais [Carneiro et al. 2004] [Ramanujam e Capretz 2005] por não necessitarem de intervenção no código-fonte do SGBD monitorado; a arquitetura Embutida tem tido boa aceitação em pesquisas de SMA que interagem com SGBD de código aberto [Milanés e Lifschitz 2005] [Ogeer 2004].

Para o desenvolvimento do DBSitter-AS optou-se por manter a implementação na arquitetura Em Camada, já adotada no projeto do DBSitter original, e portanto ter um SMA externo ao SGBD que se beneficie principalmente da possibilidade da aplicação a múltiplos SGBD.

3.3 Discussão

Aplicações baseadas em SMA, cada vez mais, têm sido usadas nas áreas de Gerenciamento de *Workflow* e Processos de Negócio, Comércio Eletrônico, Monitoração de Ambientes (e.g. tráfego de veículos, estradas, imagens de satélite, cálculo de trajetórias), Recuperação e Gerenciamento de Informação (e.g. Robôs de busca na Internet, distribuição automática de notícias), Sistemas de Auxílio Especializado em Recomendação (e.g. Assistentes Eletrônicos ou Elaboradores de Diagnóstico), Sistemas Baseados em Ambientes Virtuais (e.g. Simuladores de Vôo) e

Sistemas de Gerenciamento Autônomo de Ativos (e.g. Gerenciamento de recursos de redes de computadores).

De acordo com o apresentado neste capítulo, podemos afirmar que agentes inteligentes e SMA podem ter um papel importante no desenvolvimento futuro dos sistemas computacionais. A combinação de técnicas de Engenharia de Software e Inteligência Artificial pode expandir a área de alcance dos SMA, o que os potencializa a lidarem com ambientes de natureza complexa, distribuída e concorrente.

A administração automatizada de SGBD é uma área que se encaixa nos requisitos para ser implementado em formato de SMA. Topologias de arquiteturas já foram propostas [Lifschitz e Macêdo 2004] e evoluções vêm sendo experimentadas tanto no âmbito acadêmico como no comercial. A evolução desses conceitos e técnicas são passos que podem levar à construção de SMA que possibilitem que SGBD tradicionais tornem-se SAGBD.

No próximo capítulo, iremos apresentar o *Framework* DBSitter-AS: um projeto arquitetural baseado em SMA, estruturado em topologia Em Camada, que propõe uma forma de construção de componentes autônomos para fornecimento prevenção e resolução de problemas relacionados ao gerenciamento e monitoração de SGBD.

4 O *Framework DBSitter-AS*

Neste capítulo apresentamos o Framework[10] DBSitter-AS. O Framework proposto define uma especificação arquitetural para construção de componentes que forneçam autonomia de administração para SGBD. Organizada como um SMA, esta especificação arquitetural é compatível com os princípios da Computação Autônoma apresentados no Capítulo 2 e provê uma forma para desenvolvimento de características flexíveis de prevenção e resolução de falhas em SGBD tradicionais.

Os componentes desenvolvidos seguindo a especificação DBSitter-AS estarão localizados externamente (integração em camada) ao SGBD. Com o auxílio desses componentes, o ABD poderá ser poupado de tarefas rotineiras (deixando de apenas reagir aos problemas) e terá tempo para se dedicar ao planejamento estratégico da gestão dos dados. O objetivo principal da arquitetura proposta é manter um SGBD em funcionamento normal de maneira o mais autônoma possível, realizando prevenção, resolução de falhas e emissão de alertas. Além disso, a arquitetura propõe permitir que os componentes autônomos possam aprender com as ações realizadas mediante análise do *feedback* recebido do usuário.

A principal motivação deste trabalho foi elaborar uma especificação que estendesse os estudos realizados para o desenvolvimento do DBSitter [Carneiro et al. 2004] e criasse um modelo capaz de modularizar o desenvolvimento de componentes de autonomia para SGBD. Um benefício direto de se ter uma especificação arquitetural é a possibilidade se desenvolver os diferentes componentes da arquitetura em paralelo, cada um de acordo com uma linha de pesquisa diferente, mas que possam ser integrados em um produto final coerente, de forma padronizada.

Na Seção 4.1 apresentaremos a estrutura organizacional do *framework*. Na Seção 4.2 são apresentados requisitos e a metodologia utilizada na concepção conceitual do *framework* e principais artefatos da sua especificação formal.

4.1 A Estrutura Organizacional

Algumas metodologias para modelagem SMA (e.g. Tropos [Castro et al. 2002] e GAIA [Zambonelli et al. 2003]) utilizam-se da metáfora organizacional para análise e raciocínio do problema de modelagem. Esta abordagem tem como intuito facilitar a compreensão do software a ser desenvolvido, tratando-o como uma instanciação computacional de uma organização do mundo real (e.g. uma empresa, um sistema, um grupo de indivíduos autônomos e interativos). À medida que a complexidade de

[10] Definimos Framework como um conjunto de estruturas e especificações arquiteturais.

uma organização aumenta, sugere-se, baseado nos princípios de modularidade e encapsulamento, dividir a organização em suborganizações com subconjuntos de agentes que desempenham papéis funcionalmente correlatos, mas que possivelmente possam ser envolvidos além da fronteiras da sua suborganização [Zambonelli et al. 2003].

Tendo em vista a metáfora organizacional, o DBSitter-AS foi concebido como uma abstração de uma organização administradora de SGBD, formada por agentes e composta por três suborganizações: "Detecção, prevenção e correção de falhas", "Notificações e registros" e "Aprendizagem e sugestão", conforme mostrado na Figura 4-1. O ambiente onde o SMA de arquitetura DBSitter-AS atuará, é composto pelo SGBD alvo de gerenciamento (e.g. Oracle, PostgreSQL, SQL Server), o sistema operacional (SO) do servidor onde o SGBD é executado e uma Base de Conhecimento e Registro. Essa base auxiliar pode instanciada em um SGBD gratuito, como o Firebird[11] ou MySql[12], por exemplo.

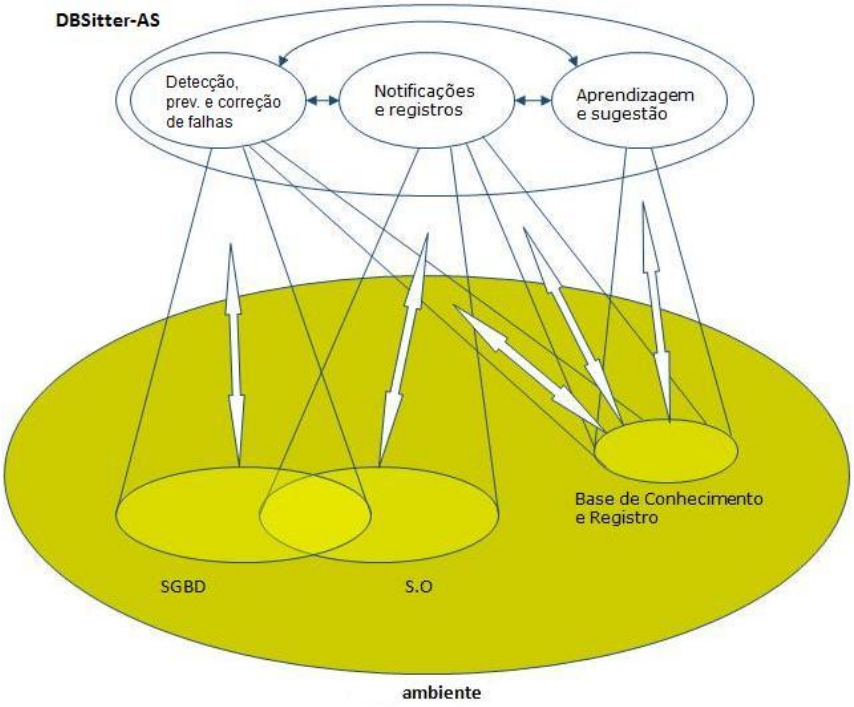

Figura 4-1: Estrutura Organizacional do SMA DBSitter-AS

A suborganização de "Detecção, prevenção e correção de falhas" atua no SGBD e no SO, assim como a suborganização de "Notificação e registros". As três suborganizações utilizam-se da Base de Conhecimento e Registro. Desta forma há

[11] www.firebirdsql.org
[12] www.mysql.com

uma separação funcional entre os agentes que farão parte das suborganizações, o que refletirá nas suas capacidades e papéis desempenhados.

As capacidades dos agentes de suborganizações diferentes são complementares e estes colaboram para a realização de seus objetivos. Por exemplo, na suborganização de "Detecção, prevenção e correção de falhas" haverá agentes que perceberão falhas no SGBD e necessitam que um alerta seja dado ao usuário, então um agente especializado em enviar comunicação (e.g. e-mail, SMS, alertas visuais), que faz parte da suborganização "Notificações e Registros", tem seus serviços solicitados.

Em particular, a suborganização "Detecção, Prevenção e Correção de Problemas" por sua complexidade e variedade de problemas existentes em bancos de dados, foi subdividida em mais duas suborganizações, a saber:

- Detecção e correção de Problemas de SGBD
  - Gerenciamento de Estruturas Controle
  - Gerenciamento de Estruturas Físicas
  - Gerenciamento de Estruturas Lógicas
- Detecção e correção de Problemas de Infra-estrutura
  - Gerenciamento de Sistema Operacional
  - Gerenciamento de Rede

A classificação feita busca caracterizar as diversas falhas que podem acontecer em um SGBD. A suborganização "Detecção e correção de Problemas de SGBD" foi dividida em três novas suborganizações de gerenciamento: "Gerenciamento de Estruturas de Controle", "Gerenciamento de Estruturas Físicas" e "Gerenciamento de Estruturas Lógicas". Estruturas de controle são usadas comumente pelos SGBD para definir seus comportamentos, como arquivos de inicialização, parâmetros para definição de linguagem ou nível de acesso dos usuários. Estruturas físicas dizem respeito aos arquivos físicos utilizados pelos SGBD para armazenar os dados. Estruturas lógicas são os objetos manipuláveis para tratamento dos dados, como tabelas, índices, catálogos, *stored procedures*.

A suborganização "Detecção e correção de Problemas de Infra-estrutura" foi dividida em "Gerenciamento de Sistema Operacional" (SO) e "Gerenciamento de Rede", referentes a ambientes onde também podem ocorrer erros que afetem o desempenho do SGBD. Em relação ao SO podem ocorrer picos de processamento que identificados ajudem no diagnóstico de perdas de desempenho ou mesmo a necessidade de interrupção de processos. Em relação à rede, há a necessidade do controle de estruturas responsáveis pela comunicação do SGBD com o mundo exterior, como portas de comunicação ou processos "*Listeners*" que precisem ser iniciados ou cancelados.

Para cada uma dessas suborganizações identificadas há grupos de agentes com perfis, habilidades e responsabilidades semelhantes, que interagem com os das

outras suborganizações através de solicitações de serviços e informações para atingir de forma harmônica, os objetivos globais da organização DBSitter-AS.

Na próxima seção detalharemos os passos seguidos na metodologia usada para modelagem do DBSitter-AS e apresentaremos os artefatos gerados.

4.2 Metodologia e Modelagem

A construção de um produto de software envolve uma série de atividades, muitas vezes com um alto grau de complexidade, que são desenvolvidas a partir da visão de várias pessoas que detêm o conhecimento sobre alguma parte do sistema. Estas pessoas podem ou não conhecer bem o produto que vai ser gerado, mas contribuem com informações para a obtenção do resultado mais adequado possível.

Para que todas as informações coletadas durante o processo de desenvolvimento do software possam ser identificadas, analisadas corretamente e justificadas perante seus contribuintes, é necessário o uso de métodos e padrões que permitam a correta documentação das informações em todas as fases do ciclo de vida do produto.

Para sistemas complexos, em especial SMA, existe a necessidade de técnicas de modelagem de sistemas específicas, que permitam que sua complexidade seja efetivamente gerenciada e metodologias para guiar a análise e projeto do sistema. Métodos tradicionais de desenvolvimento de software não possuem flexibilidade suficiente para lidar com conceitos de alto nível tais como o controle dinâmico de um agente sobre seu próprio comportamento, a habilidade de representar interações cooperativas, e mecanismos de representação de mudanças internas, crenças, objetivos e incerteza inerentes às interações no mundo real [Fisher et al. 1997].

Na especificação do DBSitter-AS, foi utilizada a metodologia Tropos [Bresciani et al. 2004] [Castro et al. 2002]. Em virtude da metodologia Tropos ter, correntemente, duas abordagens, a da Itália [Bresciani et al. 2004] e a versão Brasil/Canadá [Castro et al. 2002], procedimentos das duas versões foram utilizados; a cada etapa do projeto, foram escolhidos os que mais se ajustassem ao desenvolvimento de um SMA com as características do DBSitter-AS. O trabalho de definição de um guia de etapas unificado para o Tropos [Silva et al. 2007] fez parte de um trabalho correlato ao DBSitter-AS, focado principalmente nas etapas de análise de requisitos e serve de referência para a seqüência de atividades realizadas durante o nosso processo da modelagem. Esta publicação refere-se à arquitetura DBSitter-AS por XAADB (eXternal Architechture for Autonomous DataBase administration), denominação dada nas primeiras versões do trabalho.

Nas seções a seguir são apresentados os artefatos mais importantes para compreensão global das atividades de modelagem realizadas. Lembramos que na Seção 3.1.2 foram apresentados os conceitos básicos da metodologia Tropos.

4.2.1 Modelagem de Pré-requisitos *(Early Requirements)*

A metodologia Tropos é fortemente orientada a requisitos e para tanto tem duas fases específicas (*Early Requirements* e *Late Requirements*) direcionadas para levantamento de pré-requisitos e requisitos de sistema e do raciocínio por trás das entidades interessadas (atores e agentes).

Para modelar todos os requisitos de um sistema eminentemente computacional (mais que informacional) e sujeito a grande quantidade de requisitos não-funcionais [Cysneiros 2001] foi preciso entender as regras de negócio que influenciam a atividade de Administração de Banco de Dados, os atores envolvidos e interesses implícitos na automatização dessas tarefas. O raciocínio dessa etapa da modelagem encontra-se representado em artefatos que demonstram as dependências estratégicas entre atores.

A fase de "*Early Requirements*" destina-se a capturar o contexto em que o sistema que será desenvolvido está inserido. Nesta fase identificamos as motivações para o desenvolvimento do sistema.

Atividades da Fase

A primeira atividade desta fase consiste em identificar os *stakeholders* (atores) do sistema e suas necessidades. Em seguida, as dependências entre os atores são explicitadas. Nas atividades restantes o raciocínio interno de cada ator é refinado. A fase *Early Requirements* é dividida em quatro atividades identificadas com prefixo **ER**, seguido dos seus números seqüenciais.

ER.01 - Identificar os *stakeholders* do domínio da aplicação e as suas intenções como atores sociais que querem obter metas

Artefato de entrada: *Registros de reuniões, documentos organizacionais*

Contou-se com informação de três especialistas na área de Administração de Bancos de Dados e experiência nos SGBD Oracle, SQL Server e PostgreSql como fonte principal de informações sobre o problema. Foram realizadas quatro reuniões para esclarecimento da configuração organizacional onde o problema está inserido e as necessidades de seus *stakeholders*. Nessas reuniões procurou-se caracterizar as necessidades comuns dos ABD e as expectativas das empresas nas quais trabalham.

Um ABD experiente também atuou permanentemente como consultor para a modelagem do *framework* DBSitter-AS. O Quadro 4-1 enumera os *stakeholders* levantados, suas metas e necessidades.

Artefato de saída: *A1 - Lista de stakeholders*

Quadro 4-1: Lista de stakeholders e suas intenções

Stakeholder	***Metas / Necessidades***
ABD	Condições de serviço melhoradas; Tempo poupado para tarefas estratégicas; Ferramenta com: desempenho aperfeiçoado, usabilidade fornecida, flexibilidade fornecida, manutenção facilitada, multiplataforma, confiável e segura;
Cliente / Empresa	Gerenciamento de dados executado; Gerenciamento de dados eficiente; Eficiência em gerenciamento de dados provida; Continuidade de serviços garantida; ROI antecipado; Custo reduzido;

ER.02 - Decompor metas a partir de uma análise orientada a metas

Artefato de entrada: *A1 - Lista de stakeholders*
Artefato de saída: *A2 - Diagrama de Atores*

Definimos *stakeholders* nesta dissertação como sendo atores sociais externos ao sistema a ser modelado, mas a partir desta atividade usaremos indistintamente o termo ator para atores externos ou internos ao sistema.

A Figura 4-2 mostra dois atores, cliente e ABD, suas interdependências e suas metas pessoais. Lembramos que as metas dividem-se em *hardgoal*s (requisitos funcionais) e *softgoals (*metas que não podem ser mensuradas e normalmente representam requisitos não-funcionais*).* O cliente tem o *softgoal* de ter o "Gerenciamento de dados eficiente" e "Custo reduzido".

O cliente depende do ABD para alcançar o *hardgoal* de ter o "Gerenciamento de dados executado" e os *softgoals* de ter "Eficiência no gerenciamento de dados provida" e "Continuidade de serviços garantida". O ABD depende do cliente para ter "Condições de serviço melhoradas".

A legenda apresentada junto à Figura 4-2 servirá também como auxílio para os outros diagramas que aparecerão na seqüência do capítulo.

Figura 4-2: Diagrama de atores

ER.03 – Analisar o raciocínio para cada meta dos *atores* através de análise *means-ends*, decomposição de metas e contribuições.

Artefato de entrada: *A2 - Diagrama de atores*
Artefato de saída: *A3 - Diagrama de metas*

A Figura 4-3 mostra a análise do raciocínio do ator Cliente (representada pela expansão do ator). Nesta etapa ER.03 concluiu-se que a meta ter "Gerenciamento de dados eficiente" pode ser alcançada através dos planos "Adquirir novas ferramentas", "Contratar profissionais qualificados" ou "Treinar ABD" e que o primeiro contribui positivamente para ter o "Lucro aumentado". As contribuições são representadas nos diagramas por setas marcadas com sinais, que indicam uma maior ou menor contribuição positiva ou negativa.

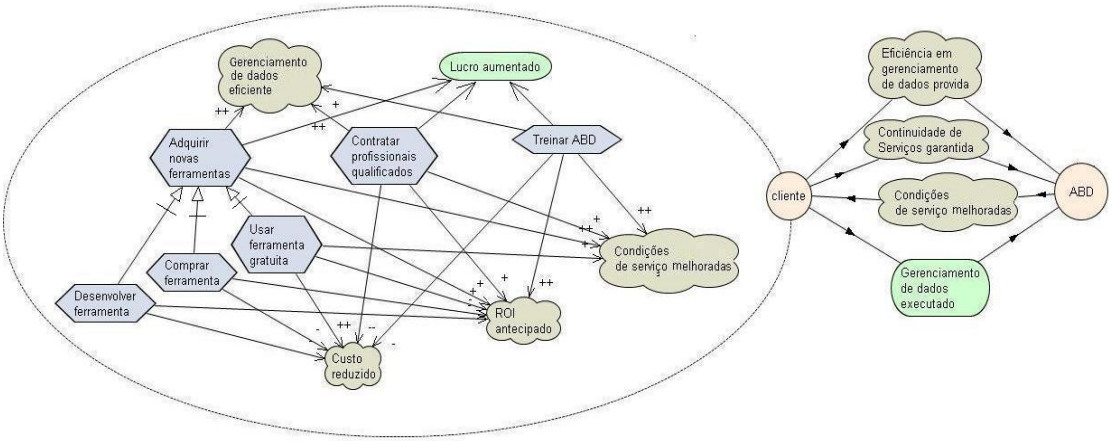

Figura 4-3: Diagrama de metas do Ator Cliente

A Figura 4-4 mostra a análise do raciocínio do ator ABD. Concluiu-se que a meta de ter "Gerenciamento de dados executado" pode ser alcançada tanto com o

plano "Automatizar tarefas" quanto com o plano "Usar administração de dados tradicional". Os meios para operacionalizar o plano "Automatizar tarefas" podem ser "Usar ferramenta gratuita" ou "Usar ferramenta paga". "Automatizar tarefas" contribui mais positivamente na "Eficiência em gerenciamento de dados provida" e para "Continuidade de serviços garantida" que a "Administração de dados tradicional".

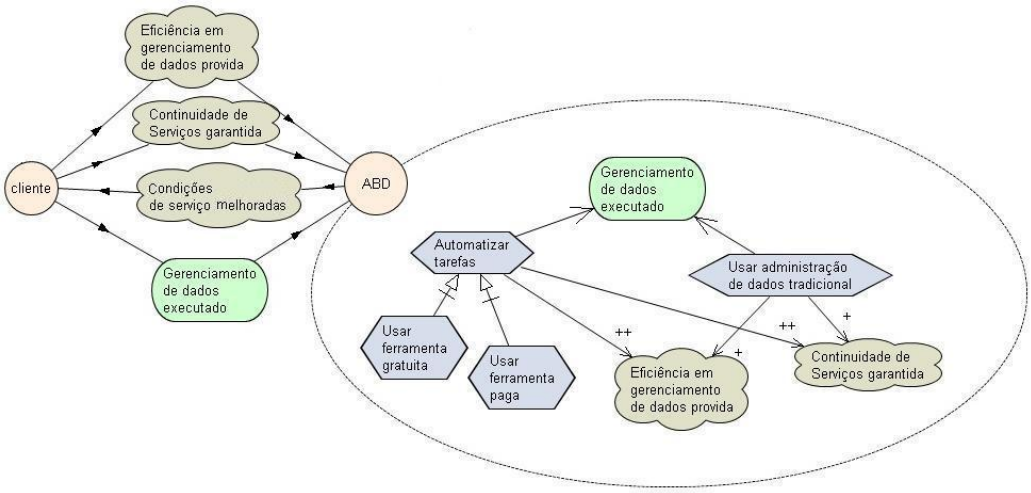

Figura 4-4: Diagrama de metas do Ator ABD

ER.04 - Identificar planos que contribuem positivamente ou negativamente para os softgoals.

Artefato de entrada: *A3 - Diagrama de metas*

Artefato de saída: *A3 - Diagrama de metas*

O resultado desta etapa pode ser visto ainda na Figura 4-3. O plano "Adquirir novas ferramentas" pode ser alcançado pelo plano "Desenvolver ferramenta" ou pelo plano "Comprar ferramenta" ou por "Usar ferramenta gratuita". Sendo que os planos "Comprar Ferramenta" e "Desenvolver Ferramenta" contribuem negativamente com o *softgoal* de "Custo reduzido" e o plano de Ferramenta gratuita contribui positivamente para este mesmo *softgoal*. Da mesma forma cada um dos planos apresentam contribuições positivas ou negativas para os *softgoals* de "ROI (*Return Of Investiment*) antecipado" e "Condições de serviço melhoradas".

4.2.2 Modelagem de Requisitos *(Late Requirements)*

A fase de "*Late Requirements*" destina-se a descrever o sistema dentro de seu ambiente operacional, a partir de requisitos funcionais relevantes e requisitos de qualidade ou não-funcionais.

Atividades da Fase

A primeira atividade desta fase consiste em identificar as dependências entre os outros atores e o ator sistema (DBSitter-AS). A segunda atividade consiste em analisar e refinar as metas do ponto de vista do ator sistema. A terceira atividade identifica as contribuições para os *softgoals*. A quarta atividade completa o comportamento interno do sistema com a análise de seus planos e por último as dependências entre os atores são revisadas.

A fase de "*Late Requirements*" se divide em cinco atividades identificadas com prefixo **LR**, seguido dos seus números seqüenciais e geram dois artefatos de saída que são extensões aos artefatos gerados durante a fase *early requirements*.

LR.01 - Identificar o ator sistema e a dependência dos outros atores da organização em relação a ele

Artefato de entrada: *A2 - Diagrama de atores*
Artefato de saída: *A2 - Diagrama de atores (estendido com o ator sistema)*

Esta atividade identifica as dependências dos outros atores da organização em relação ao sistema. O ator ABD depende do ator sistema (DBSitter-AS) para alcançar a meta de "Prevenção e resolução autônoma de falhas providas" e que este atende aos *softgoals* de "Desempenho aperfeiçoado", "Usabilidade fornecida", "Flexibilidade fornecida", "Manutenção facilitada", "Multiplataforma atendida", "Confiabilidade garantida" e "Segurança garantida". Além disso, o ator DBSitter-AS fornece ao ator ABD "Tempo poupado para tarefas estratégicas".

Na Figura 4-5 podemos ter uma visão parcial do diagrama de atores, focando a parte estendida do ator DBSitter-AS, que representa o sistema que será modelado. Nela podemos ver que foram acrescentados os atores SO e SGBD na organização, que complementam dependências identificadas para o ator DBSitter-AS. Uma vez que o raciocínio do ator DBSitter-AS foi analisado, foi identificado que depende do ator SO para alcançar a meta de "Recursos e serviços de SO fornecidos" e que ele depende do ator SGBD para alcançar a meta de "Recursos e serviços de SGBD providos".

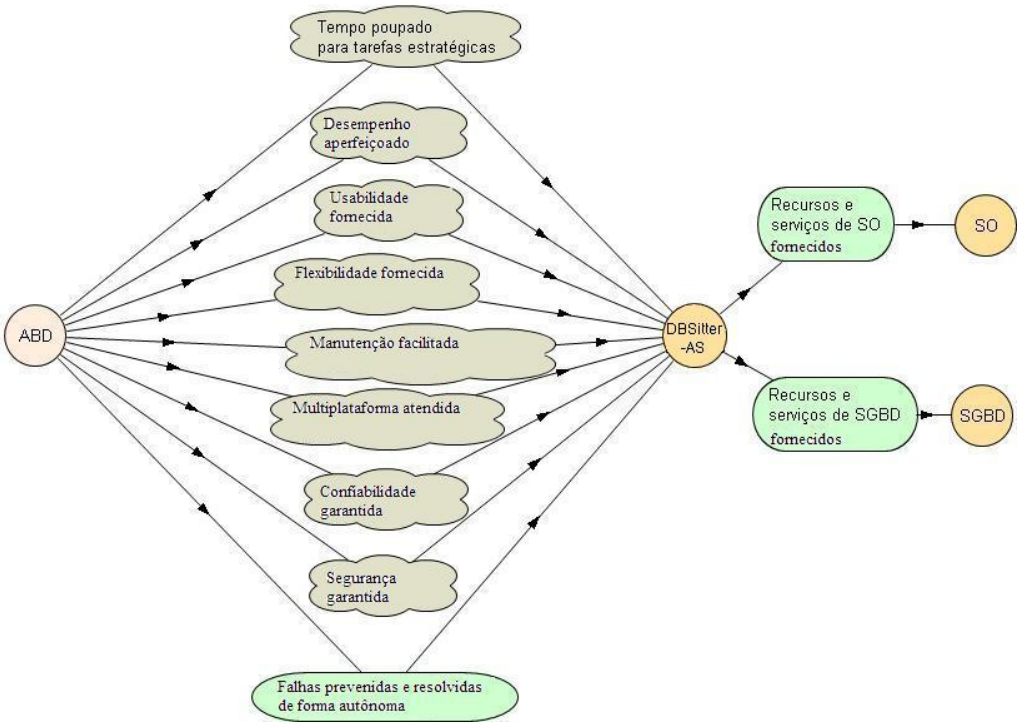

Figura 4-5: Diagrama de atores estendido

LR.02 - Analisar as metas do ponto de vista do ator sistema por decomposições e contribuições

Artefato de entrada: *A2 - Diagrama de atores*
A3 - Diagramas de metas
Artefato de saída: *A4 - Diagrama de metas estendido*

Na atividade LR.01 analisamos as dependências do ator DBSitter-AS para atingir as metas indiretas. Ou seja, metas dos quais outros atores da organização dependem do ator DBSitter-AS, para alcançar. Vamos analisar agora novas decomposições e contribuições. Um *hardgoal* pode ser decomposto em outros *hardgoals* que se configurem em submetas que facilitem seu entendimento e modelagem. Essas submetas podem ter que ser cumpridas em conjunto (decomposição "E") ou podem precisar que apenas uma seja comprida (decomposição "OR").

A Figura 4-6 mostra o Diagrama *de metas do ator sistema*. Nesta atividade foi analisado que o *hardgoal* "Falhas prevenidas e resolvidas de forma autônoma" poderia ser decomposta nos seguintes *hardgoals* (elipses em verde): "Falhas em objetos físicos de SGBD monitoradas, prevenidas e resolvidas", "Falhas em objetos lógicos de SGBD monitoradas, prevenidas e resolvidas", "Falhas de conectividade monitoradas e resolvidas", "Problemas de SO monitorados e controlados", "Eventos notificados",

"Informação e conhecimento gerenciados", "Usuários autenticados", "Falhas em estruturas de controle de SGBD monitoradas e resolvidas".
Para o cumprimento de uma meta, é necessário se especificar "planos" para que ela seja cumprida. Os planos são os meios para se atingir uma meta, que seja *softgoal* ou *hardgoal*. As decomposições de metas em planos são representadas por decomposições "means-end", conforme legenda da Figura 4-6.

Os planos (exagonos em azul) identificados devem ter analisadas suas contribuições para outros *softgoals*.Por exemplo, foi considerado que seria preciso uma decomposição do *hardgoal* "Usuários autenticados" em um plano "Autenticar Usuário" que contribui positivamente para alcançar o *softgoal* "Segurança garantida". O plano "Autenticar usuário" foi decomposto em dois subplanos: "Controlar acesso ao SGBD" e "Controlar acesso ao SO".

Também foi analisado que o *hardgoal* "Falhas em objetos físicos de SGBD monitoradas, prevenidas e resolvidas" pode ser alcançada através dos planos "Monitorar e sugerir solução para falhas em objetos físicos" e "Monitorar, prever e resolver falhas em objetos físicos".

LR.03– Identificar as contribuições dos softgoals

De forma análoga, todas as outras metas foram analisadas e foram levantados planos para atingi-las. Também foram analisadas contribuições positivas e negativas para os softgoals, conforme se pode acompanhar em detalhes na Figura 4-6.

LR.04 - Decompor os planos do ponto de vista do ator sistema

Artefato de entrada: *A3 - Diagrama de metas*
Artefato de saída: *A4 - Diagrama de metas estendido (com planos)*

Da mesma forma que as metas podem ser decompostas, podemos decompor planos em subplanos que facilitem sua compreensão. Essas decomposições também podem ser tipo "OU" ou tipo "E". Nesta atividade os planos identificados na atividade LR.02 foram decompostos e foram analisadas as suas contribuições.
O plano "Autenticar Usuário" pode ser executado tanto pelo plano "Controlar acesso ao SO" quanto pelo plano "Controlar acesso ao SGBD". O plano "Registrar configuração" é decomposto nos planos "Registrar casos de falha e soluções", "Pegar *feedback* de solução de correção", "Registrar políticas organizacionais e regras de negócio" e "Registrar configuração de usuário". As duas primeiras decomposições contribuem positivamente para o *softgoal* "Auto-aprendizagem". De forma semelhante, conforme também se pode acompanhar naFigura 4-6, foram analisados,

decompostos e foram verificadas as contribuições do restante dos planos identificados na atividade anterior.

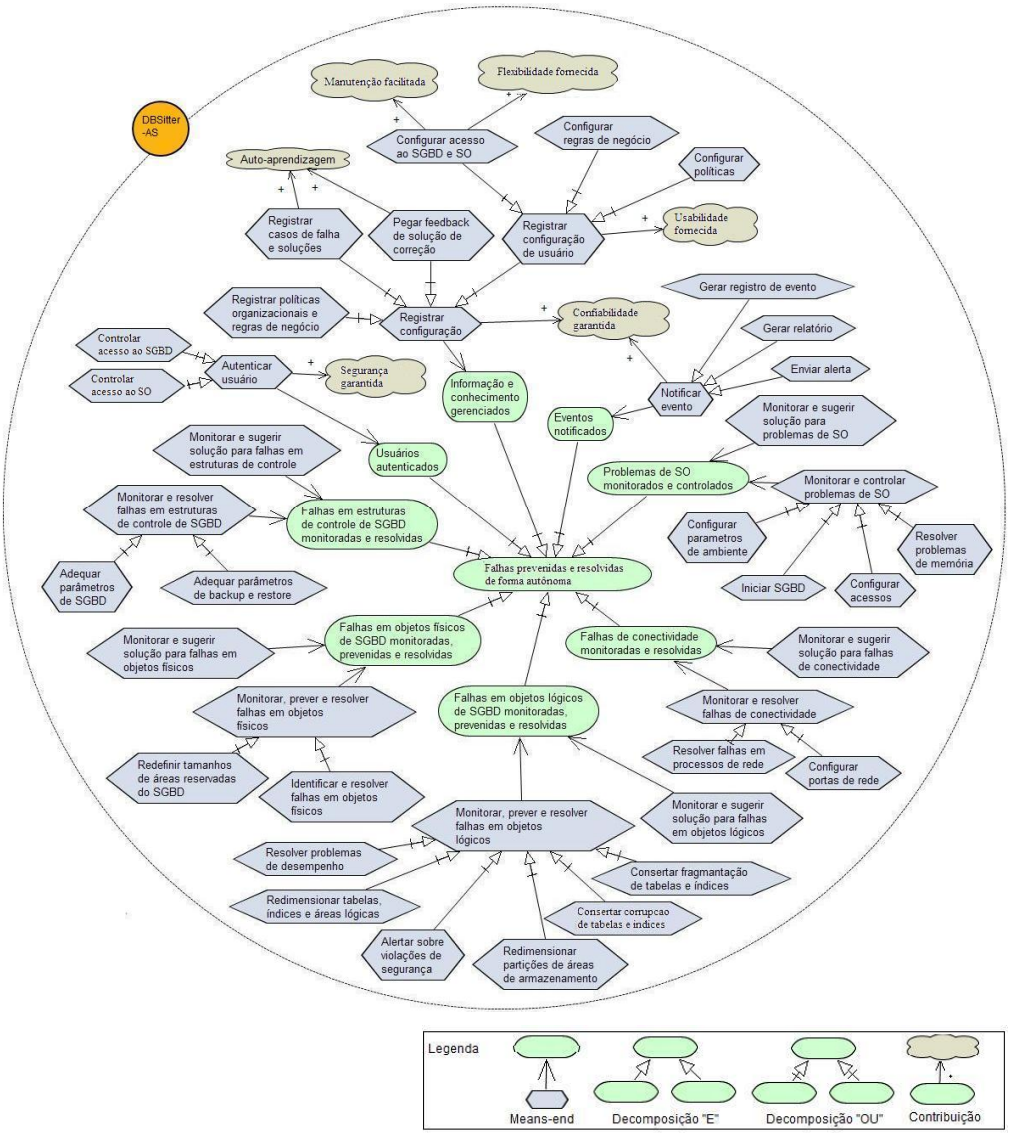

Figura 4-6: Diagrama de metas do ator DBSitter-AS estendido

É importante salientar que esse diagrama não representa a totalidade exaustiva de planos e subplanos possíveis. Procurou-se contemplar os planos essenciais que permitam uma visão ampla das atividades que serão desempenhadas pelo SMA. Devem-se acrescentar outros planos e metas, mesmo em fases mais adiantadas da metodologia, à medida que forem sendo identificadas e o raciocínio por trás deles refinado.

Discussão

Nesta altura da modelagem, temos identificados os atores externos, seus interesses e dependências e o mapeamento dos requisitos do sistema. Baseado nesses requisitos foram estabelecidas as metas do ator sistema DBSitter-AS e mapeados todos os planos para que essas metas sejam atingidas.

Na próxima fase da metodologia Tropos, vamos modelar como metas e planos podem ser distribuídas funcionalmente entre papéis e agentes que as realizem, para cumprir os objetivos globais do DBSitter-AS.

4.2.3 Modelagem do Projeto Arquitetural *(Architectural Design)*

A fase de projeto arquitetural define a arquitetura global do sistema em termos de subsistemas interconectados através de dados e fluxos de controle. A arquitetura do sistema constitui um modelo de estrutura do sistema relativamente pequeno que descreve como os componentes trabalham.

Atividades da Fase

Os diagramas foram projetados seguindo as seis atividades selecionadas para esta fase identificadas com prefixo **AD**, seguido dos seus números seqüenciais, e são gerados seis artefatos, conforme descrito a seguir.

AD.01 - Incluir novos atores (sub-sistemas) e delegar a eles sub-metas do sistema

Artefato de entrada: *A4 – Diagrama de metas estendido.*

Artefatos de saída: *A6 – Diagrama de ator para a arquitetura do sistema*
A7 – Estilos arquiteturais selecionados.

A arquitetura DBSitter-AS possui várias metas que foram identificadas na fase de análise dos requisitos (artefato A4 - *Diagrama de metas estendido*). As metas foram agrupadas de forma a representar funcionalidades correlatas e atribuídas a cinco novos atores, conforme resumido no Quadro 4-2.

Quadro 4-2: Relação entre as metas identificadas e atores

Metas identificadas na fase de requisitos	Metas agrupadas por funcionalidades comuns	Ator responsável pela satisfação da meta
Falhas em estruturas de controle de SGBD monitoradas, prevenidas e resolvidas	Solucionar Problemas de Banco de Dados	Administrador de SGBD
Falhas em objetos físicos de SGBD monitoradas, prevenidas e resolvidas	Solucionar Problemas de Banco de Dados	Administrador de SGBD
Falhas em objetos lógicos de SGBD monitoradas, prevenidas e resolvidas	Solucionar Problemas de Banco de Dados	Administrador de SGBD
Problemas de SO monitorados e controlados	Solucionar Problemas de SO	Administrador de SO
Falhas de conectividade monitoradas e controladas	Solucionar Problemas de conectividade	Administrador de Conectividade
Informações e conhecimento gerenciados	Coletar Informações do usuário e organização	Administrador de Conhecimento
Eventos notificados	Repassar Informações para usuários	Administrador de Comunicação
Usuários autenticados	--	Todos os Administradores

A partir desta análise foram introduzidos os atores apresentados na Figura 4-7, que representa o diagrama preliminar da arquitetura DBSitter-AS. Esses novos atores internos ao ator sistema DBSiter-AS serão a base da modelagem de papéis e agentes do SMA. Papel é uma abstração do comportamento de um ator que pode ser implementada por um ou mais agentes. Na Figura 4-7, a meta atendida pelo ator Administrador de SGBD representa uma aglutinação das três metas de responsabilidade desse ator, constantes do Quadro 4.2, para simplificação da representação. Já a meta "Usuários Autenticados" não foi colocada explicitamente por ser responsabilidade comum a todos os atores internos ao ator sistema DBSitter-AS.

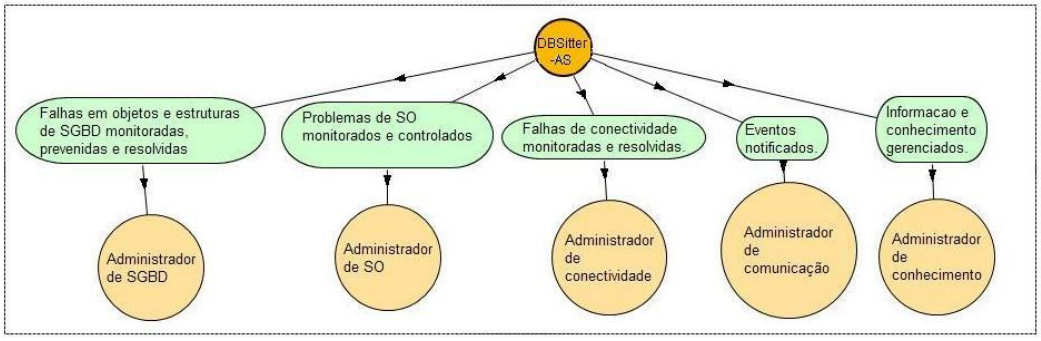

Figura 4-7: Diagrama preliminar de arquitetura

AD.02 – **Selecionar estilo arquiteturais alternativos.**

Em Tropos são definidos estilos arquiteturais organizacionais para aplicações distribuídas, cooperativas e dinâmicas como sistemas multiagentes para guiar o projeto da arquitetura do sistema [Castro et. al 2002]. Estes estilos arquiteturais são baseados em conceitos e alternativas de projetos provenientes de pesquisas em gestão organizacional. *Flat strucuture*, *Pyramid*, *Joint-venture*, *Structure-in-5*, *Takeover*, entre outros são exemplos destes estilos [Fuxman et. al 2001].

Castro e colegas [Castro et al. 2002] avaliam os estilos organizacionais usando atributos não-funcionais de qualidade de software, identificados para arquiteturas envolvendo componentes autônomos e coordenados, tais como previsibilidade (1), segurança (2), adaptabilidade (3), coordenabilidade (4), cooperatividade (5), disponibilidade (6), integridade (7), modularidade (8) ou agregabilidade (9). O Quadro 4-3 apresenta um exemplo de correlação entre alguns estilos arquiteturais e os atributos de qualidade de software. As notações +, ++, -, -- significam, respectivamente contribuições parcial/positivo, suficiente/positivo, parcial/negativo e suficiente/negativo.

Quadro 4-3: Correlações entre Estilos arquiteturais e atributos de qualidade, adaptado de [Castro et al. 2002]

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Structure-in-5	+	+		+	-	+	++	++	++
Pyramid	++	++	+	++	-	+	--	-	
Joint-venture	+	+	++	+	-	++		+	++
Bidding	--	--	+	-	++	-	--	++	
Take over	++	++	-	++	--	+		+	+

Na fase de projeto arquitetural é selecionado um estilo arquitetural usando os critérios de qualidade do software representados na fase "*Late Requirements*" como "*softgoals*" (requisitos não-funcionais) no diagrama de atores.

Na arquitetura DBSitter-AS são considerados como atributos de qualidade de software as seguintes características expressas como *softgoals*:

a. **Usabilidade**: O sistema deve estar em conformidade com os padrões definidos, ter uma interface amigável para os usuários, disponibilizar ferramentas de auxilio, ser didático e ser documentado.

b. **Confiabilidade**: O sistema deve prover a integridade dos dados e estar sempre disponível. Confiabilidade está relacionada à integridade e à disponibilidade.

c. **Desempenho**: O sistema deve especificar os parâmetros de medida de desempenho explícitos, como o tempo de resposta, quantidade de acessos e área de armazenamento.

d. **Manutenibilidade**: O sistema deve especificar os parâmetros de implementação e extensibilidade. Manutenibilidade está associado à adaptabilidade.

e. **Segurança**: O sistema deve especificar os parâmetros de segurança, tais como perfis de acesso, sigilo e auditoria.

Uma vez definidos os atributos de qualidade mais relevantes para o DBSitter-AS, podemos comparar com os critérios de qualidade do software dos estilos organizacionais através do quadro de correlação. Podemos observar que os estilos que mais se adaptam ao DBSitter-AS são o *Joint-venture* e o *Structure-in-5*. Esta decisão foi tomada mediante as contribuições dos atributos de qualidade, como demonstrado na Figura 4-8.

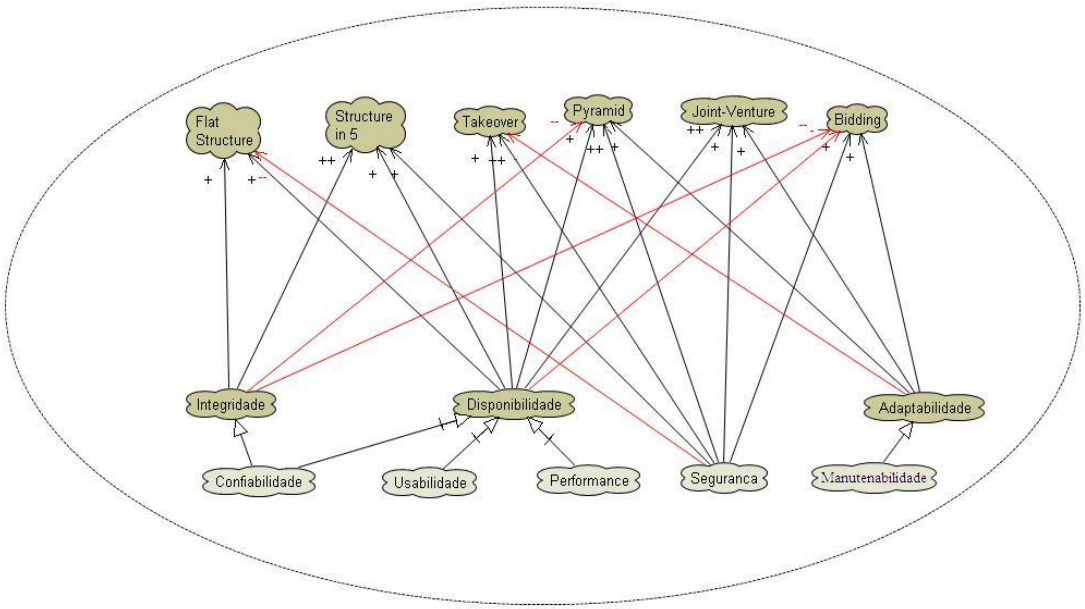

Figura 4-8: Seleção do Estilo Arquitetural

O *Structure-in-5* é um típico modelo de estrutura organizacional que se divide em componentes de execução estratégica, componentes de coordenação e componentes operacionais. O *Joint-venture* é um estilo mais descentralizado que envolve uma concordância entre dois ou mais parceiros principais para obter os benefícios de operar em larga escala e reusar a experiência e conhecimento dos parceiros. Cada parceiro pode gerenciar e controlar a si mesmo em uma dimensão local e interagir com os outros para trocar informações.

Os dois modelos se aplicam igualmente ao DBSitter-AS, embora não satisfaçam todos os *softgoals* completamente. O estilo *Joint-venture* foi selecionado, devido à

necessidade do sistema de prover agentes autônomos que não dependam de uma coordenação estritamente centralizada.

O estilo Joint-Venture dá bom suporte à coordenabilidade [Kolp et al. 2002] já que cada participante interage para a tomada de decisões estratégicas, via coordenador. Os participantes indicam seus interesses e o coordenador responde com informações estratégicas que levam em conta outros participantes. Por outro lado, sendo os participantes heterogêneos, a coordenação de atividades conjuntas pode ser uma tarefa difícil. A posição central do coordenador implica a responsabilidade de resolver conflitos e diminuir a imprevisibilidade das ações dos participantes. Novos participantes podem facilmente se adaptar à estrutura, bastando registrarem-se junto ao coordenador. Os participantes têm liberdade trocar recursos e informações diretamente entre si, embora o coordenador não possa ser removido, devido ao seu papel central.

Uma vez selecionado o estilo arquitetural, deve-se adaptar os atores já identificados para formarem a arquitetura do sistema desejado. Como o estilo *joint-venture* estabelece a existência de um ator Coordenador, foi acrescentado o Coordenador DBSitter-AS, conforme mostra o diagrama de arquitetura da Figura 4-9 em representação do DBSitter-AS em estilo *Joint-venture*.

AD.03 *– Incluir novos atores baseados no estilo arquitetura selecionado.*

Artefato de entrada: *A4 – Diagrama de atores.*
A7 – Estilos arquiteturais selecionados.
Artefatos de saída: *A8 – Diagrama de ator para a arquitetura do sistema*

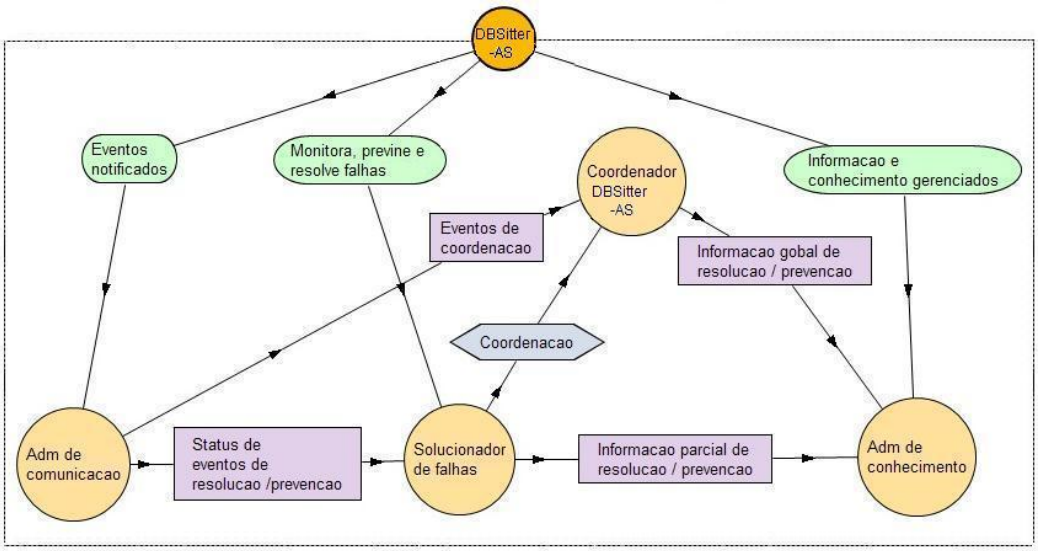

Figura 4-9: Diagrama de arquitetura DBSitter-AS

A responsabilidade do coordenador, segundo o estilo *Joint-venture* é de coordenar tarefas, fazer planos globais e gerenciar o compartilhamento de conhecimento e recursos. Recursos são representados por retângulos no diagrama. O "Coordenador DBSitter-AS" se relaciona com o ator "Solucionador de Falhas", coordenando as tarefas que são delegadas a ele e gerenciando o compartilhamento de informações. O coordenador depende do "Solucionador de Falhas" para realização de tarefas de reparo e prevenção e para perceber falhas no ambiente. Estas mesmas dependências ocorrem entre o "Coordenador DBSitter-AS" e os demais atores do sistema: "Administrador de Comunicação" e "Administrador de Conhecimento".

O ator "Solucionador de falhas" é uma abstração de três atores (Administradores de conectividade, SGBD e SO), que por terem características comuns na arquitetura (mesmas atribuições de detecção de falhas, prevenção e resolução) são representados dessa maneira para fins de simplificação dos diagramas que representam a arquitetura. A Figura 4-10 abaixo apresenta as categorias do ator "Solucionador de Falhas".

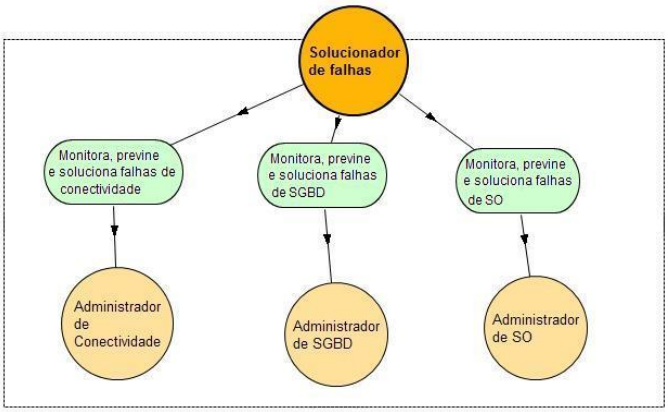

Figura 4-10: O ator "Solucionador de Falhas"

AD.04 - Inclusão de novos atores usando os padrões sociais de sistemas multiagentes.

Artefato de entrada: *A8 - Diagrama de ator para a arquitetura do sistema*
Artefatos de saída: *A9 – Diagrama de arquietura estendido*

O Próximo passo da metodologia é o detalhamento de cada um dos atores já identificados. Através de expansões do comportamento interno de cada ator, identificamos como será seu comportamento e modelamos possibilidades de implementação. São confeccionados, então, diagramas de arquitetura estendidos que explicitem o comportamento e as comunicações de agentes que desempenham as funcionalidades dos atores.

Também nesta etapa, realiza-se uma análise de *padrões sociais* para os atores. Padrões sociais são representações que descrevem comportamentos baseados nas funcionalidades básicas de agentes, como *broker, mediator, wrapper, embassy, monitor* [Hyden et al. 1999]. Por exemplo, um agente tipo *mediator* responsabiliza-se por encapsular a especificação de como os agentes interagem, mantendo em um único ponto a natureza das interações, um exemplo de agente tipo *mediator* é um agente coordenador.

A Figura 4-11 apresenta a modelagem de uma expansão do diagrama de arquitetura, tendo como foco o ator "Administrador de SGBD" e incorporação de comportamento para resolução de falhas em objetos lógicos de SGBD.

O comportamento interno do ator "Administrador de SGBD" está contido no quadrado pontilhado que delimita sua abrangência no sistema. Para cumprir a meta parcial (reduzida para melhor observação diagramática) de monitorar falhas de objetos lógicos no SGBD (e.g. tabelas, índices) o ator Administrador de SGBD tem sua definição implementada por agentes que realizam dois papéis: "Detector de falha em objetos lógicos", "Solucionador de falha em objeto lógico" (representados por círculos com linha curva na parte de baixo) e um agente "*Wrapper* SGBD".

A representação pelo papel, introduzida neste diagrama significa que a arquitetura deixa a implementação livre, podendo ser um agente que realize vários papéis ou um papel realizado por mais de um agente. Acompanhando o diagrama, vemos que o agente "Coordenador DBSitter-AS" é responsável por criar instâncias dos agentes que vão desempenhar os papéis de "Detector de Falha em Objeto Lógico" e "Solucionador de Falha em Objeto Lógico". Ao papel "Detector de falha em objetos lógicos" é delegada a tarefa de perceber ou prever (mediante aplicação de técnicas de análise de tendência, por exemplo) falhas em objetos lógicos no SGBD e notificar ao "Coordenador DBSitter-AS".

O padrão social *wrapper* tem a função de ser uma camada de *interface* com algum sistema externo (e.g. SGBD, sistema legado). No diagrama apresentado foram introduzidos os agentes "*Wrapper Base de Conhecimento"*, "*Wrapper SGBD"* e "*Wrapper Adm. Comunicação"* para atuarem como interface com os sistemas com os quais se comunicam, a saber: Repositório de Conhecimento (SGBD auxiliar que contém todas as metainformações sobre casos de falha, agentes, resolução, acessos e políticas organizacionais), SGBD alvo da monitoria (e.g. Oracle, PostgreSql) e meios externos de comunicação (e.g. e-mail, SMS).

O papel "Detector de falha em objetos lógicos" usa o padrão social *Monitor* cujas ações são monitorar status de objetos lógicos do SGBD, via comunicação com o agente "*Wrapper* SGBD"; alertar o coordenador sobre falhas ou iminência de falhas e responder sobre status de objetos lógicos para outros agentes. O papel "Solucionador de falhas em objeto lógico" tem incumbência de executar as ações de correção no

SGBD, através da troca de mensagens (contendo comandos e resultados em SQL) com o agente "*Wrapper* SGBD".

Complementam essa representação, os atores "Administrador de SO" e "Administrador de conectividade" que colaboram enviando informação sobre seus objetos monitorados para o agente coordenador e para agentes solucionadores de falhas.

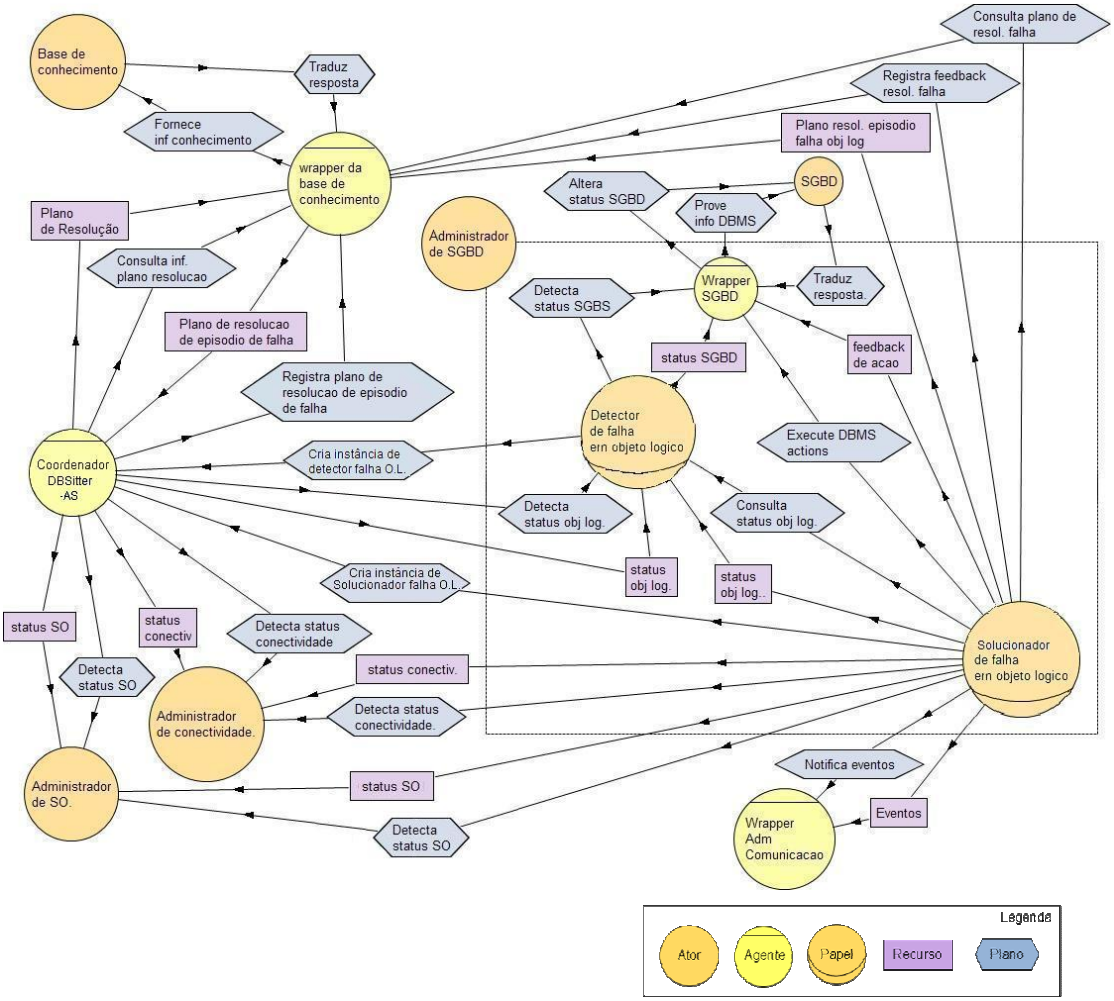

Figura 4-11: Diagrama de arquitetura estendido para o ator Administrador de SGBD

Conforme visto no Quadro 4-2, foi atribuído ao ator "Administrador de SGBD", além de monitorar objetos lógicos, também monitora objetos físicos de SGBD (e.g. arquivos físicos de dados) e estruturas de controle de SGBD (e.g. arquivos de parâmetros). Isso significa a definição similar de papéis Detector e Solucionador de falhas em objetos físicos de SGBD e Detector e Solucionador de falhas em estruturas de controle de SGBD. Para cada um desses comportamentos (atuação em objetos físicos e estruturas de controle), um diagrama semelhante ao Figura 4-11 deve ser

confeccionado, levando-se em conta as necessidades de registro de documentação e lacunas de entendimento da equipe de programação que irá codificar o SMA.

A seqüência completa de diagramas[13] dessa fase da Metodologia Tropos (não contemplada no presente documento) é completada com o detalhamento interno dos outros atores (e.g. Administrador de SO, Administrador de conectividade) da mesma forma que foi demonstrado parcialmente para o ator "Administrador de SGBD".

AD.05 - Identificar um conjunto de capacidades necessárias aos agentes para cumprir suas metas e planos.

Artefato de entrada: *A6, A7, A8 e A9*
Artefatos de saída: *A10 – Quadro de Capacidades*

Seguindo a metodologia Tropos, no intuito de melhor definir o SMA, a próxima atividade é caracterizar capacidades que os agentes devem ter. As capacidades representam a habilidade de um ator de definir, escolher e executar um plano para atendimento de uma meta, mediante certas condições ambientais e na presença de um evento específico [Bresciani et al. 2004]. Para cada capacidade, o agente tem um conjunto de planos que pode realizar. As capacidades não são derivadas automaticamente da análise means-ends de cada meta (executadas por um ou mais planos), mas podem ser facilmente identificadas analisando-se os diagramas de atores estendidos. Normalmente cada relacionamento means-ends dará lugar a uma ou mais capacidades. Por exemplo, na Figura 4-11 podemos ver que o agente "Coordenador DBSitter-AS" desempenha os planos "Cria instância solucionador falha O.L. (Objeto Lógico)" e "Cria instância detector falha O.L.", logo podemos identificar uma capacidade de "Instanciar outros agentes", a ser registrada numa lista específica, como o Quadro 4-4.

Após identificar todas as capacidades, estas são agrupadas gerando um conjunto de padrões que levam a classificar os tipos de agentes do modelo. O Quadro 4-4 mostra as capacidades para os agentes que colaboram para a implementação das funcionalidades do ator "Administrador de SGBD". A análise das capacidades permite um aprofundamento na modelagem dos agentes e o registro antecipado de capacidades que ainda estão implícitas no modelo, mas que atendem a requisitos de implementação (e.g. capacidade "Registrar serviço em páginas amarelas" que atende a forma de comunicação do SMA) ou que atendem a requisitos do sistema (e.g. "Identificar tendência de falha", que atende ao requisito de previsão de falhas), mas ainda não estão representadas em diagramas.

[13] A seqüência completa dos diagramas pode ser encontrada em http://www.cin.ufpe.br/~dbsitter/dbsiter-as

Quadro 4-4: Quadro de Capacidades (parcial)

Agente	Capacidade	Tipo de agente
Coordenador DBSitter-AS	Instanciar outros agentes Receber status de objeto Receber informação de falha Elaborar plano para resolução de episódio de falha	Coordenador
Solucionador de falha em objeto lógico	Receber status de objeto Notificar eventos de resolução Corrigir falhas de problemas lógicos Sugerir solução Solicitar serviços de outros agentes solucionadores	Solucionador falha em objeto lógico
Detector de falha em objeto lógico	Receber status de outros objetos Informar falha objeto lógico Identificar tendência de falha Informa status de objetos lógicos	Detector de falha em objeto lógico
Wrapper SGBD	Emite comandos SQL no SGBD alvo Responde com resultados de comandos SQL	*Wrapper* do SGBD
Todos	Registrar serviço em pág. amarelas	Todos

Considerações de Modelagem e Implementação

Por se tratar de um *framework* arquitetural, consideramos que a especificação do DBSitter-AS vai até o nível de detalhe apresentado. O framework DBSitter-AS não tem o intuito de tolher a flexibilidade para variações de implementação possíveis.

Em linhas gerais podemos resumir a arquitetura como sendo um SMA em estilo *joint-venture* composto por um agente coordenador e agentes que monitoram e previnem e resolvem falhas em um SGBD alvo. Os agentes que atuam no SGBD alvo estão divididos funcionalmente em agentes que atuam em estruturas de controle, objetos físicos ou objetos lógicos. Adicionalmente temos uma base de dados auxiliar (o Repositório de Conhecimento) que se responsabiliza pela persistência das informações do SMA (e.g. casos de falha e soluções, histórico de atuações, *feedbacks* do usuário, regras de negócio e políticas organizacionais, tipos de agentes e planos de resolução de episódios de falha). Para interação com o SGBD alvo e Repositório de Conhecimento, devemos ter dois agentes *wrappers.* Também há especificados agentes exclusivos para comunicação externa (e.g. notificações para o usuário) e agentes que detectam e resolvem falhas no sistema operacional e na camada de rede correlatas com falhas no SGBD alvo.

Alguns elementos da arquitetura ficaram especificados apenas como papel, para que a implementação seja livre, podendo mais de um agente implementar um único

papel ou um agente implementar mais de um papel. Questões como redundância de agentes também não foram especificadas propositadamente, embora recomendemos fortemente esquemas de redundância, onde um tipo de agente possa fazer as vezes de outro. A existência da persistência externa ao agente é um primeiro passo para implementação de redundâncias, já que todas as informações vitais sobre os agentes e atuações estarão guardadas.

Completa a especificação do *framework*, a modelagem do Repositório de Conhecimento, que será apresentado na próxima seção.

4.3 O Modelo de Persistência DBSitter-AS

De forma a prover o *framework* de características que permitam a manutenção da informação sobre realização de atividades e cadastros de casos de falhas, procedimentos de resolução de falhas ou prevenção, foi desenvolvido um modelo relacional capaz de armazenar essas informações. O modelo de persistência DBSitter-AS deve ser "instalado" como parte da realização do ator Base de Conhecimento, apresentado na arquitetura DBSitter-AS. O ator Base de Conhecimento configura-se pelo Repositório de Conhecimento em conjunto com o Agente *Wrapper* da Base de Conhecimento.

O Agente *Wrapper* da Base de Conhecimento, que é um agente especializado em manipulação do SGBD no qual o modelo de persistência foi instalado. Tipicamente esse agente *wrapper* é dotado de interface de comunicação para linguagem de consulta SQL (e.g. JDBC, nos casos de implementação do agente em Java) e autorizações de acesso para consulta e manipulação do repositório. Como forma de aumentar a segurança das informações manipuladas, apenas esse agente *wrapper* possui o privilégio de consulta ao repositório, devendo todos os outros agentes, serem autorizados pelo agente coordenador, em atividade planejada, e apenas para acesso às informações necessárias.

O Repositório de Conhecimento é implementado através da modelagem dos elementos do SMA em uma base de dados relacional auxiliar, preferencialmente gratuita e de código aberto (e.g. Firebird, MySql). O modelo concebido armazena todas as informações relacionadas aos cadastros de casos de falhas no SGBD alvo (o SGBD a ser gerenciado), passos para resolução ou prevenção de casos de falhas, informações acerca dos tipos de agentes disponíveis (sensores, solucionadores) e capacidades de atuação (e.g. sensor de objetos lógicos Oracle, solucionador de falha em estrutura de controle do PostgreSql). Adicionalmente há estruturas de armazenamento de informações específicas para regras de negócio e políticas organizacionais que serão úteis para montagem dos planos de resolução de falhas e registros de todas as atividades realizadas, agentes responsáveis e *feedback* do

usuário sobre a efetividade da atuação feita. O *feedback* mencionado fará parte dos mecanismos de aprendizagem cuja existência estão previstas no *framework*.

O Modelo Relacional de Persistência

O Repositório de Conhecimento está modelado conforme a Figura 4-12, e é composto pelas seguintes tabelas e funcionalidades:

***Caracterização de agentes**:* São tabelas que descrevem os agentes modelados (tabelas AGENTE, TIPO-AGENTE e AREA-ATUACAO) e são usadas pelo agente Coordenador para "iniciar" a execução do SMA. Uma vez instanciados, cada agente deve se registrar no componente de páginas-amarelas utilizado pelo SMA (esse componente, em algumas implementações também pode ser um agente). Uma tabela espelho para o *Páginas-amarelas* (tabela PAGINAS-AMARELAS) é então preenchida com as informações de agentes e serviços disponíveis.

Caracterização de serviços: Cada *serviço* (tabela SERVICO) também é registrado na base de persistência (e.g. Desfragmentação de uma *tablespace*[14] Oracle) e também as ações necessária para resolvê-lo. Os serviços podem possuir uma ou mais ações (tabela SERVICO-ACAO) de resolução (e.g. uma desfragmentação de *tablespace* no Oracle pode ser realizada através de quatro ações: exportar os dados contidos na *tablespace* para um arquivo externo, excluir a *tablespace*, recriar a *tablespace* e importar os dados de volta). As *ações* (tabela ACAO) são caracterizadas por tipos de ação (tabela TIPO-ACAO) que guiam os agentes que vão realizá-las (e.g. comando SQL, execução de script externo, comando do sistema operacional). Cada agente tem suas habilidades compatíveis e previamente cadastradas para executarem as ações constantes dos seus serviços.

Caracterização de resoluções de falha: Uma vez detectada alguma *ocorrência* de irregularidade no SGBD alvo que justifique uma intervenção corretiva ou preventiva (tabelas OCORRENCIA_MANUTENCAO, TIPO-OCORRENCIA), o agente Coordenador necessita planejar a execução desta intervenção. Os planos de resolução têm seus modelos cadastrados previamente (tabela PLANO_ACAO), identificando que tipo de agente será responsável pela sua execução e quais serviços serão realizados (tabela SERVICO_PLANO).

Planejamento e execução de resoluções de falha: O planejamento da resolução da ocorrência de irregularidade (tabela PLANO_RESOLUCAO_OCORRENCIA) é

[14] Tablespaces são áreas lógicas de armazenamento do SGBD Oracle. Objetos como tabelas e índices são atreladas às *tablespaces* e estas aos arquivos físicos no Sistema operacional.

cadastrado pelo agente coordenado e atribuído a um agente responsável. Na hora da confecção desse plano, o agente coordenador aplica políticas e regras organizacionais que serão restrições à execução do plano (tabelas TIPO_REGRA, POLITICAS_E_REGRAS). Um exemplo de regra do tipo *temporal*, seria não poder retirar o SGBD alvo do ar em horário menor que 22h e maior que 6h.

Avaliação de resoluções de falha: Os registros das tentativas de resolução (tabela TENTATIVA_RESOLUCAO) e das avaliações (tabela AVALIADOR) das ações de resolução de falha também estão previstas e complementam as informações dos planos de resolução.

A utilização de um modelo de persistência permite que pacotes de serviços possam ser cadastrados para serem tratados pelos agentes. Por exemplo, normalmente grande parte de erros em objetos lógicos de SGBD são resolvidos através de comandos SQL ou extensões de SQL. Visto que os agentes atuadores em objetos lógicos possuem habilidade de manipular *strings* de comandos SQL, a estrutura básica de um agente dessa natureza não precisa ser alterada para ser aplicada às ações de SGBD diferentes. Basta que os comandos SQL correspondentes às ações de resolução de falha sejam cadastrados na base de dados do Repositório de Conhecimento e sejam atribuídos tipos de agentes e serviços apropriados para executá-los.

Da mesma forma, grande parte dos erros em estruturas de controle, falhas no sistema operacional e em elementos de rede podem ser tratados com scripts pré-formatados. Isso permite que pacotes específicos para determinados tipos de problema e SGBD sejam criados e aos poucos distribuídos para utilização com componentes de arquitetura DBSitter-AS (e.g. pacote de resolução de erros em objetos lógicos do SQL Server, pacote de resolução de erros em estruturas de controle para PostGreSQL).

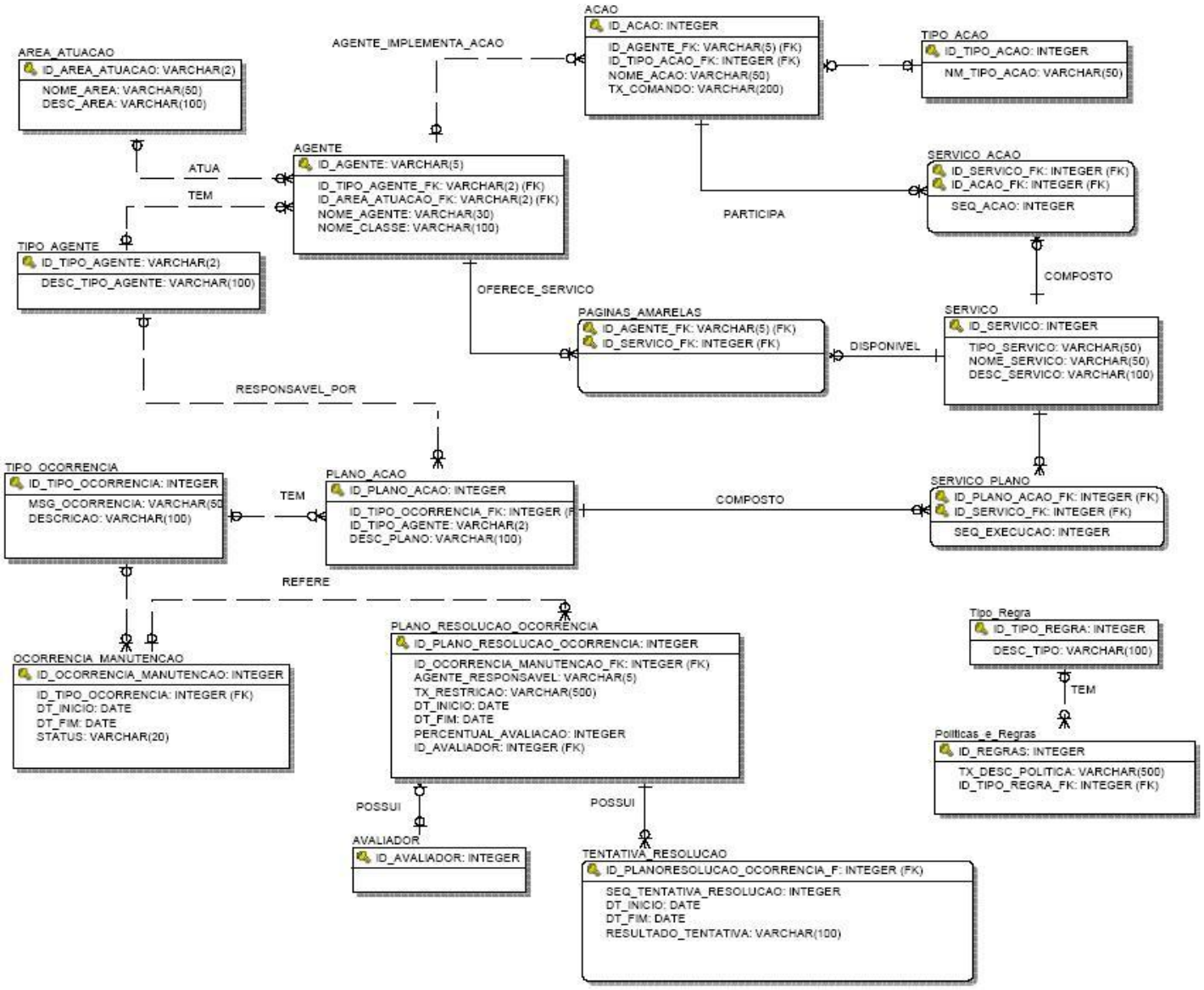

Figura 4-12: O Modelo de Persistência DBSitter-AS

4.4 Considerações

Neste Capítulo apresentamos o DBSitter-AS, uma especificação arquitetural baseada em SMA que visa o desenvolvimento de componentes para gerenciamento autônomo de SGBD. A concepção do *framework* DBSitter-AS teve como base o respeito aos princípios do Projeto DBSitter [Carneiro et al. 2004], a saber: ser estruturado em plataforma aberta, ser concebido como um SMA, ser multiplataforma de execução (e.g. Windows ou Unix) e ter capacidade de monitorar tipos diferentes de SGBD. Além destes, a arquitetura DBSitter-AS busca o máximo de aderência aos oito princípios básicos da Computação autônoma, com a definição das seguintes funcionalidades:

- ***Conhecer a si próprio:*** *a camada de persistência procura registrar todas as informações necessárias do SMA, responsabilidades e funções;*
- ***Capacidade de se configurar e reconfigurar****: as atividades de monitoração são específicas do SGBD alvo (SGBD e versão), cadastradas de acordo com a necessidade. Otimização de ações mediante feedback do usuário também estão previstas;*
- ***Realizar constante auto-otimização****: Previstas funcionalidades de análise de feedback e aprendizagem, que podem ser implementadas dando maior poder de raciocínio ao agente Coordenador* DBSitter-AS *para concepção dos planos de resolução de falhas e prevenções;*
- ***Capaz de se reparar****: Além das funcionalidades de reparação nos SGBD alvo, a camada de persistência registra todas as informações correntes, permitindo recuperação do status após paradas anormais. A Atuação de agentes, em threads* paralelas*, também pode prover redundância capaz de compensar interrupções de atividades. Agentes diferentes, dependendo do projeto, podem assumir papéis de outros (e.g. um solucionador assumir o papel de coordenação temporária);*
- ***Capaz de se proteger:*** *Os agentes só realizam atividades mediante autorização do Agente Coordenador e autenticações necessárias. Esquemas de coordenação replicada podem ser implementados.*
- ***Auto-adaptação****: está considerada a possibilidade de implementação do agente coordenador com capacidade de aprendizagem (e.g. utilização de motores de inferência e regras de produção)* baseada no *feedback* do usuário;

- ***Solução não-proprietária, aberta e baseada em padrões****: está prevista utilização de repositório gratuito, plataforma Java de desenvolvimento aberta e padronização FIPA[15] para comunicação entre os agentes;*
- ***Prever e antecipar recursos****: Funcionalidades de análise de tendência são previstas para implementação nos agentes detectores de falha em objetos lógicos e físicos do SGBD alvo.*

A idéia da implementação de software para computação autônoma baseada em Agentes Inteligentes e SMA é corroborada por pesquisadores e teóricos da Computação Autônoma [White et al. 2004] e Agentes Inteligentes [Wooldridge 2001]. O desenvolvimento da camada de gerenciamento externa ao software do SGBD gerenciado confere uma maior liberdade de aplicação de técnicas de IA para os componentes provedores de autonomia. A não necessidade de alteração de código do SGBD gerenciado também possibilita a utilização do *framework* para o desenvolvimento de componentes que gerenciem SGBD proprietários, que são normalmente os mais utilizados no mercado.

Complementando o uso da metodologia Tropos, temos a fase de Projeto Detalhado, onde papéis serão mapeados para agentes e agentes são especificados em detalhe. Em virtude de ser muito ligada à implementação, esta etapa foi desenvolvida como parte dos trabalhos desta dissertação a título de experimento prático do desenvolvimento seguindo o framework. Também a implementação de um SMA para resolução de um caso de falha para o SGBD Oracle foi realizada. Essas atividades são apresentadas no Capítulo 5.

[15] http://www.fipa.org/

5 Uma Implementação do DBSitter-AS

Neste capítulo, apresentaremos um experimento realizado com o *framework* DBSitter-AS com o objetivo de exemplificar as atividade da fase de Projeto Detalhado da Metodologia Tropos e a implementação da especificação arquitetural proposta.

Em virtude de estarem muito ligadas às soluções de implementação do SMA, as atividades desta fase de Projeto Detalhado serão demonstradas de forma específica para modelagem da resolução do caso de falha selecionado como exemplo. O exemplo desenvolvido tem como objetivo ser uma referência e ponto de partida para futuros trabalhos de implementações de componentes de arquitetura DBSitter-AS.

Do ponto de vista do SMA desenvolvido, procuramos selecionar um caso que envolvesse uma participação mais abrangente de agentes, que precisassem colaborar para resolver a falha.

Nas seções a seguir, apresentaremos o caso de falha e depois os artefatos de Projeto Detalhado que modelam sua resolução. Posteriormente, apresentaremos a implementação do experimento, as tecnologias usadas e os resultados obtidos.

5.1 O Caso de Falha Selecionado

Como exemplo de desenvolvimento baseado nas especificações DBSitter-AS, selecionamos um caso de falha que pudesse configurar-se como típicas de um SGBD de mercado. O SGBD alvo selecionado foi o Oracle, um dos mais utilizados correntemente.

A falha a ser solucionada pela comunidade de agentes é o *dimensionamento incorreto do buffer cache do SGBD Oracle*. O *buffer cache* é uma área de memória utilizada para armazenar os blocos lidos a partir dos discos. Um *buffer cache* pequeno irá fazer com que o SGBD precise remover do *cache* os blocos de dados seguindo uma lista dos "menos recentemente usados" (LRU, Last Recently Used). Dependendo da freqüência que isso acontece, poderá causar uma queda de desempenho.

O *buffer cache* pode ser avaliado calculando-se o *buffer hit ratio* (razão de acerto do *buffer*), usando a seguinte fórmula:

```
Hit_ratio =     (dB_block_gets + consistent_gets - physical_reads) /
                (dB_block_gets + consistent_gets)
```

Onde,

db_block_gets é o número de vezes que o bloco de dados foi requisitado para o *buffer cache*;

consistent_gets é o número de vezes que uma leitura consistente foi requisitada para blocos no *buffer cache*;

physical_reads é o número total de blocos de dados lidos do disco para o buffer cache;

Mediante uma consulta SQL ao dicionário de dados do Oracle, é possível identificar o *buffer hit ratio*:

```
SELECT  1 - (phy.value / (cur.value + con.value)) "Cache Hit Ratio"
  FROM  v$sysstat cur,v$sysstat phy,v$sysstat con
 WHERE  cur.name = 'db block gets'
   AND  phy.name = 'physical reads'
   AND  con.name = 'consistent gets';
```

Caso a consulta acima retorne menos de 90%[16], deve-se aumentar o tamanho do *buffer cache*. Isso significa que é desejável que nove entre dez leituras possam ser satisfeitas sem a necessidade de ir ao disco.

Para simplificação do exemplo, caso a taxa de acerto seja abaixo de 90%, o tamanho do *buffer cache* será aumentado em 10%, como medida preventiva.

Para realizar esse objetivo, um ABD humano necessita retirar o SGBD Oracle do ar (*shutdown*), editar o arquivo de parâmetros de inicialização do SGBD Oracle (arquivo init.ora) e substituir o valor indicado no parâmetro *dB_block_buffers* pelo novo valor estimado. Depois se deve colocar novamente o SGBD Oracle no ar (*startup*).

5.2 Projeto Detalhado (Detailed Design)

O Projeto Detalhado trata a especificação mais detalhada dos agentes, isto é, descreve as metas dos agentes, crenças, capacidades, assim como as comunicações entre os mesmos. Esta fase, além de muito associada à escolha da implementação, também depende das características da linguagem de programação adotada.

Atividades da Fase

Dividimos esta fase em três atividades identificadas com prefixo **DD**, seguido dos seus números seqüenciais, e são gerados três tipos de artefatos, conforme descrito a seguir.

[16] Este indicador em 90% é consenso entre especialistas em desempenho do SGBD Oracle.

DD.01 - Modelar as capacidades de um agente específico

Artefato de entrada: *A10 – Quadro de Capacidades*
Artefatos de saída: *A11 – Diagrama de capacidade (Diagrama de atividades UML)*

Durante a fase de projeto detalhado Tropos, utilizamos diagramas UML (Unified Modelling Language) [Booch et al. 1999] para representação das ações desempenhadas pelos agentes. O diagrama de atividade UML permite modelar uma capacidade do ponto de vista de um agente ou papel.

Para exemplificar esta fase foi modelada a capacidade "Elaborar plano de resolução de episódio de falha", do papel "Coordenador DBSitter-AS". A

Figura **5-1** apresenta este diagrama. Para o agente Coordenador montar um plano ele deve receber a identidade de uma falha que foi detectada por algum agente Detector. Em seguida é feita uma consulta ao Repositório de Conhecimento para saber o procedimento a ser tomado para resolução. Caso a falha tenha solução (há um modelo (*template*) de resolução cadastrado para cada tipo de erro) são aplicadas regras de negócio e políticas organizacionais que orientam a resolução do episódio de falha (e.g. falha que necessite retirar o SGBD do ar só poderá ser agendada para após as 22h). Posteriormente o agente responsável pela resolução da falha será informado do seu agendamento. Caso não haja solução prevista, apenas um alerta será gerado e enviado para o agente Administrador de Comunicação enviar informação da maneira mais adequada.

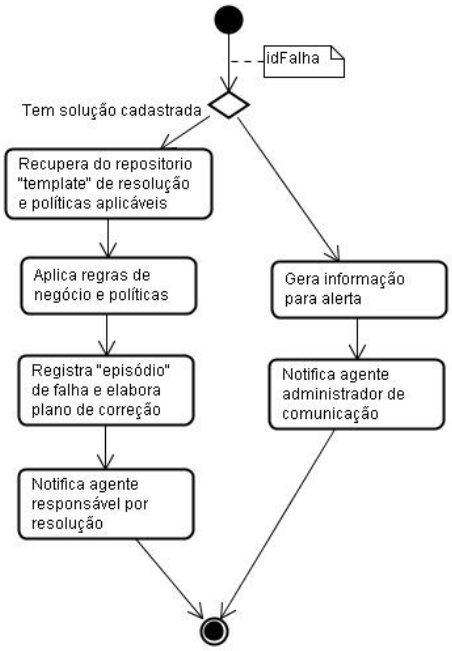

Figura 5-1: Diagrama da capacidade "Elaborar plano de resolução de episódio de falha", do agente Coordenador DBSitter-AS

DD.02 - Especificar cada plano do diagrama de capacidade

Artefato de entrada: *A11 - Diagrama de capacidade*
Artefatos de saída: *A12- Diagrama de planos (Diagrama de atividades de UML)*

Cada plano (ou ação) que compõe um nó de um diagrama de capacidade pode ser especificado mais detalhadamente usando diagramas de atividade UML [Booch et al. 1999]. A Figura 5-2 mostra o diagrama de planos para o plano "Registra episódio de falha e elabora plano de correção". Para executar este plano o agente recebe os parâmetros de correção da falha (e.g. o nome de um índice que esteja fragmentado além do aceitável) e o template do script para correção desta falha. Para cada passo do "template" de resolução desse tipo de falha são aplicadas as restrições de regras de negócio e políticas organizacionais. Em seguida é determinado o agente Solucionador de Falha que será responsável pela execução do plano e acionado o agente Wrapper do Repositório de Conhecimento, para que o plano de resolução seja registrado.

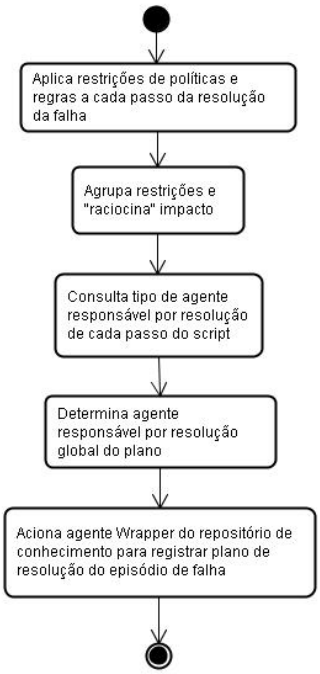

Figura 5-2: Diagrama de Planos para "Registra episódio de falha e elabora plano de correção"

DD.03 - Modelar as interações (ou eventos) do sistema

Artefato de entrada: *A9 – Diagrama de Metas estendido*
Artefatos de saída: *A13 - Diagramas de Interações (Diagramas de Interações AUML)*

Nesta atividade as interações entre os agentes são representadas por diagramas de seqüência de UML. Agentes correspondem a objetos, cuja linha de vida não depende da interação específica e setas de comunicação correspondem a mensagens assíncronas.

No Diagrama de Seqüência representado na Figura 5-3 está especificada a seqüência de mensagens trocadas entre os agentes para a realização da correção de falha em dimensionamento do *buffer cache* do SGBD Oracle. Neste diagrama temos o Agente bufferCacheAgent que foi modelado para desempenhar os papéis de "Detector de Falha em Objeto Lógico" e "Solucionador de Falha em Objeto Lógico" em relação à memória *cache* do SGBD Oracle, o agente coordinatorAgent, o Agente firebirdWrapperAgent (*wrapper* da Base de Conhecimento) e o agente SGBDMonitor (responsável por verificar se o SGBD alvo está no ar, pará-lo ou iniciá-lo).

A seqüência de atividades do diagrama será descrita a seguir. Os números entre parêntesis correspondem ao passo da atividade.

Verificamos no diagrama que, após a detecção de anomalia no dimensionamento do *buffer cache* (1), o agente bufferCacheAgent avisa o coordinatorAgent sobre a existência da falha; (2). O agente coordinatorAgent, por sua vez, deve consultar regras de negócio e planos de resolução na base de conhecimento (2.1 a 2.3) (via consulta ao agente firebirdWrapperAgent) para montar o plano global de resolução do episódio de falha.

Para resolver um episódio de falha, um agente "Solucionador" deve seguir um plano de resolução de ocorrência de falha, que especifica horários, ações e tipos de agentes que irão desempenhá-los (essas ações podem ser desempenhadas pelo agente responsável pela resolução da ocorrência de falha ou por outro tipo de agentes, mediante solicitação deste). No caso do diagrama em análise, para o bufferCacheAgent solucionar a falha deve, como primeiro passo, solicitar ao agente SGBDmonitor que retire o SGBD do ar (esta atividade pode sofrer uma restrição de regra de negócio ou política organizacional, como, por exemplo, ter que aguardar até as 22h) (4). O agente SGBDMonitor deve confirmar que há um plano prevendo retirada de SGBD do ar nas condições correntes para atender a solicitação (4.1) antes de realizar a atividade (4.2). Uma vez confirmado e o SGBD fora do ar, o bufferCacheAgent altera o arquivo de *c*onfiguração do SGBD aumentando a memória disponível (5) e em seguida solicita (6) que o agente SGBDMonitor recoloque o SGBD Oracle no ar (6.1). Uma checagem de efetividade deve ser feita (7) (isso é realizado através de nova detecção de sintoma de mau dimensionamento da memória compartilhada) e este *feedback* é repassado ao agente coordenador (8) que por sua vez registrará o episódio de atuação e resultado na base de conhecimento (8.1).

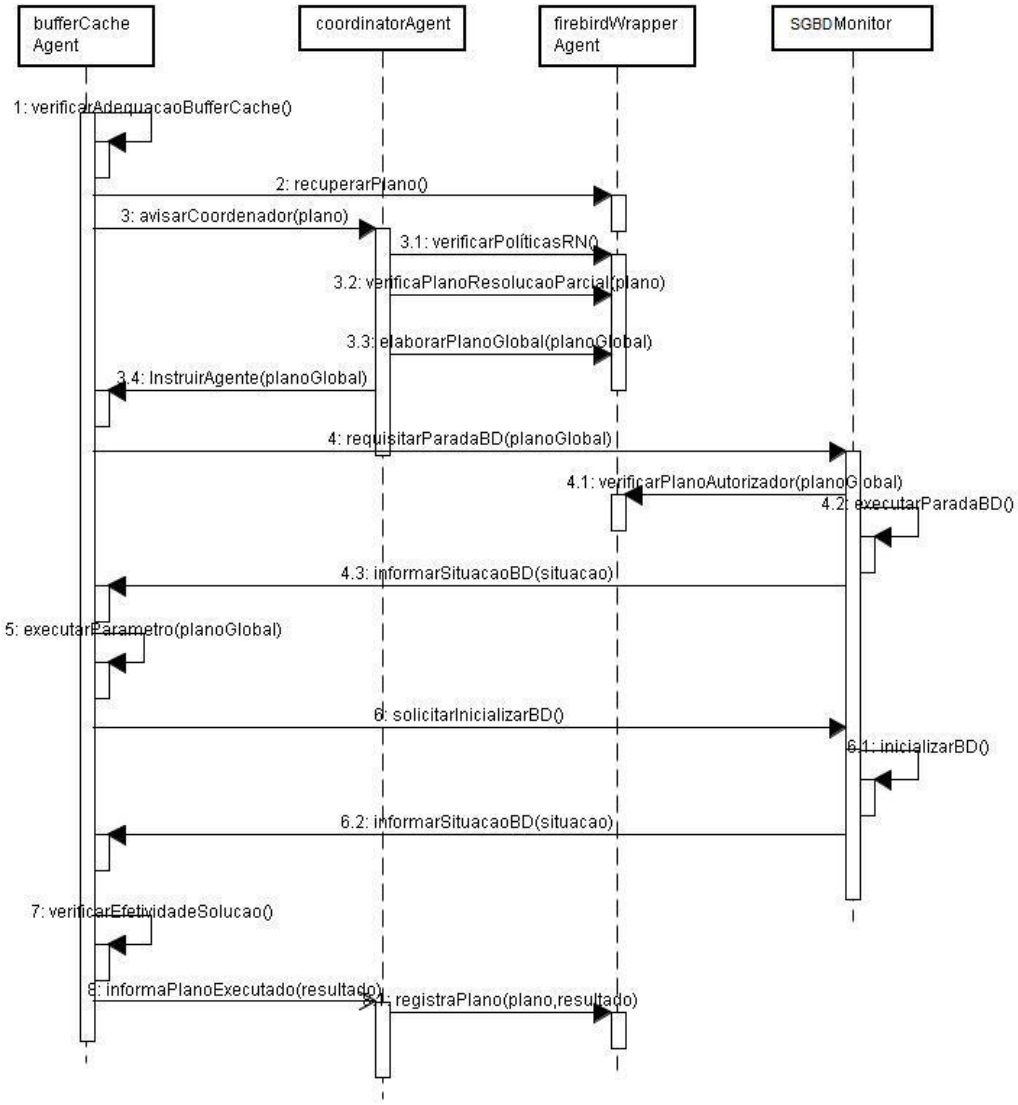

Figura 5-3: Diagrama de interação de agentes para resolução de falha em dimensionamento do buffer cache do SGBD Oracle

Complementações aos Modelos

Diagramas de classe em UML [Booch et al. 1999] podem complementar a modelagem dos agentes, uma vez que normalmente cada agente a ser implementado será instanciado em uma classe numa linguagem de desenvolvimento orientada a objetos. Um exemplo de um diagrama de classes que representa a classe abstrata *agent* e a classe derivada *AgentDBAvailable* pode ser visto na Figura 5-4. Na implementação do nosso experimento, os diagramas de classe não foram usados, visto que utilizamos um ambiente de programação para o desenvolvimento dos agentes, o JADE [JADE 2007], que auxilia a construção de classes de agentes de maneira intuitiva. O Jade será abordado nas próximas seções.

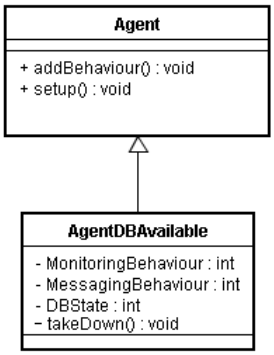

Figura 5-4: Diagrama de Classe agente AgentDBAvailable

Já as especificações de papéis, que não são explicitamente adotadas pela metodologia Tropos, mas pela metodologia Gaia [Zambonelli et al. 2003] foram utilizadas para complementar a documentação da especificação arquitetural. Essas especificações detalham os papéis com descrições em texto, os protocolos e atividades realizados, permissões e responsabilidades (*liveness* – o que o papel deve realizar para cumprir suas metas e *safety* – o que deve ser respeitado na comunidade).

As especificações de papéis estilo Gaia foram usadas como um detalhamento das capacidade levantadas pela metodologia Tropos na atividade AD.04, apresentadas na Seção 4. Um exemplo de especificação de papel pode ser acompanhado no

Quadro 5-1. A lista completa da especificação dos papéis pode ser encontrada no Apêndice 1.

Quadro 5-1: Especificação do Papel Coordenador

Papel: Coordenador DBSitter-AS (Coordinator)	
Descrição	Esse papel irá coordenar a ação dos outros papéis, elaborando planos globais de resolução de falhas e a participação e seqüência de cada papel (planos parciais). Além disso, ele se responsabilizará pelo seguimento de normas organizacionais e regras de negócio (contexto), que comporão restrições às resoluções dos planos (e.g. restrições de horário). O papel também se responsabilizará pela orientação dos papéis Notificador e Gerente de Conhecimento, os autorizando a realizar modificações nas suas respectivas bases de dados.
Protocolos e Atividades	SendInformation, ReceiveInformation, ReceiveEpisodeFeedback, // recebe feed-back dos episodios de correcao Notify, RegisterServiceDirectory, CreatePlan,
Permissões	Reads KnoledgeBase // registra casos de falha // registra episódios de correção
Responsabilidades	
Liveness	COORDINATOR = RegisterServiceDirectory (ReceiveEpisodeFeedback | CreatePlan | SendInformation | ReceiveInformation | Notify) $^{\omega}$
Safety	Ter acesso à base de conhecimento

5.3 A Implementação do Caso

Para resolução autônoma do caso de falha selecionado (descrito na Seção 5.1) e cuja resolução foi modelada na atividade **DD.03** da Seção 5.2, foi necessário o desenvolvimento de agentes concebidos para desempenharem os papéis de acordo com as especificações DBSitter-AS. O

Quadro 5-2 abaixo apresenta a matriz de Papéis x Agentes, utilizada no experimento realizado.

Quadro 5-2: Matriz Papéis x Agentes implementados no experimento

Papel	**Agente**
Coordenador DBSitter-AS	coordinatorAgent
Detector de falha em Objeto Lógico	SENMonitorBuffer
Solucionador de falha em Objeto Lógico	EXEAjustaMemoriaBuffer
Detector camada de rede	SENrede
Solucionador de falha em Estrutura de Controle	SGBDMonitor
Wrapper do Repositório de Conhecimento	FirebirdWrapperAgent
Wrapper do SGBD	oracleWrapper
Páginas Amarelas	jadeDirectFacilitator

No experimento temos os seguintes agentes participantes:

coordinatorAgent: agente responsável por ser o coordenador do *Joint-venture* implementado. É responsável pela criação dos outros agentes, pelo planejamento da resolução de falhas detectadas e pela interpretação das regras de negócio e políticas organizacionais sobre os planos a serem realizados. Regras de negócio podem ser, por exemplo, temporais, de acesso, de subordinação e sua interpretação é dependente da implementação do agente coordenador, que pode usar regras de produção, utilizar um engenho de inferência, grafos ou usar encadeamentos simples de condições para aplicá-las quando do planejamento de ações.

SENMonitorBuffer: agente responsável pela detecção de mau dimensionamento do buffer cache. É especializado em consultas ao dicionário de dados do Oracle (via solicitação de informação ao agente oracleWrapper) e interpretação dos resultados.

Obs.: No diagrama de sequência da atividade DD.03 o agente bufferCacheAgent está modelado acumulando as funcionalidades dos agentes EXEajustaMemoriabuffer e SENMonitorBuffer. Por opção de implementação, o instanciamos em dois agentes.

EXEajustaMemoriabuffer: agente que desempenha o papel de solucionador da falha e para tanto possui capacidades de manipulação do arquivo de parâmetros de inicialização do Oracle, *'init.ora'* onde, em formato texto, o parâmetro de tamanho do buffer cache é armazenado.

SENrede: agente que realiza a verificação se a camada de rede do Oracle está no ar através da verificação do processo *listener* do Oracle.

SGBDMonitor: agente capaz de realizar *startups* e *shudowns* do SGBD Oracle.

firebirdWrapperAgent: agente responsável por toda a interação com a camada de persistência da arquitetura DBSitter-AS. Realiza leituras e manipulação de dados no SGBD Firebird utilizado como repositório e base de conhecimento.

oracleWrapper: agente Wrapper para o SGBD Oracle, dotado de permissões de acesso e capacidade de realizar consultas SQL para atendimento a outros agentes.

jadeDirectFacilitator: agente que implementa o serviço de páginas amarelas, onde os agentes se registram e registram seus serviços.

Tecnologias Utilizadas

Seguindo o requisito do DBSItter-AS ser implementado em plataforma aberta (open source), optou-se por utilizar Java[17] como linguagem de programação e o framework JADE [JADE 2007] para implementação dos agentes.

O JADE é uma ferramenta específica para ser utilizada no desenvolvimento de sistemas multiagentes. Segue os padrões estabelecidos pela FIPA[18] para comunicação de agentes e é totalmente escrita em Java. Os agentes nela desenvolvidos podem ser distribuídos em máquinas com sistemas operacionais diferentes, atendendo aos requisitos do DBSitter-AS de ser totalmente independente de plataforma. O principal objetivo do JADE é simplificar o desenvolvimento de SMA, garantindo um padrão de interoperabilidade entre eles através de um grande conjunto de agentes de serviços de sistema. Um dos principais serviços utilizado pelos agentes implementados é o chamado serviço de "Páginas Amarelas" que, no JADE, é fornecido pelo agente Directory Facilitator (DF). Com ele, cada agente, inclusive o coordenador, pode cadastrar os serviços que oferece e procurar por serviços oferecidos por outros agentes. Uma vez que um agente sabe que um determinado serviço está sendo oferecido e por quem ele é fornecido, ele pode interagir com um desses agentes através da troca de mensagens padrão FIPA-ACL (Fipa - Agent Communication

[17] http://java.sun.com
[18] http://www.fipa.org

Language)[19].

Os elementos básicos de um agente implementado através do framework Jade são: os comportamentos (*behaviors*), que implementam as ações (sejam elas atômicas ou compostas) executadas pelo agente; e a fila de mensagens, que gerencia individualmente o envio e o recebimento das mensagens de cada agente. A execução de um agente é controlada por um *scheduler*, ou escalonador, que controla o agendamento da execução dos comportamentos dos agentes.

O SGBD alvo do monitoramento foi o Oracle 10g Express Edition[20]. O ambiente de execução, o Sistema Operacional Microsoft Windows Vista[21] e o ambiente de desenvolvimento Java foi o Eclipse[22] 3.2. Como repositório de conhecimento que pudesse armazenar a camada de pesistência, especificada no DBSitter-AS, escolhemos o SGBD Firebird[23] 2.0, também seguindo as orientações de ser um software gratuito. No repositório, foram catalogados os sintomas a serem monitorados e atividades de resolução exigidas pelo experimento, bem como informações sobre os agentes e regras de negócio a serem respeitadas, conforme modelo de persistência apresentado no Capítulo 4. A Figura 5-5 apresenta uma tela de consulta às tabelas do repositório, implementadas no SGBD Firebird.

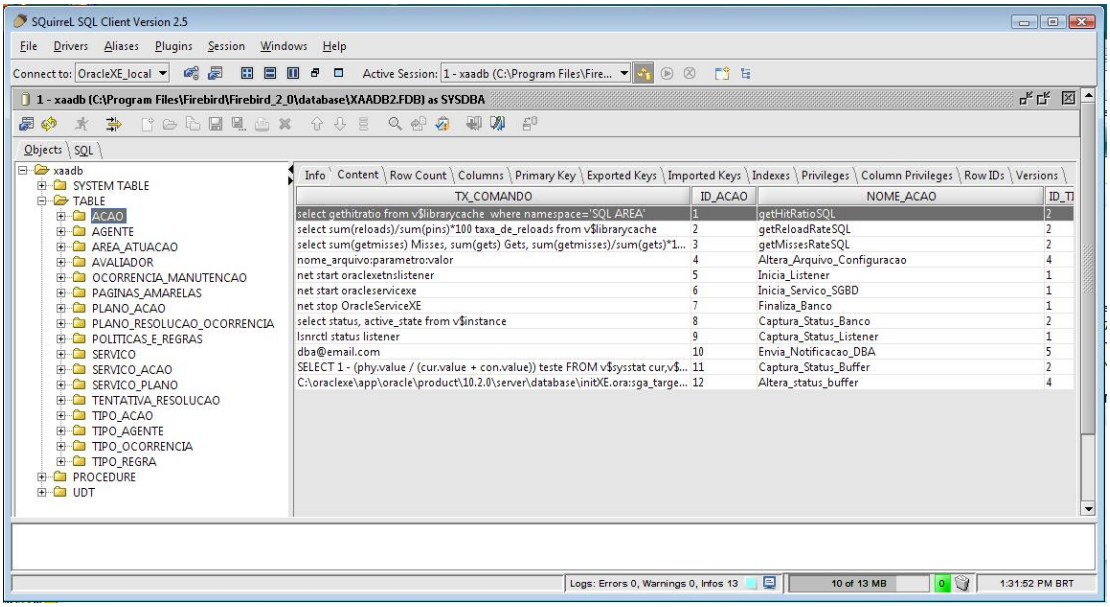

Figura 5-5: Interface de consulta apresentando tabelas do repositório, implementado no Firebird

[19] http://www.fipa.org/specs/fipa00061/
[20] http://www.oracle.com/database
[21] http://www.microsoft.com
[22] http://www.eclipse.org/
[23] http://www.firebirdsql.org

5.4 Resultados Experimentais

Nesta seção, apresentaremos os resultados obtidos com o exemplo de implementação e a adequação do caso selecionado. Consideramos um limite menor que 95% do *hit ratio* para disparar o evento de prevenção para aumento do tamanho do *buffer cache* do Oracle (para que não atinja o limite máximo de 90%).

Descrição dos Resultados

Na Figura 5-6 podemos acompanhar o *hit ratio* inicial alto. Para forçar atingir um limite do *hit ratio* menor que 95% mais rapidamente, foram submetidas consultas com grande volume de dados retornados, com execuções em paralelo.

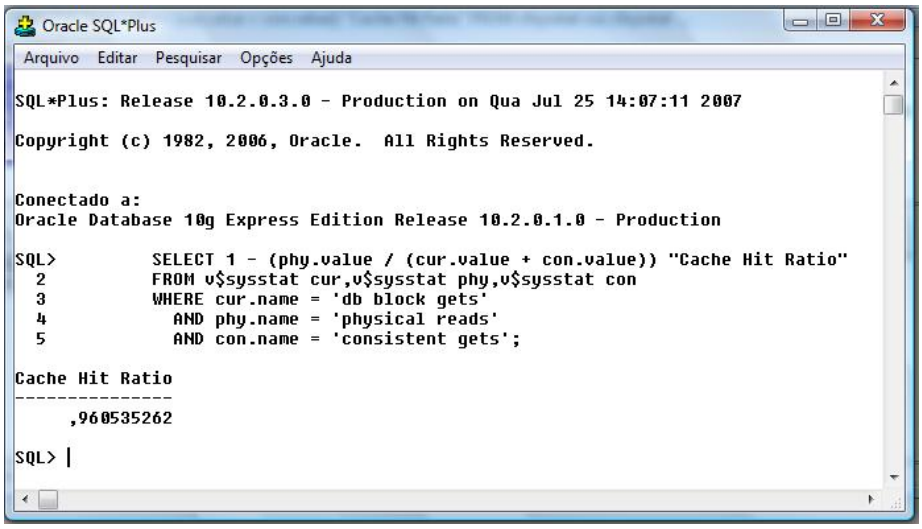

Figura 5-6: Consulta inicial do hit ratio

Em seguida, foi acionada a execução do SMA, que passou a monitorar a queda progressiva do *hit ratio,* a medida que as consultas com grandes volumes de dados eram executadas. A Figura 5-7, mostra a interface gráfica do JADE exibindo os agentes registrados no agente Páginas-amarelas.

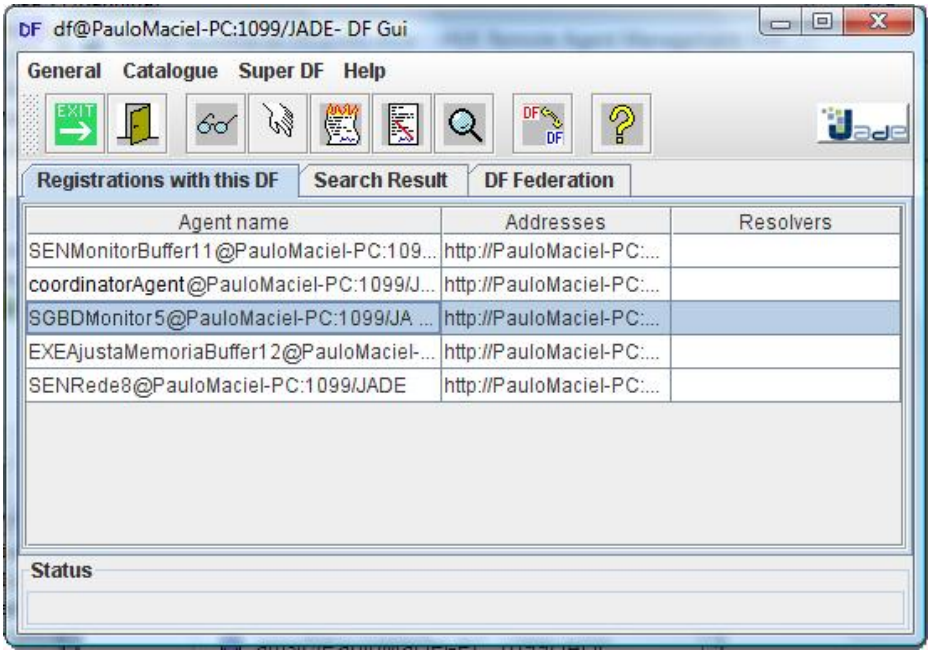

Figura 5-7: Agentes registrados no Páginas Amarelas (Interface Gráfica Jade)

O JADE possui uma interface gráfica chamada *JADE Inspector*, que rastreia todas comunicações entre os agentes. A Figura 5-8 mostra a comunicação trocada entre os agentes do SMA modelado para o experimento. Nela podemos acompanhar os agentes trocando mensagens com o agente Páginas-amarelas (direct facilitator – DF) e solicitando informações sobre qual agente disponibiliza determinado serviço. Logo após cada consulta ao DF, podemos acompanhar as solicitações de serviços e informações entre os diversos tipos de agentes. O fluxo segue o diagrama de seqüências modelado como exemplo na atividade **DD.03** da Seção 5.2.

Os resultados dos testes mostraram que o sensor do *buffer cache* (SENMonitorBuffer) identificou corretamente quando o limite estabelecido para o *hit ratio* foi atingido, e a comunicação com o agente Coordenador ocorreu com sucesso. Uma vez que o agente Coordenador recebeu a notificação, ele comunicou aos agentes sensores daquele setor do SGBD para descontinuarem o monitoramento até que o problema fosse resolvido a fim de evitar conflitos entre as ações dos agentes no ambiente (um agente sensor de banco fora do ar, por exemplo, poderia interpretar que havia uma falha no funcionamento do SGBD no momento em que o agente executor estivesse resolvendo o problema do *buffer cache*).

Em seguida, o agente coordenador requisitou os serviços do agente *wrapper* do repositório (firebirdWrapperAgent) para recuperar o *template* do plano de ações cadastrado para resolver aquele tipo de falha, o agente responsável por executá-lo, assim como as políticas e regras de negócio cadastradas (para os testes, apenas a restrição temporal citada anteriormente foi cadastrada).

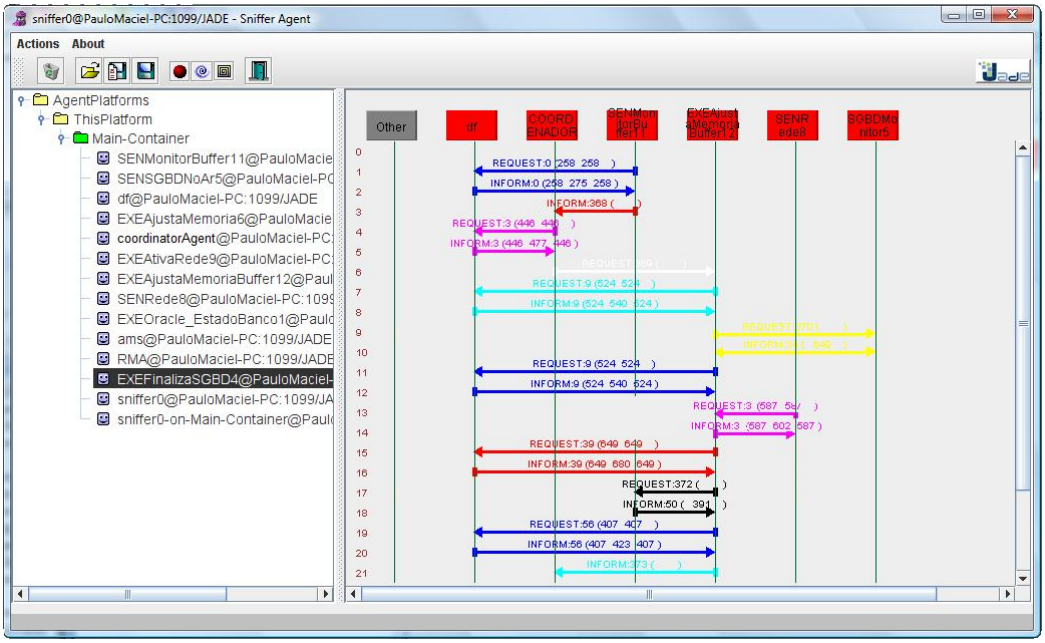

Figura 5-8: Comunicações entre os agentes, no JADE Inspector

Como o coordenador identificou que havia uma restrição temporal relacionada à ação de tirar o banco do ar, uma das ações planejadas para resolver o problema, o agente modificou o plano recuperado inicialmente, adicionando uma ação que dizia exatamente a hora que o agente executor deveria começar a executar o plano global. Em linhas gerais, o plano gerado foi:

1. Ajustar parâmetro buffer – EXEajustaMemoriaBuffer

    1.1. Retirar o SGBD do Ar - SGBDMonitor
        1.1.1. RESTRIÇÃO: INICIAR ÀS 20:01[24].

    1.2. Alterar arquivo 'init.ora' - EXEajustaMemoriaBuffer

    1.3. Reiniciar Listener – SENRede

    1.4. Iniciar serviço do Banco – SGBDMonitor

    1.5. Verificar se o *buffer cache* está ok – SENMonitorBuffer

Em seguida, o Coordenador atribuiu esse plano global ao agente responsável, o EXEajustaMemoriaBuffer, e voltou a um estado apto a receber novas notificações de falhas e previsões.

Na hora determinada, o agente EXEajustaMemoriaBuffer iniciou o processo de requisição de serviços dos agentes capazes de realizar as ações planejadas no plano a ele delegado.

[24] Para efeito de testes, o agendamento foi transformado apenas em um retardo de 30 segundos, na execução.

Ao final da execução, o arquivo de parâmetros de inicialização do SGBD Oracle, 'init.ora', foi alterado apropriadamente (neste teste, considerou-se uma alteração do parâmetro dB_block_buffers, que guia a alocação do *buffer cache*, para valor superior em 10% ao que estava alocado anteriormente) . Uma nova consulta ao agente sensor do buffer foi realizada para confirmar que o hit ratio estava em valor correto e o Coordenador foi notificado do sucesso da operação. Em seguida, o agente coordenador cadastrou os resultados da operação nas tabelas apropriadas no Repositório DBSitter-AS e voltou ao estado em que recebe novas notificações dos agentes.

Para esse experimento foi considerado que as requisições de serviços nunca são maliciosas e que os agentes podem atender aos pedidos de outros agentes sem verificar se eles têm permissão para pedi-los.

5.5 Conclusão

Neste capítulo apresentamos a fase de Projeto Detalhado e um exemplo simples de implementação de um SMA seguindo o *Framework* DBSitter-AS. Ambos foram guiados pelas atividades de resolução de um caso de falha de dimensionamento de memória para o SGBD Oracle.

O caso selecionado (mau dimensionamento de áreas de memória, no caso o *buffer cache* do SGBD Oracle) configura-se como inerente a todos os SGBD e é uma preocupação real dos ABD. Do ponto de vista do SMA desenvolvido, procurou-se um caso de falha cuja resolução envolvesse a participação obrigatória de agentes de vários tipos e que forçasse a comunicação e solicitação de serviços complementares entre eles. Tivemos a participação, além dos agentes coordenador, páginas-amarelas e *wrappers* do repositório e do SGBD alvo, dois agente da Suborganização de Gerenciamento de Estruturas de Controle (EXEajustaMemoriaBuffer e SGBDmonitor), um agente da Suborganização de Gerenciamento de Estrutruturas Lógicas (SENmemoriaBuffer) e um agente da Suborganização de Gerenciamento de Rede (SENrede). Com isso, embora a atuação seja resolvida de forma simples tem uma boa abrangência dentro do SMA.

Buscou-se também a incidência de uma política organizacional aplicada ao caso: não ser possível a retirada do ar do SGBD durante horário comercial, que aplicada ao plano de resolução (pelo agente coordenador) induzisse o agendamento da atuação para agente solucionador. Todas as comunicações entre os agentes seguiram o padrão das especificações FIPA-ACL para comunicação.

No próximo capítulo, apresentaremos as conclusões desta dissertação e abordaremos as contribuições oferecidas pelo *Framework* DBSitter-AS e possibilidades de trabalhos futuros que implementem ou estendam as especificações atuais.

6 Conclusões e Trabalhos Futuros

Este capítulo apresenta uma síntese do trabalho desenvolvido e apresenta as principais contribuições desta dissertação. Também relatamos as dificuldades encontradas e as possibilidades de pesquisa que estendam o modelo proposto.

O objetivo principal da dissertação foi propor um *framework* que sirva de especificação arquitetural para a construção de componentes que forneçam autonomia de gerenciamento para SGBD convencionais. Inicialmente, mostramos um panorama da área de Computação Autônoma, seus princípios, histórico e evolução até as pesquisas mais recentes. Esses trabalhos buscam o desenvolvimento de características de autogerenciamento para os *software* desenvolvidos. Detalhamos algumas pesquisas específicas sobre aplicação da Computação Autônoma aplicada a SGBD nos meios acadêmicos e as evoluções dos SGBD comerciais.

Apresentamos os fundamentos teóricos do paradigma da Orientação a Agentes, dos Sistemas Multiagentes e Metodologias de Desenvolvimento SMA e discutimos as correlações entre Computação Autônoma e SMA.

Com o intuito de criar uma especificação arquitetural inspirada nos trabalhos do Sistema DBSitter [Carneiro et al. 2004], propusemos no Capítulo 4, o *Framework* DBSitter-AS, modelado formalmente como um SMA. O *framework* proposto procura estar alinhado ao máximo aos princípios da Computação Autônoma e seguir os requisitos originais do Projeto DBSitter.

Para demonstrar a efetividade do *framework* como uma especificação capaz de guiar o desenvolvimento de um SMA que gerencie de forma autônoma um SGBD de mercado, implementamos um experimento pequeno, embora representativo de SMA que contém agentes atuantes camada de rede, em algumas estruturas de controle e solucionador de problemas na memória *buffer* do SGBD ORACE. Posteriormente, apresentamos os resultados conseguidos através de uma implementação que usou o Framework JADE para desenvolvimento SMA.

6.1 Contribuições

A principal contribuição apresentada por esta dissertação foi:

- A criação da especificação arquitetural DBSitter-AS, um modelo de uma sociedade de agentes que detectam, previnem e corrigem falhas de SGBD, de forma autônoma; Esta especificação contempla:

- Definição de papéis e funcionalidades a serem desempenhados por cada agente e a forma de comunicação entre eles;
- Definição da forma de coordenação dos agentes;
- Utilização de especificação arquitetural baseada em estilo organizacional *joint-venture* e aplicação de padrões sociais aos agentes, o que permite a criação de padrões para resolução de falhas em SGBD;
- Estabelecimento de um modelo de persistência de informações que permite maior robustez à arquitetura;

As seguintes contribuições também foram apresentadas:

- Proposta de uma especificação que usa tecnologias de código aberto e são multiplataforma, além de poder ser aplicada a qualquer SGBD.
- A estruturação do raciocínio sobre os problemas encontrados em SGBD em uma visão organizacional, com Suborganização de Gerenciamento de SGBD subdividida em Objetos Físicos, Lógicos e Estruturas de Controle e Suboganizações de Gerenciamento de Rede e Gerenciamento de Sistema Operacional.
- Extensão da primeira especificação do DBSitter para considerar padronização FIPA para comunicação dos agentes.
- Inovação na consideração de Políticas e Regras Organizacionais aplicadas ao gerenciamento autônomo de SGBD.
- Exemplificação do desenvolvimento de um SMA DBSitter-AS.

6.2 Limitações e Trabalhos Futuros

A criação de um *framework* de especificações arquiteturais por si só configura-se em uma tarefa grande e trabalhosa. Mais ainda aplicada ao complexo domínio do gerenciamento autônomo de SGBD, onde a cada ação correspondem reações e efeitos colaterais. Minúcias e situações específicas devem ser tratadas nas etapas de especificação da implementação, sendo o DBSitter-AS um guia de nível mais alto para a compreensão do problema e consideração dos componentes envolvidos.

Assim sendo, complementações e detalhamentos das especificações dos componentes do DBSitter-AS são possíveis e consideradas extensões às especificações arquiteturais, dentre as quais podemos destacar:

- Especificação e testes de formas de raciocínio que possibilitem ao agente Coordenador gerenciar efeitos colaterais, realizar análises de impacto,

gerenciar concorrência de atividades e replanejamentos de atividades de correção e prevenção de falhas;

- Redefinição do mecanismo de aprendizagem para utilizar outras formas de inferência, além do *feedback* do usuário. A utilização de estruturas auxiliares como motores de inferência ou ontologias para classificação de erros e sintomas são aderentes às especificações atuais.
- Especificação de um modelo para armazenar, classificar e interpretar as regras de negócio e políticas organizacionais. Isso estenderia a forma incipiente baseada em classificações de restrições temporais ou de acesso utilizada no nosso experimento.
- Especificação de um *Datawarehouse* para *datamining* de informações sobre desempenho da ferramenta (com elaboração de critérios de qualidade e indicadores de desempenho). Isso permitiria uma avaliação constante do sistema autônomo e análises comparativas com realizações humanas ou de outras ferramentas.
- Criação de "pacotes" de regras de resolução de falhas por SGBD e área de atuação (e.g. pacote de casos de resolução de falhas em objetos lógicos para o SGBD SQL Server);
- Implementação de novos experimentos e protótipos de componentes para automação de gerenciamento de SGBD;

6.3 Considerações Finais

A despeito dos recentes avanços, o surgimento de um SGBD realmente autônomo ainda necessitará de grande esforço de pesquisa e que mesmo o advento do SAGBD não significará a extinção das funções de Administrador de Dados ou Administrador de Banco de Dados.

O surgimento de SGBD cada vez mais autônomos significará a liberação dos ABD das tarefas rotineiras e repetitivas, para a ocupação desses profissionais com atividades de caráter mais estratégico, como funções de definição de políticas, integração global do tratamento de dados e informações, e definição de processos organizacionais.

Embora os desafios que levam ao SAGBD sejam enormes, percebemos que este é o caminho por onde os SGBD estão seguindo. Soluções proprietárias dos grandes fabricantes de SGBD aparecerão em maior número, a cada versão lançada no mercado.

Nesse contexto, aumenta a importância de soluções multiplataforma, de código-aberto e gratuitas para as pessoas que vão administrar os SGBD e para

empresas, interessadas num custo menor. Foi buscando propiciar um pequeno passo em direção de se atingir essa melhor qualidade de trabalho que o DBSitter-AS foi desenvolvido.

Referências Bibliográficas

[Autoadmin 2007] "AutoAdmin: Self-Tuning and Self-Administering Databases". Disponível em http://www.research.microsoft.com/dmx/autoadmin/. Último acesso em 7/02/2007.

[Autonomic Computing 2007] Disponível em http://www.autonomiccomputing.org/. Último acesso em 7/02/2007.

[Bahati et al. 2006] Bahati R., Bauer M., Vieira E. e Baek O. (2006). "Using Policies to Drive Autonomic Management". Em International Workshop on Wireless, Mobile, Multimedia Proceedings of the 2006 International Symposium on World of Wireless, Mobile and Multimedia Networks.

[Bauer et al. 2001] Bauer B., Muller J. P. e Odell J. (2001). "Agent UML: A Formalism for Specifying Multiagent Interaction". Em Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge
eds., Springer, Berlin, pp. 91-103, 2001.

[Birman et al. 2003] Birman K.P., van Renesse R., Vogels W. (2003). "Navigating in the storm: using astrolabe for distributed self-configuration, monitoring and adaptation". Em: Proceedings of the autonomic computing workshop, 5th international workshop on active middleware services (AMS 2003), Seattle. pp 4–13

[Bratman et al. 1987] Bratman M. (1987). "Intention, plans and practical reason". Ed. Harvard Press, 1987.

[Bresciani et al. 2004] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. e Perini, A. (2004). "Tropos: An Agent-Oriented Software Development Methodology". Em Autonomous Agents and Multi-Agent Systems, 8(3), pp 203–236.

[Booch et al. 1999] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language: User Guide. Addison-Wesley, 1999.

[Busetta et al. 2004] Busetta P., Rönquist R., Hodgson A. e Lucas A. (1998). "JACK Intelligent Agents – Components for Intelligent Agents in Java". Relatório técnico, Agent Oriented Software Pty. Ltd, Melbourne, Austrália.

[Carneiro et al. 2004] Carneiro A., Passos R., Belian R., Costa T., Tedesco P. e Salgado A. C. (2004). "DBSitter: An Intelligent Tool for Database Administration". Em DEXA – 15th International Conference and Workshop on Database and Expert Systems Applications.

[Castro et al. 2002] Castro, J., Kolp, M. e Mylopoulos, J. (2002) "Towards Requirements-Driven Information Systems Engineering: The Tropos Project". Em Information Systems News, Elsevier, vol. 27, pp. 365-89.

[Chess et al. 2004] Chess D., Segal A., Whalley I. e White S. (2004)."Unity: experiences with a prototype autonomic computing system. "Proceedings of IEEE first international conference on autonomic computing, New York May 2004. p. 140–7.

[Crawford e Dan 2003] Crawford, C.H. Dan, A. (2002). "eModel: addressing the need for a flexible modeling framework in autonomic computing". Em Modeling, Analysis and Simulation of Computer and Telecommunications Systems. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on Publication,
p.p. 203- 208

[Costa et al. 2003] Costa, R., Lifschitz, S e Salles, M. (2003). "Index Self-Tuning with Agent-based Databases". Em CLEI Electronic Journal, 6(1). p.p.22.

[Constantinescu 2003] Constantinescu, Z. (2003). "Towards an Autonomic Distributed Computing System". Em Proceedings of the 14th International Workhop on Database and Expert Systems Applications (DEXA 'OS).

[Cysneiros 2001] Cysneiros, L., (2001). "Requisitos não funcionais: Da Elicitação ao Modelo Conceitual". Dissertação de Doutorado – PUC Rio, fev. 2001.

[DeLoach et al. 2001] DeLoach, S., Wood, M. e Sparkman C., (2001). "Multiagent Systems Engineering". Em International Journal of Software Engeneering and Knowledge Engenieering, Vol 11, No.3, pp.231-258.

[Dong et al. 2003] Dong X. Hariri, S., Xue L., Chen H., Zhang M., Pavuluri, S. Rao, S. (2003). "Autonomia: an autonomic computing environment". Em Performance, Computing, and Communications Conference. Conference Proceedings of the 2003 IEEE, p.p. 61- 68

[Elnaffar et al. 2003] Elnaffar, S., Powley W., Benoit D. e Martin P. (2003). "Today's DBMSs: How autonomic are they?" Em DEXA - 14th International Workshop on Database and Expert Systems Applications.

[Ferber 1999] Ferber, J. (1999). "Multi-Agent Systems – An Introduction to Distributed Artificial Intelligence". Editora Addison-Wesley.

[Fisher et al. 1997] Fisher, M.; Muller, J.; Schroeder, M.; Staniford, G.; and Wagner, G. (1997). "Methodological foundations for agentbased systems" Em Knowledge Engineering Review 12(3):323—329.

[Franklin e Graesser 1996] Franklin S. e Graesser. A. (1996). "Is it an agent or just a program? A Taxonomy for Autonomous Agents". Em Institute for Intelligent Systems, University of Memphis - Proceedings of the Third International Workshop on Agent Theories Architectures, and Languages, Springer-Verlag, 1996.

[Fuad e Oudshoorn 2006] Fuad, M. M., Oudshoorn M. J. (2006). "An Autonomic Architecture for Legacy Systems". Em Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems (EASE'06).

[Fuxman et al. 2001] Fuxman, A., Giorgini, P., Kolp, M. e Mylopoulos, J. (2001). "Information systems as social structures". Em Proc. of the 2nd Int. Conf. on Formal Ontologies for Information Systems, FOIS'01, Ogunquit, USA.

[Ganek 2003] Ganek, A. G. (2003). "Autonomic Computing: Implementing the Vision". Em Keynote presentation at the autonomic computing workshop, (AMS 2003), Seattle, WA.

[Ganek e Corbi 2003] Ganek, A.G. e Corbi, T.A. (2003). "The dawning of the autonomic computing era". Em IBM Systems Journal, Vol. 42, No. 1.

[Horn 2001] Horn, Paul. (2001). "Autonomic Computing: IBM's Perspective on The State of Information Technology" - IBM Corporation. Disponível em: http://www.research.ibm.com/autonomic/. Último acesso em 7/02/2007.

[Hübner e Sichman 2003] Hübner F. Jomi e Sichman J. S. (2003), "Organização de Sistemas Multiagentes". Em In Renata Vieira, Fernando Osório, and Solange Rezende, editors, III Jornada de Mini-Cursos de Inteligência Artificial (JAIA'03), volume 8, p.p. 247-296. SBC, Campinas, 2003.

[Huhns e Stephens 1999] Huhns M. e Stephens L. (1999). "Multiagent Systems: A modern approach to distributed artificial intelligence". Ed. Weiss, G.,MIT Press. Cap. 2.

[Hyden et al. 1999] Hayden, S., Carrick, C. e Yang, Q. (1999) "Architectural design patterns for multiagent coordination", Em Proceedings of the 3rd International Conference on Autonomous Agents, Agents'99, Seattle, USA.

[IBM Autonomic Blueprint 2006] "An architectural blueprint for autonomic computing" Quarta Edição (2006), Disponível em http://www-03.ibm.com/autonomic/pdfs

[IBM Autonomic Toolkit 2007] IBM Autonomic Computing Toolkik. Disponível em: http://www.ibm.com/developerworks/autonomic/overview.html. Último acesso em 07/02/2007.

[Jennings 1996] Jennings N. R. (1996). "Coordination Techniques for Distributed Artificial Intelligence". Em Foundations of Distributed Artificial Intelligence. O'HARE, Greg; JENNINGS, Nicholas (Eds.): John Wiley and Sons, 1996. cap.6.

[Jennings 2000] Jennings, N. R. (2000), On agent-based software engineering. Em Artificial Intelligence, v.117 n.2, p.277-296, Março 2000.

[JADE 2007] Java Agent DEvelopment Framework. Disponível em: http://jade.cselt.it. Último acesso em 07/02/2007.

[Khargharia et al. 2003] Khargharia B., Hariri S., Parashar M., Ntaimo L., e Kim B. U. (2003). "vGrid: A framework for building autonomic applications" . Em the Int. Workshop Heterogeneous and Adaptive Computing—CLADE 2003 Seattle, WA, 2003.

[Kephart 2002] J. Kephart (2002). "Software agents and the information economy". Em Proceedings of the National Academy of Sciences U.S.A., Volume 99, Supplement 3, 7207-7213, Maio, 2002.

[Kephart e Chess 2003] Kephart, Jeffrey O. e Chess, David (2003). "The Vision of Autonomic Computing", IEEE Computer Society, Volume 36, Issue 1, Jan 2003 pp 41-50.

[Koehler et al. 2003] Koehler, J., Giblin, C., Gantenbein, D. e Hauser R. (2003). "On Autonomic Computing Architectures". Em Research Report (Computer Science) RZ 3487 (#99302), IBM Research (Zurich).

[Kolp e Mylopoulos 2001] Kolp, M. e Mylopoulos, J. (2001). "Architectural Styles for Information Systems: An Organizational Perspective". Em the 13th Conference on Advanced Information Systems Engineering (CAiSE'01), Interlaken, Switzerland, 2001.

[Kolp et al. 2002] Kolp, M., Giorgini J. e Mylopoulos (2002). "Organizational Patterns for Early Requirements Analysis". Em the 1EEE Joint International Requirements Engineering Conference 2002, RE 2002, Agosto 2002.

[Kumar 2004] Kumar, M. (2004). "Oracle Database 10g: The Self-Managing Database"; Oracle White Paper #40090; Oracle Corporation 2004.

[Lifschitz e Macêdo 2004] Lifschitz S., Macêdo J. A. F., (2004). "Agent-based Databases and Parallel Join Load Balancing". Em: Sistemas de Información e Ingeniería de Software, cap. 6, pp. 69 a 88. Temas Selectos – Ed. Centro Estudios Informatica, Univ. Simon Bolivar (Merida) Venezuela, 2004.

[Lifschitz et al. 2004] Lifschitz S., Milanés A. e Salles M. (2004). "Estado da Arte em Auto-sintonia de SGBD Relacionais". Em Relatório técnico, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio).

[Lightstone et al. 2003] Lightstone S., Schiefer B., Zilio D. e Kleeewein J. (2003) "Autonomic Computing for Relational Databases: The Ten-Year Vision". IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA' 2003), Banff AB.

[Lohman e Lightstone 2002] Lohman G.M e Lightstone S.S. (2002). "SMART: Making DB2 (More) Autonomic." Em 28th VLDB Conference – Very Large Database Endowment.

[Lomet et al.] Lomet, D., Barga R., Chaudhuri S., Larson P. e Narasayya V.: "The Microsoft Database Research Group". Disponível em http://www.research.microsoft.com/reasearch/db

[Menon et al. 2003] Menon J., Pease D., Rees R., Duyanovich L e Hillsberg B. (2003) "IBM storage tank - A heterogeneous scalable SAN file system". IBM Syst Journal; 42(2):250–67.

[Microsoft SQL Server] Microsoft SQL Server. Disponível em www.microsoft.com/sql. Último acesso em 07/02/2007.

[Milanés 2004] Milanés, A. (2004). "Uma arquitetura para auto-sintonia global de SGBDs usando agentes". Dissertação (Mestrado) - Pontíficia Universidade Católica do Rio de Janeiro, Departamento de Informática (PUC-Rio).

[Milanés e Lifschitz 2005] Milanés, A. y., Lifschitz S. (2005). "Design and Implementation of a Global Self-tuning Architecture". Em 20º. Simpósio Brasileiro de Bancos de Dados – SBBD, 2005.

[Océano 2007] Océano IBM Research IBM Corp. The Océano Project. Disponível em http://www.research.ibm.com/oceanoproject/. Último acesso em 7/02/2007.

[Oceanstore 2007] "The OceanStore Project", project overview. Berkeley UC. Computer science division. Disponível em http://oceanstore.cs.berkeley.edu/. Último acesso em 07/02/2007.

[Odell et al. 2004] Odell, J., Parunak, H. V. D., Bauer, B. (2000). "Extending UML for agents". Em AOIS Workshop at AAAI, pp. 3–17, 2000.

[Ogeer 2004] Ogeer, Nyla (2004). "Multiple Buffer Pools and Dynamic Resizing of Buffer Pools in PostgreSQL". Queen's University Kingston, Ontario, Canadá. Tese de Mestrado em Ciência da Computação.

[Padgham e Winikoff 2002] Padgham, L. e Winikoff, M. (2002) "Prometheus: A Methodology for Developing Intelligent Agents". Em Proceedings of the First International Conference on Autonomous Agents and Multi-Agent Systems – AAMAS'02, Third International Workshop on Agent-Oriented Softaware Engineering AOSE-2002. pp.135-146.

[Parashar 2005] Parashar, M. e Hariri S. (2005) "Autonomic Computing: An Overview". Em Hot Topics, Lecture Notes in Computer Science 3566.

[Poellabauer 2002] Poellabauer C. (2002). "Q-fabric—system support for continuous online Qualitymanagement". Disponível em http://www.cc.gatech.edu/systems/projects/ELinux/qfabric.html. Último acesso em 07/02/2007.

[PostgreSql 2007] PostgreSQL. Disponível em: http://www.postgresql.org/. Último acesso em 7/02/2007.

[Pool 2002] Pool. R. (2002). "Natural selection: A New Computer Program Classifies Documents Automatically". Disponível em http://domino.watson.ibm.com/comm/wwwr_thinkresearch.nsf/pages/selection200.html. Último acesso em 07/02/2007.

[Ramanujam e Capretz 2005] S. Ramanujam and M. A. M. Capretz (2005). "ADAM: A Multi-Agent System for Autonomous Database Administration and Maintenance", International Journal of Intelligent Information Technologies, vol. 1, No. 3, pp.14-33, Jul.-Set. 2005.

[Russell e Norvig 2003] Russell, S. J. e Norvig, P. (2002). "Artificial Intelligence: A Modern Approach", 2nd edition. Prentice Hall.

[Salles 2004] Salles, M. (2004). "Criação Autônoma de Índices em Bancos de Dados". Pontifícia Universidade Católica do Rio de Janeiro. Tese de Mestrado em Ciência da Computação.

[Silva et al. 2007] Silva, M. J., Maciel P. R., Pinto, R. C., Alencar, F., Tedesco, P. and Castro, J. (2007) "Extracting the Best Features of Two Tropos Approaches for the Efficient Design of MAS". In: IDEAS'07 – Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software 2007.

[Stillger et al. 2001] Stillger M., Lohman G., Markl V. e Kandil M. (2001). "LEO - DB2's LEarning Optimizer", Em Proceedings. 27th Intl. Conf. On Very Large Databases, Rome, Italy, pp. 19-28.

[Tennenhouse 2000] Tennenhouse, D. (2000). "Proactive computing. Em "Communications of the ACM", 43, 5, pp. 43–50.

[TPC] TPC – Transaction Processing Performance Council. Disponível em www.tpc.org. Último acesso em 07/02/2007.

[Vlassis 2003] Vlassis, Nikos (2003). "A Concise Introduction to Multiagent Systems and Distributed AI". Em University of Amsterdam Lecture Notes, 2003. Disponível em: http://www.science.uva.nl/~vlassis/cimasdai/

[Wagner 2003] Wagner, G. (2003). "The agent-object-relationship metamodel: towards a unified view of state and behavior". Em Information Systems, v.28 n.5, pp.475-504, Julho de 2003.

[Want et al. 2003] Want R., Pering T. e Tennenhouse D. (2003). Em "Comparing
Autonomic and Proactive Computing", IBM Systems Journal,
vol. 42, no. 1, pp. 129-135.

[Web Services 2007] Web Services Activity: Disponível em www.w3.org/2002/ws/. Último acesso em 07/02/2007.

[Weiss 1999] Weiss, G. (1999). "Multiagent Systems: A modern approach to distributed artificial intelligence". Editora MIT Press. Prólogo.

[White et al. 2004] White, S.R., Hanson, J.E., Whalley, I., Chess, D.M. e Kephart, J.O. (2004). "An architectural approach to autonomic computing". Em Proceedings of the International Conference on Autonomic Computing (ICAC).

[Wood e DeLoach 2000] Wood, M. e DeLoach, S.A. (2000). "An Overview of the Multiagent Systems Engineering Methodology". Em Proceedings of the First International Workshop (AOSE-2000). Springer-Verlag, Berlin, pp.207-221.

[Wooldridge 2001] Wooldridge. M. (2001). "An Introduction to Multiagent Systems", Editora Wiley.

[Yu 1995] Yu E. (1995). "Modeling Strategic Relationships for Process Reengineering". Tese de doutorado, University of Toronto, Department of Computer Science.

[Zambonelli et al. 2000] Zambonelli, F. Jennings, N. R., Wooldridge, M. (2000). "Organizational Abstractions for the Analysis and Design of Multi-Agent Systems". Em Proceedings Of The 1st International Workshop On Agent-Oriented Software Engineering, 2000, Limerick, Ireland, Anais 2000. p.p. 127–141.

[Zambonelli et al. 2003] Zambonelli, F., Wooldridge, M. e Jennings, N. R., (2003). "Developing Multi-Agent Systems: The Gaia Methodology". Em ACM Transaction on Software Engineering and Methodology, vol. 12, No. 3, pp.417-470.

[Zilio et al. 2000] Zilio D., Lightstone S. e Lyons K. (2000) "Self-managing Technology in IBM DB2 Universal Database". Em Proceedings Of 2001 CIKM, 2000, pp. 541-543.

Apêndice 1 – Especificação de Papéis

Monitor de estruturas de controle de SGBD

Papel: Detector de Falha em Estrutura de Controle **(**MonitorControlDB)	
Descrição	Esse papel irá: - monitorar constantemente os estados das estruturas e controle do banco de dados alvo, monitorando processos do SGBD no S.O., em arquivos de logs (traces) de SGBD ou realizando consultas em dicionários de dados. Caso solicitado, ele irá se comunicar com outros papéis que requisitem informações, passando dados sobre o estado atual da estruturas monitoradas do banco. - executar ações que visam recuperar o estado das estruturas de controle do banco de dados e ações de correção de falhas. - notificar papel coordenador sobre falhas, ações e resultados.
Protocolos e Atividades	ReceiveInfRequest, ProccessInfRequest, AnswerInfRequest, QueryKB // consulta base de conhecimento Notify, AskForInformation, ReceiveInformation, HealingDB, MonitoringDB, registerServiceDirectory
Permissões	Reads DBState *// O estado do banco de dados* Reads OSState *// O estado dos processo do SO* Reads KnowlodgeBase // lê estratégias de correção de problemas Changes DBState *// Realiza operações no SGBD*
Responsabilidades	
Liveness	MONITORCONTROLDB = registerServiceDirectory.((ReceiveInfRequest . ProccessInfRequest . AnswerInfRequest) | (MonitoringDB) || (HealingDB) | (AskForInformation) | (ReceiveInformation) | QueryKB | Notify) $^{\omega}$
Safety	Ter acesso completo ao servidor, incluindo aos processos que estão sendo executados pelo SO, processos do SGBD e ao SGBD alvo. Ter acesso à base local de conhecimento.

Monitor de estruturas físicas de SGBD

Papel: Detector de Falha em Objeto Físico (MonitorPhysicalDB)	
Descrição	Esse papel irá: - monitorar constantemente os estados das estruturas físicas do banco de dados alvo em arquivos de logs (traces) de SGBD, realizando consultas em dicionários de dados e verificando estados de estruturas de memória em disco (via S.O.). Caso solicitado, ele irá se comunicar com outros papéis que requisitem informações, passando dados sobre o estado atual do banco. - Esse papel irá executar ações que visam recuperar o estado do banco de dados e ações de correção de falhas. Eventualmente, também poderá requisitar informações de outros agentes. - notificar papel coordenador sobre falhas, ações e resultados.
Protocolos e Atividades	ReceiveInfRequest, ProccessInfRequest, AnswerInfRequest, AskForInformation, ReceiveInformation, QueryKB // consulta base de conhecimento Notify, HealingDB, MonitoringDB, registerServiceDirectory
Permissões	Reads DBState *// O estado do banco de dados* Reads OSState *// O estado dos processo e áreas do SO* Reads KnowlodgeBase // lê estratégias de correção de problemas Changes DBState *// Realiza operações no SGBD*
Responsabilidades	
Liveness	MONITORPHYSICALDB = registerServiceDirectory.((ReceiveInfRequest . ProccessInfRequest . AnswerInfRequest) | (MonitoringDB) | | (HealingDB) | (AskForInformation) | (ReceiveInformation) | QueryKB | Notify) $^{\omega}$
Safety	Ter acesso completo ao servidor, incluindo aos processos que estão sendo executados pelo SO, áreas de memória em disco e ao SGBD alvo Ter acesso à base local de conhecimento.

Monitor de estruturas lógicas de SGBD

Papel: Detector de Falha em Objeto Lógico (MonitorLogicalDB)	
Descrição	Esse papel preliminar irá: - monitorar constantemente os estados das estruturas lógicas do banco de dados alvo em arquivos de logs (traces) de SGBD ou realizando consultas em dicionários de dados. Caso solicitado, ele irá se comunicar com outros papéis que requisitem informações, passando dados sobre o estado atual do banco. Eventualmente, também poderá requisitar informações de outros agentes. - executar ações que visam recuperar o estado do banco de dados e ações de correção de falhas. Eventualmente, também poderá requisitar informações de outros agentes. - notificar papel coordenador sobre falhas, ações e resultados.
Protocolos e Atividades	ReceiveInfRequest, ProccessInfRequest, AnswerInfRequest, AskForInformation, ReceiveInformation, QueryKB // consulta base de conhecimento Notify, HealingDB, MonitoringDB, registerServiceDirectory
Permissões	Reads DBState *// O estado do banco de dados* Reads OSState *// O estado dos processos e áreas do SO* Reads KnowledgeBase // lê estratégias de correção de problemas Changes DBState *// Realiza operações no SGBD*
Responsabilidades	
Liveness	MONITORLOGICALDB = registerServiceDirectory.((ReceiveInfRequest . ProccessInfRequest . AnswerInfRequest) | (MonitoringDB) | HealingDB) | (AskForInformation) | (ReceiveInformation) | QueryKB |Notify) $^{\omega}$
Safety	Ter acesso completo ao servidor, incluindo aos processos que estão sendo executados pelo SO, áreas de memória e dicionário de dados do SGBD alvo. Ter acesso à base local de conhecimento

Monitor de SO

Papel: Detector de Falha em SO (MonitorSO)	
Descrição	Esse papel preliminar irá: - monitorar os estados do Sistema Operacional, através da aquisição de informações em variáveis de ambiente, processos e áreas de armazenamento. Irá se comunicar com papéis que requisitem informações sobre os estados monitorados. - executar ações que visam recuperar o estado de elementos do S.O. - notificar papel coordenador sobre falhas, ações e resultados.
Protocolos e Atividades	ReceiveInfRequest, ProccessInfRequest, AnswerInfRequest, AskForInformation, ReceiveInformation, QueryKB // consulta base de conhecimento Notify, HealingOS, MonitoringOS, registerServiceDirectory
Permissões	Reads OSState *// estado do S.O.* Changes OSState *// operações de alteração de elementos do S.O.* Reads KnowledgeBase // lê estratégias de correção de problemas.
Responsabilidades	
Liveness	MONITOROS = registerServiceDirectory ((ReceiveRequest . ProccessInfRequest .AnswerRequest) | (MonitoringSO) | (HealingSO) | |(AskForInformation) | (ReceiveInformation)) | QueryKB | Notify)$^{\omega}$
Safety	Ter acesso completo aos recursos do sistema operacional

Monitor de rede

Papel: Detector de Falha de Conectividade (MonitorNetwork)	
Descrição	Esse papel preliminar irá: - monitorar os estados da rede de comunicação, através da aquisição de informações em variáveis de ambiente, processos, portas etc. Irá se comunicar com papéis que requisitem informações sobre os estados do monitorados. - executar ações que visam recuperar o estado de elementos da rede de comunicação. - notificar papel coordenador sobre falhas, ações e resultados.
Protocolos e Atividades	ReceiveInfRequest, ProccessInfRequest, AnswerInfRequest, QueryKB // consulta base de conhecimento Notify, AskForInformation, ReceiveInformation, HealingNetwork, MonitoringNetwork, registerServiceDirectory
Permissões	Reads OSState *// estado do S.O.* Changes OSState *// operações de alteração de elementos do S.O.* Reads KnowlodgeBase // lê estratégias de correção de problemas
Responsabilidades	
Liveness	MONITORNETWORK = registerServiceDirectory ((ReceiveRequest . ProccessInfRequest . AnswerRequest) | (MonitoringNetwork) | (HealingNetwork) | (AskForInformation) | (ReceiveInformation) | QueryKB | Notify)$^{\omega}$
Safety	Ter acesso completo aos recursos do sistema operacional

Notificador

Papel: Wrapper Administrador de Comunicação (Notifier)	
Descrição	Esse papel irá executar ações de interface entre o sistema e os usuários finais. Informações coletadas de outros papéis serão registradas em base de dados apropriada e/ou enviadas para os usuários finais.
Protocolos e Atividades	SendInformation, ReceiveInformation, RegisterInformation, registerServiceDirectory
Permissões	Changes LogBase // registra eventos
Responsabilidades	
Liveness	NOTIFIER = registerServiceDirectory (SendInformation | RegisterInformation | ReceiveInformation) $^{\omega}$
Safety	Ter acesso à base de registro de eventos.

Gerente de Conhecimento

Papel: Wrapper da Base de Conhecimento (KnowledgeManager)	
Descrição	Esse papel irá executar ações de aprendizagem, registrando na base de conhecimento novos casos percebidos, registros de casos pelo usuário, coleta de feed-back do usuário e acertos de casos usados, repassados pelo papel coordenador e também deverá inferir conhecimento através de similaridade.
Protocolos e Atividades	ReceiveEpisodeFeedback, // recebe feed-back dos episódios de correção RegisterFoultCase, InferCaseSolution, registerServiceDirectory
Permissões	Changes KnoledgeBase // registra casos de falha // registra episódios de correção
Responsabilidades	
Liveness	KNOWLEDGEMANAGER = registerServiceDirectory (ReceiveEpisodeFeedback) | InferCaseSolution | RegisterFoultCase) $^{\omega}$
Safety	Ter acesso à base de conhecimento.

Coordenador

Papel: Coordenador XAADB (Coordinator)	
Descrição	Esse papel irá coordenar a ação dos outros papéis, elaborando planos globais de resolução de falhas e a participação e seqüência de cada papel (planos parciais). Além disso, ele se responsabilizará pelo seguimento de normas organizacionais e regras de negócio (contexto), que comporão restrições às resoluções dos planos (e.g. restrições de horário). O papel também se responsabilizará pela orientação dos papéis Notificador e Gerente de Conhecimento, os autorizando a realizar modificações nas suas respectivas bases de dados.
Protocolos e Atividades	SendInformation, ReceiveInformation, ReceiveEpisodeFeedback, // recebe feed-back dos episodios de correcao Notify, RegisterServiceDirectory, CreatePlan,
Permissões	Reads KnoledgeBase // registra casos de falha // registra episódios de correção
Responsabilidades	
Liveness	COORDINATOR = RegisterServiceDirectory (ReceiveEpisodeFeedback | CreatePlan | SendInformation | ReceiveInformation | Notify) $^{\omega}$
Safety	Ter acesso à base de conhecimento

Dissertação de Mestrado apresentada por **Paulo Roberto Moreira Maciel** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"DBSitter-AS: Um Framework Orientado a Agentes para Construção de Componentes de Gerenciamento Autônomo para SGBD"**, orientada pela **Profa. Patrícia Cabral de Azevedo Restelli Tedesco** e aprovada pela Banca Examinadora formada pelos professores:

Profa. Patrícia Cabral de Azevedo Restelli Tedesco
Centro de Informática / UFPE

Prof. Marcus Costa Sampaio
Departamento de Sistemas e Computação / UFCG

Profa. Carina Frota Alves
Pesquisadora associada ao Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 9 de agosto de 2007.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.