PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 48/08

Computação Autônoma: Uma Visão sobre Arquiteturas e Infra-estruturas

Sand Luz Corrêa

Renato Fontoura de Gusmão Cerqueira

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 48/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Dezembro, 2008

Computação Autônoma: Uma Visão sobre Arquiteturas e Infra-estruturas[1]

Sand Luz Corrêa e Renato Fontoura de Gusmão Cerqueira

{scorrea, rcerq}@inf.puc-rio.br

Abstract. Driven by the increasing demand for self-management in computer systems, research in the area of autonomic computing has gained prominence. The purpose of this work is to provide an overview of the self-management architectures proposed in the past few years and discuss the underline support required to address their features.

Keywords: Autonomic computing, architecture, infrastructure

Resumo. Motivadas pela crescente demanda por sistemas capazes de gerenciarem a si próprios, pesquisas na área de computação autônoma se tornaram proeminentes. O propósito deste trabalho é fornecer uma visão geral de arquiteturas auto-gerenciáveis propostas nos últimos anos e discutir infra-estruturas de apoio necessárias para assegurar as características de tais arquiteturas.

Palavras-chave: Computação autônoma, arquitetura, infra-estrutura

[1]Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil. Processo CNPq número 140729/2006-2

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introdução

Sistemas computacionais enfrentam um rápido crescimento em tamanho e complexidade. Avanços em redes de comunicação e Internet tornam tais sistemas cada vez mais interdependentes, dando origem a uma malha de serviços digitais integrados e de grande escala que forma um ambiente de operação complexo e sujeito a várias restrições dinâmicas. Como observado em [1], o gerenciamento de ambientes integrados exige não só o controle sobre componentes individuais, mas também sobre os relacionamentos oriundos das interações entre os mesmos. Teoricamente, isto introduz um problema combinatorial, uma vez que qualquer combinação de componentes pode ser considerada na solução. Adicionalmente, ferramentas de gerenciamento, quando providas juntamente com alguns componentes do ambiente, variam consideravelmente em suas funcionalidades, dificultando uma percepção global e uniforme do comportamento do sistema.

Tendo tal cenário como motivação, em outubro de 2001, um manifesto produzido pela IBM [2] alertava para a dificuldade do gerenciamento de softwares complexos (tais como instalação, configuração, otimização e manutenção) como o principal entrave para futuras inovações na indústria de TI. De maneira geral, o manifesto ressalta o crescimento da complexidade dos sistemas atuais e a inevitável incapacidade humana de os gerenciarem, dado o grande inter-relacionamento dos diversos elementos de software e hardware que os compõem e que resulta em comportamentos emergentes difíceis de serem antecipados. Para tratar tal problema, o documento propõe o conceito de Computação Autônoma (do inglês *Autonomic Computing* - AC) - sistemas computacionais capazes de se auto-gerenciarem dado um conjunto abstrato de objetivos definidos pelo administrador. O termo foi escolhido deliberadamente e traz uma conotação biológica com o sistema nervoso autônomo. Este, por sua vez, é responsável pelo controle de funções vitais que adaptam, de forma inconsciente, o corpo humano às suas necessidades e às necessidades geradas pelo ambiente. Tal como seu equivalente biológico, computação autônoma é inserida em sistemas computacionais visando promover o auto-governo de suas funções.

Desde 2001, muitos trabalhos foram propostos tendo como objetivo introduzir habilidades autônomas em sistemas computacionais. Em especial, três áreas de aplicação têm concentrado grande parte destes trabalhos: gerenciamento de energia em grandes centros de computação, gerenciamento de ambientes em grades computacionais e computação ubíqua [3].

Em grandes centros computacionais, o alto custos operacional da infra-estrutura física de TI (como, por exemplo, gastos com eletricidade, condicionamento e refrigeração de equipamentos), resultou em um grande volume de trabalhos que exploram sistemas auto-gerenciáveis capazes de se adaptarem não só em termos de desempenho, mas também em relação ao consumo de energia. Os primeiros trabalhos tratavam exclusivamente o consumo de energia em processadores, como por exemplo em [4]. Recentemente, no entanto, modelos de gerenciamento de energia mais abrangentes têm sido propostos, visando incluir também consumo de recursos como memória, rede e dispositivos de entrada e saída [5].

A natureza heterogênea, dinâmica e de serviços integrados em grande escala, característica dos ambientes de grades computacionais, também tem servido como motivação para aplicação de computação autônoma. Duas áreas de pesquisas se destacam nestes ambientes: gerenciamento dinâmico de recursos [6, 7] e administração de sistemas [8, 9]. O primeiro, no entanto, tem concentrado a maior parte dos trabalhos. Tipicamente, grades computacionais são construídas sobre arquiteturas orientadas a serviço e, portanto,

a satisfação de QoS determina a seleção de recursos. Isto significa o compromisso com a qualidade de serviço acordada não só na alocação inicial dos recursos mas durante toda a utilização dos mesmos. Nesse sentido, AC é aplicada em gerenciamento de recursos de ambientes de grades computacionais como forma de garantir tal compromisso, mesmo diante da dinâmica e complexidade do ambiente em questão. Por outro lado, estudos centrados em administração de sistemas em grades computacionais visam introduzir autonomia administrativa em aplicações que executam em tais ambientes. Para tanto, registros de problemas e reações a adversidades (provenientes da intervenção humanas) são registrados em *logs* de operação. A partir dos *logs* é possível determinar o conjunto de ações e respostas que derivam o conjunto de políticas. Estas, por sua vez, guiarão o comportamento autônomo dos sistemas.

De maneira similar, computação ubíqua também envolve um ambiente complexo e dinâmico, em que diversos dispositivos, possivelmente heterogêneos, interagem entre si. Portanto, a complexidade de instalação e manutenção de aplicações nestes ambientes, naturalmente conduz à necessidade de sistemas auto-gerenciáveis. Um exemplo de trabalho onde computação autônoma é aplicada com este objetivo pode ser encontrado em [10].

Diante deste contexto, propomos, neste trabalho, uma revisão do estado da arte em computação autônoma. Essencialmente, abordamos os princípios que regem AC, arquiteturas para construção de sistemas autônomos e o suporte ou infra-estruturas necessários para o desenvolvimento destes sistemas. Este trabalho está organizado da seguinte forma. A Seção 2 apresenta os princípios fundamentais atribuídos à AC. A Seção 3 descreve as principais propostas de arquitetura encontradas na literatura. A Seção 4 apresenta uma revisão de mecanismos de suporte à construção de aplicações autônomas. Finalmente, a Seção 5 apresenta a conclusão final deste trabalho.

2 Definição e Propriedades

Como mencionado na seção anterior, o conceito envolvendo AC foi inspirado no corpo humano, o qual provê mecanismos efetivos para se auto-monitorar, controlar, regular e até mesmo recuperar-se de pequenos problemas físicos sem a necessidade de intervenções externas. Entretanto, diferentemente do corpo humano, em sistemas computacionais, tal auto-gerência não é desempenhada involuntariamente, mas sim, através de tarefas que administradores delegam aos sistemas de acordo com políticas adaptativas [11]. Estas determinam o tipo de ação a ser executada em diferentes situações.

A figura 1 resume as propriedades gerais de um sistema autônomo [12]. Como ilustrado na figura, o objetivo principal de AC é auto-gerência. Para atingir tal objetivo, entretanto, quatro requisitos são essenciais: auto-cura (*self-healing*), auto-otimização (*self-optimizing*), auto-proteção (*self-protecting*) e auto-configuração (*self-configuring*).

Auto-cura é a propriedade do sistema que assegura sua recuperação efetiva e automática, quando falhas são detectadas. Entretanto, ao contrário de técnicas de tolerância a falhas tradicionais, auto-cura requer não só o mascaramento da falha, mas também a identificação do problema e seu reparo imediato, sem interrupção do serviço e com o mínimo de intervenção externa.

Auto-otimização consiste na capacidade do sistema de ajustar automaticamente suas políticas de utilização de recurso a fim de maximizar a alocação e uso dos mesmos, satisfazendo às demandas dos usuários.

Auto-proteção refere-se à propriedade do sistema de defender-se de ataques acidentais ou maliciosos. Para tanto, o sistema deve ter conhecimento sobre potenciais ameaças, bem como prover mecanismos para tratá-las.

Finalmente, auto-configuração é a característica do sistema que o permite ajustar-se automaticamente às novas circunstâncias percebidas em virtude do seu próprio funcionamento ou como apoio a processos de auto-cura, auto-otimização ou auto-proteção.

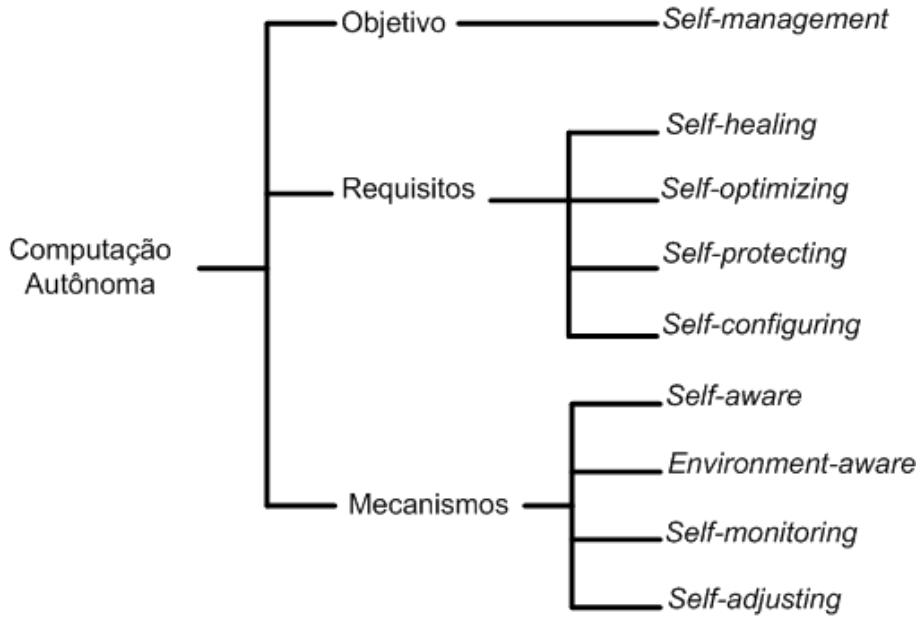

Figura 1: Requisitos para computação autônoma.

Adicionalmente, a manutenção das propriedades descritas acima exige a provisão de mecanismos de implementação que as apóiem, como, por exemplo, a ciência de contexto interno (*self-aware*) e externo (*environment-aware*), o monitoramento destes contextos (*self-monitoring*) e adaptação dinâmica (*self-adjusting*) [2, 13, 14]. Estes mecanismos permitem que sistemas computacionais tenham conhecimento dos componentes que os formam, seus estados internos e estados de suas conexões com outros componentes, bem como os recursos disponíveis no ambiente e o os estados dos mesmos.

De fato, muitos autores tratam os requisitos e mecanismos como um grupo único de propriedades, onde os requisitos são as características principais e os mecanismos as de menor relevância. Alguns trabalhos, como em [15], ressaltam também a importância de duas características adicionais: abertura (*openness*) e proatividade (*anticipatory*). A primeira refere-se à propriedade do sistema de operar em ambientes heterogêneos de forma portátil, obedecendo, portanto, a padrões e protocolos abertos. A segunda refere-se à capacidade do sistema de antecipar suas próprias necessidades e comportamento, bem como as do ambiente, e agir proativamente diante de tais informações.

3 Arquiteturas para AC

Tradicionalmente, sistemas computacionais têm sido construídos para resolver problemas específicos, provendo soluções particulares para satisfazer requisitos estritos e de forma

isolada. Entretanto, no caso de sistemas com comportamentos emergentes, requisitos, objetivos e escolhas específicas podem depender de estados e contextos os quais não são conhecidos antecipadamente. Como mencionado na seção anterior, AC trata este problema tentando assegurar algumas propriedades que garantem uma visão holística dos sistemas computacionais. Para atingir tal objetivo, algumas arquiteturas de software para AC foram propostas. Em geral, estas arquiteturas apresentam soluções para automatizar o ciclo de gerenciamento de sistemas, o qual envolve as seguintes atividades: (1) monitoramento do sistema; (2) análise dos seu comportamento e (3) tomada de decisões para assegurar convergência em relação a valores limiares de parâmetros como desempenho, disponibilidade e segurança (geralmente denominados *Service Level Objectives* ou SLOs). De maneira geral, as arquiteturas propostas são classificadas em dois grupos [16]: arquiteturas baseadas em elementos autônomos e arquiteturas baseadas em infra-estrutura. A seguir descrevemos os dois tipos de arquitetura.

3.1 Arquitetura Baseada em Elementos Autônomos

Neste tipo de arquitetura, sistemas auto-gerenciáveis são formados pela composição de elementos autônomos. Estes, por sua vez, consistem em módulos de software auto-contidos com interfaces de interação específicas e dependências de contexto explícita [17]. A figura 2 ilustra um componente autônomo. Cada elemento consiste em dois módulos principais: uma unidade funcional que executa os serviços providos pelo elemento e uma unidade de controle que monitora seu estado e contexto, analisa a condição corrente e promove a adaptação necessária. Como mostrado na figura, as partes principais de um elemento autônomo são:

- Elemento gerenciado: corresponde à unidade funcional do elemento, a qual pode ser afetada, em tempo de execução, em virtude de falhas, escassez de recursos, ataques, problemas de desempenho, etc.
- Ambiente: representa os fatores que podem afetar o elemento gerenciado, sendo formado por duas partes. O ambiente interno consiste em mudanças ocorridas no próprio elemento gerenciado e, portanto, refletem o estado do mesmo. O ambiente externo reflete o estado do ambiente de execução.
- Controle: corresponde à unidade de gerenciamento do elemento. A unidade de controle: (1) recebe definições de requisitos (desempenho, tolerância a falhas, segurança) desejados e especificados pelos usuários; (2) inspeciona e caracteriza o estado do elemento; (3) inspeciona o estado geral do sistema; (4) determina o estado do ambiente e (5) usa estas informações para controlar e adaptar a operação do elemento gerenciado a fim de atingir o comportamento especificado. Inspeções e adaptações de estado são promovidas, respectivamente, por sensores e adaptadores acoplados ao elemento gerenciado.

O módulo de gerenciamento consiste em dois laços de controle denominados MAPE (Monitora-Analisa-Planeja-Executa). O laço de controle local trata apenas estados de ambiente conhecidos, sendo baseado em conhecimento encontrado no próprio elemento gerenciado. Por esta razão, o laço local é incapaz de controlar o comportamento global do sistema. Em um cenário de gerenciamento em que todo o sistema é afetado, o laço local

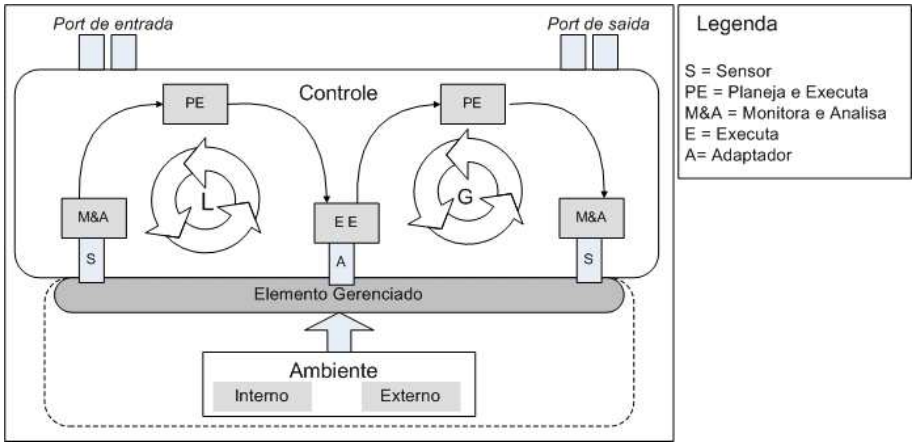

Figura 2: Estrutura de um elemento autônomo

irá repetir continuamente sua política de adaptação local, podendo levar o elemento a um comportamento caótico. Em um dado momento, variáveis essenciais do sistema atingirão seus limiares, ativando-se assim o laço de controle global. Este, por sua vez, trata estados de ambiente desconhecidos empregando técnicas de aprendizado de máquina, inteligência artificial ou mesmo intervenção humana. O novo conhecimento oriundo do laço de controle global é inserido no elemento gerenciado, tornando-o capaz de adaptar seu comportamento atual às mudanças do ambiente.

Devido a sua estrutura, um elementos autônomo deve ser auto-gerenciável, ou seja, deve ser capaz de se reconfigurar, de se recuperar de falhas internas, de otimizar seu próprio comportamento e de se proteger contra ataques externos. Um elemento autônomo deve, sempre que possível, tratar tais problemas localmente, simplificando dessa forma o gerenciamento global do sistema. Um elementos autônomo também deve ser capaz de estabelecer e manter relacionamentos com outros elementos autônomos, em especial, seus provedores ou consumidores de serviço. Tais relacionamentos são a única maneira pela qual entidades maiores, bem como o próprio sistema, são criadas. Portanto, arquiteturas baseadas em elementos autônomos são inerentemente distribuídas. Entretanto, sem uma infra-estrutura central de monitoramento, elementos autônomos publicam seu estado em canais compartilhados por outros elementos autônomos. Esta solução, por sua vez, pode levar a um grande volume de dados sendo transmitidos pelo canal.

A seguir descrevemos alguns projetos envolvendo AC que implementam uma arquitetura baseada em elementos autônomos. As arquiteturas de software destes projetos se diferenciam essencialmente pela abstração utilizada ao representar um serviço ou conjunto de serviços autônomos. Unity [18] e ABLE [19] implementam elementos autônomos como agentes inteligentes, enquanto que Accord [20, 21] utiliza a abstração de componentes de software para atingir o mesmo objetivo.

3.1.1 Unity

Em Unity, elementos autônomos são implementados como agentes Java, usando-se um conjunto de ferramentas para apoiar o desenvolvimento de aplicações AC. Elementos comunicam entre si através de interfaces *Web Services*. Para assegurar o auto-gerenciamento

do sistema a partir da operação localizada dos elementos autônomos que o constitui, alguns elementos de infra-estrutura são definidos na arquitetura:

- *containers*: elementos que representam máquinas (nós) que hospedam elementos autônomos. *Containers* são também responsáveis por iniciar os elementos da arquitetura, incluindo elementos autônomos.

- registradores: elementos que provêem mecanismos de descoberta de elementos da arquitetura. Fornecem um serviço de registro onde elementos da arquitetura podem publicar seus serviços e torná-los disponíveis para outros elementos.

- repositórios de políticas: elementos que provêem interfaces através das quais é possível manter um repositório de políticas que guiam a operação do sistema. Em geral, três tipos de políticas são suportadas: políticas baseadas em ações (tipicamente da forma *if(condition)then(action)*), políticas baseadas em objetivos (condições com maior nível de abstração que a condição anterior, uma vez que especificam situações desejadas sem no entanto especificar como obtê-las) e políticas baseadas em funções de utilidade (condições com o maior nível de abstração pois especificam funções que determinam automaticamente o objetivo mais apropriado para qualquer situação apresentada).

- agregadores: combinam dois ou mais elementos e utiliza o agregado para formar uma entidade de serviço superior

- sentinelas: provêem interfaces através das quais elementos solicitam o monitoramento da operação de outros elementos. Se o elemento monitorado se torna não responsivo, os elementos que solicitaram o monitoramento são notificados.

- árbitro de recursos: elemento que computa a alocação de recursos em elementos autônomos denominados ambientes de aplicação. Cada ambiente de aplicação representa um ambiente de execução. Um gerente interno ao elemento é responsável por obter os recursos que o ambiente necessita e comunicar-se com outros ambientes de aplicação.

Em Unity, auto-configuração é obtida através de um processo de composição guiada por objetivos. Tal processo é mais freqüente na iniciação do elemento, onde suas dependências externas são resolvidas. Entretanto, reconfigurações podem acontecer em qualquer momento do ciclo de vida do elemento. Auto-cura é implementada em agrupamentos (*clusters*) de repositórios de política. Auto-otimização é assegurada através do árbitro de recursos e ambientes de aplicação. A seguir, descrevemos resumidamente estes processos.

Composição de Serviços Guiada por Objetivos

Containers e Registradores consistem nos primeiros elementos a serem iniciados, seguidos imediatamente pelo árbitro de recursos. Este, por sua vez, solicita aos *containers* a iniciação dos repositórios de política e das sentinelas que irão monitorar os repositórios. Após iniciação, os repositórios se registram contactando um elemento registrador. Após iniciação dos elementos de infra-estrutura, elementos autônomos são iniciados. Inicialmente,

um elemento autônomo conhece apenas o papel genérico que desempenha no sistema e o endereço de um elemento registrador. Ele, então, entra em contato com o registrador para localizar outros elementos e resolver suas dependências externas. Dentre estes elementos, destacamos o repositório de políticas, o qual é localizado pelo elemento autônomo a fim de obter suas políticas de operação a partir do seu papel. Com as dependências externas resolvidas, o elemento autônomo se registra a fim de ser contactado por outros elementos que necessitam de seus serviços.

Auto-cura em Repositórios de Políticas

Repositórios de política são mantidos em agregados sincronizados onde cada alteração de estado de um repositório é imediatamente replicada para os demais elementos do agregado. Quando o sistema é iniciado, um árbitro de recurso decide quantos repositórios de políticas serão necessários e contacta containers para criarem tais repositórios e as respectivas sentinelas que irão monitorá-los. Cada repositório criado consulta um registrador para contactar os membros do agrupamento já registrados e juntar-se a eles. Durante operação dos repositórios, qualquer alteração no conjunto de políticas de um elemento do agrupamento é imediatamente comunicada aos demais membros. Se um sentinela detecta que seu repositório associado falhou, ele notifica o árbitro de recursos. Este, por sua vez, escolhe um dos membros restante para assumir as subscrições do repositório defeituoso. Em seguida, o árbitro escolhe um nó de execução e entra em contato com seu respectivo *container* para criar um novo repositório de políticas que se juntará ao agregado.

Auto-otimização de Recursos

A figura 3 ilustra os elementos envolvidos no processo de auto-otimização. Cada ambiente de aplicação possui uma função de utilidade obtida a partir de um repositório de políticas. A função de utilidade para o ambiente i é representada por $U_i(S_i, D_i)$, dado um conjunto fixo R_i. S_i e D_i representam o nível de serviço e a demanda em i, respectivamente. Ambos, são vetores que especificam valores para múltiplas classes de usuários. R_i representa um vetor em que cada elemento indica a quantidade específica de um tipo de recurso. O objetivo do gerenciamento de recurso é otimizar $\sum_i U_i(S_i, D_i)$ continuamente, sendo o controle e a otimização do conjunto fixo de recursos (R_i) em um ambiente de aplicação i feitos pelo gerenciador do próprio ambiente. Entretanto, alocações entre ambientes de aplicação são tratadas pelo árbitro de recursos. Este recebe periodicamente dos ambientes de aplicação uma função de utilidade $U^{'}(R_i)$ que estima o valor para cada nível possível de recurso R_i. Esta função de utilidade é calculada em função do nível de serviço esperado ($S_i^{'}$) e demanda esperada ($D_i^{'}$) para uma janela de tempo determinada.

Funções de utilidade são definidas pelo usuário e inseridas nos repositórios de política. A arquitetura foi validada através de protótipos, explorando cenários de gerenciamento de recursos.

3.1.2 ABLE

Em [19], ABLE (*Agent Building and Learning Environment*), um *framework* desenvolvido pela IBM como plataforma para construção de sistemas multi-agentes, foi estendido para amparar o conceito de agentes autônomos. Nesta extensão, algoritmos podem ser

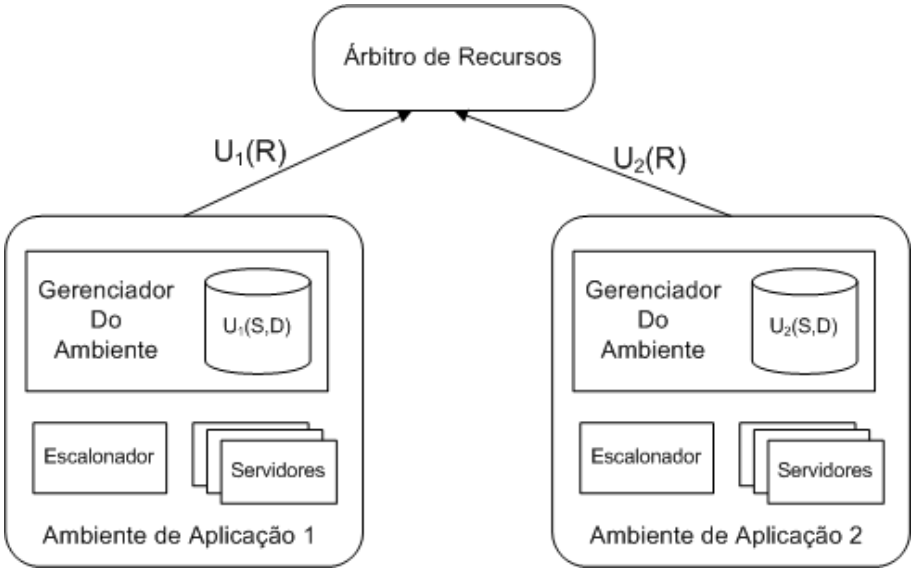

Figura 3: Arquitetura para auto-otimização em Unity

encapsulados em componentes JavaBeans e implantados como objetos Java ou agentes autônomos.

A figura 4 mostra as principais classes que formam o *framework*. *AbleBeans* são componentes *JavaBeans* ordinários. Estes componentes são conectados entre si para formar *AbleAgents*. *AbleAgents*, por sua vez, consistem em *containers* para outros *AbleBeans*, fornecendo uma abstração adequada para encapsular *AbleBeans* relacionados ou mesmo outros *AbleAgents*. Para tanto, *AbleAgents* implementam as interfaces *AbleBeanContainer* e *AbleUserDefinedFunctionManager*. *AbleUserDefinedFunctionManager* define operações que permitem a um software externo, tal como sensores e adaptadores, se integrar a *AbleAgents*.

Inteligência não é um recurso inerente aos *AbleAgents*. Ao contrário, ela deve ser explicitamente adicionada aos agentes. Para tanto, o *framework* fornece uma biblioteca de componentes *AbleBeans* que implementam serviços de filtragem de dados (*data beans*), algoritmos de aprendizado de máquina (*learning beans*) e máquinas de inferência (*rule beans*). Dessa forma, um elemento autônomo em ABLE é implementado através de um *AbleAgent* que encapsula *AbleBeans* funcionais, sensores, adaptadore e *AbleBeans* de conhecimento.

Em [19] são apresentados estudos de casos de aplicações desenvolvidas usando-se ABLE. Os cenários apresentados envolvem administração de sistemas em ambiente com servidores *web* e banco de dados.

3.1.3 Accord

Accord consiste em uma arquitetura baseada em componentes de software para a construção de sistemas autônomos. Nesta arquitetura, modelos conceituais, de implementação e de governança são definidos para utilizarem conhecimento humano sobre a aplicação a fim de guiar a execução e adaptação dos serviços. Este objetivo é alcançado adaptando-se o comportamento de serviços individuais e suas interações de acordo com políticas defini-

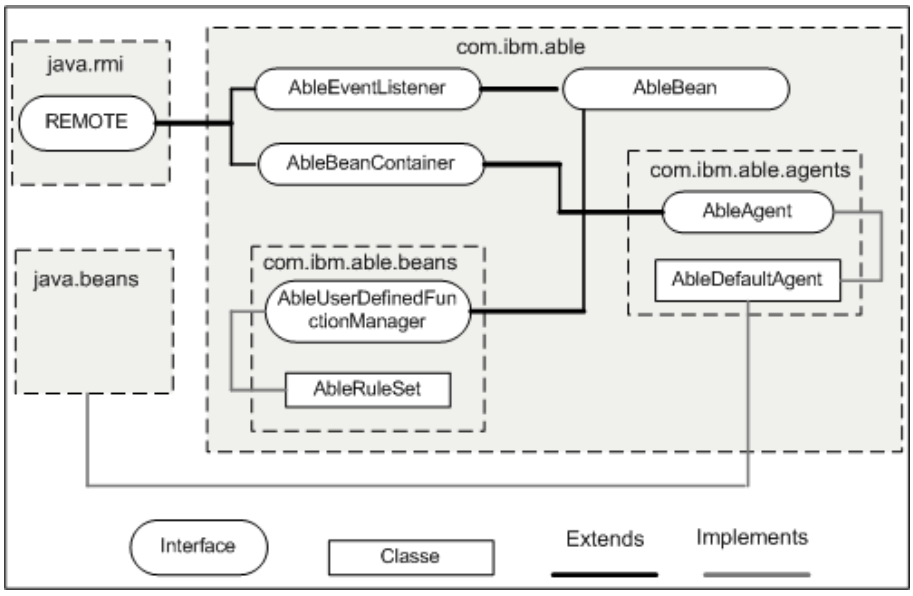

Figura 4: Estrutura do *framework* ABLE

das por usuários, visando, assim, melhor atender os requisitos dinâmicos do ambiente de execução.

A figura 5 ilustra a arquitetura de serviços autônomos em Accord. Como mostrado na figura, um serviço autônomo estende a estrutura tradicional de um componente de software, adicionando a este: (1) uma porta de controle através da qual o estado do componente pode ser monitorado e (2) um gerente de serviço que monitora e controla, em tempo de execução, o comportamento do serviço gerenciado. Uma porta de controle consiste em sensores e adaptadores que permitem, respectivamente, consultar e modificar o estado do serviço gerenciado. Portas de controle e serviço são usadas pelo gerente para controlar as funções, desempenho e interações do serviço gerenciado e são descritas usando-se WSDL (*Web Service Definition Language*).

Em Accord, políticas seguem a forma de regras *if(condition)then(action)* e são descritas pelo usuário usando-se XML. Dois tipos de regras são definidas. Regras de adaptação controlam o comportamento funcional do componente, incluindo modificação de parâmetros de serviço, mudança de implementações para alcançar requisitos de QoS, correção de erros e proteção de serviços. Estas adaptações são locais, ou seja, ocorrem em serviços individuais. Regras de interação controlam a relação entre componentes, entre componentes e seus ambientes e a coordenação de uma aplicação autônoma. Composições autônomas são obtidas através da combinação de regras de adaptação e interação. De maneira geral, quando regras de adaptação não podem satisfazer objetivos globais, regras de interação são usadas para modificar a composição da aplicação.

Infra-estrutura de Execução

Em Accord, a infra-estrutura de execução é constituída por um gerente de composição, os serviços autônomos e uma máquina de governança que opera de forma descentralizada.

O gerente de composição decompõe o *workflow* de uma aplicação em regras de interação

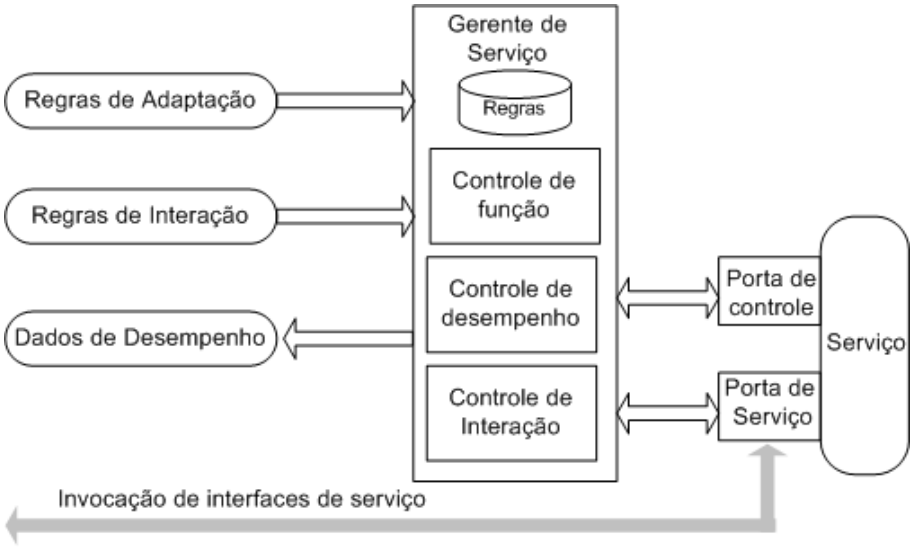

Figura 5: Elemento autônomo em Accord

para serviços individuais. Este processo de decomposição consiste em mapear padrões de *workflow* em templates de regras correspondentes. Alguns templates básicos são fornecidos pelo próprio Accord, enquanto que templates mais complexos podem ser construídos a partir de templates simples. As regras de interação são então injetadas nos gerentes dos respectivos serviços, onde são executadas para estabelecer a comunicação e coordenação entre os serviços envolvidos. Regras de adaptação também são injetadas nos gerentes de serviço. No gerente de serviço, a execução de uma regra envolve três fases: consulta à condição, avaliação da condição e resolução de conflitos e invocação da ação. Regras de resolução de conflitos também são definidas pelo usuário.

É importante ressaltar que a infra-estrutura de execução de Accord é descentralizada. Enquanto regras de interação são definidas pelo gerente de composição, as interações reais ocorrem nos gerentes de serviço de forma descentralizada e paralela. Relacionamentos de coordenação entre serviços podem ser dinamicamente modificados substituindo-se regras de interação. Entretanto, uma grande dificuldade da arquitetura consiste na grande dependências do domínio da aplicação para a definição das regras de adaptação e interação. Para resolver este problema, Accord oferece um serviço de controle baseado em modelos como forma de complementar as estratégias baseadas em regras.

Controle Baseado em Modelos

Como mencionado anteriormente, um modelo de controle formal, denominado LLC (*limited look-ahead control*), é adicionado aos gerentes de serviço, complementando-se as estratégias baseadas em regras. Estes controladores são inseridos para aumentar a efetividade das regras definidas, já que estas são suscetíveis a erros. A abordagem por LLC permite que múltiplos objetivos (QoS) e restrições do sistema sejam representadas explicitamente no problema de otimização, em cada passo de controle. Dado um passo de controle k, o controlador encontra o menor valor que minimiza a função de custo $\sum_{i=k+1}^{k+N} J(x(i), u(i))$, sujeita às restrições do sistema. N representa um horizonte de pre-

visão, $x(i)$ representa o estado do sistema no passo k e $u(i)$ representa as variáveis de controle e parâmetros do ambiente no tempo k.

Accord é um protótipo gerado no projeto AutoMate, o qual tem por objetivo a construção de um ambiente de grades computacionais autônomas. Neste contexto, algumas aplicações para gerenciamento de recursos dinâmicos em grades computacionais foram desenvolvidas usando-se o *framework*.

3.2 Arquitetura Baseada em Infra-estrutura

Arquiteturas baseadas em infra-estruturas correspondem ao grupo de soluções nas quais os elementos que compõem o sistema não são inerentemente autônomos. Ao contrário, as propriedades autônomas são providas pela infra-estrutura através de modelos que descrevem e analisam o comportamento do sistema. Dessa forma, existe uma separação clara entre a infra-estrutura que provê as habilidades autônomas e o sistema em questão. Tipicamente, modelos arquiteturais consistem em grafos que descrevem os componentes do sistema e o seus relacionamentos. O nível de abstração envolvido na noção de componente no modelo é determinado pelo projetista da arquitetura.

A figura 6 ilustra uma arquitetura de software genérica em que habilidades autônomas são fornecidas pela infra-estrutura. Sondas (*probes*) são inseridas no sistema em execução a fim de monitorá-lo. Tais sondas geralmente são localizadas e instrumentam partes específicas do sistema. Dados provenientes do monitoramento efetuado pelas sondas são agrupados em observações de mais alto-nível presentes no modelo arquitetural. Esta tarefa é executada por componentes calibradores (*gauges*) que se situam entre o sistema monitorado e o gerente de adaptação, responsável por controlar adaptações no nível arquitetural. Informações fornecidas pelo calibrador permitem atualizar o modelo com base no estado corrente do sistema. Dessa forma, quando uma propriedade é atualizada em virtude de dados coletados pela sondas, o modelo é novamente analisado a fim de determinar se o sistema ainda opera adequadamente. Caso contrário, um plano reparador é criado contendo uma estratégia adaptativa para o sistema em execução.

Uma vantagem deste tipo de arquitetura em relação àquelas baseada em elementos autônomos consiste na separação clara entre a infra-estrutura que fornece as habilidades autônomas e o sistema monitorado. Esta separação facilita a incorporação de mecanismos de auto-gestão em sistemas já existentes. Entretanto, é importante ressaltar que sondas, adaptadores e até mesmos os modelos arquiteturais podem ser dependentes da aplicação. Uma desvantagem deste tipo de arquitetura é que sua natureza centralizadora (a infra-estrutura concentra os mecanismos que provêem a autonomia do sistema alvo) pode dificultar sua implantação em ambientes distribuídos. A seguir descrevemos dois projetos que utilizam este tipo de abordagem.

3.2.1 KX

Em [22] é introduzida uma arquitetura denominada KX (*Kinesthetics eXtreme*) cujo objetivo é inserir uma infra-estrutura de monitoramento-controle-realimentação em sistemas já existentes. Dados sobre o sistema em execução são coletados a partir de sondas e interpretados por calibradores, que os mapeiam em variáveis de modelos do sistema em execução. Uma camada de controle e decisão analisa o efeito dos novos valores de variáveis no comportamento global do sistema, podendo decidir-se pela inserção ou remoção de sensores e

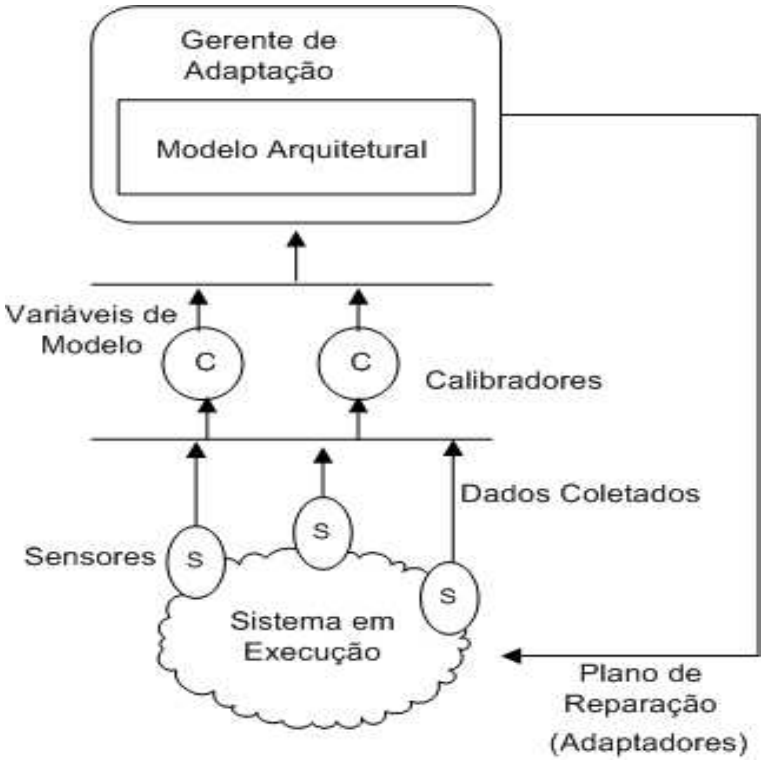

Figura 6: Arquitetura genérica de abordagens baseadas em infra-estrutura

calibradores ou reconfiguração de componentes ou módulos do sistema via adaptadores.

Em KX, sondas consistem em códigos de instrumentação inseridos dinamicamente em *byte code* Java. Calibaradores consistem em dois tipos diferentes de componentes: formatadores e reconhecedores de padrão. Formatadores transformam dados originais provenientes de sondas em um formato padronizado, uma vez que sondas podem representar dados de forma independente. Reconhecedores de padrão relacionam, de forma temporal, eventos provenientes de sondas distintas.

O mecanismo de adaptação se baseia em uma máquina de processamento de *workflow*, denominada *workflakes*. Esta máquina é responsável pela implantação de agentes móveis, denominados *worklets*, nos componentes gerenciados. No sistema em execução, *worklets* são executados através de uma máquina virtual. Adaptadores especializados, então, transformam comandos *worklets* em ações específicas. Apesar da existência de mecanismos para processar *workflows*, a arquitetura não oferece nenhum suporte para a sua geração.

Vários estudos de casos foram desenvolvido para validar a arquitetura proposta, incluindo um serviço de mensagem instantânea, processamento adaptativo de dados multimídia e detecção e recuperação de falhas em um sistema de informação geográfica (denominado GeoWorlds).

3.2.2 Park et al.

Outro exemplo de projeto envolvendo AC que implementa uma arquitetura baseada em infra-estrutura pode ser encontrado em Park et al. [23]. Como em Unity, a arquite-

tura proposta é baseada em sistemas multi-agentes. Entretanto, diferentemente de Unity, onde cada agente representa um elemento autônomo, nesta arquitetura, agentes formam a infra-estrutura que provê as habilidades autônomas para um sistema em execução. Esta abordagem é utilizada com o objetivo principal de minimizar os recursos necessários para a obtenção das habilidades autônomas.

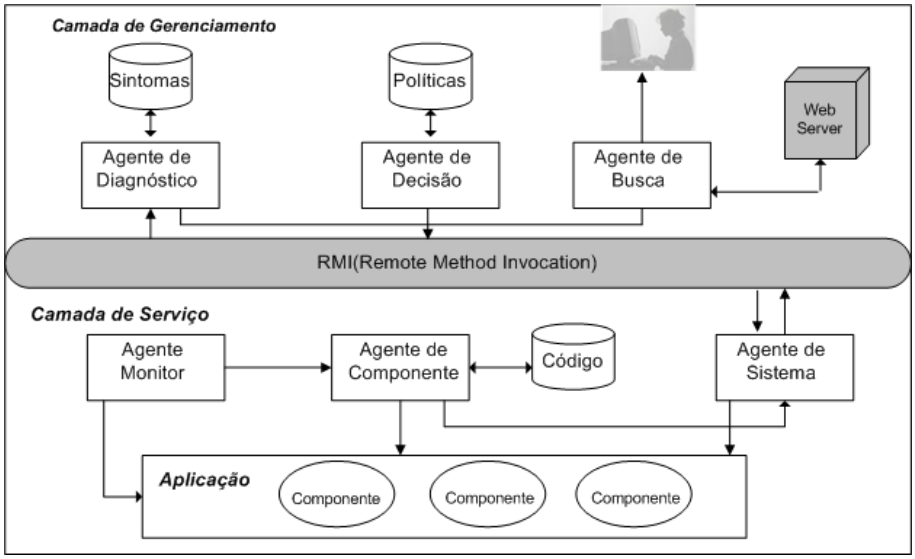

Figura 7: Arquitetura multi-agente que implementa uma abordagem baseada em infra-estrutura

A figura 7 ilustra a estrutura da arquitetura proposta em Park et al.. Esta arquitetura baseia-se em seis processos consecutivos: monitoramento, filtragem, tradução, diagnóstico, e decisão.

Na fase de monitoramento, um agente monitor, implementado como um processo único, é responsável por observar o estado dos recursos (CPU, memória, etc) de uma máquina ou nó de execução e monitorar o tamanho dos históricos gerados pelos componentes do sistema. Para tratar problemas causados por componentes que não geram históricos de execução, o agente monitora também eventos de erro gerados pelo sistema operacional. Caso o agente observe algum evento suspeito nos estados dos recursos, históricos dos componentes ou eventos gerados pelo sistema operacional, uma fase de filtragem é iniciada.

Na fase de filtragem, um agente de componentes coleta o contexto observado pelo agente monitor e o submete a um processo de filtragem, onde apenas contextos de erros são selecionados. Contextos de erros são filtrados a partir de palavras-chave: "erro", "não", "notificação", etc. Em seguida, o contexto filtrado é traduzido para o formato CBE (*Common Based Event*). A filtragem é um processo importante para acelerar a execução das fases seguintes, uma vez que, em geral, a conversão para o formato CBE produz arquivos maiores que os históricos originais. Após a conversão, o contexto de erro é classificado segundo uma escala de prioridades.

Após a filtragem, é iniciada a fase de execução, controlada por um agente de sistema. Este, por sua vez, recebe o contexto de erro no formato CBE e coleta os estados dos recursos. O agente de sistema mantém valores limiares de utilização dos recursos. Estes valores são comparados com a informação de estados coletada e uma política apropriada

é executada em função do resultado da comparação. A informação coletada e o contexto CBE são então enviados para um agente de diagnóstico, dando origem à fase de análise.

O contexto CBE, a informação sobre os estados dos recursos e os componentes correlacionados com o contexto são entradas para o processo de diagnóstico de problemas realizado pelo agente de diagnóstico. Este utiliza um algoritmo de agrupamento (*clustering*) para correlacionar sintomas observados e o problema que gerou tais observações. O resultado do diagnóstico é enviado para um agente de decisão.

O agente de decisão toma uma ação de auto-cura a partir do diagnóstico obtido na fase anterior. Para atingir este objetivo, o agente de decisão usa uma estratégia determinística, mantendo uma tabela (repositório de políticas) que mapeia problemas em decisões. Entretanto, quando o problema não pode ser resolvido deterministicamente (isto é, não existe um mapeamento para o problema em questão), o agente de decisão usa um algoritmo de árvore de decisão para encontrar a ação mais apropriada. A solução fornecida pelo algoritmo é "aprendida" pelo agente, criando-se uma entrada para o problema e sua solução no repositório de políticas. Finalmente, a solução, isto é o conjunto de código a ser executado a fim de se recuperar do problema diagnosticado, é enviada ao agente de sistema. Adicionalmente, a arquitetura fornece um agente de busca para acessar o *site* do fornecedor à procurar da solução de um problema. O resultado da busca é enviada para o administrador do sistema.

A arquitetura proposta em Park et al. foi implementada em Java e os agentes foram desenvolvidos usando a plataforma JADE. Políticas foram implementadas através de regras *if(condition)then(action)*) escritas em XML. Um cenário de detecção de falhas em um ambiente de transações *web* foi implementado para validar a arquitetura.

4 Infra-estrutura de Suporte

Como discutido nas seções anteriores, a complexidade emergente dos sistemas computacionais atuais requer arquiteturas de software adaptáveis em diversos atributos e funcionalidades. Tais arquitetura, entretanto, se tornam possíveis somente através de infra-estruturas complexas. Nesta seção discutimos a infra-estrutura necessária para a construção de arquiteturas autônomas. Como proposto em [24], tratamos a complexidade dividindo a infra-estrutura em três níveis: tecnológica, de serviços e de políticas. De maneira geral, a infra-estrutura tecnológica descreve abstrações básicas em que serviços são construídos. A infra-estrutura de serviço descreve o conjunto de requisitos não funcionais (autônomos ou não) que compõem parte do comportamento das aplicações. Finalmente, a infra-estrutura de políticas descreve um conjunto de regras que coordenam e controlam o comportamento das aplicações. A seguir, discutimos cada dimensão dentro de uma perspectiva *top-dow*, ou seja, partindo-se da infra-estrutura de mais alto nível.

4.1 Infra-estrutura de Políticas

Políticas são especificações declarativas de regras ou regulamentações que determinam o comportamento de componentes de aplicações, de forma totalmente desacoplada de suas implementações [25]. Por esta razão, políticas são representações concisas, de fácil compreensão e verificação, podendo ser dinamicamente atribuídas. Tipicamente, o grau de adaptabilidade de uma aplicação cresce com o número de módulos controlados por

políticas. Em AC, políticas são essenciais, uma vez que elas constituem a forma na qual os administradores expressam seus objetivos para o sistema.

Em geral, políticas podem ser representadas de três maneiras: ações, objetivos e funções de utilidade. Políticas baseadas em ações são representações simples de regras que tomam a forma *if(condition)then(action)*. Entretanto, uma dificuldade neste tipo de representação é a possibilidade de ocorrência de conflitos entre políticas, quando muitas regras são especificadas. Políticas baseadas em objetivos são especificações com maior nível de abstração que especificações baseadas em ações, uma vez que definem situações desejadas sem, no entanto, especificar como obtê-las. Em tempo de execução, planos são gerados contendo ações a serem tomadas para atingir o objetivo esperado. Contudo, uma dificuldade neste tipo de representação consiste na classificação dos estados possíveis para uma situação, o qual é expresso apenas através de dois valores: desejado ou indesejado. Dessa forma, estados intermediários são impossíveis de serem atingidos através de objetivos. Políticas baseadas em funções de utilidade resolvem este problema pela definição de níveis de satisfação para cada estado. Porém, funções de utilidade são difíceis de serem definidas.

Políticas são criadas e mantidas por infra-estruturas de gerenciamento. Uma infra-estrutura de gerenciamento de políticas deve incluir: uma linguagem de especificação de regras, um mecanismo de ativação de ações baseado no contexto corrente e um modelo de distribuição e implantação de políticas em pontos específicos de controle. Idealmente, linguagens de especificação de regras devem ser rica em formalismo, uma vez que uma semântica formal facilita a automatização de mecanismos de análise e ativação de ações. Tal linguagem deve também ser de fácil extensão, permitindo a criação de novos tipos de políticas a partir de regras já existentes. Por outro lado, mecanismos de análise e ativação de ações envolvem o monitoramento do contexto em que os componentes ou sistema gerenciado executam e a seleção da política a ser aplicada de acordo com o contexto observado. Tais mecanismos estão diretamente relacionados à infra-estrutura de serviço (monitoramento e adaptação). Mecanismos de implantação, por sua vez, descrevem como as novas políticas serão introduzidas nos componentes gerenciados e também estão fortemente relacionados à infra-estrutura de serviço.

Adicionalmente, uma infra-estrutura de gerenciamento de políticas deve também prover mecanismos para assegurar a correção das políticas especificadas, fornecendo apoio à descrição de meta-políticas (regras e restrições aplicadas às próprias políticas especificadas), verificação de regras e detecção e resolução de possíveis conflitos.

4.2 Infra-estrutura de Serviços

Este tipo de infra-estrutura compreende os serviços fundamentais que apóiam a construção de sistemas distribuídos tradicionais (como, por exemplo, serviço de implantação e registro), bem como serviços específicos, diretamente relacionados com a construção de aplicações autônomas. Dentre os serviços específicos, destacamos o de monitoramento e adaptação como os mais importantes.

Monitoramento consiste no ato de coletar informações relacionadas a características e estados de objetos de interesse [26]. Particularmente, em AC, monitoramento consiste na captura de propriedades do ambiente que são significativas para os requisitos de auto-gerenciamento. Entidades do sistema encarregadas do processo de monitoramento são denominadas sensores. O processo de monitoramento pode ocorrer de forma passiva, pela

simples observação e relato de eventos que acontecem no ambiente, ou de forma ativa, através, por exemplo, de ferramentas que modificam e adicionam código às implementações de aplicações.

Devido ao grande volume de dados gerado por processos de monitoramento, uma característica desejável para sensores é adaptabilidade. Nesse sentido, sensores devem ser capazes de ajustar a freqüência de monitoramento e, portanto, o volume de dados gerados, de forma a minimizar a sobrecarga do processo, sem contudo, comprometer a consistência das informações monitoradas.

Adaptação consiste no processo em que softwares são alterados dinamicamente para atender um uso corrente ou condição ambiental. Tipicamente, em AC, adaptação ocorre em função da necessidade de reparo de erros, melhora de desempenho, tolerância a falhas ou proteção em virtude de ataques. Duas abordagens gerais têm sido aplicadas para alcançar adaptação dinâmica: adaptação paramétrica e adaptação estrutural [27].

Adaptação paramétrica envolve a modificação de variáveis que determinam o comportamento do software. Em sistemas em que o comportamento é representado através de um modelo arquitetural, adaptação paramétrica constitui-se em uma forma direta de controle de tal comportamento. Entretanto, uma desvantagem desta abordagem é á incapacidade de adoção de estratégias de adaptação que não foram inicialmente projetadas.

Ao contrário, adaptação estrutural consiste na troca de algoritmos ou partes estruturais de um software, possibilitando a uma aplicação a adoção de novas estratégias (algoritmos) para tratar situações que não foram inicialmente previstas na sua construção. Exemplos de adaptação estrutural incluem a recomposição dinâmica de um sistema em função da limitação de recurso, adição de um novo comportamento em sistemas já instanciados a fim de acomodar situações emergentes ou substituição de componentes para reparo de erros.

4.3 Infra-estrutura Tecnológica

No nível mais básico de infra-estrutura, o apoio à construção de sistemas auto-gerenciáveis tem sido provido principalmente por tecnologias como separação de interesses (*separation of concerns*), reflexão computacional, programação baseada em componentes de software e programação baseada em agentes inteligentes.

Separação de interesses é um estilo de programação no qual os comportamentos funcionais e não funcionais da aplicação são concebidos de forma separada, aumentando-se assim a reuso dos módulos projetados. Em AC, separação de interesses é aplicada no projeto de serviços autônomos com o objetivo de obter um baixo acoplamento entre módulo de gerenciamento e o comportamento funcional dos serviços.

Reflexão computacional refere-se à habilidade de um software em prover estruturas para sua própria representação, de tal forma que seu comportamento e a estrutura que o descreve estejam causalmente conectados. Tal conexão determina que o comportamento do sistema é controlado através da manipulação da sua representação, a qual é constantemente atualizada de acordo com o próprio comportamento. Dessa forma, reflexão computacional pode ser aplicada em sistemas autônomos para prover mecanismos de monitoramento e análise de comportamento que determinam adaptações paramétricas e estruturais.

Programação baseada em componentes de software ressurgiu recentemente como uma forma de complementar o modelo orientado a objetos e prover meios de diminuir suas deficiências. Dessa forma, o modelo de componentes de software incorpora vários conceitos do paradigma de objetos, como encapsulamento e separação entre interface e implementação,

mas além disso também torna explícitos conceitos como dependências e conexões entre componentes. Estes podem ser definidos como unidades de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, definidas através de um conjunto de conectores. A composição entre componentes de software pode ser feita de forma estática (em tempo de compilação) ou dinâmica (em tempo de execução). Dessa forma, programação baseada em componentes contribui para o desenvolvimento de sistemas autônomos de duas formas principais: possibilitando a interação entre serviços autônomos heterogêneos, através da separação clara entre interface e implementação, e facilitando o processo de adaptação estrutural, uma vez que a noção de composição dinâmica é inerente ao próprio paradigma.

Finalmente, agentes de software consistem em entidades cognitivas que percebem o ambiente em que estão inseridas e reagem a ele. Tais agentes podem utilizar diferentes técnicas de inteligência artificial para determinar uma reação apropriada. Esta característica torna agentes de software entidades proativas, capazes não só de atenderem requisições, mas também alterar o ambiente em que estão inseridos. Dessa forma, o princípio de autonomia é algo inerente à abstração de agentes.

As tabelas 1 e 2 resumem as características principais das arquiteturas apresentadas neste trabalho, em relação às propriedades de auto-gerenciamento consideradas, aplicações desenvolvidas e infra-estruturas (de polícas, serviços e tecnlógicas) utilizadas.

Característica	**Unity**	**ABLE**	**Accord**
propriedades	*self-healing* *self-optimizing* *self-configuring*	*self-healing* *self-configuring*	*self-optimizing* *self-configuring*
infra-estrutura de política	funções de utilidade definidas pelo usuário	ações e objetivos	ações definidas pelo usuário ou apoiadas por modelos
serviço de monitoramento	sentinelas	implementado pelo usuário	implementado pelo usuário
serviço de adaptação	paramétrica	paramétrica e estrutural	paramétrica e estrutural
infra-estrutura tecnológica	agentes de software	agentes de software formados por componentes JavaBeans	componentes de software
aplicações desenvolvidas	protótipo envolvendo gerenciamento de recursos	aplicações envolvendo administração de servidores *web* e banco de dados	gerenciamento de recursos em grades computacionais

Tabela 1: Exemplos de arquiteturas baseadas em elementos autônomos - caracteríticas principais

Característica	**KX**	**Park et al.**
propriedades	*self-healing* *self-configuring*	*self-healing* *self-configuring*
infra-estrutura de política	ações baseadas em *workflows*	ações escritas em XML
serviço de monitoramento	sondas ativas e calibradores de evento	agente monitor
serviço de adaptação	paramétrica e estrutural	paramétrica
infra-estrutura tecnológica	objetos Java	agentes de software
aplicações desenvolvidas	administração de sistemas legados	protótipos de administração de sistemas

Tabela 2: Exemplos de arquiteturas baseadas em infra-estrutura - caracteríticas principais

5 Conclusão

Desde 2001, quando o manifesto produzido pela IBM alertou para a dificuldade de gerenciamento dos sistemas computacionais atuais e apontou a autonomia dos sistemas como a alternativa mais viável para a solução do problema, muitos trabalhos em AC foram propostos com este objetivo. Neste trabalho, apresentamos uma revisão de algumas das propostas mais relevantes da literatura em questão, dando particular importância ao aspecto arquitetural e de infra-estrutura adotados por tais soluções. A revisão destas soluções nos permitiu as seguintes conclusões:

- De fato, computação autônoma é um grande desafio, pois requer conhecimentos em diversos campos, especialmente composição dinâmica, arquitetura de software, modelagem de sistemas computacionais, governança, engenharia de software e inteligência artificial (planos, aprendizagem, representação de conhecimento, etc). Portanto, a realização plena da visão de AC exige uma combinação efetiva entre arquiteturas e infra-estrutura. Entretanto, como mostrado nete trabalho, nenhuma arquitetura proposta atingiu plenamente esta visão, já que apenas algumas propriedades de auto-gerenciamento são efetivamente suportadas.

- Cooperação e interação são conceitos fundamentais em AC, uma vez que entidades maiores, inclusive o próprio sistema, só são possíveis através da composição de entidades menores. Neste sentido, um requisito essencial consiste em padronização. Entretanto, como observado nas arquiteturas apresentadas neste trabalho, a área de AC ainda sofre com soluções amplamente *ad-hoc*, em que diferentes infra-estruturas e paradigmas são aplicados.

- Políticas são essenciais para construção de sistemas autônomos, uma vez que elas constituem a forma na qual os administradores expressam seus objetivos para o sistema. Grande parte das arquiteturas apresentadas neste trabalho utilizam políticas baseadas em ações, em que objetivos são expresso diretamente através de ações que são executadas quando uma determinada condição é observada. Contudo, para atingirmos completamente a visão de AC, representações de mais alto nível são necessárias.

- Finalmente, é importante ressaltar que a construção e refinamento de protótipos ou, preferencialmente, aplicações reais, é uma parte essencial no processo de amadurecimento de AC. Neste sentido, muito há para ser explorado, já que poucos trabalhos exercitam cenários complexos de gerenciamento em protótipos ou aplicações reais.

Referências

[1] SHEHORY, O.. **The Role of Agents in Enterprise System Management: a Position Paper**. In: AAMAS '06: PROCEEDINGS OF THE FIFTH INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, p. 1530–1533, New York, NY, USA, 2006. ACM.

[2] **Autonomic Computing: IBM Perspective on the State of Information Technology, IBM T.J. Watson Labs, NY, 15th October 2001. Presented at AGENDA 2001, Scotsdale, AR.**, 2001. Disponível em http://www.research.ibm.com/autonomic/.

[3] HUEBSCHER, M. C.; MCCANN, J. A.. **A Survey of Autonomic Computing—Degrees, Models, and Applications**. ACM Computing Surveys, 40(3):1–28, 2008.

[4] KANDASAMY, N.; ABDELWAHED, S. ; HAYES, J.. **Self-optimization in Computer Systems via On-line Control: Application to Power Management**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'04), p. 54–61, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[5] KHARGHARIA, B.; HARIRI, S. ; YOUSIF, M.. **Autonomic Power and Performance Management for Computing Systems**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'06), p. 145–154, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[6] AGARWALA, S.; CHEN, Y.; MILOJICIC, D. ; K.SCHWAN. **QMON: QoS- and Utility-aware Monitoring in Enterprise Systems**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'06), p. 124–133, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[7] PARASHAR, M.; KLIE, H.; CATALYUREK, U.; KURC, T.; BANGERTH, W.; MATOSSIAN, V.; SALTZ, J. ; WHEELER, M.. **Application of Grid-enabled Technologies for Solving Optimization Problems in Data-driven Reservoir Studies**. Future Gener. Comput. Syst., 21(1):19–26, 2005.

[8] ZENMYO, T.; YOSHIDA, H. ; KIMURA, T.. **A Self-healing Technique based on Encapsulated Operation Knowledge**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'06), p. 25–32, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[9] BRODIE, M.; MA, S.; LOHMAN, G.; SYEDA-MAHMOOD, T.; MIGNET, L. ; MODANI, N.. **Quickly Finding Known Software Problems via Automated**

Symptom Matching. In: PROCEEDINGS OF THE SECOND INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'05), p. 101–110, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[10] MCCANN, J. A.; HUEBSCHER, M. C. ; HOSKINS, A.. **Context as Autonomic Intelligence in a Ubiquitous Computing Environment**. International Journal of Internet Protocol Technology (IJIPT) Special Edition on Autonomic Computing, 2(1):30–39, 2007.

[11] **An Architectural Blueprint for Autonomic Computing**. Technical report, IBM, 2003.

[12] STERRITT, R.; BUSTARD, D.. **Autonomic Computing- A Means of Achieving Dependability?** In: PROCEEDINGS OF IEEE INTERNATIONAL CONFERENCE ON THE ENGINEERING OF COMPUTER BASED SYSTEMS (ECBS'03), p. 247–251, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[13] KEPHART, J. O.; CHESS, D. M.. **The Vision of Autonomic Computing**. IEEE Computer, 36(1):41–50, 2003.

[14] GANEK, A.; CORBI, T. A.. **The Dawning of the Autonomic Computing Era**. IBM Syst. J., 42(1):5–18, 2003.

[15] PARASHAR, M.; HARIRI, S.. **Autonomic Computing: An Overview**. In: UNCONVENTIONAL PROGRAMMING PARADIGMS, INTERNATIONAL WORKSHOP UPP 2004, LE MONT SAINT MICHEL, FRANCE, SEPTEMBER, volumen 3566 de **Lecture Notes in Computer Science**, p. 257–269. Springer, 2004.

[16] MCCANN, J. A.; HUEBSCHER, M. C.. **Evaluation Issues in Autonomic Computing**. In: PROCEEDINGS OF GRID AND COOPERATIVE COMPUTING WORKSHOP, p. 597–608. Springer Berlin/Heidelberg, 2004.

[17] WHITE, S. R.; HANSON, J. E.; WHALLEY, I.; CHESS, D. M. ; KEPHART, J. O.. **An Architectural Approach to Autonomic Computing**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'04), p. 2–9, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[18] TESAURO, G.; CHESS, D. M.; WALSH, W. E.; DAS, R.; SEGAL, A.; WHALLEY, I.; KEPHART, J. O. ; WHITE, S. R.. **A Multi-agent Systems Approach to Autonomic Computing**. In: AAMAS '04: PROCEEDINGS OF THE THIRD INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, p. 464–471, Washington, DC, USA, 2004. IEEE Computer Society.

[19] BIGUS, J. P.; SCHLOSNAGLE, D. A.; PILGRIM, J. R.; III, W. N. M. ; DIAO, Y.. **ABLE: A Toolkit for Building Multiagent Autonomic Systems**. IBM Syst. J., 41(3):350–3718, 2002.

[20] LIU, H.; PARASHAR, M. ; HARIRI, S.. **A Component Based Programming Framework for Autonomic Applications**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'04), p. 10–17, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[21] BHAT, V.; PARASHAR, M.; LIU, H.; KHANDEKAR, M.; KANDASAMY, N. ; ABDELWAHED, S.. **Enabling Self-Managing Applications using Model-based Online Control Strategies**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON AUTONOMIC COMPUTING (ICAC'06), p. 15–24, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[22] KAISER, G.; PAREKH, J.; GROSS, P. ; VALETTO, G.. **Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems**. In: AUTONOMIC COMPUTING WORKSHOP AT THE FIFTH ANNUAL INTERNATIONAL WORKSHOP ON ACTIVE MIDDLEWARES SERVICE (AMS'03), p. 22–30, 2003.

[23] PARK, J.; YOO, G. ; LEE, E.. **Proactive Self-healing System based on Multi-Agent Technologies**. In: SERA '05: PROCEEDINGS OF THE THIRD ACIS INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS, p. 256–263, Washington, DC, USA, 2005. IEEE Computer Society.

[24] EYMANN, T.. **The Infrastructures of Autonomic Computing**. The Knowledge Engineering Review, 21(3):189–194, 2006.

[25] PHAN, T.; HAN, J.; SCHNEIDER, J.-G.; EBRINGER, T. ; T.ROGERS. **A Survey of Policy-Based Management Approaches for Service Oriented Systems**. In: 19TH AUSTRALIAN CONFERENCE ON SOFTWARE ENGINEERING (ASWEC 2008), p. 392 – 401, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[26] ZANIKOLAS, S.; SAKELLARIOU, R.. **A Taxonomy of Grid Monitoring Systems**. Future Gener. Comput. Syst., 21(1):163–188, 2005.

[27] MCKINLEY, P. K.; SADJADI, S. M.; KASTEN, E. P. ; CHENG, B. H.. **Composing Adaptive Software**. Computer, 37(7):56–64, 2004.