Capítulo

# 4

# Computação Autônoma: Conceitos, Infra-estruturas e Soluções em Sistemas Distribuídos

Sand Corrêa, Renato Cerqueira
Departamento de Informática – PUC-Rio

***Abstract***

*The growth in size and complexity of today's computer system makes them unmanageable by human beings. The main reason for that is the large interrelationship among theirs inumerous hardware and software components. Autonomic Computing deals with this issue by moving many administrative tasks to the own system, which generates softwares that are able to manage themselves based on high-level guidelines. This chapter presents an overview about state of art in Autonomic Computing in the context of Distributed Systems, tackling the main features and challenges related to the development of self-management distributed systems and applications. It is presented the motivations for this novel computing paradigm, basic principles, proposed architectures, and technological infrastructures and middleware services. It is also presented the main recent proposals in the literature, research challenges and future trends.*

***Resumo***

*O crescimento do tamanho e da complexidade dos sistemas computacionais atuais, dado o grande inter-relacionamento dos diversos elementos de software e hardware que os compõem, resulta na inevitável incapacidade humana de os gerenciarem. Computação Autônoma trata tal problema transferindo-se grande parte das responsabilidades administrativas para o próprio sistema, criando-se assim softwares capazes de se auto-gerenciarem a partir de diretivas de alto nível. Este capítulo apresenta uma visão geral do estado da arte em Computação Autônoma em Sistemas Distribuídos, abordando as principais características e desafios envolvendo a construção de sistemas e aplicações distribuídas auto-gerenciáveis. São apresentadas as motivações para este novo paradigma de computação, seus princípios básicos, arquiteturas propostas e as infra-estruturas tecnológicas e de serviços (middleware) necessárias para a sua completa realização. Também são apresentadas as principais soluções propostas atualmente na literatura, desafios ainda não superados e tendências futuras.*

## 4.1. Introdução

O surgimento de ambientes computacionais distribuídos de alto desempenho e grande escala, tais como os sistemas de informação pervasiva e grades computacionais, propiciou uma nova geração de aplicações científicas e de negócios cujas características principais incluem dinamismo e complexidade [Liu et al. 2004]. A primeira característica está relacionada principalmente à ubiqüidade e alta escala, que tornam o ambiente de execução e a necessidade de recursos da aplicação variáveis ao longo do tempo. A segunda deve-se, em grande parte, a uma arquitetura de software orientada a serviços [Papazoglou and Georgakopoulos 2003] e composta por inúmeros elementos de software e hardware distribuídos geograficamente. Esses serviços são heterogêneos em suas funcionalidades, interfaces com outros serviços e usuários humanos, além de serem fornecidos por diferentes fabricantes.

Adicionalmente, requisitos de qualidade de serviço (QoS) e emergência também constituem características importante dessas aplicações. Uma preocupação crucial em ambientes orientados a serviços consiste na satisfação de requisitos estritos de qualidade envolvendo, por exemplo, corretude, desempenho, confiabilidade, dentre outros.

Emergência, por sua vez, refere-se a comportamentos não antecipados, já que, tradicionalmente, sistemas distribuídos estão sujeitos a alterações de comportamento provenientes de falhas de recursos (as quais se tornam mais freqüentes à medida que o sistema cresce em escala), heterogeneidade e concorrência, bem como carga variável no tempo, a qual pode levar a modelos imprecisos de desempenho. No entanto, a noção de composição por serviços, inerentes à arquitetura, introduz alguns problemas adicionais para a determinação do comportamento do sistema. Dentre eles ressalta-se, primeiramente, a dificuldade de antecipar, em tempo de projeto, todas as configurações em que os serviços serão utilizados ao longo do ciclo de vida da aplicação, especialmente em sistemas de grande escala. Igualmente penosa é a atividade de prever todas as interações possíveis entre os serviços em questão. Por fim, a dificuldade de monitoramento completo é outro problema que afeta a determinação do comportamento do sistema, já que, como unidades independentes de implantação, cada serviço pode prover práticas de gerenciamento específicas e consideravelmente diferentes entre si.

Juntas, essas características impõem grandes desafios à administração dos sistemas de informação atuais e futuros. De fato, estudos [Salehie and Tahvildari 2005] demonstram que aproximadamente metade dos recursos destinados à tecnologia de informação são gastos em manutenções preventivas, recuperações de falhas e localização e determinação de problemas.

Tendo esse cenário como motivação, em outubro de 2001, um manifesto produzido pela IBM [Horn 2001] alertava para a dificuldade do gerenciamento de softwares complexos (tais como o descrito acima) 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[1] (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 humano. Este 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 [White et al. 2004] [Tesauro et al. 2004] [Sterritt and Bustard 2003] [Parashar and Hariri 2004]. Outros trabalhos exploraram AC sob a perspectiva de engenharia de software [Lin and Leaney 2005], métricas de avaliação [McCann and Huebscher 2004], produtos [Salehie and Tahvildari 2005] e desafios [Kephart 2005]. Ainda sobre trabalhos relacionados, é importante ressaltar um capítulo da edição de 2006 desta publicação, [Braga et al. 2006], em que se discute o projeto de redes autonômicas: redes capazes de gerenciarem a si próprias. Esse trabalho apresenta em detalhes os principais aspectos relacionados ao paradigma da computação autônoma, a aplicação desse paradigma para a construção de sistemas gerais e, em particular, de redes de computadores (cabeadas, sem fio e redes de sensores). No entanto, conforme descrito pelos autores, a solução de gerenciamento autônomo deve ser desenvolvida considerando-se os aspectos particulares de cada projeto. Neste sentido, as soluções apresentadas para projetos no nível de rede são inadequadas para tratar auto-gerenciamento em camadas de aplicação.

Dentro desse contexto, este capítulo, apresenta uma visão geral do estado da arte em Computação Autônoma na área de Sistemas Distribuídos. São abordadas as principais características e desafios envolvendo a construção de aplicações distribuídas auto-gerenciáveis e as infra-estruturas tecnológicas e de serviços (*middleware*) necessárias para a sua completa realização. Também são apresentadas as principais soluções propostas atualmente na literatura, desafios ainda não superados e tendências futuras.

O restante deste trabalho está organizado da seguinte maneira. As Seções 4.1.1 e 4.1.2 apresentam, respectivamente, os princípios básicos que regem AC e tipos de aplicações em que seu uso tem sido mais freqüente. A Seção 4.2 aborda os requisitos necessários para construção de sistemas autônomos, analisando-se, em particular, três perspectivas: arquiteturas de software para construção de sistemas autônomos, infra-estruturas de apoio à viabilização dessas arquiteturas e modelos arquiteturais para representação de conhecimento. A Seção 4.3 apresenta o estado da arte das soluções e iniciativas propostas visando sistemas de informação autônomos. Também é apresentado um estudo comparativo dessas soluções, as quais serão analisadas segundo os requisitos descritos na seção anterior. Finalmente, a Seção 4.4 apresenta conclusões sobre o estado atual de maturidade dos sistemas estudados, desafios ainda não superados pela área e tendências futuras.

[1]Os termos "Computação Autônoma" e "Computação Autonômica" são ambos usadas na literatura. Neste trabalho, adotamos o termo Computação Autônoma.

### 4.1.1. Definição e Propriedades

Como mencionado na seção anterior, o conceito envolvendo AC foi inspirado no corpo humano, o qual provê mecanismos efetivos para monitorar, controlar e regular a si próprio 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 [ibm 2003]. Estas determinam o tipo de ação a ser executada em diferentes situações.

A Figura 4.1 resume as propriedades gerais de um sistema autônomo. Conforme ilustrado na figura, o objetivo principal de AC é auto-gerenciamento. Para atingir esse objetivo, entretanto, quatro propriedades são essenciais: auto-cura (*self-healing*), auto-otimização (*self-optimizing*), auto-proteção (*self-protecting*) e auto-configuração (*self-configuring*). Essas propriedades são comumente denominadas propriedades *self*-*.

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 recursos 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.

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*) [Kephart and Chess 2003] [Horn 2001] [Ganek and Corbi 2003]. Esses 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 estado 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 [Parashar and Hariri 2004], 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 com-

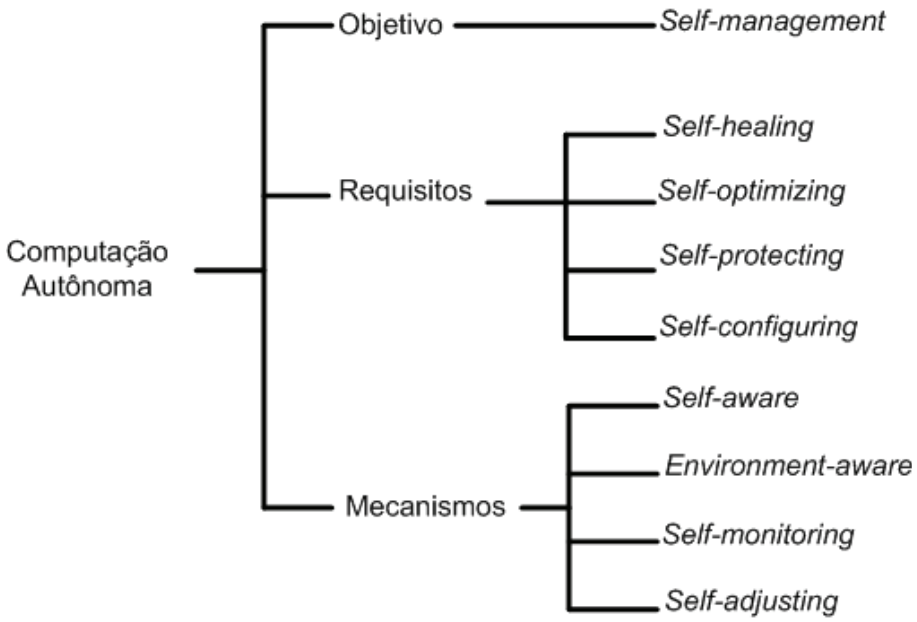


**Figure 4.1. Propriedade Gerais para Computação Autônoma [Sterritt and Bustard 2003].**

portamento, bem como as do ambiente, e agir proativamente diante de tais informações.

Mais recentemente a comunidade de AC tem discutido o que realmente pode ser considerado auto-gerenciamento, uma vez que alguns serviços isolados propiciam momentos em que um sistema pode gerenciar a si próprio. Serviços como esses podem ser encontrados, por exemplo, em um módulo de otimização de consulta de um SGBD (Sistema de Gerenciamento de Banco de Dados), um gerente de recursos de um sistema operacional ou o software de roteamento de uma rede. No entanto, o termo AC é aplicado para designar sistemas em que mudanças de políticas ocorrem para refletir o ambiente de execução corrente. Ou seja, mudanças devem ocorrer dinamicamente. Além disso, para distinguir sistemas autônomos de sistemas puramente adaptativos (como, por exemplo, alguns sistemas multimídia que se adaptam a flutuações da largura de banda disponível), os primeiros devem exibir mais de uma característica de gerenciamento [Huebscher and McCann 2008].

### 4.1.2. Tipos de Aplicações

Nos últimos anos, vários trabalhos têm incorporado propriedades de auto-gerenciamento como forma de obter maior autonomia às aplicações. Em especial, três áreas de aplicação têm concentrado grande parte desses trabalhos: gerenciamento de energia em grandes centros de computação, gerenciamento de ambientes em grades computacionais e computação ubíqua.

A Figura 4.2 ilustra os custos que compõem o TCO (*Total Cost of Ownership*) de um *rack* típico de um *data center* [APC 2003]. Conforme ilustrado na figura, os gastos com eletricidade, condicionamento e refrigeração de equipamentos representam um custo de investimento significativo para os grandes centros de computação. Esse fato 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 [Kandasamy et al. 2004]. 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 [Khargharia et al. 2006].

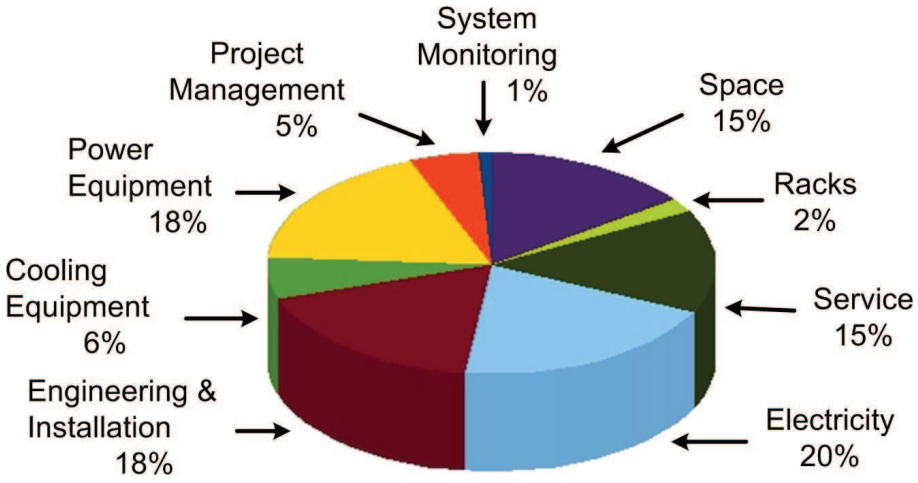


**Figure 4.2. Componentes do TCO de um *data center* típico [APC 2003].**

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 [Parashar et al. 2005] e administração de sistemas [Zenmyo et al. 2006, Brodie et al. 2005]. 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 negociada 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 esse compromisso, mesmo diante do dinamismo 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 nesses ambientes. Para tanto, registros de problemas e reações à adversidades (provenientes da intervenção humanas) são reportados em *logs* de operação. A partir dos *logs* é possível determinar um 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 para a criação de um ambiente inteligente. Portanto, a complexidade de instalação e manutenção de aplicações nesses ambientes, naturalmente conduz à necessidade de sistemas auto-gerenciáveis. Um exemplo de trabalho onde computação autônoma é aplicada

com esse objetivo pode ser encontrado em [McCann et al. 2007].

## 4.2. Requisito para Construção de Sistemas Autônomos

Uma vez apresentados os desafios inerentes à Computação Autônoma, em função da necessidade de assegurar as propriedades *self-**, nesta seção discutem-se os requisitos essenciais para promover sua realização efetiva. De maneira geral, tais requisitos são abordados segundo três perspectivas: arquiteturas de software para construção de sistemas autônomos (Seção 4.2.1), modelos arquiteturais para representação de conhecimento (Seção 4.2.2) e infra-estruturas de apoio (Seção 4.2.3).

### 4.2.1. Arquiteturas de Software 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 4.1.1, AC trata esse 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, essas arquiteturas apresentam soluções para automatizar o ciclo de gerenciamento de sistemas (veja Figura 4.3), o qual envolve as seguintes atividades:

- monitoramento ou medição: função que coleta, agrega, correlaciona e filtra dados sobre recursos gerenciados. Dados coletados incluem: informações de topologia, eventos, métricas, propriedades de configuração, etc. Recursos gerenciados incluem servidores, unidades de armazenamento, banco de dados, servidores de aplicação, serviços, aplicações, etc.
- análise e decisão: a função de análise examina os dados coletados e determina se devem ser feitas mudanças sobre as políticas ou estratégias correntes. Essa tomada de decisão assegura convergência em relação a valores limiares de parâmetros como desempenho, disponibilidade e segurança (geralmente denominados *Service Level Objectives* ou SLOs).
- controle e execução: a função de controle escalona e executa as mudanças identificadas como necessárias pela função de análise e decisão.

Quando esse ciclo de gerenciamento é automatizado, um laço de controle inteligente, ou ciclo de gerenciamento autônomo, é criado de forma a automatizar um conjunto de tarefas comumente executadas por administradores de sistemas. De maneira geral, um ciclo de gerenciamento autônomo pode ser implementado através de dois tipos de arquiteturas [McCann and Huebscher 2004]: as baseadas em elementos autônomos e as baseadas em infra-estrutura.

#### Arquitetura Baseada em Elementos Autônomos

Neste tipo de arquitetura, sistemas auto-gerenciáveis são formados pela colaboração de elementos autônomos. Estes, por sua vez, consistem em módulos de software auto-

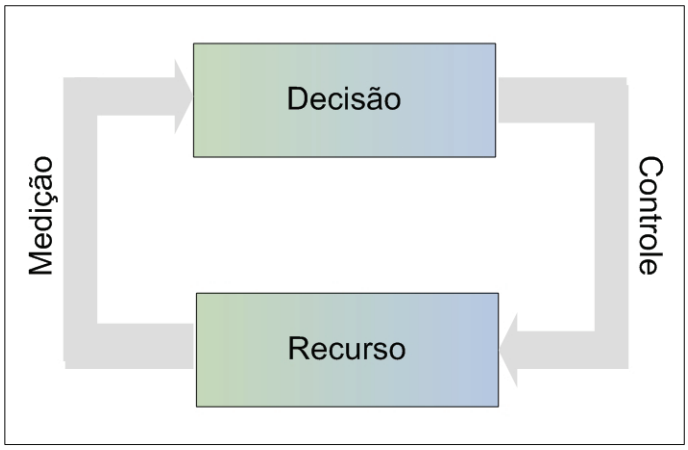


**Figure 4.3. Ciclo de gerenciamento de sistemas.**

contidos com interfaces de interação específicas e dependências de contexto explícita [White et al. 2004]. A Figura 4.4 ilustra um elemento 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. Conforme ilustrado 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 (*effectors*) acoplados ao elemento gerenciado.

Em um elemento autônomo, o módulo de gerenciamento consiste em dois laços de controle denominados MAPE (Monitora-Analisa-Planeja-Executa). O laço de controle local trata apenas estados conhecidos de ambiente, 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 irá repetir continuamente sua política de adaptação local,

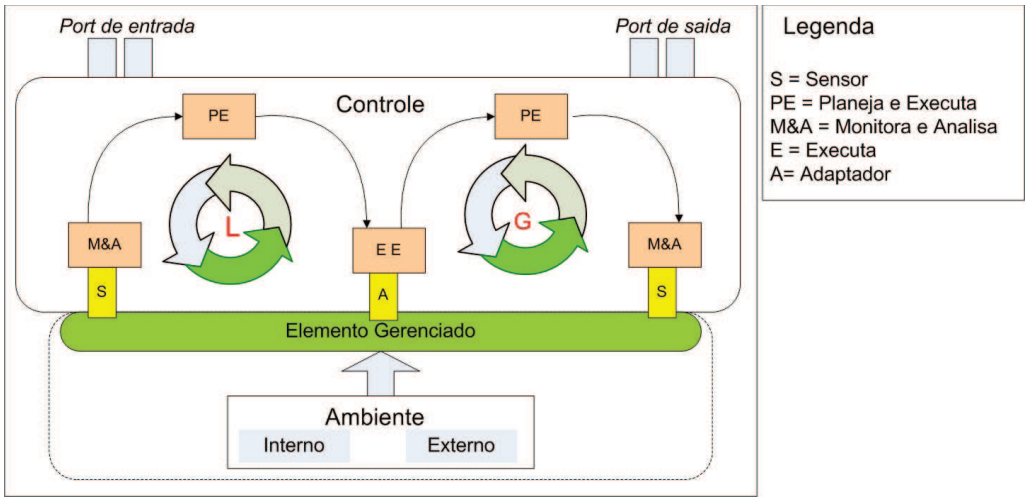


**Figure 4.4. Estrutura de um elemento autônomo [Parashar and Hariri 2004].**

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.

Tradicionalmente, ferramentas de gerenciamento monitoram e controlam recursos através de interfaces de gerenciamento variadas, como arquivos de *log*, eventos, comandos, APIs e arquivos de configuração. O acesso a essas interfaces pode variar consideravelmente de acordo com o tipo de recurso gerenciado ou fabricante, dificultando enormemente o gerenciamento dos recursos. Para evitar problemas dessa natureza, a unidade de controle de um elementos autônomo interage com elementos gerenciados através de interfaces padronizadas denominadas *touchpoint*. Essas interfaces constituem blocos de construção para sensores e adaptadores, determinando o comportamento esperado para esses elementos. As operações usadas para observar informações sobre elementos gerenciados são encapsuladas em sensores, os quais suportam dois estilos de interação com a unidade de controle:

- *Request-Response*: operações que expõem informações (tais como propriedades e relacionamentos) sobre o estado corrente do elemento gerenciado.
- *Send-Notification*: conjunto de eventos que são acionados quando o elemento gerenciado atinge um estado que deve ser reportado.

De maneira similar, as operações usadas para mudar o estado ou comportamento de um elemento gerenciado são encapsuladas em adaptadores, cujos estilos de interação incluem:

- *Perform-Operation*: operações que permitem que o estado ou comportamento de um elemento gerenciado seja alterado.

- *Solicit-Response*: operações que permitem que um elemento gerenciado requisite informações para a sua unidade de controle.

A granulosidade de um elementos gerenciado é dependente do domínio da aplicação. Dessa forma, um servidor, por exemplo, pode ser projetado como um elemento gerenciado composto por recursos de menor granulosidade, como CPU, disco e memória. Neste caso, o bloco *touchpoint* do elemento gerenciado deverá exportar um sensor e um adaptador para cada recurso.

Devido a sua estrutura, um elemento 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 elemento autônomo também deve ser capaz de estabelecer e manter relacionamentos com outros elementos autônomos, em especial, seus provedores ou consumidores de serviço. Esses 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. Dessa forma, um modelo de colaboração interativa natural para esta arquitetura é o modelo *peer-to-peer*. No entanto, é importante ressaltar que esta solução pode levar a um grande volume de troca de mensagens. Uma alternativa, então, consiste em um modelo hierárquico, no qual alguns elementos autônomos gerenciam um grupo de outros. Em ambos os casos, entretanto, a interação entre elementos autônomos é habilitada apenas através de sensores e adaptadores, como mostra a Figura 4.5.

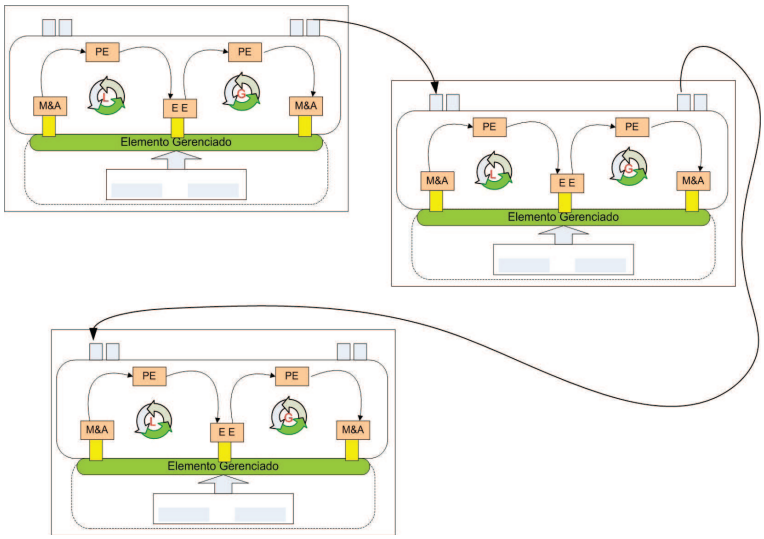


**Figure 4.5. Interação colaborativa entre elementos autônomos.**

Na seção 4.3 são apresentadas algumas aplicações (ou projetos) que implementam uma arquitetura baseada em elementos autônomos. De maneira geral, as arquiteturas de software destes projetos diferem-se essencialmente pela abstração utilizada ao representar um serviço ou conjunto de serviços autônomos. Enquanto alguns projetos implementam

elementos autônomos como agentes inteligentes, outros utilizam a abstração de componentes de software para atingir o mesmo objetivo.

### 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 4.6 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. Essas 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 centralizada (a infra-estrutura concentra os mecanismos que provêem a autonomia do sistema alvo) pode dificultar sua implantação em ambientes distribuídos. Exemplos de aplicações que implementam esse tipo de arquitetura são apresentadas na Seção 4.3.

## 4.2.2. Representação de Conhecimento em AC

Um requisito essencial para a automatização de um ciclo de gerenciamento de sistemas é a existência de conhecimento que governe a forma como as três funções (monitoramento, análise e execução) serão executadas. Nesse contexto, em AC, conhecimento se torna um termo mais amplo que em outras área (como, por exemplo, inteligência artificial), podendo representar qualquer tipo de dado estruturado ou informação que possa ser usada na execução de um processo, especialmente os automatizados. Nesse escopo ampliado, estruturas como arquivos de *log*, dados obtidos a partir de sensores, dados sobre escalonamento de mudanças, etc, são consideradas conhecimento. Tipicamente, três tipos de conhecimento são largamente utilizados em AC:

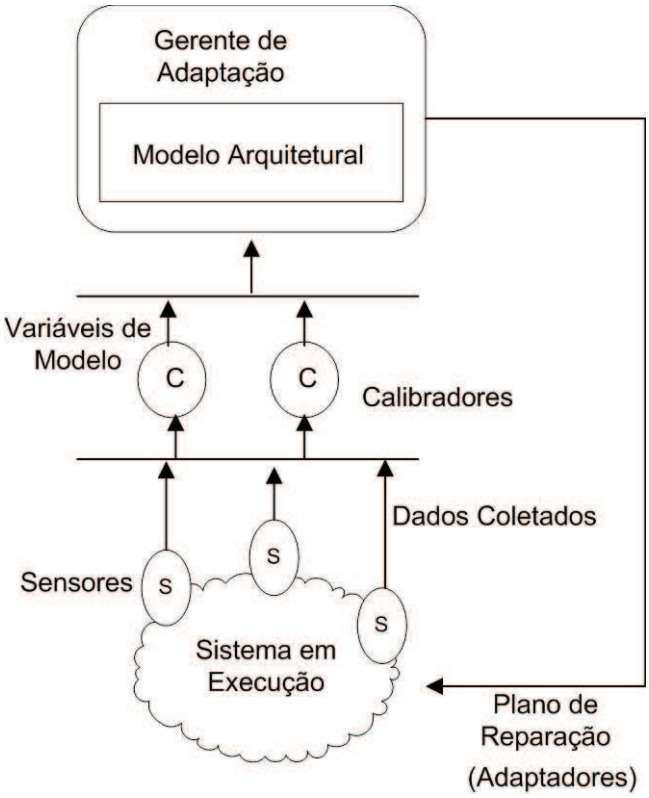


**Figure 4.6. Arquitetura genérica de abordagens baseadas em infra-estrutura [McCann and Huebscher 2004].**

- Conhecimento referente a topologia, como, por exemplo, descritores de entidades
- Conhecimento referente a políticas
- Conhecimento referente a determinação de problemas

A Figura 4.7 representa um ciclo de gerenciamento autônomo permeado por conhecimento. Sintomas são gerados pela função de monitoramento e indicam uma condição associada a um elemento ou recurso gerenciado. A função de análise e decisão correlaciona sintomas a fim de diagnosticar o problema. Uma vez que o problema foi determinado, políticas são acessadas para governar as ações que serão tomadas, gerando-se assim um plano de mudança. Esse plano é executado pela função de controle e execução. É importante ressaltar que, neste contexto, o conhecimento referente a determinação de problemas envolve não só processos de auto-cura mas também auto-proteção, auto-otimização e auto-configuração. A seguir são apresentados modelos de representação para este tipo de conhecimento. Representação de conhecimento referentes a políticas são discutidas na Seção 4.2.3.

Como uma classificação didática dos modelos que têm sido usados para representar conhecimento referente a determinação de problemas, este trabalho propõe o agrupamento nas seguintes categorias:

- Modelos analíticos: incorporam modelos *a priori* da estrutura e do comportamento

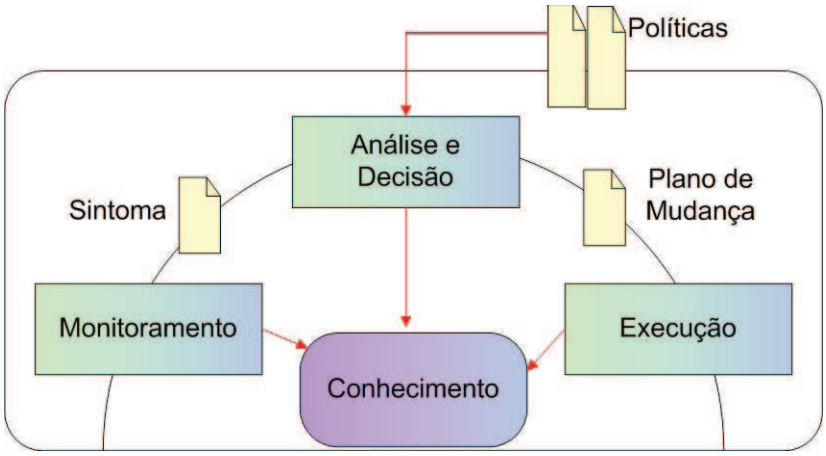


**Figure 4.7. Ciclo de gerenciamento de sistemas permeado por conhecimento**

do sistema, os quais podem ser representados quantitativamente ou como um conjunto de regras do tipo *evento-condição-ação*. Nessa categoria incluem-se modelos clássicos, tais como teoria de filas, redes de petri e teoria de controle. Apesar de estáveis e matematicamente fundamentados, esses modelos podem ser de difícil derivação manual, especialmente para domínios complexos. O processo de modelagem pode requerer um esforço humano substancial, tornando-o sujeito a erros ou mesmos suposições irreais. Adicionalmente, esses modelos são incapazes de operar de forma precisa quando encontram condições não previstas em tempo de projeto.

- Modelos de aprendizado de máquina: recentemente, alguns trabalhos têm explorado o uso de técnicas de aprendizado de máquina como bloco de construção para mecanismos de correlação de dados instrumentados e determinação de problemas. Esses modelos se baseiam principalmente em abordagens de aprendizado estatístico, não assumem envolvimento humano e requerem pouco ou mesmo nenhuma análise dependente do domínio modelado. Nesses modelos, a análise consiste em um processo de aprendizado *offline*, em que modelos do comportamento do sistema são derivados a partir de dados de treinamento disponíveis antecipadamente. Redes bayesianas, árvores de decisão, técnicas de *clustering* e redes neurais são alguns exemplos de modelos desta natureza.

- Modelos de aprendizado *online*: uma grande dificuldade de processos de aprendizado *offline* consiste na natureza não incremental do aprendizado. Em muitas aplicações, o monitoramento do ambiente e tomada de decisão são processos constantes. Isto exige uma abordagem incremental que atualize continuamente o modelo corrente à medida que mais informações se tornem disponíveis. Contudo, grande parte das técnicas de aprendizado *offline* é projetada para processamento em lote de um grande volume de dados, sendo o conjunto de treinamento lido e processado repetidamente em busca de uma solução ótima. Portanto, abordagens *offline* tornam-se inviáveis para aplicações que necessitam de aprendizado incremental e dinâmico. Algumas abordagens foram propostas na literatura para resolver este problema. De maneira geral, elas podem ser classificadas em dois grupos: aprendizado incremental e algoritmos *online*. A primeira abordagem refere-se a um con-

junto de soluções que adaptam técnicas de aprendizado *offline* para serem aplicadas incrementalmente. Cada passo ou incremento do processo de aprendizado faz uso de estruturas de dados eficientes em termos de consumo de recursos computacionais (especialmente memória). A segunda abordagem refere-se a uma área de pesquisa própria (*Online Algorithms*) e independente da área de aprendizado de máquina. Ambas, porém, compartilham o mesmo objetivo: uma tomada de decisão presente, baseada somente em experiências obtidas no passado.

À medida que estudos em aprendizado de máquina são aplicados na resolução de problemas do mundo real, algumas suposições simplistas, que guiaram a área durante anos, são abandonadas. Dentre tais suposições, destaca-se a hipótese (comumente assumida por técnicas de aprendizado de máquina) que considera que todas as observações ambientais usadas em um processo de aprendizado encontram-se disponíveis antecipadamente. O relaxamento de tal suposição cria o conceito de aprendizado incremental, no qual observações são assimiladas à medida que se tornam disponíveis. Em modelos de aprendizado não incremental, todo o conhecimento sobre o ambiente é induzido de forma monolítica e em um único momento. Como conseqüência, tais modelos apresentam as seguintes características: (1) o conhecimento induzido não pode ser atualizado intermitentemente; (2) o algoritmo de aprendizado realiza buscas extensivas sobre os dados a fim de assegurar que a acurácia do preditor tende para uma solução ótima; (3) privilegia-se a acurácia da solução em detrimento à eficiência; (4) normalmente, o conhecimento adquirido não é organizado de forma a facilitar sua recuperação. Ao contrário, modelos de aprendizado incremental assumem como hipótese fundamental que observações ambientais são obtidas ao longo do processo de aprendizado. Portanto, o ambiente pode mudar ao longo do processo, exigindo-se uma atualização contínua do conhecimento adquirido. Esta necessidade, por sua vez, exige do modelo de aprendizado um compromisso entre acurácia e eficiência. Tais características tornam modelos de aprendizado incremental abordagens promissoras para tomada de decisão em sistemas dinamicamente adaptáveis.

Nesse sentido, também são promissoras algumas técnicas de Algoritmos *Online*. Esta constitui-se em uma área própria e independente da área de Aprendizado de Máquina. O interesse em comum por problemas de tomada de decisão envolvendo incerteza, motivou o levantamento de uma coleção de problemas em Aprendizado de Máquina que podem ser adequadamente tratados por um *framework* de algoritmos *online* [Blum 1996]. Dentre tais problemas, destacam-se os que envolvem aprendizado a partir de exemplos. Em Algoritmos *Online*, o aprendizado a partir de exemplos assume a forma de uma seqüencia consecutiva de rodadas. Em cada rodada, uma pergunta é apresentada ao módulo de aprendizado, o qual deve fornecer uma resposta. Para responder à pergunta, o módulo de aprendizado usa um mecanismo de previsão, denominado hipótese, que funciona como um mapeamento entre conjunto de perguntas e o conjunto de respostas admissíveis. Após prever uma resposta, a resposta correta é apresentada ao módulo de aprendizado. A qualidade da resposta do módulo de aprendizado é aferida por uma função de perda que mede a discrepância entre a resposta dada e a correta. O objetivo final do módulo de aprendizado é minimizar a perda acumulada ao longo de sua execução. Para atingir tal objetivo, o módulo de aprendizado deve atualizar sua hipótese após cada rodada. Dessa forma, cada rodada pode ser dividida em três etapas: primeiramente o algoritmo recebe uma instância (pergunta); em seguida, o algoritmo prevê um rótulo para a instân-

cia (resposta), baseando-se em uma hipótese; finalmente, o rótulo (resposta correta) é apresentado ao algoritmo que, então, atualiza sua hipótese [Shalev-Shwartz 2007].

Um algoritmo *online* produz uma seqüencia de hipótese $f = (f_1, ..., f_m)$. $f_1$ é uma hipótese inicial arbitrária e $f_i$, para $i > 1$, é a hipótese escolhida após a análise dos $(i-1)$ exemplos anteriores. $l(f_t(x_t), y_t)$ é a função de perda que descreve o erro do algoritmo ao prever $y_t$, baseando-se em $x_t$ e os exemplos anteriores $(x_1, y_1), ..., (x_{t-1}, y_{t-1})$. Em geral, algoritmos *online* não assumem qualquer suposição estatística sobre a seqüencia de dados, a qual pode ser determinística, estocástica ou mesmo arbitrariamente adaptativa. A maior dificuldade destas abordagens, no entanto, consiste na realimentação contínua de rótulos (ou apresentação da resposta correta), já que, em muitos problemas, não é possível fornecer tal realimentação de forma precisa e em um futuro próximo.

A Seção 4.3 revisa vários trabalhos da literatura que fazem uso dos três modelos para representar conhecimento referente a determinação de problemas.

### 4.2.3. 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 discute-se a infra-estrutura necessária para a construção de arquiteturas autônomas. Como proposto em [Eymann 2006], a complexidade é tratada 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, discute-se cada dimensão dentro de uma perspectiva *top-dow*, ou seja, partindo-se da infra-estrutura de mais alto nível.

#### 4.2.3.1. 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 [Phan et al. 2008]. 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 e diretas de regras que tomam a forma *evento-condição-ação*. Esse tipo de representação tem sido usado intensamente no gerenciamento de sistemas distribuídos tradicionais. Um exemplo que merece destaque é a linguagem Ponder para especificação de regas. Entretanto, uma dificuldade neste tipo de representação é a possibilidade de ocorrência de conflitos

entre políticas, os quais podem ser difíceis de serem detectados, especialmente quando muitas regras são especificadas. Políticas baseadas em objetivos são especificações com maior nível de abstração que especificações baseadas em ações, já que definem situações desejadas sem, no entanto, especificar como obtê-las. Em tempo de execução, planos são gerados contendo as ações a serem tomadas para atingir o objetivo esperado. Por esse motivo, políticas baseadas em objetivo exigem mais recursos para serem processadas. Em geral, sistemas multi-agentes utilizam largamente esse tipo de representação. 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. Uma maneira de resolver esse problema consiste em representar políticas como funções de utilidade. Essas são capazes de definir um nível quantitativo de satisfação para cada estado. No entanto, um problema associado a funções de utilidade é a dificuldade em definí-las, uma vez que cada aspecto que influencia a decisão da função deve ser quantificado. Funções de utilidade têm sido intensamente exploradas em aplicações envolvendo gerenciamento e alocação de recursos.

Políticas são criadas e mantidas por infra-estruturas de gerenciamento, as quais devem incluir: um mecanismo de especificação de políticas, um mecanismo de ativação de ações baseado no contexto corrente e um modelo de distribuição e implantação de políticas em pontos específicos de controle. Tipicamente, políticas podem ser definidas usando-se linguagens de especificação, modelos de informação, representações lógicas ou regras escritas em XML. 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. Esses 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.

## Especificação de Políticas

Como mencionado anteriormente, políticas podem ser especificadas através de linguagens de especificação, modelos de informação, representações lógicas ou regras escritas em XML. Linguagens de especificação fornecem uma gramática adjacente na qual é possível definir um conjunto de abstrações para criar e gerenciar políticas. Idealmente, tais linguagens devem ser ricas semanticamente, provendo noções como tipos básicos (autorização, obrigação, delegação), composição de políticas (através do uso de papéis e relacionamentos) e abstrações para organizar hierarquicamente um conjunto de objetos e restringir e controlar o reforço de uma política em tempo de execução. Devem também ser de fácil extensão, permitindo a criação de novos tipos de políticas a partir de regras já existentes. Como exemplo de linguagem de especificação de políticas destaca-se Ponder.

Uma alternativa para a notação de políticas é o uso de modelos de informação. Nesse caso, políticas são representadas através de modelos UML. No top da hierarquia, um conjunto de classes fornece um *framework* conceitual e extensível para descrever o esquema associado ao ambiente gerenciado. Semânticas refinadas também podem ser suportadas, no entanto, com maior dificuldade de manutenção que em linguagens de especificação. Nos modelos de informação, cada componente da política é representado como uma classe e, portanto, expressões complexas levam a um grande número de classes manuseadas. Como conseqüência, o mecanismo de especificação se torna lento e verborrágico, dificultando, dessa forma, o gerenciamento dos repositórios de políticas. Por esse motivo, o uso de modelos de informação pode ser uma boa escolha para mapeamento de políticas com baixo nível de abstração (como, por exemplo, políticas dirigidas a dispositivos) mas se torna menos favorável a políticas de alto nível. Um exemplo de padrão que utiliza esse tipo de representação de política é o PCIM (*Policy Core Information Model*).

É possível também definir políticas através de representações lógicas. Esse tipo de definição está fundamentalmente ligada à segurança e ao controle de conflito. Representações lógicas são livres de ambigüidade e são mais adequadas para a checagem automática de consistência. No entanto, por serem abordagens formais, representações lógicas são pouco intuitivas e não mapeiam facilmente para mecanismos de implementação.

Por fim, existem também várias iniciativas que utilizam XML como mecanismo de definição de políticas, tirando proveito da simplicidade e interoperabilidade da linguagem. No entanto, esse tipo de representação sofre pela verbosidade demasiada.

## Arquiteturas de Gerenciamento baseadas em Políticas

Tradicionalmente, redes de comunicação e sistemas distribuídos têm sido guiados por arquiteturas baseadas em políticas. Dentre essas arquiteturas, o modelo IETF/DMTF, proposta pela *Engineering Task Force (IETF) / Distributed Management Task Force (DMTF)*, tornou-se uma arquitetura de referência, sendo a mesma comporta por quatro componentes principais: Ferramenta de Gerenciamento (permite que administradores definam as políticas a serem utilizadas), Repositório de Políticas (utilizado para armazenar as políticas geradas), Ponto de Decisão de Política (ponto intermediário responsável por se comunicar-se com o repositório, recuperar e interpretar as políticas armazenadas e tomar decisões baseadas nestas) e Pontos de Reforço (mecanismo capaz de aplicar e executar as políticas provenientes do ponto de decisão. No modelo IETF/DMTF, políticas podem ser representadas usando-se Ponder ou PCIM.

Em 2005, um plataforma genérica de gerenciamento baseado em políticas e especialmente projetada para sistemas autônomos foi introduzida pela IBM. Essa plataforma, denominada PMAC (*Policy Management for Autonomic Computing*), oferece apoio de gerenciamento a arquiteturas baseadas em elementos autônomos e os recursos gerenciados por estes. PMAC é implementado em Java e fornece dois tipos de linguagem de especificação de políticas. A primeira, denominada ACPL (*Autonomic Computing Policy Language* é baseada em XML, e, portanto, verborrágica. A segunda linguagem é denominada SPL (*Simplified Policy Language*) e consiste em uma notação concisa e amigável.

#### 4.2.3.2. 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 transparência de localização, 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**

Monitoramento consiste no ato de coletar informações relacionadas a características e estados de objetos de interesse. Particularmente, em AC, monitoramento consiste na captura de propriedades do ambiente que são significativas para os requisitos de autogerenciamento. Conforme exposto em [Zanikolas and Sakellariou 2005], um mecanismo geral para monitoramento de sistemas distribuídos envolve os seguintes componentes:

- entidade: consiste em um recurso útil e único do sistema, tendo um tempo de vida considerável e de uso geral. Exemplo de entidades incluem processadores, memórias, meios de armazenamento, processos, etc.
- evento: coleção de dados tipados e com estampa de tempo, sendo a mesma associada a uma entidade. A estrutura tipada de cada evento, bem como sua semântica, é capturada em um modelo denominado esquema. Um esquema garante o mapeamento de qualquer evento para um estrutura de dados e sua interpretação semântica.
- sensor: processo que monitora uma entidade, gerando eventos. Sensores são classificados em passivos e ativos. Os primeiros usam medições geralmente disponibilizadas por facilidades de sistemas operacionais, enquanto que os últimos estimam medidas usando-se *benchmarks* customizados, e, portanto, intrusivos.

O monitoramento consiste em um processo de quatro etapas: (1) geração de eventos - sensores consultam entidades e codificam as medições obtidas de acordo com o esquema definido; (2) processamento de eventos - esta etapa é dependente da aplicação e pode incluir atividades como filtragem ou sumarização; (3) distribuição - transmissão dos eventos gerados para as entidades interessadas; (4) consumo, onde as entidades interessadas utilizam os eventos que lhes foram notificados.

É importante ressaltar, no entanto, que o projeto de mecanismos de monitoramento deve atender os seguintes requisitos:

- escalabilidade: mecanismos de monitoramento devem ser projetados de forma escalável, assegurando que o aumento do número de recursos, eventos e usuários não irá comprometer o desempenho do sistema monitorado. Em geral, escalabilidade é obtida como resultado de um bom desempenho e baixo grau de intromissão. O primeiro refere-se à garantia de que o sistema atingirá a vazão necessária dentro de um tempo de resposta aceitável em diferentes cenários de carga. O segundo

refere-se à sobrecarga imposta pelo mecanismo de monitoramento às entidades monitoradas. Em geral, esta sobrecarga é medida calculando-se a utilização de recursos (da máquina e da rede) gerada para atender as demandas do mecanismo de monitoramento durante a coleta, processamento e distribuição dos eventos.

- extensibilidade: mecanismos de monitoramento devem ser extensíveis em relação aos recursos monitorados e os eventos gerados por estes. No entanto, para ser extensível, um mecanismo de monitoramento deve utilizar um modelo de codificação de eventos também extensível e auto-descritivo. Além disso, o esquema de eventos deve ser capaz de tratar modificações de forma dinâmica e controlada. Similarmente, os protocolos de interação entre produtores e consumidores de eventos utilizados pelo mecanismo devem ser capazes de acomodar novos tipos de eventos.

- diversidade de modelos de coleta e publicação de dados: mecanismos de monitoramento podem tratar eventos estáticos ou dinâmicos. O padrão de medição (por exemplo, coleta em intervalos de tempos regulares ou por demanda) também pode variar enormemente em função do tipo de evento monitorado. Portanto, uma característica desejável em mecanismos de monitoramento é o uso de diferentes políticas de medição de dados. Adicionalmente, devido ao grande volume de dados gerados por processos de monitoramento, uma característica desejável para sensores é adaptabilidade. 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. Esse ajuste, no entanto, não pode comprometer a consistência das informações monitoradas. Dessa forma, o mecanismo de monitorameto deve prover diferentes modelos de publicação de dados, os quais serão escolhidos em função do tipo de evento e o contexto corrente.

- portabilidade: portabilidade em mecanismos de monitoramento está associada principalmente aos sensores, os quais devem monitorar diferentes tipos de recursos de forma independente de plataforma. Entretanto, portabilidade também se aplica aos eventos gerados, uma vez que métricas devem ser encapsuladas de maneira independente de plataforma.

- relógio global: como definido anteriormente, eventos consistem em uma coleção cronológica de dados tipados. Nessa coleção, a estampa de tempo determina quão recente é um evento. Portanto, a noção de um relógio global é essencial para mecanismos de monitoramento.

De maneira geral, os componentes de um mecanismo de monitoramento podem ser vistos como produtores e consumidores de eventos. Dessa forma, um requisito importante é a forma como ocorre a interação entre esses dois componentes. Tipicamente dois modelos de interação se destacam: *pull* e *push*. No primeiro modelo, o consumidor utiliza um padrão de comunicação do tipo requisição-resposta para obter eventos a partir dos produtores, enquanto que no segundo modelo eventos são difundidos ativamente pelos produtores para todos os consumidores interessados.

Comumente, algumas entidades podem desempenhar tanto o papel de produtor quanto de consumidor para alguns eventos. Nesse caso, tais entidades desempenham um

papel de mediadores entre os produtores originais do evento e seus consumidores finais, executando atividades como filtragem, sumarização, *broadcast* ou *cache*.

### Adaptação de Contexto

Adaptação de contexto 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 [McKinley et al. 2004].

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. Um bom exemplo de adaptação paramétrica ocorre no TCP. Esse protocolo ajusta seu comportamento modificando-se os valores que controlam a janela de gerenciamento e retransmissão em resposta a um possível congestionamento na rede. Entretanto, adaptação paramétrica possui uma limitação: a incapacidade de adoção de estratégias de adaptação que não foram inicialmente projetadas. Portanto, adaptação paramétrica permite o redirecionamento para uma estratégia diferente, porém, existente. Novas estratégias não podem ser adotadas.

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.

Grande parte dos trabalhos envolvendo software adaptativos concentra-se em *middleware*, uma camada de serviços que separa aplicações de sistemas operacionais e protocolos de rede. Muitos *middlewares* adaptativos se baseiam em soluções orientados a objetos tradicionais, como, por exemplo, CORBA, Java RMI e DCOM. Além disso, muitas abordagens de *middlewares* adaptativos se apóiam em técnicas que envolvem algum nível de indireção na interação entre as entidades da aplicação. A tabela 4.1 ilustra algumas técnicas com esse objetivo. Conforme indicado na tabela, algumas abordagens usam padrões de projeto para implementar a indireção, enquanto outras usam o conceito de aspectos ou reflexão computacional (veja Seção 4.2.3.3). Duas técnicas de *middleware* são apresentadas na tabela: interceptação e integração. Ambas introduzem modificações na interação entre as partes envolvidas. Entretanto, a interceptação é uma técnica transparente para a aplicação, enquanto que na integração a aplicação explicitamente faz uma chamada aos serviços de adaptação.

O termo *composer* é usado para designar a entidade que usa técnicas, como as descritas na tabela 4.1, para adaptar o comportamento de uma aplicação. Particularmente, em AC, é desejável que, cada vez mais, *composers* correspondam a módulos de software.

| | |
|---|---|
| Ponteiro para função | Caminhos de execução são dinamicamente redirecionados através de ponteiros para função. (Exemplos: Vtables em COM, funções de *callback* em CORBA) |
| *Proxies* | *proxies* são usados no lugar de objetos da aplicação e redirecionam chamadas de métodos para diferentes implementações. (Exemplos: ACT, AspectlX) |
| *Strategy* e *virtual pattern* | padrões de projeto que encapsulam implementações, possibilitando a substituição transparente de uma implementação por outra. (Exemplos: Dynamic TAO, ACE) |
| Protocolo de metaobjetos | mecanismo que permite interceptação e introspecção para modificação de comportamento. (Exemplos: OpenCOM, OpenJava, OpenORB) |
| Aspectos | fragmentos de código que implementam requisitos não funcionais (que atravessam toda a aplicação) são combinados com a aplicação dinamicamente. (Exemplo: AscpectJ) |
| Interceptação por *middleware* | chamadas de método e respostas que passam por uma camada de *middleware* são interceptadas e redirecionadas. (Exemplo: CORBA) |
| *Middleware* integrado | aplicações fazem chamadas explicitas a serviços de adaptação fornecidos por uma camada de *middleware*. (Exemplos: Adaptive Java, Orbix) |

**Table 4.1. Técnicas de adaptação de software [McKinley et al. 2004].**

#### 4.2.3.3. 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. Essa 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.

## 4.3. Projetos e Aplicações Autônomas

O objetivo desta seção é capturar o estado da arte de projetos e soluções propostas para Computação Autônoma em Sistemas Distribuídos, discutindo-se suas características principais. As propostas são apresentadas segundo o modelo de representação de conhecimento utilizado. Dessa forma, a Seção 4.3.1 apresenta projetos que fazem uso de modelos analíticos; a Seção 4.3.2 discute projetos que utilizam modelos de aprendizado de máquina, enquanto que na Seção 4.3.3 são apresentadas propostas que fazem uso de aprendizado *online*. Ao final da seção é apresentado um estudo comparativo das propostas estudadas, analisando-as segundo os seguintes critérios: domínio de aplicação, propriedades *self-** implementadas, tipo de arquitetura de software adotada, infra-estruturas utilizadas, técnica de representação de conhecimento utilizada.

### 4.3.1. Modelos Analíticos

#### 4.3.1.1. Unity

Em Unity [Tesauro et al. 2004], 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, políticas baseadas em objetivos e políticas baseadas em funções de utilidade.

- 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, esses processos são descritos resumidamente.

### 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 4.8 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.

### 4.3.1.2. Accord

Accord [Bhat et al. 2006] [Liu and Parashar 2004] 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 definidas por usuários, visando, assim, melhor atender os requisitos dinâmicos do ambiente de execução.

A Figura 4.9 ilustra a arquitetura de serviços autônomos em Accord. Como

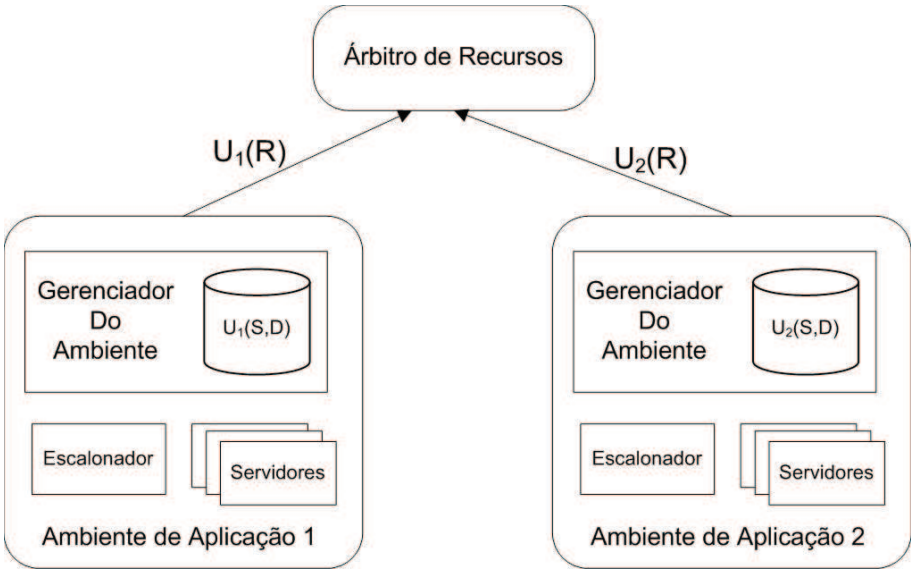


**Figure 4.8. Arquitetura para auto-otimização em Unity [Tesauro et al. 2004].**

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 para serviços individuais. Este processo de decomposição consiste em mapear padrões de *workflow* em templates de regras correspondentes. Alguns templates básicos

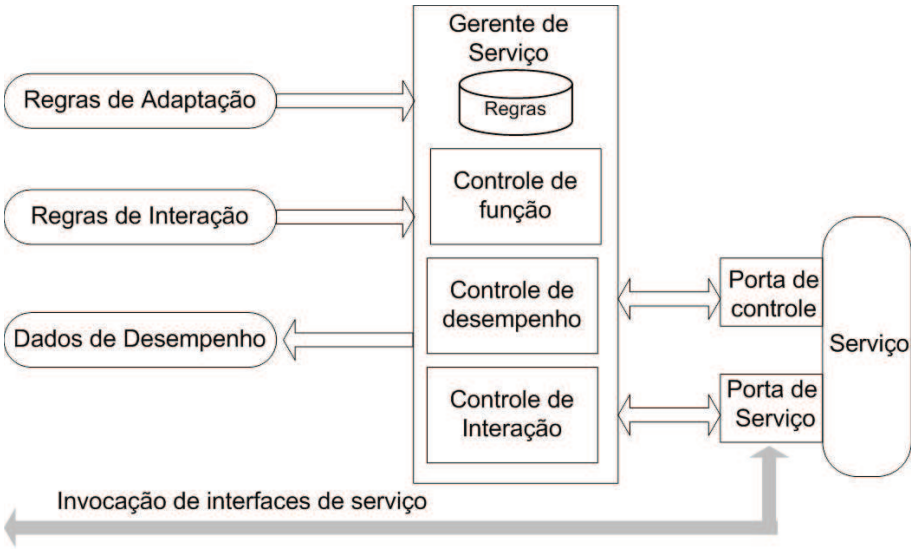


**Figure 4.9. Elemento autônomo em Accord [Bhat et al. 2006].**

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. Nestes, 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*.

#### 4.3.1.3. KX

Em [Kaiser et al. 2003] é introduzida uma arquitetura denominada KX (*Kinesthetics eXtreme*) cujo objetivo é inserir uma infra-estrutura de monitoramento, controle e 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 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).

#### 4.3.1.4. R-Capriccio

R-Capriccio [Zhang et al. 2007] é um *framework* para planejamento da capacidade de sistemas, utilizando-se cargas provenientes de ambientes reais de produção. A partir de tal planejamento, um modelo de utilização de CPU é construído para caracterizar o comportamento normal da CPU em relação à composição de carga do sistema. Anomalias são detectadas comparando-se o comportamento corrente do sistema com o modelo construído. Para isto, R-Capriccio consiste basicamente em três ferramentas: *WorkLoad*

*Profiler*, usada para caracterização da carga do sistema, ou seja os tipos de transações e sessões mais comuns; *Solucionador baseado em Regressão*, ferramenta usada para calcular a demanda de CPU para cada tipo de transação; e um *Modelo analítico*, ferramenta usada para o planejamento efetivo da capacidade do sistema. Estas ferramentas são utilizadas considerando-se as seguintes suposições sobre o sistema:

- O sistema tem uma arquitetura multi-camada
- O gargalo do sistema é a CPU
- Existe uma restrição de tempo de resposta a ser obedecida (um valor limiar de tempo de resposta do sistema)
- Existem logs de aplicações, refletindo todas as requisições e atividades executadas por um cliente.
- É possível mensurar a utilização da CPU em cada camada do sistema.

Para a caracterização da carga, são definidos os conceitos de transação, sessão, capacidade do sistema e tempo de *think*. Uma transação consiste no acesso a uma página Web. Uma sessão consiste num conjunto de transações individuais compondo um serviço. A capacidade do sistema é definida como o número de sessões clientes concorrentes suportadas, sem violar a restrição do tempo de resposta do sistema. O tempo de *think* representa o tempo entre o momento em que o cliente recebe a resposta do servidor e o momento em que ele coloca a próxima requisição. O log contendo os dados da carga é gerado pelo *WorkLoad Profiler*, o qual, em intervalos regulares de tempo, coleta as seguintes informações: a utilização da CPU no período ($U^m_{CPU}$), o número de transações do tipo $i$ executadas no período ($N_i$), o número de sessões concorrentes ($N$) e o tempo médio de *think* ($Z$).

Em seguida, o *Solucionador* calcula o custo (demanda) de CPU ($C_i$) para processar cada tipo de transação $i$, em uma janela de tempo $T$. Para este cálculo, usa-se um método de regressão estatística, tendo como entrada $N_i$, $T$ e $U^m_{CPU}$. Uma vez calculado o valor de cada $C_i$, é possível calcular o valor da utilização de CPU esperada, ou seja, $U^e_{CPU} = (C_0 + \sum_i N_i.C_i)/T$.

O planejamento da capacidade do sistema utiliza um modelo analítico de teoria de filas. Neste modelo, um sistema multi-camada é normalmente modelado como um sistema fechado (o número de clientes é constante). Cada camada é modelada como um conjunto de pares *(servidor, fila)* e a carga é balanceada entre os recursos de uma mesma camada. O tempo de *think* é modelado como um servidor infinito. Uma vez calculado o tempo de serviço de cada fila, em função de $C_i$, é possível resolver o sistema usando-se MVA (*Mean-Value Analysis*). Entretanto, MVA supõe que o tempo de serviço em cada fila é constante. Para resolver este problema, aplica-se MVA em cada janela de tempo $T$ e, posteriormente, os resultados obtidos são combinados entre si.

### 4.3.2. Modelos de Aprendizado de Máquina

#### 4.3.2.1. ABLE

Em [Bigus et al. 2002], 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 encapsulados em componentes JavaBeans e implantados como objetos Java ou agentes autônomos.

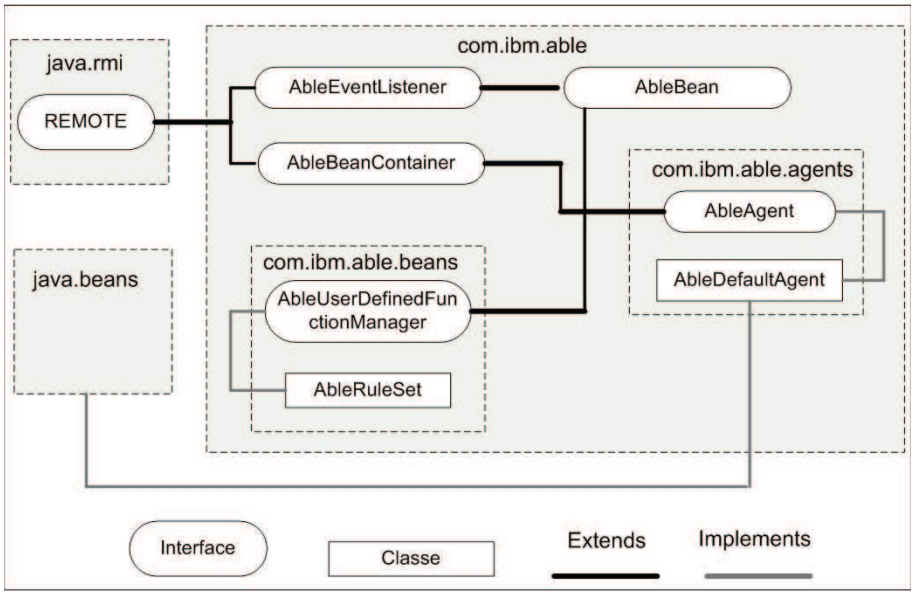


**Figure 4.10. Estrutura do *framework* ABLE [Bigus et al. 2002].**

A Figura 4.10 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*. Esta 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, adaptadores e *AbleBeans* de conhecimento.

Em [Bigus et al. 2002] 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.

#### 4.3.2.2. Park et al.

Em [Park et al. 2005] é apresentada uma arquitetura para AC baseada em infra-estrutura. Como em Unity, a arquitetura 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.

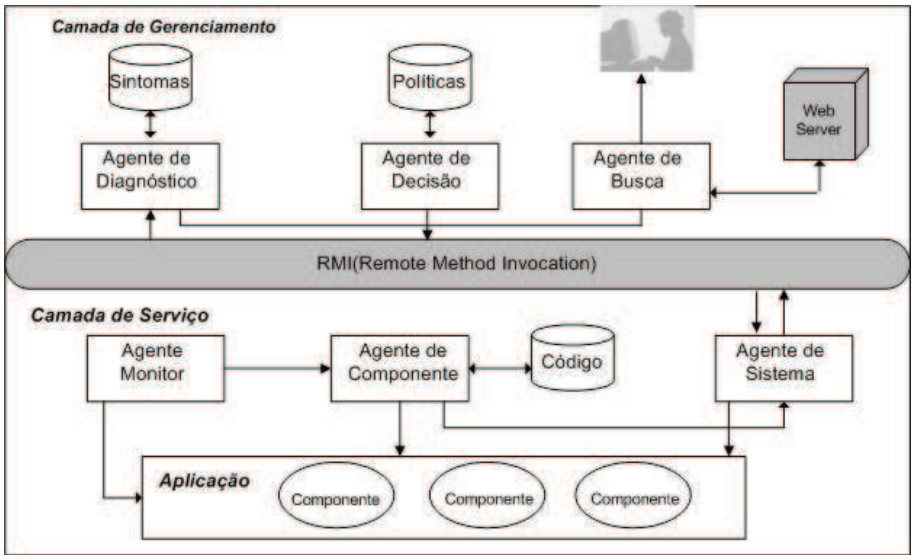


**Figure 4.11. Arquitetura multi-agente que implementa uma abordagem baseada em infra-estrutura [Park et al. 2005].**

A Figura 4.11 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 *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.

#### 4.3.2.3. Magpie

Magpie [Barham et al. 2004] é um protótipo, desenvolvida pela Microsoft, com a finalidade de automatizar o diagnóstico e determinação de problemas de desempenho em sistemas voltados para Internet. A ferramenta foi projetada tendo como meta dois objetivos principais. O primeiro consiste na extração de informações sobre o consumo de recursos e fluxo de controle de cada requisição processada. Neste sentido, Magpie provê informações detalhadas sobre como a requisição foi servida e quanto tempo e recursos foram gastos em diferentes pontos do processamento da requisição. O segundo objetivo consiste no uso das informações de requisições individuais para a construção de modelos apropriados para planejamento de carga, depuração de degradação e detecção de anomalias. Para isto, o protótipo foi implementado como um conjunto de ferramentas e seu funcionamento pode ser dividido em três etapas:

1. Instrumentação: O *framework* de instrumentação de Magpie implementa a contabilização da utilização dos recursos do sistema. Essa contabilização e feita por

requisição e ocorre à medida que eventos são gerados pelos componentes, tanto no espaço de usuários quanto no modo *kernel*. A atribuição dos eventos às requisições correspondentes se apóia na ordenação das estampas de tempo (*timestamp*) de criação dos eventos. Cada evento é representado pela estampa de tempo de geração, um nome e um ou mais atributos que o caracterizam. Normalmente, o *framework* de instrumentação de Magpie é implementado pelo *Event Tracing for Windows* (ETW), o qual é capaz de contabilizar consumo de CPU, acesso a disco e envio/recebimento de pacotes de dados em uma máquina. Instrumentação de aplicações e *middlewares* ocorre quando recursos são multiplexados/demultiplexados ou na transferência do fluxo de controle entre componentes.

2. Extração de carga: a instrumentação gera um *stream* alternado de eventos de diferentes requisições, uma vez que estas ocorrem concorrentemente. Como resultado, o primeiro passo na extração da carga é o agrupamento dos eventos de uma mesma requisição para dar início à contabilização. Uma ferramenta denominada *request parser* é usada com este objetivo. Ela identifica os eventos pertencentes a requisições individuais através da aplicação de uma operação de junção (*join*) temporal sobre os mesmo, de acordo com regras especificadas em um esquema de eventos. Durante este processo, a ordenação causal dos eventos não é modificada.

3. Análise de comportamento: Em seguida, os dados obtidos na extração da carga são analisados para construção dos modelos de predição e depuração. Entretanto, antes da análise proceder, os dados são transformados para uma forma canônica, eliminando-se as informações de como a requisições são servidas. A partir da forma canônica, requisições são serializadas de forma determinística para uma representação de *string*, a qual é utilizada como entrada para um algoritmo de *clustering*. *Strings* similares são agrupadas num mesmo *cluster*, segundo a distância de Levenshtein. *Strings* que não pertencem a nenhum *cluster* suficientemente grande, são consideradas requisições anômalas. Para identificar a causa do problema, Magpie constrói uma máquina de estado probabilística que aceita o conjunto de *strings* representando as requisições possíveis. Requisições anômalas são processadas por esta máquina que identifica todas as transições com baixa probabilidade e os eventos que correspondem a tais transições. Esses eventos são considerados a causa do problema.

Uma grande contribuição de Magpie consiste no *framework* de extração de carga, que relaciona os eventos às requisições sem a necessidade de alocação de identificadores únicos para as requisições.

#### 4.3.2.4. Pinpoint

Pinpoint [Chen et al. 2002] é um *framework* muito semelhante a Magpie. Entretanto, ao contrário deste, Pinpoint se destina especificamente à detecção e localização de falhas em aplicações cliente-servidor, não sendo seu objetivo a detecção de problemas de desempenho. O *framework* funciona da seguinte forma:

1. Pinpoint primeiramente grava *logs* de componentes envolvidos no processamento de requisições. *Logs* são obtidos a partir de sistemas comerciais que provêem instrumentação no nível do *middleware* e de aplicações. Destes *logs*, são extraídos dois tipos de comportamento do sistema: caminhos de requisição e interações de componentes. Ambos fornecem visões diferentes do comportamento do sistema. Como em Magpie, um caminho representa uma seqüencia ordenada dos componentes usados para servir a requisição. Entretanto, ao contrário de Magpie, a associação de um evento à requisição é feita através de identificadores únicos. O comportamento anormal dos caminhos é capturados por uma gramática probabilística que calcula a probabilidade de diferentes seqüências terem sido geradas por uma mesma linguagem. Por outro lado, a análise das interações de componentes se apóia na ponderação de cada *link* que representa uma interação entre o componente e outros componentes do sistema. Nesta estratégia, interações anômalas são percebidas comparando-se a contabilização dos *links* que entram e saem de um componente observado, com os valores correspondentes no modelo construído.

2. Em seguida, Pinpoint determina se cada requisição foi completada com sucesso ou falha. Para isto, ele utiliza detectores de falhas externos (como *ping* para detectar queda de máquina) e detectores de falhas internos (que localizam falhas de algoritmo). Como detectores de falhas internas, Pinpoint oferece os dois métodos de determinação de comportamento descritos no item anterior.

3. Finalmente, Pinpoint utiliza duas técnicas independentes para localização de problemas: *clustering* e árvores de decisão. Ambas as técnicas são utilizadas para encontrar correlações entre a presença de um determinado componente em uma requisição e a falha da mesma.

#### 4.3.2.5. iManage

iManage [Kumar et al. 2007] consiste num *framework* para diagnóstico e determinação de problemas de desempenho, voltados particularmente para sistemas de grande escala. Para isto, iManage estende o trabalho de Cohen et al. [Cohen et al. 2004], propondo a divisão do espaço de estado do sistema em partes menores e, portanto, mais fáceis de serem gerenciadas. O comportamento das partições é então capturado através da construção de micro-modelos, onde uma rede bayesiana é construída para cada partição.

O algoritmo de particionamento do estado do sistema consiste em uma técnica de *clustering*, onde os estados do sistema representados em cada instância de um mesmo grupo são próximos entre si. Considerando-se $S$ o conjunto de estados do sistema, $V$ o espaço de estado e $V_\alpha \subset V$, o conjunto de variáveis que quando modificadas afetam o estado do sistema, a distância entre dois estados $s_1$ e $s_2$ é dado por $v(s_1,s_2) = \eta x \delta_v(s_1,s_2) + \mu x \theta(s_1,s_2)$. $\mu$ e $\eta$ são parâmetros definidos pelo usuário; $\delta$ é a distância em relação à dimensão $V$ e $\theta$ é a distância em relação a dimensão $V_\alpha$. Um espaço de estado $P$ deve ser particionado se existem $V_\alpha$ e $V_\alpha^{'}$ tais que:

- $\sum_{\forall s_i,s_j \in S} \delta_{V_\alpha} - V_\alpha^{'}(s_i,s_j) \leq \Delta_{max}$

- $Cardinalidade(V'_{\alpha}) \leq f$
- $f$ (número máximo de partição) e $\Delta_{max}$ (erro máximo permitido) são definidos pelo usuário

Uma vez particionado o espaço de estado, um micro-modelo é construído para cada partição, com o objetivo de inferir os valores de $V_{\alpha}$, dados os valores de $V - V_{\alpha}$. Para isto, uma rede bayesiana é induzida sobre $\{V - V_{\alpha}, c\}$, onde $c$, a variável de classificação, representa todos os valores possíveis que a variável $V_{\alpha}$ pode assumir. Como o espaço em que a rede é construída foi particionado, é possível fazer tal enumeração de forma eficiente.

O particionamento do espaço de estado do sistema é a grande contribuição deste trabalho. Validações feitas pelos autores mostraram que a técnica de partição não somente simplifica a fase de aprendizagem da rede, mas também aumenta a acurácia do modelo representado.

### 4.3.3. Modelos de Aprendizado *Online*

#### 4.3.3.1. MESO

MESO (*Muti-Element Self-Organizing tree*) [Kasten and McKinley 2007] é um sistema baseado em memória perceptual[2], projetado para apoiar processos de aprendizado incremental e tomada de decisão *online* em sistemas autônomos. Para atingir este objetivo, MESO implementa um algoritmo de *clustering* que envolve duas funções principais: treinamento e classificação. Durante treinamento, padrões são adicionados à memória perceptual, permitindo a construção de um modelo dos dados amostrados. Em geral, padrões compreendem observações relacionadas à qualidade de serviço ou contextos ambientais. Cada amostra do conjunto de treinamento consiste em um par $(x_i, y_i)$, onde $x_i$ é um vetor de padrões e $y_i$ corresponde a uma meta-informação, definida pelo usuário, associada ao padrão. Um exemplo de meta-informação consiste na ação a ser executada, quando o padrão associado é reconhecido.

Na memória perceptual, padrões são organizados em uma estrutura hierárquica que facilita sua recuperação. Na fase de classificação, a estrutura é consultada fornecendo-se o conceito a ser classificado. Este é comparado com os padrões armazenados e a meta-informação associada ao padrão de maior grau de similaridade em relação ao conceito fornecido é retornada.

#### Esferas Sensitivas

MESO permite a possibilidade de treinamento incremental em sistemas de monitoramento contínuo. Periodicamente, padrões capturados são adicionados ao modelo de aprendizado. Entretanto, para limitar o consumo de recursos computacionais, especialmente memória e processamento, padrões similares são agrupados em pequenos grupos

[2]memória perceptual refere-se a um tipo de memória de longo prazo, usada para armazenar padrões de estímulos externos

**Listagem 4.1. Algoritmo para criação de esferas sensitivas**

```
initialize cluster center, δ = 0, f
input pattern x(t)
find nearest center, w_i
if d(x_i, w_i) ≤ δ then
        update cluster center
else
        create new center w_j = x(t)
        grow_δ = ((d − δ) δ/d f) / (1 + ln(d − δ + 1)^2) .
        δ = grow_δ
end
next pattern
```

denominados esferas sensitivas. Esferas sensitivas são organizadas em um estrutura hierárquica que agiliza os processos de treinamento e classificação, bem como provê uma significativa compressão de dados. No entanto, para manter a acurácia do classificador, esferas sensitivas crescem incrementalmente.

A Listagem 4.1 ilustra o algoritmo de *clustering* usado em MESO. Para cada padrão lido, calcula-se a distância ($d$) entre o padrão e o centro da esfera mais próxima ($w_i$). Se esta distância for menor ou igual a $\delta$ (inicialmente nulo), o padrão é inserido na esfera, cujo centro é recalculado; caso contrário, uma nova esfera é criada contendo o padrão lido. O centro de uma esfera é calculado como a média de todos os padrões que a compõem. Entretanto, ao contrário de algoritmos de *clustering* tradicionais, em MESO, o valor de $\delta$ muda dinamicamente. Efetivamente, $\delta$ representa a sensibilidade do algoritmo em relação à distância entre os padrões.

Uma consideração importante é a relação entre o valor de $\delta$ e seu impacto no funcionamento do sistema. Se $\delta$ é muito pequeno, a taxa de compressão é baixa e muitas esferas serão criadas. Como conseqüência, o tempo para treinamento aumenta consideravelmente. Se $\delta$ é muito grande, poucas esferas são criadas, afetando a acurácia do classificador. Portanto, a função de crescimento para $\delta$ deve permitir um balanceamento entre criação de esferas e crescimento das mesmas. Na Listagem 4.1, tal função é representada por $grow_\delta$. O parâmetro $f$ é uma função de balanceamento fornecida pelo usuário.

## Estrutura Hierárquica

Para tornar a busca por um padrão mais eficiente, muitos classificadores os organizam em uma estruturas hierárquicas. Em MESO, no entanto, esferas sensitivas, e não padrões individuais, são organizadas em uma estrutura de árvore. Como mostrado na Figura 4.12, esta árvore é construída, inicialmente, atribuindo-se o conjunto de esferas sensitivas ao nó raiz. Um algoritmo recursivo divide cada nó da árvore em subconjuntos de esferas similares e cada subconjunto se torna um nó filho. Estes também são divididos até que

cada nó filho contenha apenas uma esfera. Subconjuntos de esferas similares são criados ao particionar um nó. Para cada nó filho é atribuída uma partição e uma esfera pivô. As demais esferas do nó pai são então distribuídas entre os nós filhos de acordo com a distância em relação às esferas pivô.

A árvore de esferas sensitivas produz um modelo hierárquico dos dados de treinamento. Quanto mais profundo o nível da árvore, maior o grau de similaridade entre as esferas numa mesma partição. Dessa forma, durante uma classificação, o conceito fornecido é comparado com o pivô, a partir da raiz até um nó folha, seguindo o caminho de maior similaridade.

É importante ressaltar que a árvore de MESO pode ser construída incrementalmente, à medida que novos padrões são capturados pelo modelo. Como a árvore manipula esferas (e não padrões individuais) que são criadas controladamente, é possível manter uma estrutura hierárquica de dados comprimidos de forma eficiente. O uso de MESO como ferramenta de apoio à tomada de decisão em softwares adaptáveis foi avaliado através de um estudo de caso envolvendo uma aplicação de *stream* de áudio. Esta deveria adaptar-se diante de diferentes condições de rede, tais como perda de pacote, atraso, disponibilidade de banda. Particularmente, MESO foi utilizado como o módulo de tomada de decisão da aplicação, decidindo como e quando adaptações aconteciam.

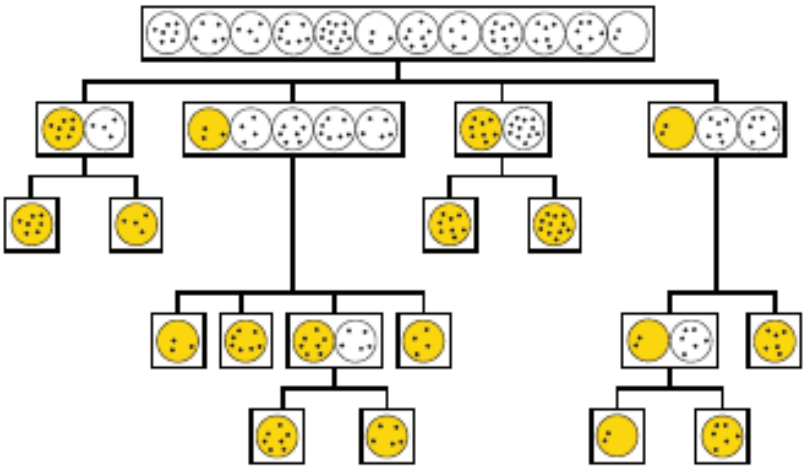

**Figure 4.12. Organização de esferas sensitivas em MESO [Kasten and McKinley 2007].**

#### 4.3.3.2. Rish et al.

Em [Rish et al. 2005] é proposta uma técnica de mensuração adaptativa, denominadas *probes* ativos, cujo objetivo é prover um mecanismo eficiente de diagnóstico de falhas e localização de problemas em aplicações distribuídas. Para alcançar este objetivo, tal mecanismos se baseia em um processo de aprendizado e inferência *online* que continuamente selecionam um conjunto de *probes* mais representativos para um diagnóstico em um dado momento. Um *probe* consiste em uma sonda ou teste cujo resultado depende de alguns componentes do sistema. Exemplos de *probes* incluem: transações enviadas para

um servidor, comandos enviados pela rede, mensagens de correio eletrônico, requisições de serviço, consulta a um banco de dados, etc.

A seleção ativa de *probes* informativos provê um mecanismo de instrumentação leve e eficiente, já que apenas um conjunto pequeno de sondas são enviadas e processadas. A seleção é implementada por um módulo de tomada de decisão que utiliza uma abordagem de aprendizado incremental e contínuo, onde o diagnóstico é construído e atualizado à medida que novas observações se tornam disponíveis. A seguir, descreve-se como o módulo de tomada de decisão é implementado.

### Seleção Ativa de *Probes*

A Figura 4.13 ilustra a abordagem proposta em Rish et al.*Probes* são enviados para o sistema monitorado e os resultados são coletados para análise. Se a inferência sobre os dados coletados indicar algum problema, *probes* mais informativos são selecionados e enviados, com base na análise dos resultados dos *probes* anteriores. Este processo é repetido até que o problema seja completamente diagnosticado.

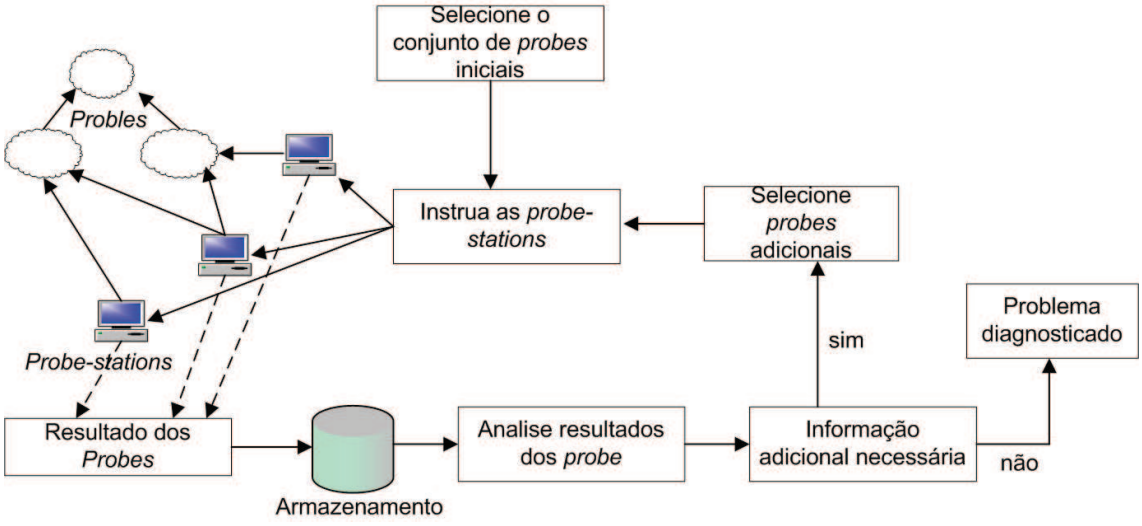


**Figure 4.13. Diagnóstico adaptativo através de *probes* ativos [Rish et al. 2005].**

A Listagem 4.2 implementa a seleção ativa de *probes*. Este algoritmo toma como entrada um conjunto de probes $T$ e uma distribuição de probabilidade à priori $P(X)$ sobre os estados do sistema. O algoritmo mantém um conjunto de crença (*belief*) sobre o estado do sistema, $Belief(X) = P(X|T_a)$, onde $T_a$ é o conjunto corrente de *probes* e seus resultados. O algoritmo calcula o próximo *probe* $Y^*$ que maximiza o ganho de informação sobre o estado $X$. Entretanto, diferentemente das abordagens passivas, o algoritmo conhece o resultado dos probes anteriores. Dessa forma, na iteração $k$, o cálculo é condicionado ao conjunto $T_a = \{Y_1^* = y_1^*, ..., Y_{k-1}^* = y_{k-1}^*\}$, enquanto que em abordagens passivas, o cálculo é condicionado a todos os resultados possíveis para o conjunto de *probes* selecionados anteriormente ($T_a = \{Y_1^*, ..., Y_{k-1}^*\}$). Portanto, a seleção ativa de *probes* encontra o $Y$ que maximiza o ganho de informação sobre $X$, dado que $T_a$ ocorreu, ou seja, $I(X;Y|T_a)$ (passo 1 da Listagem 4.2).

**Listagem 4.2. Algoritmo para seleção ativa de probes**

```
input: a set of available probes T,
       a prior distribuition P(X)
output: a set T_a of probes and their outcomes,
        Belief(X)
initialize: Belief(X) = P(X), T_a = ∅
repeat
        1. Y* = argmax_{Y∈T\T_a} I(X;Y|T_a)
        2. execute Y*; it returns Y* = y* (0 or 1)
        3. update T_a = T_a ⋃ {Y* = y*}
        4. update Belief(X) = P(X|T_a)
until ¬ (∃Y ∈ T such that I(X;Y|T_a) > 0)
return T_a, Belief(X)
```

Após encontrar $Y^*$, o algoritmo espera pelo seu resultado (passo 2). Ele então atualiza o conjunto de *probes* executados (passo 3) e a crença corrente (passo 4). Os passos 1 a 4 são repetidos até que não se possa obter mais informações sobre o estado do sistema e, portanto, melhorar o diagnóstico corrente.

Para o passo 4, atualização de crença, os autores usam um processo de inferência probabilística baseada em redes bayesianas dinâmicas. Redes bayesianas dinâmicas (DBN) estendem redes bayesianas tradicionais (BN), introduzindo-se no modelo a noção de fatias de tempo e especificando-se probabilidades de transição entre fatias consecutivas, ou seja, $P(X^t|X^{t-1})$, onde $X^t = \{x_1^t, ..., x_n^t\}$. Como conseqüência, DBNs podem ser representadas como duas BN interconectadas: a BN que representa a dependência dos estados do sistema no tempo $t$ ($BN_t$, intra-dependência) e a BN que representa a dependência entre $BN_t$ e $BN_{t-1}$ (inter-dependência). A inter-dependência foi modelada assumindo-se que o nó $X_i$ em $NB_t$ é unicamente dependente do nó $X_i$ em $BN_{t-1}$.

Em [Rish et al. 2005], um pequeno cenário envolvendo diagnóstico de falhas em transações Web foi desenvolvido como prova de conceito para os algoritmos propostos.

#### 4.3.3.3. Seshia

Em [Seshia 2007] é apresentado um *framework* baseado em algoritmos *online* para a construção de sistemas autônomos. O *framework* proposto é especialmente projetado para aprendizado a partir de falhas determinísticas e auto-recuperação. A Figura 4.14 apresenta uma visão geral do *framework* proposto.

O módulo de implementação $M$ consiste na versão original do sistema, a qual será implantada e testada. Este módulo recebe dados de entrada diretamente do ambiente em que está inserido e gera saídas. Entretanto, para aplicação do *framework*, assume-se que o comportamento de $M$ possa ser dividido em rodadas, cada uma das quais começando em um estado válido. Um estado inicial válido, por sua vez, é definido por uma invariante $I_{start}$. Formalmente, um comportamento correto de $M$ é representado por uma seqüencia

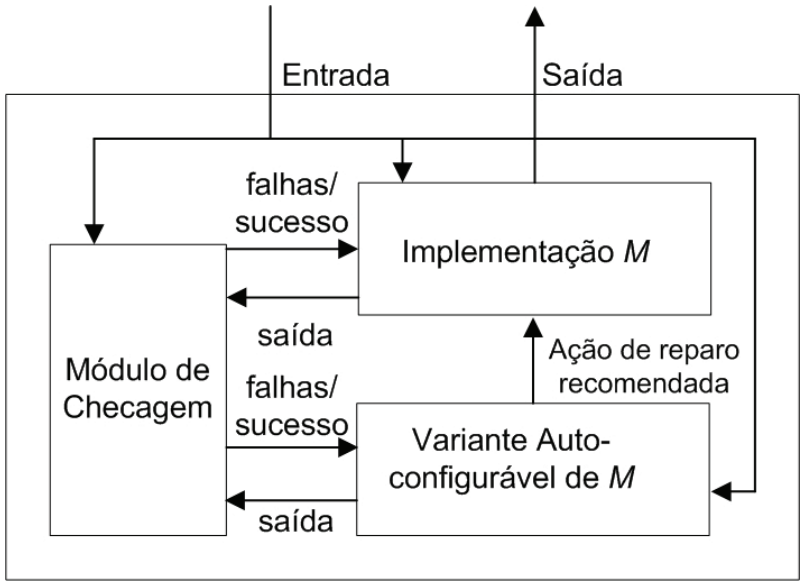


**Figure 4.14. Componentes de um sistema capaz de auto-reparo em Seshia [Seshia 2007].**

de estados e ações, como mostrado abaixo.

$$s_0 a_1 s_1 a_2 \ldots a_{i_1} s_{i_1} a_{i_1+1} s_{i_1+1} \ldots a_{i_2} s_{i_2} a_{i_2+1} s_{i_2+1}$$

onde:

- $s_0 \in I$, $I$ é o conjunto de estados iniciais do sistema
- $s_i = \delta(s_{i-1}, a_i)$, $\delta$ é a função de transição que descreve o próximo estado do sistema que resulta da execução da ação $a_i$ no estado corrente $s_{i-1}$
- $\forall_j \in 1,2,3,\ldots, s_{i_j} \in I_{start}$

Uma rodada $t$ é definida como uma seqüencia finita de estados começando em $s_{i_t}$ e terminando em $s_{i_{t+1}}$. Dessa forma, exige-se que o sistema retorne para um estado válido em cada rodada, evitando-se assim a propagação de erros para rodadas posteriores, o que poderia tornar o sistema irrecuperável.

O módulo de checagem verifica se a saída gerada por $M$ é correta. Esse módulo pode ser implementado como uma versão mais simples e formalmente verificada de $M$ ou como uma coleção de monitores que checam continuamente se as invariantes e asserções estão sendo violadas. O resultado da verificação (falha ou sucesso) é retornado para $M$, de forma que o mesmo possa tomar ações corretivas, quando necessário.

O último componente do *framework* consiste em uma variante de $M$ instrumentada de forma a modificar a função de transição de estados $\delta$ de acordo com o modelo de falhas. Em cada rodada, a variante, denotada por $\bar{M}$, executa paralelamente a $M$, processando as mesmas entradas. Entretanto, suas saídas são direcionadas apenas para o módulo de checagem, o qual as verificam. Baseado na resposta do módulo de checagem, $\bar{M}$ decide

como e quando modificar a função de transição para a próxima rodada. O algoritmo que $\bar{M}$ usa para tomar esta decisão é denominado estratégia de recuperação.

A estratégia de recuperação deve escolher uma ação de reparo a partir de um espaço possível de ações, definido pelo modelo de falhas. Para explorar as conseqüências de um reparo particular, $\bar{M}$ se auto-configura a cada rodada, mesmo na ausência de falhas. Posteriormente, se $M$ falha, ele se recupera carregando uma ação de reparo recomendada por $\bar{M}$. Dessa forma, $M$ pode se auto-ajustar baseando-se na experiência obtida por $\bar{M}$ em várias rodadas de execução.

## O Problema do Bandido Multi-armado

No *framework* proposto, o problema a ser resolvido consiste na escolha de uma ação de reparo, baseando-se na história de desempenho de ações candidatas no passado. A escolha da ação de reparo a ser executada é implementada por um algoritmo de aprendizado *online*, sendo a escolha tratada como um processo de aprendizado por erros. Cada rodada corresponde a uma tentativa na qual o aprendizado prossegue. A solução utilizada baseia-se em um algoritmo *online* clássico denominado o problema do bandido multi-armado (*the multi-armed bandit problem*). Neste algoritmo, um jogador deve escolher, em cada rodada, uma máquina dentre $m$ candidatas. No final de cada rodada, o jogador recebe uma recompensa de acordo com a escolha feita. Recompensas podem ser não positivas e não se assumem suposições estatísticas sobre elas.

Adaptando-se o algoritmo original para resolver o problema do *framework* proposto, o jogador corresponde ao módulo $M$ e as máquinas correspondem às ações de reparo. Se a ação escolhida evita a presença de erros durante a rodada, o módulo recebe uma recompensa de 1; caso contrário 0. Este problema ilustra o compromisso entre exploração de alternativas e aproveitamento das melhores escolhas. Em geral, exploração é desejável, porém, em demasia, pode levar a um baixo aproveitamento das melhores alternativas. O *framework* proposto trata tal dilema, paralelizando-se operações de exploração, atribuídas a $\bar{M}$, e de aproveitamento, realizada por $M$.

A estratégia de seleção de ações de reparo utilizada pelo *framework* é ilustrada na Listagem 4.3. A idéia básica do algoritmo consiste em usar pesos para monitorar o desempenho de $m$ ações de reparo em uma seqüencia de rodadas. Um peso alto para uma ação indica uma história de bom desempenho. No início de cada rodada $t$, o algoritmo sorteia uma ação $i_t$ de acordo com uma distribuição de probabilidade $p_1(t), p_2(t), ..., p_m(t)$ sobre as ações. Esta distribuição é uma mistura de normal e uma distribuição que atribui a cada ação uma probabilidade associada ao seu peso. A distribuição baseada em peso facilita o aproveitamento das melhores alternativas, enquanto que a distribuição uniforme assegura a exploração do espaço de ações. Se a ação escolhida $i_t$, provoca uma falha no sistema, gera-se um custo de 1 unidade para a ação $i_t$ e seu peso é decrementado por um fator exponencial.

O *framework* proposto foi validado através de um estudo de caso envolvendo monitoramento de dados trafegados numa rede. O monitor é responsável por detectar tráfego malicioso e eliminar os pacotes afetados. Duas implementações do monitor foram fornecidas, sendo a mais otimizada correspondendo ao módulo $M$ e a versão mais simples

**Listagem 4.3. Algoritmo para seleção de ações de reparo**

```
parameter: γ ∈ (0,1)
t = 1
loop
   for i = 1 to m
      p_i(t) = (1−γ) w_i(t) / Σ_{j=1}^{m} w_j(t) + γ/m
   end
   Draw action i_t from the set {1,2,...,m}
   using p_1(t), p_2(t), ..., p_m(t)
   if i_t failed then C_{i_t} = 1
   else C_{i_t} = 0
   end
   for j = 1 to m
      if j = i_t then Ĉ_j(t) = C_j(t)/p_j(t)
      else Ĉ_j(t) = 0
      end
      w_j(t+1) = w_j(t).exp(−(γ/m) Ĉ_j(t))
   end
   t = t+1
end
```

ao módulo de checagem. Também foi fornecido uma versão adaptativa de $M$, correspondendo ao módulo de reparo $\bar{M}$.

As tabelas 4.2, 4.3 e 4.4 resumem as características principais dos trabalhos apresentados nesta seção, em relação às propriedades de auto-gerenciamento consideradas, arquiteturas adotadas, infra-estruturas (de políticas, serviços e tecnlógicas) utilizadas e aplicações desenvolvidas.

## 4.4. 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, uma revisão de algumas das propostas mais relevantes da literatura em questão foi apresentada, dando particular importância ao aspecto arquitetural e de infra-estrutura adotados por essas soluções. A revisão desses trabalhos nos permitiu tirar várias conclusões. Em relação aos aspectos arquiteturais e de infra-estrutura para AC, conclui-se que:

- 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

| Característica | Unity | KX | Accord | R-Capriccio |
|---|---|---|---|---|
| propriedades | *self-healing*, *self-optimizing*, *self-configuring* | *self-healing*, *self-configuring* | *self-optimizing*, *self-configuring* | *self-optimizing* |
| arquitetura | elemento autônomo | infra-estrutura | elemento autônomo | infra-estrutura |
| infra-estrutura de política | funções de utilidade definidas pelo usuário | ações baseadas em *workflows* | ações definidas pelo usuário ou modelos de teoria de controle | ações definidas pelo usuário |
| serviço de monitoramento | sentinelas | sondas ativas e calibradores de evento | implementado pelo usuário | implementado pelo usuário |
| serviço de adaptação | paramétrica | paramétrica e estrutural | paramétrica e estrutural | paramétrica |
| infra-estrutura tecnológica | agentes de software | objetos Java | componentes de software | web service |
| aplicações desenvolvidas | protótipo envolvendo gerenciamento de recursos | administração de sistemas legados | gerenciamento de recursos em grades | gerenciamento de recursos na web |

**Table 4.2. Trabalhos baseadas em modelos analíticos - características principais**

arquiteturas e infra-estrutura. Entretanto, como mostrado neste trabalho, nenhuma arquitetura proposta atingiu completamente 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 utiliza políticas baseadas em ações, em que objetivos são expressos 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.

Em relação aos modelos usados para representação de conhecimento, conclui-se que:

- Os modelos construídos ainda não conseguem capturar com abrangência as várias características e aspectos comumente presentes em sistemas complexos. Como

| Característica | ABLE | Park | Magpie | Pinpoint | iManage |
|---|---|---|---|---|---|
| propriedades | *self-healing*, *self-configuring* | *self-healing*, *self-configuring* | *self-optimizing* | *self-healing* | *self-optimizing* |
| arquitetura | elemento autônomo | infra-estrutura | infra-estrutura | infra-estrutura | infra-estrutura |
| infra-estrutura de política | ações e objetivos | ações escritas em XML | ações definidas pelo usuário | ações definidas pelo usuário | ações definidas pelo usuário |
| serviço de monitoramento | implementado pelo usuário | agente monitor | implementado pelo usuário | implementado pelo usuário | implementado pelo usuário |
| serviço de adaptação | paramétrica e estrutural | paramétrica | paramétrica e estrutural | paramétrica e estrutural | paramétrica e estrutural |
| infra-estrutura tecnológica | agentes de software | agentes de software | objetos | objetos | objetos |
| aplicações desenvolvidas | administração de sistemas | protótipos de administração de sistemas | gerenciamento de recursos na web | administração de sistemas | gerenciamento de recursos |

**Table 4.3. Trabalhos baseadas em aprendizado de máquina - características principais**

conseqüência, as soluções apresentadas ainda são protótipos imaturos que tratam isoladamente alguns aspectos relevantes para o auto-gerenciamento.

- A grande maioria das soluções não estão efetivamente preparadas para tratar adaptabilidade. Os modelos construídos não são facilmente atualizados e, poucas soluções levam em consideração um processo de aprendizado incremental e dinâmico.

- Os mecanismos de inferência ainda são limitados, complexos e de difícil parametrização.

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 e grande escala em protótipos ou aplicações reais. Dentre algumas tendências para pesquisas futuras incluem-se:

- Avanços no que se refere à padronização das interações colaborativas entre elementos autônomos (no caso de arquiteturas baseadas nestes elementos) ou mesmo entre os serviços específicos para AC providos por uma infra-estrutura e as aplicações que os utilizam (no caso de arquiteturas baseadas em infra-estruturas). Além de padrões para interações colaborativas, destaca-se também a necessidade de padronização para: a formatação de eventos, descritores de dependência, especificações de sintomas usados em processos de determinação de problemas, formatação consistente

| Característica | MESO | Rish | Seshia |
|---|---|---|---|
| propriedades | *self-configuring* | *self-healing*, *self-configuring* | *self-healing*, *self-protecting* |
| arquitetura | infra-estrutura | infra-estrutura | infra-estrutura |
| infra-estrutura de política | ações definidas pelo usuário | ações definidas pelo usuário | ações definidas pelo usuário |
| serviço de monitoramento | implementado pelo usuário | probes ativos | implementado pelo usuário |
| serviço de adaptação | paramétrica e estrutural | paramétrica e estrutural | paramétrica e estrutural |
| infra-estrutura tecnológica | objetos | objetos | objetos |
| aplicações desenvolvidas | protótipo de um sistema adaptativo | diagnóstico de falhas | diagnóstico de tráfego malicioso |

**Table 4.4. Trabalhos baseadas em aprendizado *online* - características principais**

para captura de políticas e padrões para obter informações de configuração dos diversos componentes do sistema.

- Investigação de métodos de correlação e análise de dados monitorados que assegurem a operação das propriedades de auto-gerenciamento de forma continuada.
- Avanços em abordagens automatizadas para representação de políticas, criação de planos de ação, negociação e resolução de conflitos.
- Estudos sobre interação homem-máquina que visam entender como administradores gerenciam os sistemas atuais. Em geral, existe a necessidade de novas linguagens e abstrações que possibilitem aos humanos realizarem tarefas de monitoramento, visualização e controle de sistemas autônomos a partir da definição de objetivos de alto nível.

## Agradecimento

O desenvolvimento deste trabalho recebe o apoio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e Tecgraf/PUC-Rio.

## References

[ibm 2003] (2003). An Architectural Blueprint for Autonomic Computing. Technical report, IBM.

[APC 2003] APC (2003). Determining total cost of ownership for data center and network room infrastructure.

[Barham et al. 2004] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 259–272, Berkeley, CA, USA. USENIX Association.

[Bhat et al. 2006] Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N., and Abdelwahed, S. (2006). Enabling Self-managing Applications using Model-based Online Control Strategies. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 15–24, Washington, DC, USA. IEEE Computer Society.

[Bigus et al. 2002] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., III, W. N. M., and Diao, Y. (2002). ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Syst. J.*, 41(3):350–3718.

[Blum 1996] Blum, A. (1996). On-line Algorithms in Machine Learning. In *In Proceedings of the Workshop on On-Line Algorithms, Dagstuhl*, pages 306–325. Springer.

[Braga et al. 2006] Braga, T. R. M., Silva, F. A., Ruiz, L. B., and ao, H. P. A. (2006). *XXIV Simpósio Brasileiro de Redes de Computadores*, chapter Redes Autonômicas. Sociedade Brasileira de Computação (SBC).

[Brodie et al. 2005] Brodie, M., Ma, S., Lohman, G., Syeda-Mahmood, T., Mignet, L., and Modani, N. (2005). Quickly Finding Known Software Problems via Automated Symptom Matching. In *Proceedings of The Second International Conference on Autonomic Computing (ICAC'05)*, pages 101–110, Los Alamitos, CA, USA. IEEE Computer Society.

[Chen et al. 2002] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA. IEEE Computer Society.

[Cohen et al. 2004] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. (2004). Correlating Instrumentation Data to System States: a Building Block for Automated Diagnosis and Control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 231–244, Berkeley, CA, USA. USENIX Association.

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

[Ganek and Corbi 2003] Ganek, A. and Corbi, T. A. (2003). The Dawning of the Autonomic Computing Era. *IBM Syst. J.*, 42(1):5–18.

[Horn 2001] Horn, P. (2001). 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. Disponível em http://www.research.ibm.com/autonomic/.

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

[Kaiser et al. 2003] Kaiser, G., Parekh, J., Gross, P., and Valetto, G. (2003). 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)*, pages 22–30.

[Kandasamy et al. 2004] Kandasamy, N., Abdelwahed, S., and Hayes, J. (2004). Self-optimization in Computer Systems via On-line Control: Application to Power Management. In *Proceedings of The International Conference on Autonomic Computing (ICAC'04)*, pages 54–61, Los Alamitos, CA, USA. IEEE Computer Society.

[Kasten and McKinley 2007] Kasten, E. P. and McKinley, P. K. (2007). MESO: Supporting Online Decision Making in Autonomic Computing Systems. *IEEE Trans. on Knowl. and Data Eng.*, 19(4):485–499.

[Kephart 2005] Kephart, J. O. (2005). Research Challenges of Autonomic Computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 15–22, New York, NY, USA. ACM.

[Kephart and Chess 2003] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50.

[Khargharia et al. 2006] Khargharia, B., Hariri, S., and Yousif, M. (2006). Autonomic Power and Performance Management for Computing Systems. In *Proceedings of The Third International Conference on Autonomic Computing (ICAC'06)*, pages 145–154, Los Alamitos, CA, USA. IEEE Computer Society.

[Kumar et al. 2007] Kumar, V., Cooper, B., Eisenhauer, G., and Schwan, K. (2007). iManage: Policy-Driven Self-Management for Enterprise-Scale System. In *Proceedings of The 8th International Middleware Conference*, pages 287–307. Springer Berlin / Heidelberg.

[Lin and Leaney 2005] Lin, P. and Leaney, A. M. J. (2005). Defining Autonomic Computing: A Software Engineering Perspective. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 88–97, Washington, DC, USA. IEEE Computer Society.

[Liu and Parashar 2004] Liu, H. and Parashar, M. (2004). A Component-based Programming Model for Autonomic Applications. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 10–17, Washington, DC, USA. IEEE Computer Society.

[Liu et al. 2004] Liu, H., Parashar, M., and Hariri, S. (2004). A Component Based Programming Framework for Autonomic Applications. In *Proceedings of The International Conference on Autonomic Computing (ICAC'04)*, pages 10–17, Los Alamitos, CA, USA. IEEE Computer Society.

[McCann and Huebscher 2004] McCann, J. A. and Huebscher, M. C. (2004). Evaluation Issues in Autonomic Computing. In *Proceedings of Grid and Cooperative Computing Workshop*, pages 597–608. Springer Berlin/Heidelberg.

[McCann et al. 2007] McCann, J. A., Huebscher, M. C., and Hoskins, A. (2007). 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.

[McKinley et al. 2004] McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. (2004). Composing Adaptive Software. *Computer*, 37(7):56–64.

[Papazoglou and Georgakopoulos 2003] Papazoglou, M. P. and Georgakopoulos, D. (2003). Introduction. *Commun. ACM*, 46(10):24–28.

[Parashar and Hariri 2004] Parashar, M. and Hariri, S. (2004). Autonomic Computing: An Overview. In *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September*, volume 3566 of *Lecture Notes in Computer Science*, pages 257–269. Springer.

[Parashar et al. 2005] Parashar, M., Klie, H., Catalyurek, U., Kurc, T., Bangerth, W., Matossian, V., Saltz, J., and Wheeler, M. (2005). Application of Grid-enabled Technologies for Solving Optimization Problems in Data-driven Reservoir Studies. *Future Gener. Comput. Syst.*, 21(1):19–26.

[Park et al. 2005] Park, J., Yoo, G., and Lee, E. (2005). 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*, pages 256–263, Washington, DC, USA. IEEE Computer Society.

[Phan et al. 2008] Phan, T., Han, J., Schneider, J.-G., Ebringer, T., and T.Rogers (2008). A Survey of Policy-Based Management Approaches for Service Oriented Systems. In *19th Australian Conference on Software Engineering (ASWEC 2008)*, pages 392 – 401, Los Alamitos, CA, USA. IEEE Computer Society.

[Rish et al. 2005] Rish, I., Brodie, M., Sheng, M., Odintsova, N., Beygelzimer, A., Grabarnik, G., and Hernandez, K. (2005). Adaptive Diagnosis in Distributed Systems. *IEEE Transactions on Neural Networks*, 16(5):1088–1109.

[Salehie and Tahvildari 2005] Salehie, M. and Tahvildari, L. (2005). Autonomic Computing: emerging trends and open problems. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7.

[Seshia 2007] Seshia, S. A. (2007). Autonomic Reactive Systems via Online Learning. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 30, Washington, DC, USA. IEEE Computer Society.

[Shalev-Shwartz 2007] Shalev-Shwartz, S. (2007). *Online Learning: Theory, Algorithms, and Applications*. PhD thesis, Hebrew University.

[Sterritt and Bustard 2003] Sterritt, R. and Bustard, D. (2003). Autonomic Computing-A Means of Achieving Dependability? In *Proceedings of IEEE International Conference on the Engineering of Computer Based Systems (ECBS'03)*, pages 247–251, Los Alamitos, CA, USA. IEEE Computer Society.

[Tesauro et al. 2004] Tesauro, G., Chess, D. M., Walsh, W. E., Das, R., Segal, A., Whalley, I., Kephart, J. O., and White, S. R. (2004). A Multi-agent Systems Approach to Autonomic Computing. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 464–471, Washington, DC, USA. IEEE Computer Society.

[White et al. 2004] White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., and Kephart, J. O. (2004). An Architectural Approach to Autonomic Computing. In *Proceedings of The International Conference on Autonomic Computing (ICAC'04)*, pages 2–9, Los Alamitos, CA, USA. IEEE Computer Society.

[Zanikolas and Sakellariou 2005] Zanikolas, S. and Sakellariou, R. (2005). A Taxonomy of Grid Monitoring Systems. *Future Gener. Comput. Syst.*, 21(1):163–188.

[Zenmyo et al. 2006] Zenmyo, T., Yoshida, H., and Kimura, T. (2006). A Self-healing Technique based on Encapsulated Operation Knowledge. In *Proceedings of The Third International Conference on Autonomic Computing (ICAC'06)*, pages 25–32, Los Alamitos, CA, USA. IEEE Computer Society.

[Zhang et al. 2007] Zhang, Q., Cherkasova, L., Mathewes, G., Greene, W., and Smirni, E. (2007). R-Capriccio: A Capacity Planning and Anomaly Detection Tool for Enterprise Services with Live Workloads. In *Proceedings of The 8th International Middleware Conference*, pages 244–265. Springer Berlin / Heidelberg.