COPPE/UFRJ

DYNAFLOW: WORKFLOWS DISTRIBUÍDOS EM AMBIENTES P2P COM AUXÍLIO DE AGENTES

Vinícius Antônio Gomes Marques

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Geraldo Bonorino Xexéo

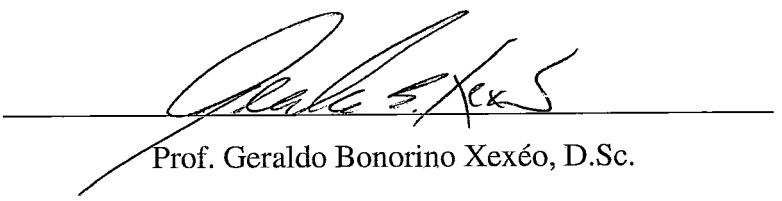
Rio de Janeiro
Outubro de 2009

DYNAFLOW: WORKFLOWS DISTRIBUÍDOS EM AMBIENTES P2P COM AUXÍLIO DE AGENTES

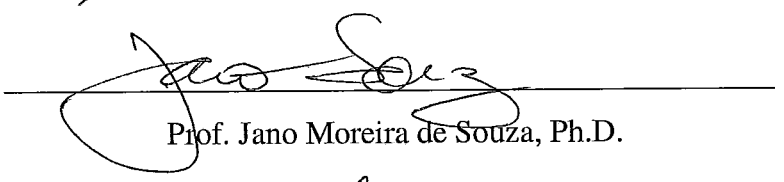
Vinícius Antônio Gomes Marques

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

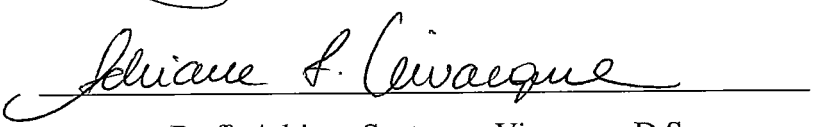
Aprovada por:



Prof. Geraldo Bonorino Xexéo, D.Sc.



Prof. Jano Moreira de Souza, Ph.D.



Profª. Adriana Santarosa Vivacqua, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

OUTUBRO DE 2009

Marques, Vinícius Antônio Gomes

Dynaflow: Workflows distribuídos em ambientes P2P com auxílio de agentes/ Vinícius Antônio Gomes Marques. – Rio de Janeiro: UFRJ/COPPE, 2009.

XI, 110 p.: il.; 29,7 cm.

Orientador: Geraldo Bonorino Xexéo

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2009.

Referências Bibliográficas: p. 107-110.

1. Peer-to-Peer (P2P). 2. Sistemas multi-agentes. 3. Workflows. I. Xexéo, Geraldo Bonorino. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Titulo.

Não sei aonde eu poderia chegar sem eles, apenas sei que sou muito feliz.
E sei que isto é conseqüência dos seus esforços em dedicação, carinho e amor.
Pai e Mãe, eu dedico esta conquista a vocês, como um singelo agradecimento!

Muita coisa mudou na minha vida, depois que ela recebeu esta pessoa.
E, após estes anos, vejo como esta pessoa é muito importante para mim.
Como agradecimento por tudo que você representa,
Esta conquista também é dedicada a você,
Meu amor, *Monique*!

Agradecimentos

Aos meus pais, e ídolos, *José e Fátima*. Pai e mãe, sem vocês, nem esta, nem outras conquistas seriam possíveis na minha vida. Tudo que sou, devo eternamente a vocês. Emociono-me muito com suas histórias de vida. Vindos de Portugal, conheceram-se aqui e criaram seus três filhos com muito esforço, carinho e dedicação. Não puderam seguir adiante com os seus estudos, mas sempre apoiaram, incondicionalmente, a educação dos seus filhos. Eu não poderia ter melhor exemplo. Que muitas sejam as oportunidades de retribuí-los, ao menos um pouquinho do que recebi de vocês. O meu eterno muito obrigado!

À minha esposa e amor da minha vida (*daqui até a eternidade...*), *Monique*. Sempre ao meu lado, ela preenche a minha vida com alegria e não mede esforços para me fazer feliz, aceitando até o que poucas esposas compreenderiam (como uma sala dedicada para *home-theater*). Sua dedicação impede que eu seja um completo enrolado e seu apoio foi fundamental para este trabalho ser concluído. Agradeço-te eternamente por todos estes anos de amor, carinho, dedicação, amizade e orientação.

Às minhas irmãs *Dília e Marta*, que sempre se preocupam com o seu irmão caçula e estão sempre lá, torcendo muito por mim. E também devo agradecer às minhas duas lindas sobrinhas, *Camila e Flavinha*, pelas horas de diversões e aos meus cunhados, *Henrique e Romano*, pelas bocas-livres nas suas casas.

Ao meu orientador, *Geraldo Xexéo*, que me atura já há alguns anos. Agradeço ao *Xexéo* por todo o seu apoio, não apenas no Mestrado, mas também pelas oportunidades de trabalho na COPPETEC. Atrasei sua saída da COPPE inúmeras vezes, quase sempre passando das 19 ou 20 horas. Mesmo com as suas inúmeras atribuições de coordenador do PESC, ele sempre foi muito solícito. Além do seu lado orientador e profissional, o *Xexéo* sempre nos brinda com o seu bom humor e o seu jeitão aluno de ser. Gostaria de agradecê-lo pelas inúmeras ajudas (e não foram poucas). Também gostaria de desculpar-me pelo meu atraso na realização deste trabalho. ☺

A todo o pessoal da linha de BD com quem convivi durante este tempo. Ao professor *Jano Souza*, pela dedicação à frente da linha de Banco de Dados e por ter aceitado participar da banca examinadora. À *Adriana Vivacqua*, pela sua receptividade e simpatia, sempre apoiando a galera da linha de BD, e também por ter aceitado participar da banca. Ah, claro, não posso esquecer sua ajuda quando este ainda era um trabalho de disciplina. Ao *José Rodrigues*, o *Zé*, pelo seu espírito colaborativo, sempre ajudando, dando idéias e interessado em ajudar a todos.

À galera da PETROBRÁS, com quem trabalho há quase dois anos. Gostaria de agradecer aos amigos *Sérgio Kriz*, *Rodrigo Águas* e *Ester* pelo companheirismo durante este tempo. E também à *Ana Paula* pela sua preocupação e por ter me liberado diversas vezes para este trabalho, mesmo com o nosso calendário sempre apertado.

"*E a tese?*" é uma pergunta que as pessoas costumam fazer sorrindo... Agradeço àquelas pessoas que se lembraram de fazer esta pergunta, como o *Manuel*, amigo que sempre me incentivou a ir mais longe. Ao *Patola*, meu padrinho e futuro afilhado de casamento, por estes anos de amizade. Ao *Zizi* (Von Held), pelos anos de amizade, viagens maneiras e doideiras em geral. Agradeço a compreensão dos amigos dos quais me distanciei um pouco durante esta jornada.

À minha Sogra, *Teresinha*, e ao meu Sogro, *César*, por ter emprestado o apartamento para morarmos ~~(aliás, prometo não quebrar nenhuma parede do novo apartamento)~~. Também agradeço a minha cunhada *Renata* ("cara estranha"), pelo seu jeito engraçado e pelas suas frases doidas.

Gostaria de fazer um agradecimento especial a uma pessoa muito cativante que passou nas nossas vidas e deixou muitas saudades: meu cunhado *Magno*. Esses poucos meses têm sido muito difíceis sem a sua alegria e a sua companhia. Perdi um amigo que topava qualquer parada, mesmo que na primeira fileira da montanha russa do Hulk. Muito obrigado por ter trazido tanta alegria e pela companhia na inesquecível viagem a Disney.

A Deus, pela saúde e, principalmente, pela família que tenho. Finalmente, o meu muito obrigado a todos!!!

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DYNAFLOW: WORKFLOWS DISTRIBUÍDOS EM AMBIENTES P2P COM AUXÍLIO DE AGENTES

Vinícius Antônio Gomes Marques

Outubro/2009

Orientadores: Geraldo Bonorino Xexéo

Programa: Engenharia de Sistemas e Computação

Tradicionalmente, os sistemas de gestão de workflows adotam uma arquitetura cliente-servidor, e, como conseqüência, a coordenação das tarefas e dos dados é centralizada. Tal abordagem possui desvantagens potenciais, tais como um único ponto de falha e gargalos de desempenho, que podem impactar na escalabilidade do sistema. Este trabalho apresenta uma arquitetura descentralizada multi-agentes para a coordenação de workflows em um ambiente puramente distribuído (P2P).

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DYNAFLOW: DISTRIBUTED WORKFLOWS IN P2P ENVIRONMENT WITH AGENTS SUPPORT

Vinícius Antônio Gomes Marques

October/2009

Advisors: Geraldo Bonorino Xexéo

Department: Computer Science and Engineering

Traditionally, the workflow management systems adopt a client-server architecture, and, as consequence, the coordination of task and data is centralized. This approach has potential drawbacks as a single point of failure and performance bottlenecks, which can affect the system scalability. This work brings a multi-agent descentralized architecture to coordinate workflows in pure descentralized environment (peer-to-peer).

ÍNDICE

1 - Introdução

1.1 – O Problema

Os sistemas de gestão de workflows têm o papel de automatizar processos de negócio, podendo envolver a coordenação de um grande número de recursos, atores e ferramentas, e transpor limites geográficos e organizacionais. Conforme observado por Alonso *et al.* (1995), as aplicações de workflow são inerentemente distribuídas e até mesmo descentralizadas.

Tradicionalmente, os sistemas de gestão de workflows (abreviado por WfMS, *workflow management systems*) adotam a arquitetura cliente-servidor. Uma das razões para tal escolha é o fato da arquitetura cliente-servidor satisfazer aos requisitos funcionais dos sistemas de gestão de workflows. Entretanto, esta é uma arquitetura centralizada, e que, portanto, possui algumas limitações inerentes que podem restringir a robustez, o desempenho, a segurança e a distribuição dos WfMS (GRUNDY *et al.*, 1998, ALONSO *et al.*, 1995).

Por centralizar a coordenação da execução e os dados dos workflows, o servidor de um sistema de gestão pode se tornar um gargalo para todo o sistema. A dificuldade em escalar e a degradação do desempenho são, portanto, problemas potenciais. A adoção do esquema centralizado também afeta a robustez, devido à existência de um ponto simples de falha.

1.2 - Objetivo

O objetivo deste trabalho é o projeto de uma arquitetura descentralizada multi-agentes para coordenar a execução de workflows em um ambiente puramente descentralizado.

O interesse por uma arquitetura descentralizada vem do descasamento de impedância existente entre a natureza distribuída dos processos de negócio e a característica centralizada das soluções tecnológicas dominantes na área de workflows.

A pesquisa atual sobre soluções centralizadas resolve parcialmente os problemas inerentes, porém aumentando ainda mais a complexidade dos sistemas (YAN *et al.*,

2006). De fato, a idéia de aplicar arquiteturas descentralizadas na gestão de workflows encontra vários defensores na literatura (EDER, 1999, HEINL *et al.*, 1999, MUTH, 1998).

Para avaliar os resultados alcançados e verificar se o objetivo foi atendido, algumas metas foram traçadas:

- A arquitetura deve suportar um ambiente puramente descentralizado e por isto ela será voltada a ambientes *peer-to-peer*;

- Serão empregados conceitos de sistemas multi-agentes (MAS) e de sistemas adaptativos complexos (CAS) na elaboração da arquitetura. Os agentes deverão obedecer aos seguintes requisitos:

    - Os agentes devem possuir comportamento minimalista, especializados apenas em realizar tarefas simples;
    - Os agentes devem utilizar apenas as informações vitais ao seu funcionamento, com uma visão restrita das informações existentes em seu ambiente.

- A implementação de um protótipo para a validação da arquitetura requer uma plataforma que forneça abstração de serviços *peer-to-peer* e que suporte conceitos de sistemas emergentes. O *framework* COPPEER (MIRANDA, XEXEO *et al.*, 2006), resultado de uma pesquisa realizada na linha de Banco de Dados do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, será utilizado como plataforma de desenvolvimento por suprir tais requisitos. Uma das metas deste trabalho é validar a aplicação e o funcionamento do *framework* COPPEER.

- A mera aplicação de agentes no projeto de uma arquitetura não diz muito sobre a qualidade do projeto. Projetar agentes requer atenção na interação com outros agentes, realizada através de troca de mensagens. Este trabalho também possui uma meta educacional, que é apresentar conceitos envolvidos na modelagem baseada em agentes.

1.3 – Organização

O capítulo 2 faz uma revisão dos conceitos básicos necessários para dar prosseguimento ao trabalho. Tal capítulo é dividido em quatro subseções:

Como uma das motivações para este trabalho é o estudo e a aplicação de conceitos de sistemas *peer-to-peer* para construção de uma aplicação descentralizada e distribuída, a seção 2.1 é uma revisão de sistemas *peer-to-peer*, apresentando a definição, os conceitos envolvidos, as características, os seus benefícios e desvantagens, as formas de consultas e os protocolos e aplicações mais populares.

A seção 2.2 é dedicada a comportamentos emergentes, conceito presente em vários sistemas complexos naturais, como o cérebro humano e as colônias de insetos. Entretanto, também está presente em sistemas que não são naturais, cuja previsibilidade e controle fogem ao domínio do homem, como na Economia.

A seção 2.3 apresenta os conceitos fundamentais do problema central desta pesquisa: workflows distribuídos. Serão observados os problemas inerentes às arquiteturas centralizadas e como estes problemas podem ser amenizados com a aplicação de arquiteturas descentralizadas. Nesta seção, depara-se com uma questão chave no caminho escolhido para a solução do problema: workflows têm natureza distribuída.

Trabalhos correlatos são apresentados ao final do capítulo 2, na seção 2.4. Tais trabalhos também trazem uma abordagem *peer-to-peer* ao problema da descentralização da coordenação de workflows e, por isso, serão detalhadamente explicados.

O capítulo 3 propõe uma arquitetura multi-agentes descentralizada, em um ambiente *peer-to-peer*, para resolver o problema abordado por este trabalho.

O capítulo 4 valida o protótipo construído, apresentando sua utilização e algumas métricas coletadas a partir da execução de alguns processos.

O capítulo 5 conclui o trabalho, avaliando as metas alcançadas, apresentando algumas considerações finais e discutindo alguns possíveis trabalhos futuros baseados na presente pesquisa.

2 - Revisão bibliográfica

2.1 – Sistemas *Peer-to-Peer* (P2P)

Diversas formas de comunicação podem ser entendidas como uma comunicação entre *peers*. Uma simples ligação telefônica, por exemplo, se baseia na ligação direta entre seus participantes, uma vez estabelecida a comunicação pela central telefônica. A comunicação entre dois computadores numa rede também pode ser vista como uma comunicação entre *peers*.

Essencialmente, um sistema *peer-to-peer* é uma aplicação computacional distribuída pelos nós de uma rede, onde cada nó é autônomo e contribui com os demais provendo recursos computacionais, como armazenamento, poder de processamento, dados e programas.

2.1.1 – Condições favoráveis

Os sistemas *peer-to-peer* emergiram sob dois aspectos: técnico e social. Embora não fossem os responsáveis pela introdução das redes *peer-to-peer*, os sistemas de compartilhamento de arquivos (caso da aplicação *Emule*) que empregam tal tecnologia representaram um papel importante, revelando uma aplicabilidade popular para a tecnologia, dentro de um ambiente distribuído de larga escala, como a Internet. Tais sistemas popularizaram rapidamente o termo *peer-to-peer*.

Tecnicamente, os sistemas *peer-to-peer* conquistaram seu espaço conforme as arquiteturas centralizadas evidenciavam algumas de suas limitações. Abordagens centralizadas costumam oferecer facilidades para seu projeto, o que possibilita um foco maior nas funcionalidades do sistema. Por outro lado, quando suas as limitações vêem a tona, as soluções para contornar os problemas que surgem costumam demandar um esforço técnico considerável, o que também implica em custo econômico.

2.1.2 – Limitações das arquiteturas centralizadas

Arquiteturas centralizadas apresentam limitações intrínsecas, principalmente na presença de um grande número de clientes, o que favoreceu uma maior aceitação de

arquiteturas *peer-to-peer* em muitos domínios. A seguir, alguns problemas potenciais das arquiteturas centralizadas são apresentados:

- Baixa escalabilidade → Arquiteturas centralizadas possuem limitações inerentes para escalar. O aumento do número de clientes de um sistema cliente-servidor, por exemplo, implica em maior consumo de recursos computacionais do servidor. Mesmo que um haja um sistema de redundância que balanceie a carga dentre vários servidores, ainda sim há um limite em relação ao número de clientes. Arquiteturas hierárquicas são empregadas tradicionalmente para se obter mais escalabilidade, mas ainda sim, a escalabilidade é impactada com o acréscimo de clientes.

- Pontos simples de falha → A indisponibilidade do servidor afeta a todo o sistema. Mesmo as arquiteturas hierárquicas, empregadas para escalar sistemas centralizados, possuem pontos críticos de falha: a queda de nós de níveis importantes na árvore de servidores pode levar ao colapso do sistema.

- Baixo desempenho → Um número de requisições anormal, causado pela quantidade de clientes acima do dimensionado, provoca gargalo de desempenho no servidor e, como conseqüência, em todo o sistema.

- Custo elevado → Os servidores devem possuir poder de processamento adequado (e provavelmente elevado) para atender ao número de clientes esperado. A entrada de clientes acima do esperado implica no aumento do poder de processamento, resultando em custos para adquirir mais hardware e para mantê-lo.

2.1.3 – Características dos sistemas P2P

Em geral, a adoção de um sistema *peer-to-peer* é cercado por certas expectativas. A seguir, são apresentadas algumas características dos sistemas *peer-to-peer*, que podem auxiliar um sistema a alcançar seus objetivos (MILOJICIC *et al.*, 2002):

- Escalabilidade → A expansão da Internet possibilitou mudanças nos requisitos dos sistemas distribuídos, tornando ainda mais crítica a escalabilidade de tais sistemas, um dos principais benefícios trazidos pelo paradigma *peer-to-peer*: a entrada de novos *peers* numa rede amplia os recursos do sistema, ao contrário de um sistema centralizado, no qual a entrada de novos clientes aumenta o consumo de recursos do servidor.

- Descentralização → Característica chave de sistemas *peer-to-peer*, que influencia todo o projeto de um sistema deste tipo. Um sistema puramente *peer-to-peer* não possui pontos de centralização, consistindo numa abordagem completamente descentralizada. A ausência de uma entidade que centralize funções do sistema evita o surgimento de pontos simples de falha e gargalos de desempenho. Entretanto, conforme veremos adiante, nem sempre um sistema *peer-to-peer* é puramente descentralizado.

- Confiabilidade → Com a descentralização, os sistemas puramente *peer-to-peer* se beneficiam de uma maior tolerância a falhas, pois não recaem sobre um ponto simples de falha, como em sistemas que centralizam suas operações em servidores. Entretanto, questões como pontos críticos de falhas (que podem desconectar uma partição do grafo de nós da rede) e a saída de participantes (que afeta a disponibilidade de recursos) podem ser problemas de difícil solução;

- Autonomia →Cada nó em um sistema *peer-to-peer* é autônomo, o que significa que não existe uma entidade que administre os nós do sistema. Cada nó é responsável por si mesmo;

- Compartilhamento de custos →Nos sistemas centralizados, a maioria do custo do sistema recaem sobre seus servidores. Nos sistemas P2P, o custo é compartilhado através da agregação de recursos outrora ociosos. A divisão de custos viabiliza muitas aplicações, como armazenamento e compartilhamento de arquivos;

- Agregação de recursos →Cada nó presente numa rede *peer-to-peer* contribui com poder de processamento, dados, armazenamento e aplicações, agregando mais

recursos ao sistema como um todo. Entretanto, a agregação de recursos impõe um requisito importante: interoperabilidade;

- Dinamismo e Flexibilidade → Um sistema *peer-to-peer* deve prever um comportamento dinâmico em seu ambiente, onde os nós entram e saem da rede freqüentemente, o que não garante o fornecimento contínuo dos recursos computacionais de cada nó. Por isso, sistemas que devem suportar ambientes altamente dinâmicos se adéquam a abordagem P2P;

- Comunicação *ad-hoc* → P2P suportam a formação de ambientes *ad-hoc*, cuja infra-estrutura é construída dinamicamente, com a entrada e a saída de participantes, sem a exigência de uma infra-estrutura previamente estabelecida.

2.1.4 - Redes sobrepostas

Um dos aspectos que favoreceu a popularização das redes *peer-to-peer* foi a possibilidade de estruturá-las como redes sobrepostas (*overlay networks*), que são aquelas construídas sobre a estrutura de outra rede independente (a rede sobreposta passa a consumir os serviços oferecidos pela rede subjacente).

Os nós existentes em uma rede sobreposta são conectados entre si através de caminhos virtuais, que podem não encontrar correspondentes diretos entre os computadores da rede subjacente. A troca de uma mensagem entre dois nós conectados entre si no nível da rede sobreposta, por exemplo, pode percorrer caminhos na rede subjacente que não são contemplados pela rede sobreposta.

Geralmente, as redes *peer-to-peer* são sobrepostas a uma rede física previamente estabelecida, que passa a prover serviços básicos para a comunicação entre quaisquer pares de nós. Sendo uma rede sobreposta, as redes *peer-to-peer* podem ser vistas como uma camada de um nível mais abstrato, que consome serviços da camada inferior para oferecer seus próprios serviços. Enquanto a rede sobreposta acessa os serviços da rede subjacente, esta não tem conhecimento de quaisquer características da rede sobreposta, como seus serviços ou sua topologia.

Os sistemas *peer-to-peer* típicos são sobrepostos a redes IP, o que permite seu uso através da ampla estrutura da Internet, estabelecendo conexões lógicas entre nós existentes em sua borda. Desta forma, um sistema *peer-to-peer* é implementado no nível

da camada de aplicação da pilha de protocolos TCP/IP, utilizando a infra-estrutura do núcleo da Internet (roteadores e gateways) de forma transparente. Entretanto, para os nós pertencentes a uma rede *peer-to-peer*, a conexão física entre si, existente na rede subjacente, também é transparente: mesmo que os nós estejam conectados fisicamente (como se estivessem na mesma rede local, por exemplo), uma eventual troca de mensagens entre nós que não estão conectados diretamente no nível lógico (ou seja, dentro da rede *peer-to-peer*) não se beneficiaria da ligação física entre si.

2.1.5 - Heterogeneidade

Muitos dos sistemas existentes não podem ser definidos simplesmente como *peer-to-peer* ou cliente-servidor, sendo classificados dentro de um espaço criado por estes dois extremos. Um exemplo claro desta transição gradual entre os extremos dos modelos *peer-to-peer* e cliente-servidor é a descentralização.

Apesar de descentralização ser uma característica fundamental ao modelo *peer-to-peer*, é muito comum encontrar algum grau de centralização em sistemas que empregam tal modelo. São diversos os motivos que permitem tal relaxamento em algum ponto do projeto de um sistema, e a complexidade de projeto é um deles: ao se deparar com a complexidade causada pela implementação descentralizada de uma determinada funcionalidade, o projetista pode optar por centralizar alguns aspectos da funcionalidade para simplificar sua implementação.

Uma classificação comum para sistemas *peer-to-peer* é quanto à heterogeneidade de seus *peers*, que pode ser pura, híbrida ou baseada em super-*peers*.

Nos sistemas puros, um *peer* qualquer tem a mesma capacidade dos demais.

Nos sistemas híbridos, alguns nós passam a funcionar como servidores para os outros *peers*, gerando algum grau de centralização em certas funções do sistema.

Já nos sistemas baseados em super-*peers*, existe uma rede *peer-to-peer* que agrupa todos os super-*peers* numa abordagem pura. Ligado a cada super-*peer*, existe uma sub-rede *peer-to-peer*, cujos *peers* vêem o seu super-*peer* como um servidor, numa abordagem híbrida. Entretanto, internamente a esta sub-rede, um *peer* qualquer possui a mesma capacidade dos outros *peers* ordinários, como numa abordagem pura.

A abordagem baseada em super-*peers*, portanto, é uma escolha intermediária entre a pura e a híbrida, e seu objetivo é reduzir os problemas de desempenho dos

sistemas puros, sem cair nos problemas dos sistemas híbridos, como um ponto simples de falha.

2.1.6 - Redes estruturadas e redes não-estruturadas

Diversas taxonomias para a classificação de redes *peer-to-peer* foram propostas na literatura, mas uma das mais simples e difundidas é a classificação segundo a topologia da rede, pela qual as redes *peer-to-peer* podem ser classificadas em estruturadas ou não-estruturadas. O principal objetivo desta classificação é fornecer uma pista sobre os objetivos do sistema: em geral, as topologias estruturadas dão ênfase na localização eficiente dos dados distribuídos pela rede, uma funcionalidade chave em diversos sistemas *peer-to-peer*.

As redes *peer-to-peer* estruturadas são aquelas em que a topologia da rede sobreposta não é construída de forma aleatória, mas sim através de algum conjunto formal de regras. A construção estruturada tem como principal objetivo facilitar as buscas por recursos da rede, como consulta a dados distribuídos pelos nós, através do conhecimento das regras aplicadas durante a construção da topologia rede. O interesse por este tipo de rede cresceu em razão da necessidade do gerenciamento de grandes quantidades de dados e de consultas eficientes sobre essas massas de dados, o que colocava a escalabilidade das redes não-estruturadas à prova. Entretanto, o gerenciamento da topologia executado pelas redes estruturadas implica em algum custo computacional devido a sua complexidade. As topologias estruturadas são tipicamente baseadas em tabelas de dispersão distribuídas (DHT, *distributed hash table*).

Por outro lado, as redes não-estruturadas apresentam uma topologia formada sem procedimentos formais, aleatoriamente, sobre a qual não reside um gerenciamento sofisticado como nas redes estruturadas, apesar de algumas regras simples poderem ser empregadas para a sua construção. A aplicação Gnutella (GNUTELLA, 2001) é um exemplo disto: apesar da topologia ser construída sobre um conjunto de regras simples, é uma rede tipicamente não-estruturada, onde a consulta por arquivos não utiliza conhecimento algum sobre a topologia da rede, e recai sobre a técnica de inundação de consultas.

Apesar de ser um dos mecanismos mais populares para consultas em redes não-estruturadas, a inundação de consultas é bastante ineficiente para redes com números

elevados de nós, o que torna sistemas como Gnutella pouco escaláveis nestas circunstâncias (PORTMANN, 2001).

Apesar do baixo desempenho na descoberta de novos nós e no tratamento de consultas, o fraco gerenciamento da topologia imposto pelas redes não-estruturadas tem como uma de suas principais vantagens o suporte ao dinamismo das redes *peer-to-peer*, onde seus nós entram e saem da rede sem previsibilidade, aspecto de tratamento mais complexo nas redes estruturadas. Caso um nó de uma rede não-estruturada seja desconectado, o procedimento de conexão pode ser executado novamente, de forma simples e eficiente.

Entretanto, em uma rede estruturada que emprega o mecanismo DHT (*Distributed Hash Table*), a entrada ou saída de um nó resulta na execução de O(*log* n) operações para manter as estruturas de roteamento do algoritmo. Se a saída de um nó se dá de forma abrupta, o algoritmo ainda sofre um *overhead* extra para a detecção da ausência do nó e para a replicação de eventuais dados. Além disso, DHT traz benefício imediato para buscas baseadas em consultas exatas, mas o suporte a consultas baseadas em palavras-chave pode não ser tão eficiente devido ao custo da manutenção do índice durante a transiência dos participantes da rede. Nas soluções que também empregam *cache* para desafogar os nós que indexam os termos mais procurados, a transiência dos nós impõe mais desafios (CHAWATHE, 2003).

2.1.7 - Descoberta de recursos

Em um sistema *peer-to-peer*, seus participantes colaboram oferecendo recursos computacionais. Para que os recursos oferecidos pela rede P2P sejam utilizados por um *peer*, deve haver um mecanismo que possibilite ao *peer* obter informações sobre os recursos compartilhados, como a sua localização ou quais *peers* o oferecem. Para obter estas informações, o *peer* interessado realiza consultas que são processadas pelo sistema. Para processar consultas, um sistema *peer-to-peer* pode empregar diversas estratégias, como a seguir:

- Diretório centralizado → Forma mais simples de suporte a consultas. Nesta abordagem, os *peers* da comunidade inserem informações sobre seus recursos em um serviço de diretório central. Para processar uma consulta, um *peer* acessa o diretório, que através de uma consulta ao seu índice, retorna o identificador do *peer* que possui o recurso desejado. O *peer* que contém o recurso é contatado, e

uma conexão direta entre os dois *peers* é estabelecida. Obviamente, o ponto fraco desta abordagem é requerer uma infra-estrutura centralizada para manter informações sobre todos os participantes do sistema, o que pode resultar em problemas de escalabilidade.

- Inundação de consultas → Técnica puramente P2P, que opera de forma descentralizada na busca por recursos. A técnica de *flooding* é muito usada por sistemas descentralizados não estruturados. Um nó que deseja realizar uma consulta envia tal consulta a todos os seus vizinhos. Por sua vez, um vizinho que não pode respondê-la encaminha a consulta a todos os seus vizinhos. A propagação da mensagem segue, assim, sendo realizada entre os nós. Se um nó possui a resposta para a consulta, ele estabelece um contato com o nó que está realizando a consulta, mas sem interromper a transmissão da mensagem para seus vizinhos. Para evitar que uma consulta, obsoleta ou não, continue se propagando para todos os nós da rede, recorre-se ao mecanismo contagem *time-to-live* (TTL): a consulta possui o atributo contador *ttl*, inicializado com um valor pelo nó que lançou a consulta. Cada vez que um nó repassa uma consulta a seus vizinhos, ele decrementa o contador da instância da consulta que possui. Caso uma instância da consulta em poder de um nó tenha *ttl* nulo, a consulta não é repassada para seus vizinhos. Implementações desta técnica devem balancear duas propriedades: escalabilidade e cobertura. Se o objetivo da consulta é cobrir toda a rede (ou seja, sem uso de *time-to-live*), provavelmente a escalabilidade será comprometida. Por outro lado, para obter escalabilidade nas consultas, a cobertura será afetada.

- Roteamento de documentos →Normalmente empregado em sistemas com topologia estruturada. Nesta abordagem, os nós da rede possuem identificadores únicos, e cada novo recurso compartilhado também recebe um identificador, obtido através da aplicação de uma função de *hash* aos seus metadados. Quando um recurso é compartilhado, ele é roteado até um determinado nó, cujo identificador seja similar ao seu. Ao se realizar uma consulta, aplica-se sobre ela a mesma função de *hash* utilizada anteriormente e, como resultado, se obtém o identificador do recurso desejado. Com este identificador, é possível recuperar o nó que possui o recurso. Esta abordagem, portanto, é baseada em palavra-chave,

o que representa uma desvantagem em relação às consultas baseadas em diretório central ou inundação, já que, nestes casos, ainda é possível retornar resultados similares. Na prática, esta estratégia é implementada com o uso de tabelas de dispersão distribuídas (DHT, *distributed hash table*).

2.1.8 - Protocolos e Aplicações

A seguir, alguns exemplos de sistemas ou protocolos P2P são apresentados.

2.1.8.1 - CHORD

Chord (STOICA, 2001) é um protocolo baseado em tabelas de dispersão distribuídas (DHT), e foca na busca por recursos distribuídos em uma rede *peer-to-peer*. O seu princípio é simples: através de uma função de dispersão, mapeia uma chave a um nó, que será o responsável por armazenar o dado associado àquela chave.

Para tal, Chord utiliza uma variante da dispersão consistente (KARGER, 1997), uma técnica baseada em funções de dispersão, que ainda traz alguns benefícios, como balanceamento de carga. Para definir a localização de um recurso na rede, o protocolo Chord aplica uma função de dispersão sobre dois ingredientes: o endereço de um nó (seu endereço IP), para que seja possível localizar o recurso de forma direta, e uma chave que identifica o recurso a ser recuperado. A aplicação da função de dispersão (que emprega aritmética de módulo 2^m) resulta em identificadores de m bits, que representam posições em uma tabela circular de módulo 2^m, formado pelo espaço de endereçamento da função de dispersão. O tamanho do identificador é escolhido de forma a reduzir colisões na tabela de dispersão circular.

Os identificadores de todos os nós da rede, obtidos através da aplicação da função de dispersão sobre cada um dos endereços de nós, são mapeados em posições da tabela circular, em ordem crescente. O mesmo é feito para o identificador obtido através da aplicação da função sobre a chave que identifica o recurso. Para uma chave k, Chord define a função *sucessor(k)*, que identifica o nó responsável pelo recurso: se o identificador k de um recurso representa uma posição no espaço de endereçamento associada ao identificador de algum nó, então *sucessor(k)* devolve o identificador de tal nó. Caso não exista nó com identificador associado àquela posição no círculo, então *sucessor(k)* devolve a próxima posição do círculo que contenha um nó, seguindo pelo sentido horário do círculo:

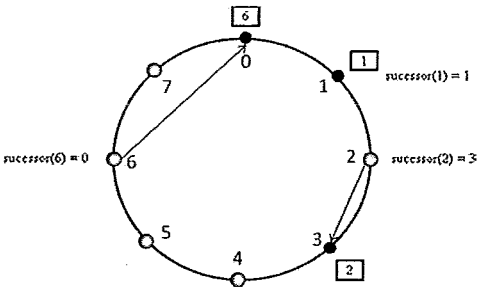


Figura 1 – Exemplo de uma topologia que utiliza o protocolo Chord

Na figura 1, três nós foram mapeados para as posições 0, 1 e 3 deste círculo, como resultado da aplicação da função de dispersão sobre seus endereços. O recurso de identificador 1 foi mapeado no nó presente na mesma posição do círculo. O recurso associado ao identificador 2 não encontrou nó presente na posição 2 do círculo, e, por isso, foi alocado na próxima posição que apresentou algum nó presente (posição 3). O mesmo aconteceu para o recurso de identificador 6, alocado no próximo nó presente, o da posição 0 do círculo.

2.1.8.2 – FreeNet

FreeNet (CLARKE *et al.*, 2001) é um sistema de compartilhamento de arquivos de topologia não-estruturada, puramente *peer-to-peer* e, portanto, completamente descentralizado. O sistema utiliza a técnica de propagação de consultas, mas de forma seletiva. Cada arquivo é associado a um identificador (em geral, através de uma função de *hash*), e o sistema procura agrupar arquivos de identificadores semelhantes em um mesmo nó. As consultas são propagadas, mas de forma seletiva, com o auxílio de tabelas de roteamento existentes em cada nó. Ao receber uma consulta, um nó redireciona a consulta (caso não possua o arquivo) ao vizinho cuja chave é mais similar a chave da consulta. Quando um arquivo é encontrado, ele é retornado recursivamente pelos nós, que armazenam uma réplica local e atualizam suas tabelas de roteamento. Arquivos pouco acessados e entradas mais antigas das tabelas de roteamento são removidos, o que beneficia arquivos populares.

2.1.8.3 – JXTA

JXTA (SUN, 1999) é uma especificação de plataforma para o desenvolvimento de aplicações *peer-to-peer*. A arquitetura é baseada em *microkernel*, o que significa que

o seu núcleo apenas coordena a comunicação entre os módulos (*plugins*) adicionados ao sistema. A plataforma tem como objetivo prover o substrato para o projeto de aplicações P2P, abstraindo a implementação dos serviços de comunicação, permitindo o foco do desenvolvimento na camada de aplicação. Dentre os serviços providos, estão: gerenciamento de grupos de *peers*, descoberta de *peers*, publicação dos serviços oferecidos por um *peer* e descoberta dos serviços providos por um *peer*. Por tratar apenas de serviços de baixo nível, vários *frameworks* foram construídos sobre o JXTA para oferecer mais camadas de abstração aos desenvolvedores.

2.1.8.4 – Napster

Napster (FANNING, PARKER, 2002) é um sistema de compartilhamento de arquivos, de topologia não-estruturada, que adota uma abordagem híbrida para suportar consultas. Quando deseja realizar uma consulta, um *peer* acessa um servidor central, que informa o endereço de *peers* que possuem os recursos desejados. O *peer* então se conecta a um dos *peers* da lista para recuperar o arquivo diretamente. Como qualquer abordagem híbrida, existe o risco de gargalos de desempenho e de falhas, devido à existência de um ponto simples de falha.

2.1.8.5 – Gnutella

Gnutella (FRANKEL, PEPPER, 1999) é um protocolo de comunicação para realização de buscas em redes *peer-to-peer* de topologia não-estruturada. A consulta é realizada através da inundação de consultas, com um valor de TTL (*time-to-live*) configurado para evitar que a consulta seja propagada além de um número de *peers*. Mais tarde, adotou-se a arquitetura de super-*peers*. Os super-*peers* auxiliam nas consultas, informando os clientes que possuem o recurso desejado.

2.1.8.6 – SETI@home

O projeto SETI@home (BERKELEY, 1999) é um sistema que analisa grandes massas de dados obtidas a partir da coleta de rádio transmissões, recebidas por um telescópio, com o objetivo de identificação de padrões de onda desconhecidos, que possam ter se originado de fontes extraterrestres. A análise utiliza o poder de processamento de computadores ociosos, conectados à Internet.

A arquitetura do sistema conta com um servidor, que particiona a massa de dados de forma que cada cliente possa processá-la. Apesar de descentralizar o processamento, esta abordagem centraliza uma tarefa essencial do sistema, da qual todos os clientes dependem (a partição dos dados e coleta dos resultados). Com o crescimento do número de clientes, e o aumento do poder de processamento dos mesmos (obedecendo à lei de Moore), rapidamente o servidor saturou. Este caso mostrou como uma abordagem híbrida pode degradar o desempenho de todo o sistema.

2.1.9 - Conclusão

A descentralização faz parte da natureza de sistemas P2P, e dá ênfase à posse e ao controle dos dados e dos recursos pelos usuários, favorecendo a igualdade entre os nós da rede, e tendo como um de seus benefícios imediatos uma escalabilidade, em geral, superior a dos sistemas centralizados.

A entrada de *peers* em um sistema P2P implica em agregação de recursos (mais recursos passam a serem oferecidos pela rede) e em escalabilidade. Portanto, é interessante o aumento do número de nós, o que não costuma ser verdade nas arquiteturas centralizadas. Sob a ótica da Economia, o número de nós em uma rede *peer-to-peer* também é importante para agregar valor à rede, através do chamado efeito rede (*network effect*) (KATZ, 1994): a demanda pelos serviços oferecidos por uma rede de comunicação também é função do tamanho esperado desta rede. Um sistema de telefonia (exemplo clássico do "efeito rede"), por exemplo, tem mais valor para um indivíduo conforme o universo de usuários do sistema seja maior.

Entretanto, é importante observar que a centralização ainda é interessante em algumas aplicações. Muitas arquiteturas não empregam *peer-to-peer* de forma pura, recaindo sobre a arquitetura cliente-servidor para a realização de algumas tarefas de desempenho crítico, como consultas. Um exemplo são alguns sistemas P2P para compartilhamento de arquivos que adotam um índice centralizado, acessado para localizar alguns nós que satisfaçam à consulta por um determinado arquivo. Estas são as abordagens híbridas, e, em muitos casos, são suficientes para tratar alguns aspectos da aplicação, mesmo que implique no surgimento de um ponto simples de falha. Outro motivo para a adoção da centralização é a facilidade de projeto: sistemas críticos em

relação a controle de acesso e a segurança devem considerar a adoção de uma abordagem centralizada, devido ao menor risco de falhas críticas no projeto.

Finalmente, após a observação dos protocolos e dos sistemas *peer-to-peer*, percebe-se que os sistemas *peer-to-peer* diferem entre si na forma como trocam seus dados, distribuem suas tarefas e localizam informações ao longo da rede.

2.2 – Sistemas Emergentes

Na natureza, conceitos como descentralização, tolerância a falhas e escalabilidade estão presentes em colônias de insetos, cardumes, alcatéias e até mesmo no cérebro humano. Este capítulo apresentará conceitos relacionados a sistemas adaptativos complexos (CAS) ou, simplesmente, sistemas emergentes. Tais sistemas buscam inspiração na forma como a natureza trata os conceitos citados e será o paradigma utilizado neste trabalho para alcançar seu objetivo.

Também será dada atenção aos conceitos que norteiam sistemas multi-agentes (MAS), com foco no principal ingrediente que utilizaremos: o agente. Aliás, os principais objetivos são a compreensão do que é um agente e como eles constituem uma solução sinergética com o paradigma P2P para aplicação em sistemas descentralizados e distribuídos.

2.2.1 - Comportamento Emergente

Em muitos casos, o comportamento de um sistema, computacional ou não, pode ser inferido a partir da compreensão do comportamento das partes menores que o compõe. Entretanto, para uma determinada classe de sistemas, a compreensão individual de seus elementos não é suficiente para inferir o comportamento sistêmico.

Comportamento emergente é aquele que não pode ser previsto através de uma análise num nível mais simples que o sistema como um todo (DYSON, 1997). Ou seja, não é possível compreender o comportamento sistêmico a partir da compreensão do comportamento de elementos menores. Dentro do conceito de emergência, o comportamento global surge de interações entre partes mais simples.

2.2.1.1 - Exemplos de Comportamentos Emergentes

Emergência não é um conceito exclusivo da área de computação, sendo estudado na meteorologia, na cosmologia, na filosofia, na economia, na sociologia, e entre várias outras áreas do conhecimento. A natureza possui vários exemplos de comportamentos emergentes, como as colônias de insetos, o cérebro humano, a formação do universo, a evolução da vida na Terra, entre outros.

O objetivo dos caminhos construídos pelas formigas é ligar seu ninho a fontes de alimentos, sempre buscando minimizar tais caminhos para economizar energia durante o transporte do alimento até o seu ninho. No entanto, estas estruturas otimizadas não são criadas individualmente por um único indivíduo, através de algoritmos: elas emergem de simples interações entre os indivíduos do ninho (PARUNAK, 1997). Na ausência de feromônios, cada indivíduo responsável pela caça de alimentos se move aleatoriamente, seguindo o chamado movimento browniano, um conceito aplicado na Física de partículas que representa o movimento aleatório de partículas em um fluido. Cada possível direção é escolhida a partir de uma distribuição uniforme sobre as possibilidades existentes. Na presença de feromônios no ambiente, o movimento também é aleatório, mas a escolha é ponderada em relação à direção de onde o feromônio emana. Ao encontrar alimentos, estando desocupada, a formiga o transporta para o ninho, guiado por um tipo distinto de feromônio emitido por ele. Ao encontrar o ninho, a formiga libera o alimento. O planejamento do caminho pelas formigas revela como um sistema natural pode ter um comportamento que emerge a partir da interação de agentes que seguem regras bem definidas e bastante simples.

2.2.2 - Sistemas Adaptativos Complexos (CAS)

Um sistema adaptativo complexo consiste de um grande número de agentes autônomos, que individualmente possuem comportamentos muito simples, e que interagem entre si também de forma simples. Apesar da simplicidade de seus componentes, o comportamento de um CAS é complexo e imprevisível, revelando-se emergente (BABAOGLU *et al.*, 2002).

Sistemas Adaptativos Complexos (CAS), ou apenas sistemas emergentes, são aqueles compostos por um grande número de agentes que podem interagir entre si, e com o seu ambiente, de forma que os objetivos sistêmicos possam ser alcançados, mesmo que alguns agentes falhem ou ocorram mudanças ambientais (MIRANDA, XEXEO *et al.*, 2006).

CAS são sistemas adaptativos porque possuem a capacidade de auto-organização. Por auto-organização, entende-se como sendo um processo dinâmico e adaptativo através do qual um sistema se mantém estruturado, sem controle externo. Portanto, tais sistemas se adaptam às mudanças impostas pelo seu ambiente. Uma

análise sobre emergência e auto-organização pode ser encontrada em (DE WOLF *et al.*, 2005).

2.2.2.1 – Estigmatismo

Em diversos sistemas naturais de comportamento emergente, a comunicação entre seus agentes é mediada pelo ambiente, através de marcas aplicadas pelos agentes no mesmo. A esta característica, dá-se o nome de estigmatismo. O estigmatismo está presente na comunicação entre formigas, através da marcação do ambiente com feromônios, o que possibilita a construção de caminhos curtos entre o ninho e as fontes de alimentos.

2.2.2.2 – Heterarquia

Em um sistema hierárquico, os elementos do sistema são classificados em níveis, e o comportamento de um determinado elemento é definido por um elemento de nível superior. Um elemento possui papel centralizador sobre seus subordinados, caracterizando-se como um ponto de centralização. Num sistema heterárquico, por outro lado, seus elementos não são subordinados a outros, mas exercem influência mútua: um agente pode exercer influência sobre o comportamento de outros agentes, através de alguma marca (estigma) aplicado no ambiente. Em tais sistemas, a coordenação é descentralizada.

Mais adiante, neste capítulo, serão apresentados sete princípios observados por Parunak em sistemas naturais que são úteis na engenharia de sistemas multi-agentes. A heterarquia é um desses princípios: mesmo em colônias de formigas, em que existe um indivíduo diferenciado, a rainha, a centralização não ocorre. A rainha não toma nenhuma decisão sobre a atuação das demais formigas, o que também é observado em colméias e em colônias de cupins.

2.2.3 - Emergência e sistemas P2P

CAS é adaptativo a mudanças em seu ambiente, é tolerante a falhas e se auto-organiza em busca de estruturas ótimas. Sistemas P2P podem ser modelados como um CAS, e um dos principais fatores para isso é poder apresentar essas propriedades sem a necessidade de suportá-las diretamente nos agentes (BABAOGLU *et al.*, 2002). Além

disso, como já foi exposto, sistemas multi-agentes (MAS) são uma ótima forma de modelar sistemas descentralizados (WANG *et al.*, 1999, WOOLDRIDGE, 1997), tal como sistemas P2P. Na prática, sistemas P2P podem ser vistos como instâncias de sistemas adaptativos complexos (BABAOGLU *et al.*, 2002).

2.2.4 - Sistemas multi-agentes (MAS)

Um sistema multi-agentes (MAS) constitui uma rede fracamente acoplada de indivíduos, os agentes, que colaboram entre si para resolver um determinado problema. MAS é considerado um ótimo paradigma para modelagem de sistemas de natureza descentralizada e distribuída (WANG *et al.*, 1999, WOOLDRIDGE, 1997), já que a sua distinção é a ausência de coordenação centralizada de atividades.

A aplicação de agentes no desenvolvimento de sistemas complexos encontra adeptos que defendem que a análise e o projeto desta classe de sistemas com o uso de agentes autônomos, interagindo entre si, traz benefícios consideráveis sobre os outros métodos de Engenharia de Software (JENNINGS, 2001, WOOLDRIDGE, 1997, PARUNAK, 1997). Weiss (2000) publicou uma extensa obra sobre a aplicação de MAS na área da Inteligência Artificial.

2.2.4.1 - Agentes

Um agente é um sistema computacional encapsulado, situado em um ambiente, capaz de ações flexíveis e autônomas naquele ambiente para atender aos seus objetivos (WOOLDRIDGE, 1997). Jennings (2001) faz algumas considerações sobre a personalidade de um agente:

- São entidades, claramente identificáveis, solucionadoras de problemas;
- Possuem fronteiras e interfaces bem definidas;
- Estão imersos em um ambiente, sobre o qual possuem controle e visão parciais;
- Possuem um papel específico, com um objetivo bem definido a ser alcançado;
- São autônomos, ou seja, possuem controle de seu estado interno e sobre seu comportamento, o que os tornam capazes de exibir um comportamento flexível na resolução de problemas para alcançar seus objetivos;
- São reativos, respondem a mudanças que ocorrem em seu ambiente;
- São proativos, podem adotar objetivos e tomar iniciativas.

As interações entre os agentes podem possuir uma semântica simples, próxima ao que ocorre na arquitetura cliente-servidor, com simples requisições e respostas, ou chegar a um nível mais complexo, com a cooperação, a coordenação e a negociação sobre ações mútuas (JENNINGS, 2001).

Segundo (RUSSEL, NORVIG, 1995), agente inteligente é tudo que possui capacidade de perceber seu ambiente através de sensores, e atuar sobre aquele ambiente.

2.2.4.2 - A orientação a agentes e a orientação a objetos

É natural que o contato com um novo paradigma induza a analogias com o mundo conhecido. Esta seção traz um breve comparativo entre a orientação a objetos e a orientação a agentes, discutido em (JENNINGS, 2001).

As duas abordagens possuem características comuns, como o encapsulamento de dados e operações, e a necessidade de interações para cumprir com seus objetivos. Entretanto, a natureza dos objetos é passiva, e apenas se tornam ativos e executam alguma ação ao receber uma mensagem para tal. Agentes são ativos por natureza, não dependem de outro fator para serem ativados. A ressalva sobre a passividade dos objetos ocorre por conta de algumas variantes da orientação a objetos (WOOLDRIDGE, 1997), como modelos de programação com objetos ativos (GUESSOUM, 1999).

Outra diferença está relacionada à seleção do comportamento que um objeto realizará. Objetos encapsulam comportamentos em métodos, que são executados dependendo de uma mensagem enviada por outro objeto. Ou seja, a seleção do comportamento de um objeto ocorre fora do seu controle. Um objeto obedece ao comando de outro objeto.

Objetos individuais representam comportamentos de granularidade muito fina e a invocação de métodos é um mecanismo muito primitivo para descrever os tipos de interações possíveis. A partir disto, surgiu necessidade de se elaborar mecanismos de abstração mais sofisticados, como os frameworks, os padrões de projeto e a componentização.

Deve-se ressaltar que implementar um sistema orientado a agentes não significa necessariamente, que objetos não serão utilizados. O modelo de programação com objetos ativos fornece autonomia aos objetos, não havendo necessidade de uma intervenção externa para ativá-los. Isto é palpável através da programação concorrente orientada a objetos, que, portanto, é apropriada para a implementação de agentes

(GUESSOUM, 1999). Ao final deste capítulo, será apresentado o *framework* COPPEER (MIRANDA, XEXEO *et al.*, 2006), cujo objetivo é o suporte a implementação de sistemas multi-agentes através da linguagem orientada a objetos Java (SUN, 1995).

2.2.5 - Princípios existentes em sistemas naturais

Parunak observou alguns princípios que permeavam a auto-organização de sistemas naturais (como as colônias de insetos, os cardumes, as alcatéias e o cérebro humano) e defende que estes princípios auxiliam o projeto de sistemas artificiais multi-agentes (MAS) de comportamento emergente. A seguir, são listados os sete princípios observados (PARUNAK, 1997):

- Modelagem orientada a agentes → Em sistemas naturais, funções emergem das interações entre seus indivíduos. Em um sistema natural, agentes são baseados em entidades distintas existentes no mundo físico (objetos do domínio) e não em funções abstratas resultantes de uma decomposição funcional. Com isso, um agente deve representar coisas, e não funções.

- Agentes minimalistas → Sistemas naturais possuem indivíduos pequenos em relação ao sistema como um todo. Neste contexto, pequeno se refere ao tamanho do código, ao tempo e ao seu escopo. Abaixo, cada um destes três aspectos é explicado:

    - Sua implementação deve ser enxuta, pois agentes compactos são de fácil entendimento e manutenção, em comparação aos módulos de um sistema monolítico. Além disto, cada comportamento trazido por um agente, em combinação com os de outros agentes produz um conjunto maior de possibilidades, resultando num comportamento mais emergente do sistema. Esta abordagem privilegia agentes especializados, em detrimento a agentes genéricos. Uma conseqüência importante é a falha de um indivíduo não comprometer todo o sistema.

    - Seguindo a filosofia dos sistemas naturais, as marcações obsoletas existentes no ambiente devem dar lugar a novas marcações, alimentando

os agentes com novas informações, tal como o feromônio liberado por insetos para demarcar trilhas, que evapora com o passar do tempo.

- O escopo de atuação de um agente deve ser reduzido, atuando apenas nas suas proximidades. Isto não impede, entretanto, que suas ações não possam afetar áreas mais distantes. Em termos de implementação, o alcance de um agente deve ter uma audiência limitada, evitando a propagação de informações para todo o sistema.

- Controle descentralizado → Não deve haver agente complexo que controle o sistema inteiramente, o que caracterizaria sua centralização. Este princípio evita a formação de pontos simples de falha, o surgimento de potenciais gargalos de desempenho, o desenvolvimento de agentes complexos, e favorece a escalabilidade. O autor destaca que mesmo em sistemas naturais, como em colônias de insetos, a centralização não ocorre. Por exemplo, em uma colméia a abelha rainha não envia ordens, mas serve como um barramento de comunicação da colônia.

- Suporte a diversificação de agentes → Agentes devem possuir algum tipo de diversificação. Isto significa ter estados internos distintos entre si que os permitam monitorar diferentes aspectos do ambiente. A diversificação pode ser obtida por aleatoriedade, como no caso da formigas, que possibilita uma diversificação espacial. Ou ainda pode ser obtida através de um mecanismo de repulsão, que ajusta o estado interno para que se afaste dos valores dos estados internos de outros agentes. A operação de caça de uma alcatéia é um exemplo de repulsão, onde os lobos se diversificam espacialmente para cercar a caça.

- Dissipação de entropia → Este princípio está relacionado com a capacidade de auto-organização de um sistema natural. A dissipação, neste caso, se refere ao que ocorre com o feromônio depositado pelos insetos no ambiente. O feromônio evapora gradualmente, depois de depositado, e são percebidos por outros agentes da colônia, que o utilizam para organizar suas ações. Estes indivíduos começam a atuar baseados em suas percepções, reorganizando o ambiente, mas também depositando mais feromônio, caracterizando um sistema de

retroalimentação. Os agentes devem ser capazes de perceber o campo existente no ambiente e se orientar a ele. A circulação de moeda é outro exemplo: empreendedores se orientam segundo o fluxo de capital, resultando na auto-organização de estruturas como um simples centro comercial, ou toda uma complexa cadeia de suprimentos.

- Compartilhamento de informação → Compartilhar informações em uma comunidade reduz custos com buscas dispendiosas, e, por isso, o sistema deve prover mecanismos que possibilitem aos agentes compartilhar e aprender informações.

- Planejar e executar concorrentemente → Sistemas naturais não planejam com antecedência, até porque a antecipação não traz muitos benefícios em ambientes em constante mudança. Ao invés disso, o planejamento realizado em concomitância com a execução é incentivado. A abordagem concorrente não é ótima, mas é uma solução possivelmente melhor do que aquela obtida com um planejamento que depende de um estado que pode nunca ser atingido em tempo de execução.

2.2.6 - Frameworks

Os conceitos de agentes podem ser aplicados a sistemas computacionais. A seguir, duas propostas para suportar o desenvolvimento de projetos baseados em agentes são apresentadas.

2.2.6.1 - O *framework* COPPEER 2.0

COPPEER (MIRANDA, XEXEO *et al.*, 2006) é um *framework* para o desenvolvimento e a execução de aplicações *peer-to-peer* baseadas em agentes (MAS). Como foi escolhido para prover a infra-estrutura básica para a implementação do protótipo deste trabalho, sua arquitetura será descrita em detalhes. A arquitetura do sistema é composta por quatro camadas (MIRANDA, XEXEO *et al.*, 2006):

- Camada de agência → Abstrai todos os demais *peers* como um conjunto de agências colaborativas, encapsulando operações de comunicação com esses. Uma *agência* é uma entidade existente em cada nó, onde residem os agentes.
- Camada de integração → Utiliza os serviços da camada de agência e oferece serviços de locação de aplicações, gerenciamento de sessão e integração remota para as camadas superiores. Locações virtuais de alta disponibilidade são fornecidas para as aplicações, devido a potencial transiência dos nós da rede. Também fornece abstrações de interação entre entidades localizadas em diferentes nós do sistema.
- Camada de colaboração →Utiliza os serviços fornecidos pela camada de integração para fornecer serviços de cooperação, coordenação e comunicação entre os usuários.
- Camada de aplicação → Aplicações do usuário.

A infra-estrutura fornece um ambiente de tempo de execução para CAS, chamado CoppeerCAS, cuja principal entidade é a agência. Uma instância de uma agência é criada em cada *peer* da rede para gerenciar qualquer entidade do sistema existente no *peer*.

Uma *célula* é um espaço compartilhado para leitura e escrita de registros (chamados *entrada*). Além disso, fornece um serviço de notificação de escrita de entradas contendo dados específicos, onde qualquer objeto pode ser inscrito como observador. Quando uma entrada (com um estado interno específico) é escrito na célula, os objetos observadores são notificados.

Uma célula pode ser conectada a qualquer outra célula conhecida. Seus serviços podem ser consumidos por agentes ou objetos da aplicação. Entradas possuem capacidade de propagação entre células, controlada por regras de propagação definidas pelo programador para cada tipo de entrada.

As células distribuídas pelas várias agências podem ser interconectadas formando conjuntos virtuais, chamados *ambiente*. Uma agência pode participar de muitos ambientes simultaneamente, mas apenas pode gerenciar uma célula por ambiente, exatamente aquela existente no seu *peer*.

Agentes são componentes associados a um ambiente, e podem realizar operações com a célula local ou células remotas vizinhas, se deslocar entre agências,

criar novos agentes e executar algoritmos definidos pelo programador. É através deles que a computação distribuída é alcançada.

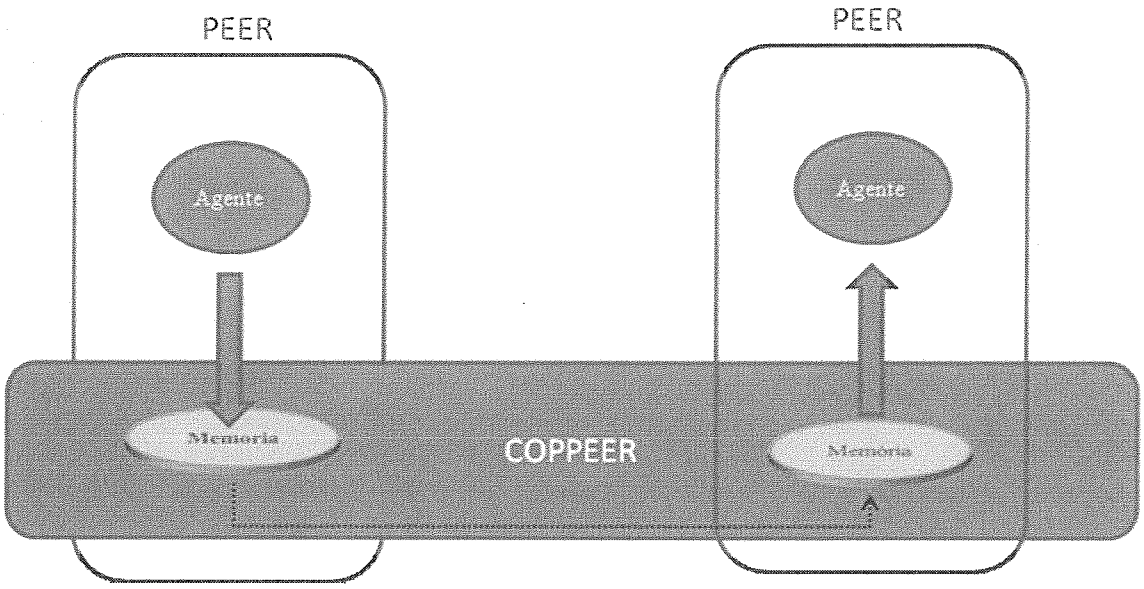


Figura 2 – Comunicação através de peers no COPPEER

2.2.6.2 - Anthill

Anthill (BABAOGLU *et al.*, 2002) é um framework para suporte ao projeto e implementação de aplicações P2P baseadas em idéias de sistemas multi-agentes, e de programação evolutiva emprestadas de CAS. Uma aplicação Anthill consiste dos nós da rede (ninhos) e de colônias de agentes adaptativos (formigas), que viajam pela rede interagindo com outros nós e cooperando com outros agentes para resolver problemas complexos. É similar a proposta do COPPEER, mas a propagação de dados não pode ser realizada sem o auxílio dos agentes.

2.2.7 - Conclusão

Neste capítulo, foram apresentados os conceitos relacionados a comportamento emergente, agentes, a sistemas multi-agentes (MAS) e a sistemas adaptativos complexos (CAS) que darão o suporte básico a este trabalho. Como mostrado, comportamento emergente é um fenômeno presente em várias áreas do conhecimento. A formação e evolução do Universo, a evolução da vida na Terra, os sistemas econômicos, o cérebro humano e vários outros sistemas complexos apresentam comportamento emergente.

Foram mostradas opiniões de autores que defendem que MAS é uma ótima forma de implementação de sistemas descentralizados e que sistemas P2P não apenas é compatível com MAS e CAS, como podem ser modelados como tal.

Um dos objetivos deste trabalho é explorar os benefícios dos sistemas P2P, e para isso, agentes serão usados para suportar descentralização e distribuição na coordenação de workflows.

2.3 – Workflows

Por volta do início dos anos 90, muitas empresas grandes perceberam o quão tinham perdido competitividade. Empresas como Toyota, Dell e Wal Mart redefiniram seus mercados através da inovação de seus processos de negócio, o que lhes garantiu uma boa vantagem competitiva frente a concorrentes que, apesar de tradicionais, não obtiveram o mesmo sucesso nas suas reações. O mercado tinha evoluído, e os gigantes, que hibernavam tranqüilos amparados na sua autoconfiança, demoraram a acordar.

Este capítulo apresenta os conceitos básicos de processos e da sua automação. Também será brevemente discutida a aplicação de workflows na área científica, e o que isto difere da sua aplicação em ambientes corporativos. Conceito de workflow flexível e seus tipos serão também apresentados.

Entretanto, o principal objetivo deste capítulo é levantar os problemas de arquiteturas centralizadas, normalmente empregadas na construção das ferramentas de automação, e discutir como tais problemas poderiam ser endereçados com a aplicação de uma arquitetura descentralizada.

2.3.1 – Processos de negócio e o foco no processo

Uma empresa possui como objetivo produzir produtos ou serviços para atender ao seu público de interesse. Para atingir o objetivo do negócio de uma empresa, um conjunto de atividades ou procedimentos, que definem regras e relacionamentos, deve ser executado para produzir os produtos ou os serviços para o seu mercado ou os seus consumidores. A este conjunto de atividades dá-se o nome de processo de negócio. Ellis (1999) define um processo simplesmente como um conjunto pré-definido de passos de trabalho (*work steps*) e a ordenação parcial desses passos. Davenport (1993) definiu processos como sendo a estrutura pela qual as organizações fazem o que é preciso para produzir valor a seus clientes. Um processo de negócio tem duração bastante variável, podendo se extender por minutos, dias ou até anos.

O movimento de convergência dos esforços das diversas áreas de uma organização, para suportar o seu negócio de forma cada vez mais eficiente, teve importância proporcional ao aumento do dinamismo da economia global, ocorrida nas últimas décadas. Com objetivo na eficiência organizacional, diversos autores aderiram à

idéia de foco no processo do negócio, ou seja, em como se faz, em detrimento ao foco no produto, ou seja, no que é feito. Segundo Davenport (1993), focar no processo implica em adotar o ponto de vista do cliente.

2.3.2 – Workflows: automatizando processos de negócio

Um workflow é a automação de um processo de negócio, completa ou parcialmente, através do qual os documentos, a informação ou as tarefas são passadas de um participante para outro por ações, de acordo com um conjunto de regras procedimentais, para contribuir com o objetivo do negócio (Hollingsworth, *et al.*, 1994).

As ferramentas que suportam a automatização podem utilizar diversas tecnologias e operar em ambientes heterogêneos, como na Internet através de *web services* (W3C, 2004).

É comum que workflows estejam relacionados a esforços de reenganharia de processos, que envolve a análise, a modelagem e a implementação dos processos de negócio chaves nas organizações. Reengenharia não implica, necessariamente, em automatizar processos, mas esta aproximação costuma ser interessante.

2.3.3 – Definições e conceitos básicos de workflows

A forte demanda por apoio da área de tecnologia da informação para suporte a processos de negócio incentivou o surgimento de diversas soluções para automação de processos de negócio no mercado. Entretanto, a ausência de padrões impossibilitava a interoperabilidade das diversas soluções de automação existentes.

Com o objetivo de concentrar os esforços em favor da padronização, foi estabelecida a *Workflow Management Coalition* (WfMC), uma organização sem fins lucrativos, formada por diversas companhias, que promove e desenvolve a tecnologia de workflows, elaborando a terminologia e os padrões para interoperabilidade e conectividade entre as diversas soluções de workflow. Ao longo do tempo, uma terminologia sobre workflows foi constituída e amplamente empregada pelas corporações. A WfMC publicou um glossário que descreve os termos usados em suas especificações, seus padrões e seus modelos, alguns destes apresentados a seguir (Hollingsworth, *et al.*, 1994):

2.3.1.1 - Sistema de Gestão de Workflow (WfMS - *Workflow Management System*)

Os sistemas de gestão de workflows são aqueles que provêem a automação de processos de negócio, possibilitando a definição, a gestão e a execução de um workflow, coordenando a sequência das suas atividades e a invocação de seus recursos (que podem ser humanos ou computacionais).

2.3.1.2 – Subprocesso

É um processo, executado ou invocado por outro processo (ou subprocesso) do qual faz parte. O conceito de passo de trabalho (*work step*) (Ellis, 1999) representa, na verdade, um subprocesso.

2.3.1.3 - Definição processos

A definição de um processo é a representação textual, gráfica ou em uma linguagem formal que possibilite seu entendimento por um sistema de gestão de workflows. Compreende um conjunto de passos discretos das atividades do processo, que podem representar ações humanas ou computacionais, com regras que governam a evolução da execução do processo. Expressa os relacionamentos entre as atividades, com informações sobre o início e fim do processo, participantes, sistemas e dados envolvidos.

2.3.1.4 - Atividade

Uma atividade representa um passo lógico dentro de um processo. Pode ser automatizada ou manual. Uma atividade automatizada é aquela que pode ser computacionalmente automatizada, com um sistema de gestão de workflows gerenciando a sua execução. Mesmo que um participante processe seu trabalho independentemente do sistema de gestão de workflow, mas notifique o sistema ao fim do seu trabalho, a atividade é dita automatizada. Atividades manuais, por outro lado, não são capazes de serem automatizadas, e permanecem fora do escopo do sistema de gestão empregado. Apesar de representadas na definição do processo, fazendo parte da modelagem do processo, não fazem parte do workflow constituído.

2.3.1.5 - Instância de um Processo ou de uma Atividade

Uma instância é a representação de uma única execução de um processo, ou de uma atividade dentro da instância de seu processo. Cada instância de um processo representa um contexto separado da execução deste processo, possuindo seu próprio estado interno e uma identidade única, que a identifica externamente. Todo o ciclo de vida de uma instância é gerenciado pelo seu sistema de gestão de workflow.

2.3.1.6 – Participante

Um participante é um recurso que executa o trabalho representado por uma instância de uma atividade do workflow. Normalmente, o termo é aplicado a recursos humanos, mas também pode ser aplicado a recursos computacionais, como agentes inteligentes. Nestes casos, o termo mais comum é "aplicação invocada" (*invoked application*).

2.3.1.7 - Item de Trabalho

Um item de trabalho representa um trabalho a ser executado por um participante, dentro do contexto de uma atividade. Em geral, uma atividade consiste de um ou mais itens de trabalho. Em geral, os itens de trabalho da atividade são apresentados ao participante através de uma lista de trabalho (*work list*), que apresenta os detalhes das tarefas exibidas ao usuário.

2.3.2 – Arquitetura de referência da WfMC

A WfMC (*Workflow Management Coalition)* possui um modelo de arquitetura de referência que apresenta uma proposta de arquitetura para um sistema de gestão de workflows, exibindo as interfaces mais importantes que a *engine* deve suportar (Figura 3). Desde a sua publicação, o modelo tem sido amplamente aceito pela indústria para o desenvolvimento das soluções de workflow, sendo que o principal motivo para sua existência é apresentar a separação das várias funções dentro de um workflow e identificar vários pontos de interface através dos quais se dá a integração entre os produtos existentes no mercado.

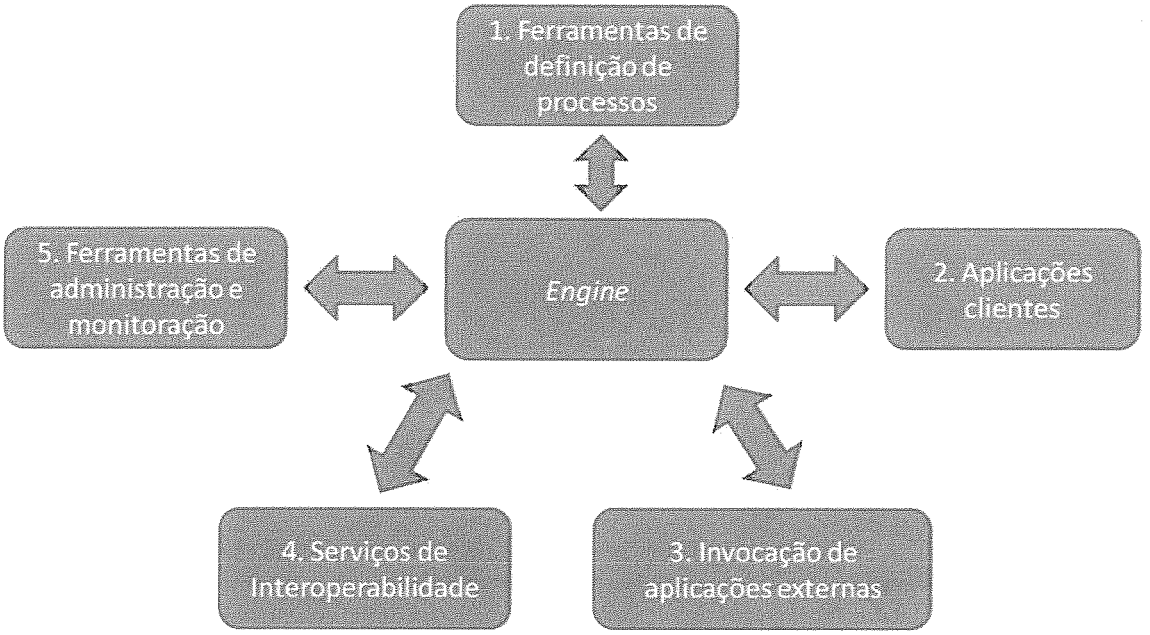


Figura 3 – Arquitetura de referência da WfMC

O modelo de referência apresenta as várias interfaces entre a *engine* e outros componentes, descritas a seguir (HOLLINGSWORTH *et al.*, 1994):

- Interface para ferramentas de definição de processos (1) → Define uma interface padrão entre as ferramentas de definição de processos e a *engine* de workflow. A ferramenta deve produzir uma especificação do processo que seja entendido pela *engine*.
- Interface para aplicações clientes (2) → Define uma interface de programação (API) comum que permite a aplicação cliente enviar requisições a *engine*.
- Interface para aplicação invocada (3) → Deve ser possível a *engine* invocar aplicações externas, o que é realizado através de uma interface de programação (API) padronizada.
- Interface de interoperabilidade (4) → Define padrões a serem seguidos pelas *engines* para possibilitar a interoperabilidade entre diferentes sistemas de gestão de workflows.
- Interface de administração e monitoração (5) → Interface que provê serviços para as tarefas de monitoração e o controle da execução da *engine*.

2.3.4 - Diferenças entre workflows científicos e workflows de negócio

Enquanto a tecnologia para gestão de workflows evoluía, surgindo diversas ferramentas de apoio, e consolidando-se diversos padrões e artefatos através da formação do WfMC *(Workflow Management Coalition)*, atendendo ao ambiente corporativo ao suprir as suas necessidades de controle dos seus processos, a área científica se absteve do emprego de tais ferramentas e padrões.

Existem requisitos de workflows que são únicos ao ambiente científico, mas, por outro lado, muitos requisitos da área científica se sobrepõem aos aplicados nos workflows de negócio.

Nos workflows de negócios, eventos e fluxo de controle recebem destaque, enquanto que o fluxo de dados é uma preocupação secundária (LUDASCHER, 2002), ao contrário dos sistemas de workflows científicos, que devem ter suporte robusto ao fluxo de dados. Tal diferença também pode ser percebida nas representações formais dos workflows de negócio, como o diagrama de atividades da UML, que possuem uma preocupação maior com fluxo de controle e representação de eventos.

Quanto ao modelo de execução, os sistemas de workflows científicos são bastante próximos às redes de processamento de fluxo de dados (*dataflow process network*), que são aplicadas a áreas como em processamento de sinais e em instruções de baixo nível de algumas arquiteturas de CPU, e, possuem uma relação estreita com linguagens funcionais (LUDASCHER, 2002). Por outro lado, o modelo de execução para workflows de negócios possuem proximidade com redes de *Petri*, vide seu emprego em análises de fluxos de controle de processos de negócio (KIEPUSZEWSKI, 2002).

2.3.5 - Workflows flexíveis

Tradicionalmente, os sistemas de gestão de workflows focam em processos bem definidos, que seguem uma metodologia mais rígida e possuem uma especificação formal. Esta visão simplificada acarretou a inflexibilidade tanto dos sistemas de gestão de workflows, quanto das linguagens de representação formal de processos, não sendo, portanto, algo intrínseco ao paradigma de workflows (JOERIS, 1999).

Como resultado, trabalhos foram realizados para buscar estratégias mais flexíveis. Surgiu o conceito de workflows flexíveis, que procuram tornar a definição dos processos mais adaptáveis a alterações durante a sua execução.

2.3.5.1 - Classificação

A seguir, é apresentada uma classificação para sistemas de gestão flexível de workflows, baseado em duas possíveis abordagens para se atingir a flexibilidade (HEINL *et al.*, 1999).

2.3.5.1.1 - Flexibilidade por Seleção (ou Ponto Aberto)

Também conhecida como ponto aberto (*open point*), flexibilidade baseada em caixa-preta, ou ainda, flexibilidade *a-priori* (JOERIS, 1999), foca na etapa de especificação do workflow, oferecendo um relaxamento na execução de um workflow ao permitir múltiplas alternativas de fluxo de execução, dando mais liberdade ao usuário. Busca contemplar prováveis futuras mudanças. É menos restritivo quanto ao avanço do fluxo de execução e é mais interessante quando um grande número de fluxos de execução é conhecido *a priori*, ou podem ser antecipados. Existem duas principais formas de se implementar este tipo de seleção, baseadas no momento em que serão aplicadas. Na modelagem antecipada, a liberdade provida para a execução do workflow já fica definida na etapa da definição, e os diversos fluxos possíveis ficam especificados explicita ou implicitamente pela linguagem de definição do workflow. Na modelagem tardia, certas partes do workflow não são modeladas durante a sua definição, ficando em abertas até o momento de sua execução. A linguagem de especificação do workflow que suporta este tipo o faz através de construções chamadas caixa-preta. Os caminhos deixados em aberto são previstos durante a definição, mas não concretizados, o que ocorre durante a execução do workflow. Em ambas as formas de modelagem, as alternativas de fluxos sempre são previstas, mas a diferença é relativa ao momento em que as alternativas serão, de fato, definidas (HEINL *et al.*, 1999).

Como destacado por (VIEIRA, 2005), a flexibilidade por seleção não tem uma adoção ampla, e a sua principal desvantagem é a dificuldade imposta ao analista do processo, que deve prever a localização dos pontos abertos.

2.3.5.1.2 - Flexibilidade por Adaptação

A flexibilidade por seleção requer previsibilidade para as alternativas de fluxos de execução, o que pode ser impossível para alguns cenários que exigem mais flexibilidade. Por exemplo, mudanças ocorridas no ambiente, que não tenham sido previstas durante a fase de especificação, não seriam suportáveis via seleção. Para tais casos, implementa-se a flexibilidade por adaptação (também conhecida como flexibilidade *a-posteriori*), que envolve a adição de novos fluxos de execução, não previstos durante a especificação, para suprir aos novos requisitos do mundo real. A adição de caminhos pode ser realizada tanto no nível da especificação, quanto no nível de instância do workflow. A modificação realizada no nível da definição do workflow, chamada de adaptação por tipo, não afetam as instâncias em execução. Já na chamada adaptação por instância, algumas ou todas as instâncias são afetadas por modificações na especificação do workflow (HEINL *et al.*, 1999). Na flexibilidade por adaptação, a consistência do processo deve ser garantida, observando-se quando e em quais circunstâncias as alterações são permitidas (JOERIS, 1999).

2.3.6 - Gestão Centralizada de Workflows

Tradicionalmente, os sistemas de gestão de workflows adotam a arquitetura cliente-servidor, que é o paradigma mais empregado na construção de sistemas distribuídos. Sua adoção é justificável, pois além de ser uma arquitetura bastante difundida e madura, satisfaz aos requisitos funcionais dos sistemas de gestão de workflows, possibilita o emprego de clientes leves, provê facilidade para monitoração e para auditorias, é atendido por mecanismos simples para sincronização de seus recursos e não impõe desafios complexos ao projeto e à implementação dos sistemas de gestão de workflows (ALONSO *et al.*, 1995). O emprego de tal arquitetura foi dominando naturalmente o desenvolvimento dos sistemas de gestão de workflows, à medida que os seus benefícios satisfaziam aos requisitos funcionais nos domínios de aplicação contemplados.

2.3.6.1 - A arquitetura Cliente-Servidor

Na abordagem cliente-servidor, um computador servidor hospeda o sistema de gestão de workflow e disponibiliza seus serviços a computadores clientes através da rede. Quando um cliente necessita de algum serviço oferecido pelo servidor, serviço é

consumido pelo cliente. Ao contrário dos servidores, os clientes não compartilham seus recursos computacionais.

Dentre as vantagens das arquiteturas cliente-servidor, estão a facilidade da gerência dos dados, o tráfego de rede reduzido e a facilidade de segurança. A gerência de dados se beneficia da centralização dos dados, permitindo que manutenções corretivas e gerenciamento de réplicas sejam menos dispendiosos. O servidor manipula grande massa de dados para processar requisições, e, em geral, sua resposta para os clientes apenas contém o necessário a estes, o que contribui para um menor impacto sobre a rede. A segurança pode ser realizada em vários níveis de permissões, protegendo recursos e possibilitando rastreamento de alterações, o que facilita questões de auditoria.

Entretanto, a arquitetura cliente-servidor apresenta algumas desvantagens importantes. Os servidores devem ser máquinas com grande poder de processamento, o que gera altos custos de manutenção. Além disso, a indisponibilidade de um servidor impacta em todos os seus clientes, por constituir-se em um ponto simples de falha, o que traz custos ainda maiores para a organização. Por ser uma solução centralizada, o excesso de requisições de clientes pode formar um gargalo sobre os servidores, o que limita bastante a escalabilidade de tais sistemas. Com a adição de mais clientes, investimentos em mais servidores, e em políticas de balanceamento de carga e de redundância se fazem necessários.

Tanto as vantagens quanto as desvantagens apresentadas são originadas a partir da centralização, própria da arquitetura cliente-servidor.

2.3.7 - Gestão descentralizada de Workflows

A adoção progressiva de sistemas de workflow pelas organizações é resultado de um melhor controle sobre seus processos obtido pelo emprego de tais sistemas, gerando melhoria no processo, e ganhos de eficiência, produtividade e flexibilidade organizacional. Apesar da existência de várias ferramentas de gestão de workflows comerciais, e da maturidade da área, demonstrada através da existência de padrões e de um comitê para concentrar os esforços para o seu desenvolvimento, alguns pontos ainda são alvo de pesquisas.

2.3.7.1 - Pontos críticos da gestão centralizada de workflows

O emprego da arquitetura cliente-servidor no projeto de um sistema de gestão de workflows traz alguns benefícios, alguns deles apresentados na seção 2.3.5. Apesar de atender aos requisitos funcionais de tais sistemas (como o gerenciamento e a coordenação do processo), o emprego da tecnologia cliente-servidor impõe algumas limitações a esses sistemas devido à centralização, característica inerente a si. Tais limitações das arquiteturas centralizadas podem afetar aos requisitos não-funcionais dos sistemas de gestão de workflows, restringindo seriamente a robustez, o desempenho, a segurança e a distribuição destes (GRUNDY *et al.*, 1998, ALONSO *et al.*, 1995).

Por centralizar a coordenação da execução e os dados dos workflows, o servidor de um sistema de gestão de workflow concentra as demandas de todo o sistema, absorvendo um peso que poderia ficar a cargo de seus clientes, cujo poder de processamento não é aproveitado. Um aumento na carga de trabalho do servidor, resultado da conexão de mais clientes, ou da execução de mais instâncias de workflows, podem causar sobrecarga no servidor, tornando-o um gargalo para todo o sistema. A dificuldade em escalar e a degradação do desempenho são, por tanto, problemas potenciais. A robustez dos sistemas de gestão de workflows também é afetada pela adoção de um esquema centralizado, por se constituir num ponto simples de falha. Empregar redundância, neste aspecto, amenizaria o problema, mas ao custo de esforço de implementação, aumentando a complexidade do sistema.

2.3.7.2 - A natureza distribuída dos workflows

Processos de negócio freqüentemente envolvem um grande número de recursos, atores e ferramentas, provavelmente distribuídos ao longo de um grande espaço geográfico, provavelmente além dos limites físicos da organização. Por sua vez, os sistemas de gestão de workflows têm o papel de automatizar a coordenação de todos esses elementos. Considerando tais observações, nota-se que as aplicações de workflow são inerentemente distribuídas e descentralizadas (YANG, 2000, ALONSO *et al.*, 1995).

O interesse pela implementação de arquiteturas descentralizadas para coordenar a execução de workflows vêm do descasamento de impedância existente entre a natureza distribuída dos processos de negócio, e a característica centralizada das soluções tecnológicas dominantes na indústria na área de workflows. A pesquisa atual

realizada sobre soluções centralizadas resolve parcialmente os problemas inerentes, e sempre aumentando ainda mais a complexidade dos sistemas existentes (YAN *et al.*, 2006). De fato, a idéia de aplicar arquiteturas descentralizadas na gestão de workflows possui vários defensores na literatura (EDER, 1999, HEINL *et al.*, 1999, MUTH, 1998).

2.3.7.3 – Os passos rumo à descentralização

Depois de apresentado os pontos críticos das soluções centralizadas, e uma breve discussão sobre a natureza distribuída dos sistemas de gestão de workflows, alguns passos necessários para a aplicação de uma solução descentralizada serão apresentados. Segundo (YAN *et al.*, 2006), tais passos são:

- Baixo acoplamento → Uma arquitetura descentralizada deve adotar uma estrutura fracamente acoplada que não implique na necessidade de um repositório de dados e de uma *engine* de workflow centrais;
- Descentralização de dados → Armazenar os dados de forma descentralizada, pelas outras máquinas do sistema;
- Descentralização de serviços → Assim como os dados, os serviços providos pelo servidor centralizado devem ser migrados para outras máquinas do sistema;
- Comunicação direta → A comunicação entre um fornecedor de serviços e um consumidor deve ser possível de forma direta, sem a necessidade de um servidor central para coordenar a comunicação;
- Suporte a SOA → Arquitetura orientada a serviços se mostra uma tendência em vários campos da computação, e o mesmo ocorre na área de gestão de workflows. Desta forma, o sistema deve suportar comunicação através de serviços.

2.3.7.4 - Gestão descentralizada de Workflows em ambientes P2P

A conectividade provida pelas estruturas de redes, como redes locais corporativas e a Internet, e poder computacional disponível em seus computadores, permitem às organizações se beneficiarem do poder coletivo de seus recursos computacionais (FAKAS, 2003). A computação *peer-to-peer* possui papel de facilitador para o compartilhamento de recursos e de serviços computacionais, permitindo às organizações obterem retorno a partir de seu poder coletivo.

Além poder agregar recursos computacionais para as organizações, sistemas *peer-to-peer* possuem características puramente descentralizadas e distribuídas que se encaixam na natureza dos workflows, e ainda oferecem as vantagens sobre as arquiteturas apresentadas no capítulo 2.

2.3.8 - Conclusão

Este capítulo apresentou alguns dos conceitos básicos existentes na área de workflows, apresentando as definições para os termos mais comuns.

Entretanto, a maior contribuição deste capítulo foi discutir alguns aspectos das arquiteturas centralizadas e identificar alguns benefícios que podem ser conseguidos pelo emprego de uma arquitetura descentralizada na coordenação de workflows. Além disto, foram apresentados autores que defendem que workflows possuem uma natureza descentralizada e distribuída e que, portanto, existe um descasamento de impedância entre as soluções centralizadas e os workflows.

2.4 - Trabalhos Correlatos

A seguir, alguns trabalhos correlatos são apresentados, e pela proximidade temática com este trabalho, seus aspectos relevantes são detalhadamente apresentados.

2.4.1 - Uma arquitetura P2P para gestão dinâmica de workflows

Em (FAKAS 2004), é proposta uma arquitetura descentralizada para a gestão de workflows, sem a existência de um elemento central de coordenação do workflow. É baseada nos conceitos de WWPD (*Web Workflow Peers Directory*) e WWP (*Web Workflow Peers*).

WWPD é um sistema de diretório que mantém uma lista dos *peers* disponíveis para participar de um workflow. Ele permite que *peers* se cadastrem e anunciem seus serviços e seus recursos para os demais *peers*. Seu funcionamento é semelhante a serviços de registro de web services UDDI (*Universal Description, Discovery and Integration*), armazenando informações relevantes dos peers registrados naquele momento. Entretanto, uma diferença fundamental os distingue dos serviços UDDI: o WWPD é um serviço de diretório que atua ativamente sobre os *peers* registrados, consultando-os continuamente sobre sua existência na rede ou sua condição atual.
WWP são os *peers* propriamente ditos, e sua principal característica é possuir toda a informação para execução das tarefas atribuídas, sendo que ainda possuem autonomia para delegar tarefas a outros *peers*.

A coordenação da execução deste sistema utiliza um mecanismo de notificações: o administrador recebe mensagens dos *peers* participantes, a fim de se manter informado sobre a execução do workflow, e envia mensagens de controle aos *peers*, para comandar a execução. Por exemplo, o administrador pode enviar uma notificação para um *peer*, informando que o tempo para execução de uma tarefa expirou. A definição do processo é descentralizada e distribuída, o que não requer servidor.

Um *WWP* possui características bem próximas a um serviço web, possuindo uma interface que descreve suas funcionalidades, e que pode ser publicada para outros *peers*. Um *WWP* pode assumir o papel de participante, quando contribui com seus serviços para um determinado processo, ou administrador de processo, quando toma a

iniciativa de iniciá-lo. Seguindo a filosofia do paradigma *peer-to-peer*, um mesmo *peer* pode assumir o papel de participante ou administrador em diferentes processos.

Um processo é totalmente descrito através de um documento XML chamado *WPD* (*Workflow Process Description*), que é anexado às mensagens trocadas entre os *WWP* para evitar o armazenamento de informações em uma entidade centralizada. Recursos que serão consumidos por atividades, ou seus resultados, não são transportados anexados às mensagens, mas referenciados no documento através de URI (*Universal Resource Identifier*). Esta abordagem traz a vantagem de possibilitar aos participantes o uso de estratégias particulares para recuperar dados e documentos, como replicação de estruturas de dados ou carga tardia.

Um *WWP* no papel de participante recebe a descrição de suas tarefas através do documento *WPD*. Cada atividade pode ser aceita ou rejeitada pelo *peer* participante, que pode ainda delegar tarefas livremente, operação que gera um sub-processo e o eleva à condição de administrador deste. Em geral, o documento *WPD* é alterado por um participante ao final da execução de sua atividade, como ao definir a localização dos dados de saída da atividade.

A administração é realizada de forma descentralizada, através de um mecanismo de notificações empregado pelo sistema. O *peer* administrador de um processo é capaz de realizar operações como a iniciação do processo, a realocação de atividades e a manutenção de informações sobre processos.

A iniciação do processo se dá com a seleção dos *peers* participantes que executarão as tarefas do processo, e o envio das atividades do processo a estes selecionados. Isto é realizado através do serviço de diretório (*WWPD*), utilizando informações relevantes sobre os *peers* registrados, como disponibilidade e qualidade de serviço (*QoS*). Na realocação de atividades, um participante notifica o administrador sobre a incapacidade de cumprir o prazo estipulado para uma atividade. O administrador realoca a atividade a um novo participante menos comprometido (consultando o serviço de diretório novamente), notificando todos os afetados pela realocação. Para gerir o processo, o administrador coleta informações relevantes sobre a execução, processando notificações enviadas pelos participantes envolvidos.

As notificações trocadas entre os peers envolvidos em um processo também são descritas como documentos XML, que contém informações como o identificador do processo, descrição, destinatários e o tipo da notificação (aceitação, rejeição,

finalização, realocação ou cancelamento de atividades, *deadline*, sobrecarga de trabalho de um participante).

2.4.2 – SwinDeW (YAN *et al.*, 2006)

SwinDeW (YAN *et al.*, 2006) é um sistema de gestão de workflows baseado no paradigma *peer-to-peer*, implementado sobre o framework JTXA (Sun, 2001), cuja proposta é um alto grau de descentralização, não empregando centralização em seu controle, e não possuindo um repositório de dados centralizado. Sua arquitetura é organizada em camadas, sendo a camada de serviços a mais importante por fornecer serviços-núcleo, envolvidos com a gerência de *peers* (que encapsula funcionalidades providas pelo framework JXTA, como suporte a grupos de *peers*) e com o suporte a funcionalidades em tempo de construção e execução do processo. Abaixo desta, está localizada a camada de dados, que consiste em repositórios distribuídos para a armazenagem de dados relacionados à execução das instâncias. A camada mais inferior é a camada de comunicação, que abstrai as funcionalidades relacionadas à troca de mensagens, como roteamento de informações. A camada mais superior é a camada de aplicação, que contém facilidades para aplicações de workflows. Uma aplicação que provê interfaces para interagir com o usuário é denominada WfPS (*Workflow Participant Software*).

Cada *peer* contém uma instância de uma aplicação WfPS e um conjunto de repositórios de dados. Um *peer* pode colaborar diretamente com os demais, fornecendo documentos, informações de controle e quaisquer artefatos. O SwinDeW agrupa logicamente os *peers* que compartilham determinados interesses, nas chamadas comunidades virtuais. Para tal agrupamento, o sistema se utiliza de um dos metadados de um *peer*: sua competência. Uma competência representa um conjunto de regras com um papel específico dentro de um processo, construídas a partir do conhecimento que o *peer* possui sobre o domínio e é identificada unicamente no sistema através de um nome, de forma que possa ser referenciado tanto na descrição do processo, quanto na coordenação dos *peers*. Uma comunidade virtual é formada, portanto, por um conjunto de *peers* que possuem uma determinada competência e é mantida dinamicamente durante a entrada e a saída de *peers*. Conforme o apresentado sobre redes sobrepostas, na seção 2.1, uma comunidade virtual pode ser vista como uma rede sobreposta à rede *peer-to-peer* subjacente.

Considerando o agrupamento em comunidades virtuais, seria natural que o serviço de descoberta de *peers* do SwinDeW fosse baseado em consultas por competências, e, de fato, isto é verdade. Cada consulta possui um atributo que indica a competência desejada, e cada *peer* consultado recupera, de seu repositório local de *peers* conhecidos, qual deles possui tal competência. Caso nenhum *peer* conhecido compartilhe de tal competência, o *peer* consultado distribui a consulta a todos esses *peers*, que prosseguem, assim, com a busca. Quando um mesmo *peer* possui mais que uma competência, ele serve como elo entre as comunidades estabelecidas, permitindo, assim, que a busca possa seguir adiante.

A definição de um processo é armazenada de forma descentralizada pelo sistema, para que cada *peer* apenas deva ter conhecimento do essencial ao seu trabalho. Para isso, são identificados os metadados de cada tarefa relevantes a sua execução (como a sua competência e suas tarefas predecessoras e sucessoras). Para cada tarefa, o serviço de descoberta de *peers* é invocado para recuperar algum *peer* que possua a competência requisitada. Cada *peer* descoberto recebe sua respectiva tarefa, baseado na sua competência, e a propaga para os demais *peers* de sua comunidade virtual, ou seja, todos os *peers* de uma comunidade virtual recebem as tarefas relacionadas à sua competência. Cada tarefa, juntamente com seus metadados, é armazenada localmente em todos os *peers* da comunidade, garantindo a sua descentralização replicada.

O *framework* deve atuar ativamente durante a entrada e saída de nós da rede para manter a descentralização da especificação de um determinado processo. A entrada de um novo *peer* é caracterizada pela sua conexão com um *peer* qualquer da rede, e é tratada por este último com a invocação do serviço de descoberta de *peers*: localiza-se algum *peer* na rede que possua a mesma competência do novo *peer*, ou seja, identifica-se a comunidade virtual da qual fará parte o novo *peer*. O novo peer passa a fazer parte da comunidade, e recebe como conseqüência, a especificação de todas as tarefas de posse da comunidade. Dentro de uma comunidade, a desconexão abrupta de um nó é detectada através da troca periódica de mensagens entre seus participantes. Outra situação possível é a alteração da competência de um nó, que pode ser visto como um caso particular da entrada de um novo nó.

Na etapa de instanciação de um processo, são criadas as instâncias de todas as tarefas definidas em tal processo, procedimento que é realizado concomitantemente com a alocação de tarefas aos devidos participantes. A instanciação de um processo não é realizada por um único *peer*, já que suas tarefas são instanciadas de forma distribuída

entre vários *peers*. Para instanciar um processo, de tal forma descentralizada, um *peer* qualquer instancia a tarefa inicial do processo. Em seguida, devem ser instanciadas as tarefas sucessoras e, para isso, o *peer* localiza, através do serviço de descoberta, um *peer* que possua competência para executar uma das tarefas sucessoras. Uma requisição para criar uma nova tarefa é enviada ao *peer* descoberto, que iniciará uma negociação, dentro de sua comunidade virtual, para definir qual membro da comunidade instanciará e executará tal tarefa. Por tanto, a instanciação de cada tarefa e a alocação são executadas em conjunto e, além disso, a alocação é baseada em negociação.

A negociação tem como principal objetivo prover balanceamento de carga entre os membros de uma determinada comunidade, selecionando aquele que possui a menor carga de trabalho no momento, otimizando o desempenho localmente. Ao considerar apenas a carga de trabalho local para distribuir tarefas, informações importantes podem ser desprezadas degradando o desempenho global do sistema. Se um *peer* faz parte de mais que uma comunidade, por possuir mais que uma competência, a sua escolha para realizar uma determinada tarefa impacta no cálculo da carga de trabalho de todas as comunidades das quais participa. Se uma destas comunidades estiver com excesso de carga de trabalho, talvez pelo número reduzido de participantes devido a sua rara competência, a seleção deste *peer* por outra comunidade degrada ainda mais o desempenho local da comunidade sobrecarregada. Observa-se que a comunidade que possui a maior carga de trabalho $\overline{w}_i$, que é definida como a média aritmética das cargas de trabalho de todos os seus membros, é a responsável pelo desempenho global do sistema. Para quantificar este impacto, o *peer* que coordena a negociação calcula w_j, que representa o maior impacto causado em alguma das comunidades das quais um *peer j* participa, devido a sua escolha. O objetivo do *peer* negociador é encontrar um *peer* de sua comunidade associado ao w_j mínimo:

$$w_j = \max\left(\frac{\overline{w}_i \times m_i + w}{m_i}\right),$$

onde $\overline{w}_i$ é carga de trabalho média da comunidade i, e é dada por $\overline{w}_i = \left(\sum_{k=1}^{m_i} w_k\right) \Big/ m_i$. A carga de trabalho de um peer, num intervalo de tempo, é somatório da carga gerada por cada tarefa neste intervalo: $w_k = \sum t_k$.

A execução de uma tarefa depende de duas condições. A primeira condição está relacionada à dependência de dados, ou seja, dados devem estar disponíveis para que a tarefa possa ser executada, e provavelmente serão fornecidos por tarefas antecessoras. A outra condição é de controle: uma tarefa apenas pode ser executada depois que todas as suas antecessoras foram concluídas. Ambas as condições devem ser satisfeitas para que uma tarefa seja iniciada. Para conduzir os dados produzidos por uma tarefa para as suas sucessoras, mensagens são trocadas diretamente entre os *peers*, através de documentos XML, sem a presença de um coordenador centralizado. Também são trocadas mensagens de controle para coordenar a dependência entre as tarefas definidas pelo processo. Tais mensagens podem sinalizar o término, cancelamento ou realocação de tarefas, ou, ainda, detecção de exceções.

3 – Dynaflow e a sua arquitetura

Este capítulo apresenta a proposta deste trabalho para o problema da coordenação da execução de workflows.

A seção 3.1 apresenta a proposta, sendo que a subseção 3.1.1 a justifica e a subseção 3.1.2 é uma visão geral da ferramenta.

As demais seções aprofundam as idéias apresentadas, detalhando a arquitetura. Elas serão apresentadas seguindo o fluxo desde a instanciação de um workflow até a sua execução. Dentro de cada passo deste fluxo, é apresentada a filosofia do sistema e uma descrição técnica que ressalta os agentes utilizados para suportar cada etapa.

Como se trata de um sistema multi-agentes, é importante que o comportamento de cada gente tenha um nível de detalhamento adequado. Os atributos e as mensagens esperadas por cada agente serão apresentados na forma de tabelas. Nos diagramas de classes apresentados ao longo deste capítulo, os agentes são identificados pelo estereótipo *agent*, e as mensagens trocadas entre agentes são identificadas com o estereótipo *entry*.

3.1 – Proposta

Propõe-se uma arquitetura multi-agentes, descentralizada e distribuída, para coordenar a execução de workflows, que utilize serviços computacionais distribuídos numa rede *peer-to-peer* na execução de tarefas.

3.1.1 – Justificativa

A escolha de um ambiente descentralizado cumpre requisitos derivados a partir de problemas potenciais observados numa arquitetura centralizada: escalabilidade, robustez, alto desempenho, flexibilidade e baixo custo. A abordagem *peer-to-peer* se encaixa nestes requisitos devido às suas características intrínsecas:

- Escalabilidade, já que o aumento do número de usuários implica no aumento do poder computacional da rede, pois cada usuário é um novo recurso para o sistema como um todo;

- Auto-organização, dispensando uma coordenação central para manutenção da rede;
- Tolerância a falhas, devido inexistência de um único ponto de falha, como nas arquiteturas centralizadas;
- Custo compartilhado dentre os *peers* participantes;
- Agregação dos recursos computacionais já existentes.

Os sistemas *peer-to-peer* se beneficiam da agregação dos recursos computacionais dispersos pela rede, já que os *peers* podem oferecer serviços para qualquer outro *peer*. Quanto mais diversificados forem os serviços, mas atrativa é a rede. Desta forma, a arquitetura deve possibilitar que um *peer* utilize os serviços dispersos pela rede para executar seus workflows, se beneficiando da agregação de recursos.

Como discutido no capítulo 2, as arquiteturas baseadas em agentes constituem um ótimo paradigma para modelagem de sistemas de natureza descentralizada e distribuída (WANG *et al.*, 1999, WOOLDRIDGE, 1997), e por isso ela será aplicada no projeto de uma arquitetura descentralizada para coordenação da execução de um workflow.

Ainda no capítulo 2, foram apresentadas algumas características importantes que os agentes devem possuir para possibilitar a emergência de um comportamento sistemático complexo, a partir da interação entre agentes mais simples. Dentre essas características, duas delas são pré-requisitos imprescindíveis para um bom projeto de uma arquitetura multi-agentes, e nortearam o projeto dos agentes deste trabalho:

- Agentes devem possuir um comportamento extremamente especializado, possuindo um comportamento minimalista;
- Agentes devem ter visão restrita sobre as informações existentes em seu ambiente, sendo sensível apenas às informações estritamente relevantes ao seu trabalho.

Portanto, os agentes da arquitetura não devem ter conhecimento de todo o workflow. Isto significa que a definição completa do processo, expresso pelo diagrama de atividades, não deve ser conhecido por um agente. Além disso, um agente deve ter comportamento minimalista. O seu papel, dentro do sistema, é colaborar para o funcionamento do todo, encapsulando comportamentos simples. Idealmente, os agentes

devem ser gerenciados por um *framework* que permita a interação entre si de forma desacoplada, como ocorre em sistemas baseados em trocas de mensagens.

3.1.2 – Visão geral

A criação de um workflow começa quando o usuário importa uma definição de processo, definido por um diagrama de atividades da UML. Em seguida, todas as tarefas do workflow devem ser mapeadas em serviços disponíveis na rede, através da interação do usuário com o sistema. Para isto, o usuário deve possuir um mecanismo que identifique os serviços de seu interesse distribuídos pela rede. Isto é realizado através da indexação dos serviços oferecidos pelo *peer*, localmente, e de um mecanismo de propagação de consultas pela rede.

Depois de mapeadas todas as tarefas, é realizado um leilão que define quais *peers* executarão quais tarefas, já que um mesmo serviço pode ser oferecido por vários *peers*. Ao término do leilão, o sistema aloca os agentes necessários a execução do sistema nos devidos *peers*.

A coordenação da execução é descentralizada e baseada em agentes. Para se definir os agentes, recorreu-se ao metamodelo do diagrama de atividades da UML. Cada elemento do metamodelo tem seu comportamento capturado por um agente específico. Desta forma, o comportamento da coordenação do workflow emerge das interações entre agentes mais simples que refletem a semântica da linguagem de definição de processos (figura 4).

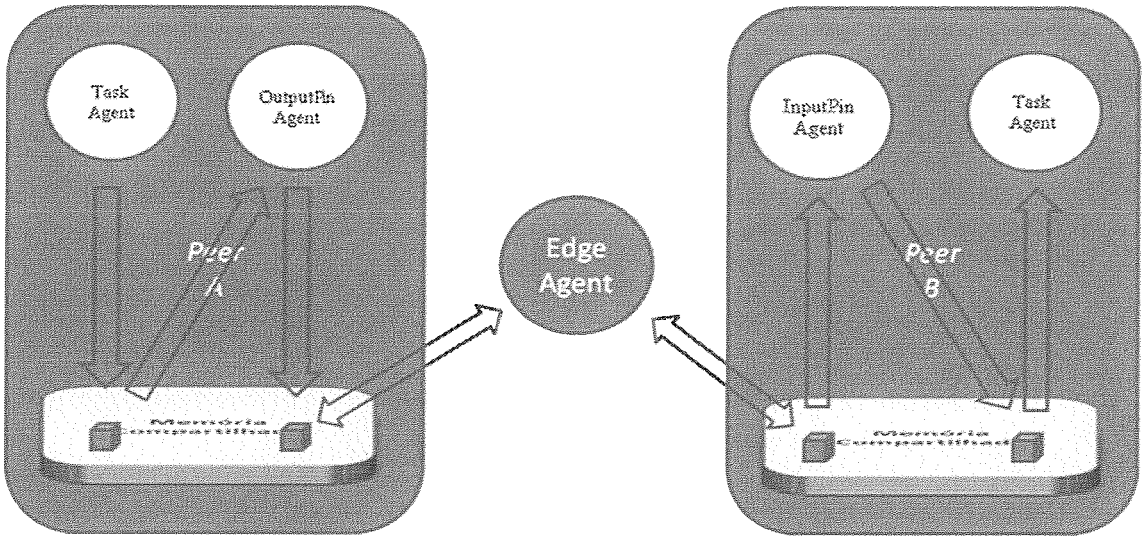


Figura 4 – Interação entre alguns agentes que coordenam a execução

3.2 - Papéis dos *peers*

Todo *peer* tem o direito de instanciar workflows e de prover serviços para os demais *peers*. Um *peer* pode assumir os papéis de publicador ou de executor no contexto do Dynaflow:

- Publicador é o papel assumido por um *peer* que irá criar um workflow, instanciando um processo de negócio.
- Executor é um *peer* que provê serviços para suportar a execução de tarefas.

Os papéis não são mutuamente exclusivos. Um *peer* pode colaborar com ambos papéis simultaneamente: enquanto participa como executor na instância de um processo, o *peer* pode instanciar um workflow, assumindo o papel de publicador para esta nova instância. O conceito de papel, portanto, não é absoluto, mas relativo à instância em questão.

Mesmo em relação a uma instância, um *peer* pode assumir papéis simultaneamente: enquanto conta com os demais *peers* da rede para suprir algumas tarefas de seu workflow, um *peer* publicador pode atuar como executor em alguma(s) tarefa(s) de seu próprio workflow. A indisponibilidade de serviços ou a redução de custo são razões para o duplo papel de um *peer*.

A filosofia apresentada está alinhada aos conceitos revisados sobre *peer-to-peer*: igualdade entre os *peers* e administração autônoma de cada *peer*.

3.3 - Macro processos: instanciação e execução

O protótipo desenvolvido pode ser visto como dois macro-processos: a instanciação de um processo e a sua execução. A instanciação de um processo realiza os passos necessários para preparar o sistema para execução de um workflow. Isto envolve a criação de todas as estruturas necessárias a etapa de execução, inclusive dos agentes que coordenam tal etapa.

A etapa de execução é a execução de uma instância de um processo, propriamente dita. Os agentes que coordenam a execução já se encontram alocados nos seus respectivos *peers* hospedeiros desde o final da etapa anterior, de instanciação.

Tais macro-processos serão apresentados e discutidos a seguir.

3.4 - A instanciação de um processo

A etapa de instanciação de um processo tem como objetivo criar todas as circunstâncias necessárias para a execução de um workflow. Isto significa:

- Definir o processo que será instanciado;
- Definir mapeamentos: para cada tarefa definida no processo, selecionar um serviço existente na rede para atendê-la;
- Alocar tarefas:
    - Negociar quais *peers* participarão da instância do processo, através do fornecimento (e da respectiva execução) dos serviços selecionados pelo usuário publicador;
    - Criar e alocar os agentes que coordenarão a execução nos *peers* selecionados.

A figura 4 apresenta um diagrama de atividades que descreve a instanciação de um processo pelo publicador. Em seguida, cada passo representado será descrito.

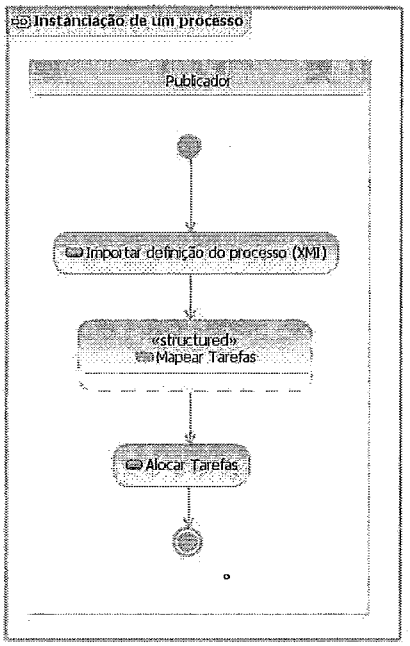


Figura 5 – Instanciação de um processo

3.4.1 - Importar definição do processo

O usuário publicador deve informar ao sistema a definição do processo a ser instanciado. Para isto, o sistema suporta definições de processos descritos por um diagrama de atividades da UML, exportados para o formato XMI (*XML Metadata Interchange*).

XMI é um padrão definido pela OMG (*Object Management Group*) para possibilitar o intercâmbio de metadados através de XML, e pode ser aplicado em metadados cujo metamodelo possa ser expresso através da especificação MOF (*Meta-Object Facility*). MOF, por sua vez, também é um padrão definido pela OMG, cujo objetivo é expressar metamodelos e surgiu da necessidade de uma arquitetura para construção do metamodelo da UML.

Algumas ferramentas possuem editor integrado, que possibilita a definição de processos graficamente. Essas soluções, em geral, trabalham com um formato próprio. Optou-se pela utilização de um padrão (o diagrama de atividades da UML 2.0) para facilitar a interoperabilidade com as diversas ferramentas de modelagem que podem exportar o diagrama para o formato XMI (*XMI Metadata Interchange*).

3.4.2 - Mapear Tarefas

A próxima etapa do macro-processo de instanciação é o mapeamento das tarefas definidas no diagrama de atividades importado no passo anterior.

Enquanto um *peer* assume o papel de publicador no contexto de uma instância de processo, os demais *peers* da rede podem colaborar na execução desta instância, ao fornecer e executar serviços, exercendo, portanto, o papel de executor. O mapeamento serve para associar as tarefas definidas pelo publicador aos serviços providos pelos demais *peers* da rede.

3.4.2.1 – Necessidade do mapeamento

Para entender melhor o porquê da necessidade de mapeamento, deve-se lembrar que um processo é definido através de um diagrama de atividades, e com isto, apenas informações sobre o fluxo de execução das suas tarefas são representadas. Cada tarefa é uma abstração de uma operação computacional, sem referência ao que deve ser feito, de fato, para concluir a tarefa. A realização de uma tarefa ocorre através dos serviços

oferecidos pelos *peers* da rede: cada tarefa é associada a um serviço disponibilizado na rede por algum *peer*. Executar uma tarefa significa, na prática, executar o respectivo serviço mapeado.

O relacionamento entre tarefas e serviços pode ser entendido como uma sobreposição entre duas camadas: uma camada é a abstração do que deve ser feito, e é representada pelas tarefas do diagrama de atividades. A camada subjacente contém as entidades que implementam o comportamento de cada tarefa do diagrama de atividades, ou seja, os serviços disponibilizados pelos *peers*.

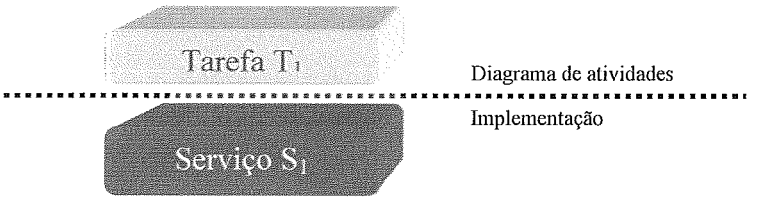


Figura 6 – Sobreposição de uma tarefa a um serviço

É atribuição do *peer* publicador a realização do mapeamento entre as tarefas definidas no processo e os serviços disponíveis na rede. Como o publicador possui a definição do processo, ele é o indivíduo mais capacitado para reconhecer quando um serviço satisfaz a uma tarefa definida no processo.

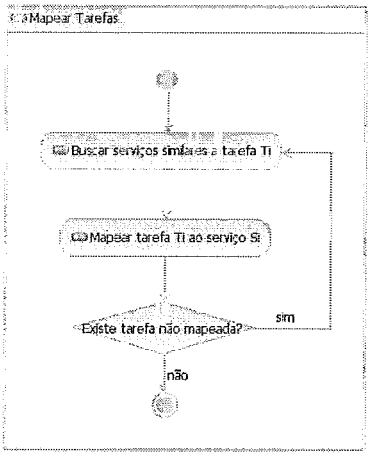


Figura 7 – Processo de mapeamento

A figura 6 apresenta os passos para realizar o mapeamento das tarefas de uma instância. Para cada tarefa do processo, o publicador busca serviços na rede e seleciona

um desses serviços para associá-lo a tarefa, através da interface do sistema. Cada passo do processo de mapeamento será discutido a seguir.

3.4.2.2 - Buscar serviços relevantes

O primeiro passo do mapeamento de uma tarefa é obter um conjunto de serviços relevantes para a execução de tal tarefa. O publicador deve possuir alguma ferramenta que o auxilie a descobrir os serviços com potencial para executar cada tarefa do workflow.

Para que o publicador possa recuperar os serviços relacionados, os executores devem disponibilizar seus serviços. Isto é feito através do emprego de mecanismos de busca e recuperação da informação: os descritores dos serviços disponibilizados por cada *peer* são indexados localmente e um mecanismo de propagação de consultas possibilita o acesso aos índices locais a cada *peer*. Quando um novo serviço a ser disponibilizado por um *peer* é registrado pelo seu usuário, os metadados presentes no descritor do novo serviço são indexados no índice local do *peer*, possibilitando a recuperação dos metadados caso uma consulta seja recebida pelo *peer*. Adiante, veremos que um descritor de serviço se trata, na verdade, de um arquivo WSDL (*Web Service Description Language*).

Quando um publicador deseja recuperar serviços disponíveis na rede, ele deve elaborar uma consulta que descreva a tarefa textualmente. Como o padrão XMI suporta a documentação de artefatos UML, o publicador poderá optar por usar a documentação da tarefa, embutida no arquivo XMI que descreve o processo, caso ela tenha sido documentada durante a construção do diagrama de atividades.

3.4.2.2.1 – Visão da descoberta de serviços

A consulta criada pelo usuário publicador é, então, propagada pela rede. *Peers* que possuam serviços, cuja descrição tenha relevância para a consulta, respondem a consulta, enviando uma resposta ao *peer* publicador. A resposta conterá metadados dos serviços relevantes à consulta, oferecidos pelo *peer* consultado.

A figura 7 apresenta um esquema conceitual para o mecanismo de descoberta de serviços, apoiado pelo framework COPPEER:

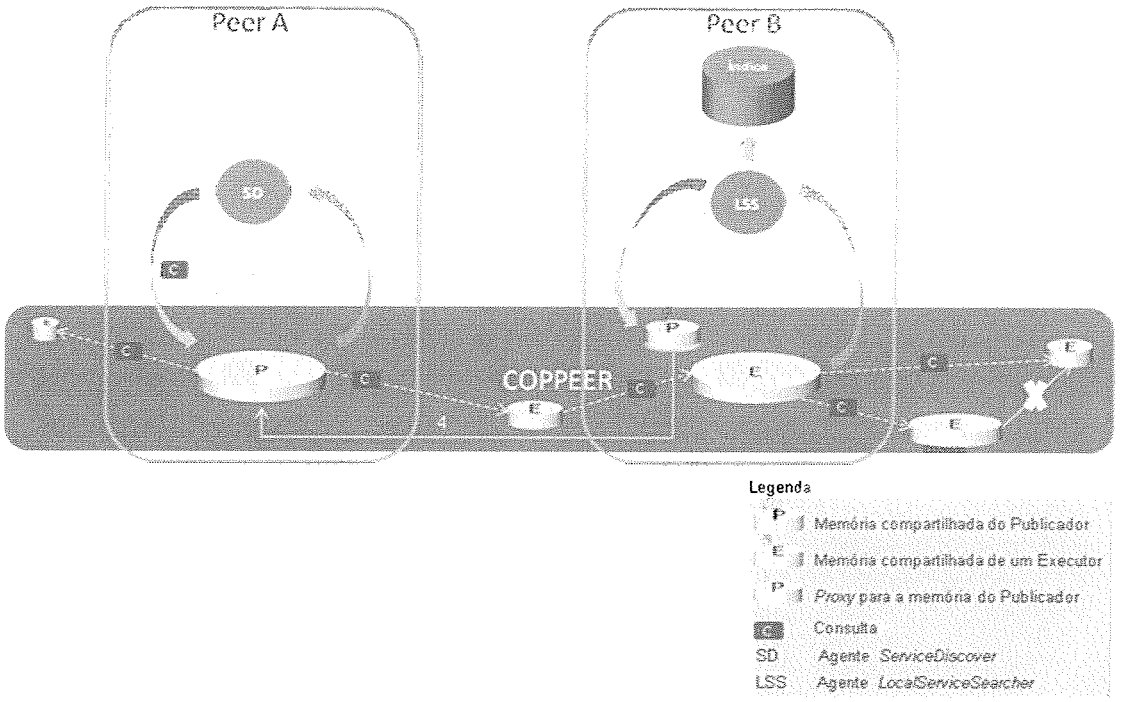


Figura 8 – Esquema do mecanismo de descoberta de serviços

O esquema apresenta as mensagens ordenadas trocadas entre os agentes e suas respectivas memórias compartilhadas. Os eventos são apresentados de forma numerada, para evidenciar as suas ordens de ocorrência.

A busca de por um serviço começa com a criação do agente *ServiceDiscover* (SD), que grava uma entrada (1), que representa a consulta realizada, na memória compartilhada de seu *peer* (que possui o papel de publicador, neste caso). O sistema sobrecai sobre o mecanismo de propagação de registros do *framework* COPPEER para propagar a consulta até todos os vizinhos do *peer* publicador. A cada chegada a um *peer*, o estado interno da entrada é avaliado pelo framework, para decidir se continua a propagação para os demais vizinhos do *peer* atual.

Quando ocorre a chegada de uma consulta à memória compartilhada de um *peer*, o agente *LocalServiceSearcher* (LSS) é notificado pelo *framework* (2). Como tratamento deste evento, o agente consulta o índice (3), recupera eventuais metadados relevantes a consulta, e cria uma entrada contendo tais resultados. O agente então grava remotamente a entrada na memória compartilhada do *peer* publicador, através de um objeto procurador (*proxy*) remoto, gerenciado pelo *framework* (4).

O agente *ServiceDiscover* (SD) é notificado pelo framework a cada resultado recebido em sua memória compartilhada (5). A agente apenas acumula os metadados

dos serviços encontrados, e o sistema os disponibiliza para o usuário através de sua interface.

3.4.2.2.2 - Pacote de descoberta de serviços

O diagrama de classes do pacote de descoberta de serviços é apresentado na figura 8, e a descrição dos seus agentes e das mensagens por eles trocadas é realizada a seguir.

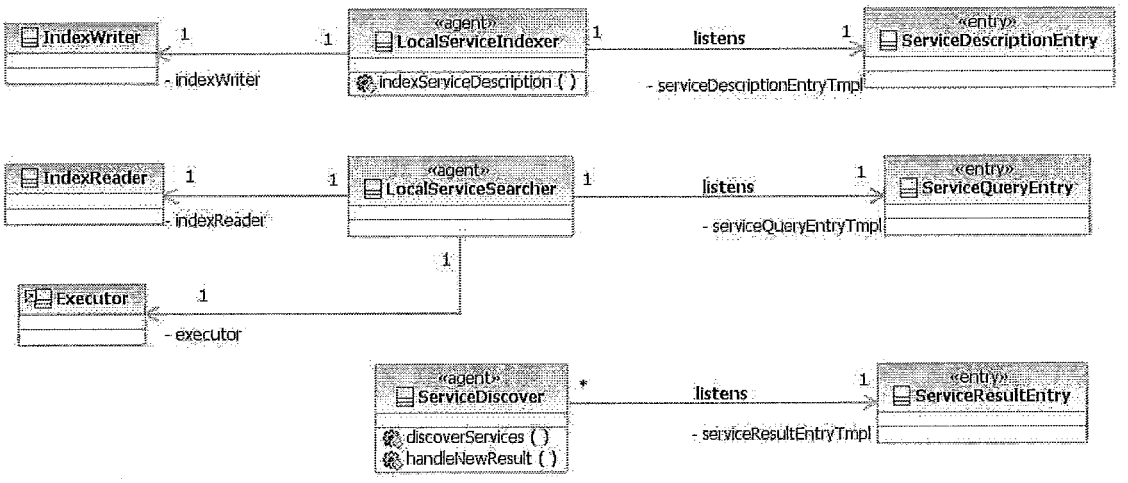


Figura 9 – Diagrama de classes do pacote de descoberta de serviços

As entidades IndexWriter e IndexReader, apresentados no diagrama de classes do pacote, abstraem as operações de leitura e escrita no índice local do *peer*.

Agente de indexação de serviços (*LocalServiceIndexer*)

Agente responsável pela manutenção do índice de descritores dos serviços providos pelo seu *peer*. É criado durante a inicialização da aplicação e permanece ativo até o encerramento da mesma. O agente abstrai o formato do descritor de serviços através de uma estrutura de objetos.

LocalServiceIndexer	
Responsabilidade	Indexar descritores dos serviços oferecidos pelo seu peer.
Mensagens de Entrada	*ServiceDescriptionEntry*

	RemoveServiceDescriptionEntry
Mensagens de Saída	-
Agentes Correlatos	-
***Peer* Hospedeiro**	Qualquer *peer* que ofereça serviços.

Eventos		
Evento	**Mensagem**	**Tratamento**
Descritor de novo serviço	*ServiceDescriptionEntry*	Indexa o descritor do serviço

Agente de descoberta de serviços (*ServiceDiscover*)

Ao ser criada uma consulta pelo *peer* publicador, uma instância deste agente é imediatamente criada para tratar apenas aquela consulta. O agente cuida da criação da consulta propagável, bem como da coleta de seus resultados (através da monitoração da memória compartilhada do seu *peer*).

ServiceDiscover	
Responsabilidade	Tratar uma consulta específica, desde a sua criação até a coleta de seus resultados.
Mensagens de Entrada	*ServiceResultEntry*
Mensagens de Saída	*ServiceQueryEntry*
Agentes Correlatos	*LocalServiceSearcher*
***Peer* Hospedeiro**	Criador da consulta (Publicador)

Eventos		
Evento	**Mensagem**	**Tratamento**
Criação do agente	-	• Envia uma mensagem **ServiceQueryEntry**, que contém a consulta, para sua memória compartilhada;

		• A mensagem será propagada automaticamente pelo COPPEER, para os demais *peers*.
Nova resposta de um *peer* à consulta	*ServiceResultEntry*	• Armazena descritores de serviços, recebidos como resultado da consulta em execução.

Agente de busca por serviços (*LocalServiceSearcher*)

Agente responsável por responder as consultas por serviços, realizadas por *peers* publicadores, consultando o índice de descritores de serviços do seu *peer*. O agente é criado durante a inicialização da aplicação e permanece ativo até o encerramento da mesma.

LocalServiceSearcher		
Responsabilidade	Busca descritores de serviços locais, relevantes a alguma consulta recebida pelo *peer*.	
Mensagens de Entrada	*ServiceQueryEntry*	
Mensagens de Saída	*ServiceResultEntry*	
Agentes Correlatos	*ServiceDiscover*	
***Peer* Hospedeiro**	Qualquer *peer* que ofereça serviços.	
Eventos		
Evento	**Mensagem**	**Tratamento**
Consulta por serviço	*ServiceQueryEntry*	• Acessa o índice, buscando serviços cuja descrição seja relevante para a consulta; • Cria uma mensagem **ServiceResultEntry**, contendo os descritores recuperados, diretamente na memória compartilhada do *peer* que gerou a consulta.

Anatomia das mensagens deste pacote

ServiceQueryEntry	
Objetivo	Informar sobre uma nova consulta.
Atributos	A consulta e o *peer* criador.
Produtores	*ServiceDiscover*
Consumidores	*LocalServiceSearcher*

ServiceResultEntry	
Objetivo	Informar sobre um novo resultado relevante para uma consulta.
Atributos	Descritor dos serviços relevantes à consulta.
Produtores	*LocalServiceSearcher*
Consumidores	*ServiceDiscover*

ServiceDescriptionEntry	
Objetivo	Informar sobre um novo serviço fornecido.
Atributos	Descritor do serviço
Produtores	Objeto Executor
Consumidores	*LocalServiceIndexer*

3.4.2.3 - Mapear tarefa a um serviço

Uma vez recuperado um conjunto de serviços relevantes para uma tarefa, o publicador deverá selecionar o serviço mais apropriado para cumprir o objetivo da tarefa em questão, através da interface do sistema. Para auxiliar o publicador, são recuperados ainda os seguintes metadados dos serviços: nome, documentação, entradas e saídas (tipos de dados e a documentação) do serviço.

Entretanto, esse não é o único mapeamento realizado pelo publicador. O diagrama de atividades permite dois tipos de ligações entre os nós de ação (que são os elementos do diagrama que representam as tarefas de um processo): fluxos de controle e fluxos de objetos. A existência de fluxos de objeto chegando às entradas de uma tarefa significa que a tarefa aguarda parâmetros para a sua execução. O recíproco se aplica às

saídas de uma tarefa. O que se pode inferir desta observação é que o serviço deve possuir o mesmo número de entradas e saídas da tarefa. A figura 9 exibe um exemplo de tarefas que retornam ou recebem objetos.

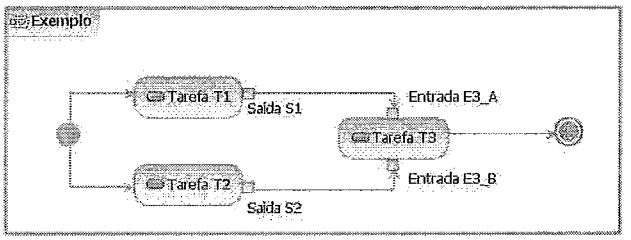


Figura 10 – Exemplos de passagem de parâmetros para execução de tarefas.

Como se pode observar na figura 9, os nós de ação T1 e T2 não recebem objetos como parâmetro, mas retornam um objeto como resultado de sua execução. Já o nó T3 recebe dois parâmetros como entrada, mas não retorna objeto como saída. Ao selecionar um serviço para suprir a tarefa T3, o usuário deverá verificar se o serviço é compatível com as entradas e saídas da tarefa.

Além da quantidade de entradas e saídas do serviço ser compatível com a quantidade de entradas e saídas da tarefa, o tipo de dado de cada entrada e saída deve ser considerado. No entanto, não é a tarefa que impõe o tipo de dado da entrada ou da saída do seu serviço: é o serviço antecessor.

Suponha que o usuário publicador inicie o mapeamento pela ordem de execução das tarefas. Aplicando esta estratégia ao diagrama da figura 9, o usuário começaria o mapeamento pela tarefa T1 ou T2. Ao associar T1 a um serviço que retorna um número inteiro, por exemplo, o usuário estará definindo que a entrada E3_4 da tarefa T3 receberá um número inteiro. Logo, ao buscar serviços que satisfaçam T3, o usuário deverá considerar esta a restrição quanto ao tipo de dado.

Ao selecionar um serviço compatível a uma tarefa, em relação ao número de entradas e saídas, e aos tipos de dados, o usuário fará o mapeamento indicando qual entrada ou saída da tarefa é associada a qual entrada ou saída do serviço selecionado. Depois de mapeadas todas as entradas e todas as saídas da tarefa, o mapeamento para aquela tarefa estará completo. O publicador segue, então, o processo apresentado na figura 3, repetindo o procedimento até que todas as tarefas do processo estejam mapeadas.

3.4.3 - Alocação

A última etapa da instanciação de um processo é a alocação. A execução do workflow é coordenada com o auxílio de agentes, e a etapa de alocação tem como objetivo mover todos estes agentes para os devidos *peers* executores. Os agentes sempre são alocados em *peers* que executarão alguma tarefa no workflow; *peers* que não estão envolvidos com a instância a ser executada não participam da alocação.

O problema da alocação se resume a decidir qual *peer*, dentre aqueles que fornecem um serviço, será o responsável por executar aquele serviço. A relação dos *peers* que fornecem um serviço é construída durante a etapa de mapeamento, quando o sistema armazena os *peers* que respondem às consultas do publicador.

3.4.3.1 - Leilão

Neste trabalho, a negociação para eleger o *peer* que executará uma tarefa é baseada em leilão. Para cada tarefa, o *peer* publicador inicia um único leilão, notificando apenas os *peers* que fornecem o serviço associado à tarefa. Como alternativa, o sistema poderia notificar todos os *peers* existentes, já que um *peer* que oferece o serviço poderia entrar na rede após o mapeamento. Entretanto, esta estratégia inundaria a rede com a propagação de mensagens.

Ao ser notificado sobre o leilão de uma tarefa, um *peer* opta por oferecer lances no leilão. Um *peer* pode oferecer quantos lances desejar, sendo que um novo lance substitui automaticamente o seu último lance.

3.4.3.3.1 – O pacote leilão

O suporte ao leilão é feito através das classes apresentadas na figura 10.

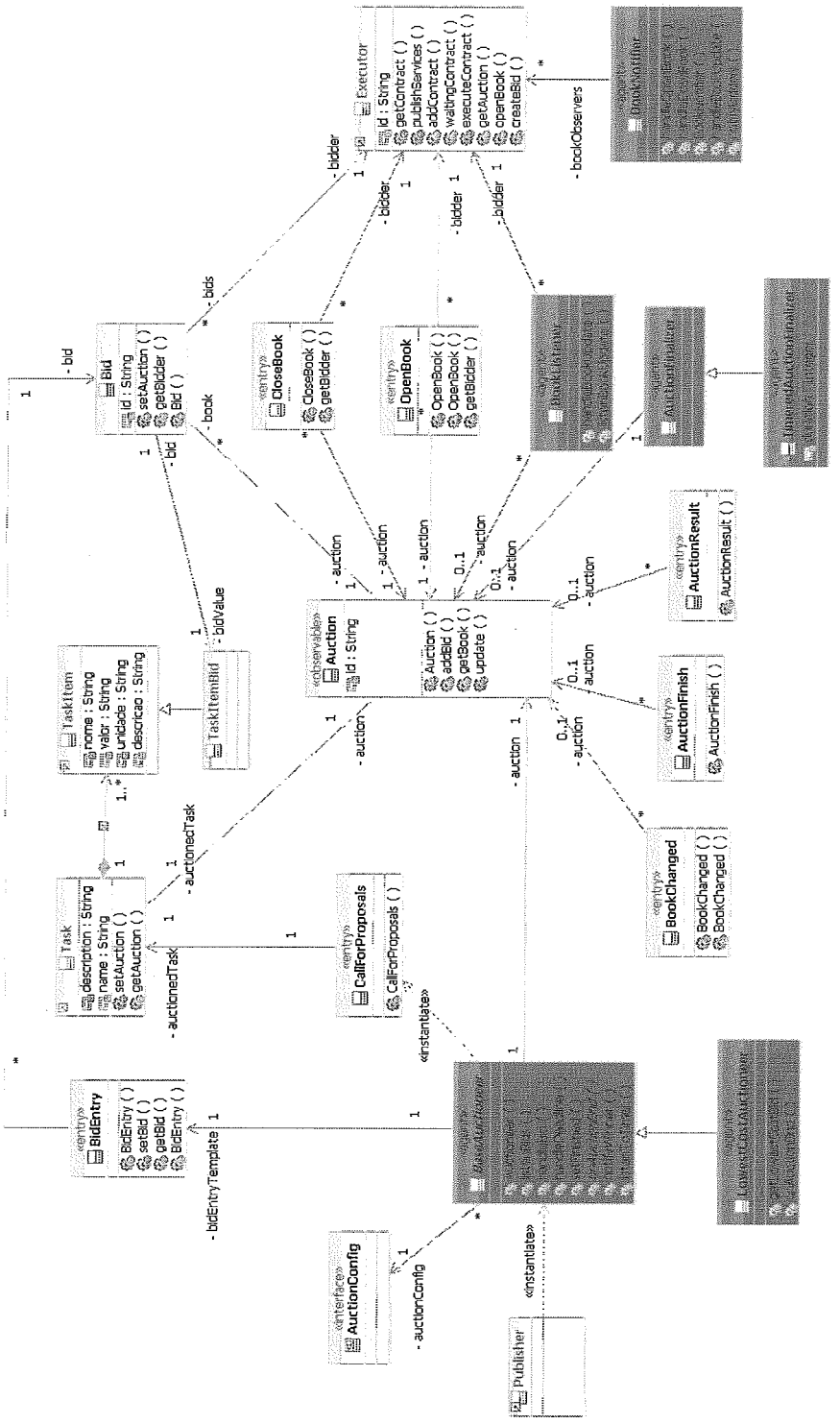


Figura 11 – Diagrama de classes do pacote de leilão (agentes na cor vermelha)

As principais entidades presentes no diagrama de classes são:

- Task → Representa a tarefa em leilão.
- Auction → Representa o leilão da tarefa.

- TaskItemBid → Configurado pelo usuário, representa o item que será leiloado, como o custo para executar a tarefa, por exemplo.
- Bid → Representa um lance oferecido por algum participante.
- Executor → Representa um executor. Não pertence ao pacote de leilão, apenas é apresentado para representar as dependências de outras classes.

O principal agente do pacote é o *BaseAuctioneer*, que exerce papel de leiloeiro, responsável por coordenar o leilão. O leiloeiro é o agente que aplica as diretivas que regem um leilão, tais como:

- Forma → Um leilão pode ser fechado ou aberto. Leilão fechado é aquele cujos lances não são públicos. No leilão aberto, os lances são públicos, o que obriga o leiloeiro a instanciar um agente especializado em notificar os *peers* interessados naquele leilão (agente *BookNotifier* do diagrama da figura 7).
- Estratégia de término → Em geral, leilões são temporizados. Nestes casos, o leiloeiro cria um agente que o notifica ao término do leilão (*TimeredAuctionFinalizer*).
- Estratégia de definição do vencedor → Para eleger o lance vencedor, o leiloeiro conta com uma função que atua sobre o conjunto de lances. A estratégia é definida especializando-se o leiloeiro (*BaseAuctioneer*). O leiloeiro *LowestCostAuctioneer* é uma especialização de *BaseAuctioneer* que elege o lance de menor custo (figura 7).

Agentes leiloeiros podem ser adicionados ao sistema sem a necessidade de alteração do código existente. Como o sistema é executado dentro do ambiente COPPEER, ele pode utilizar o suporte à injeção de dependências, oferecida pela arquitetura de *microkernel* do *framework*, para plugar agentes.

3.4.3.3.2 – Anatomia dos agentes de leilão

BaseAuctioneer	
Responsabilidade	Leiloeiro: coordena um único leilão, aplicando as estratégias para finalização e eleição do vencedor.
Mensagens de Entrada	*BidEntry, AuctionDeadline*

Mensagens de Saída	*CallForProposals, BookChanged, EndOfAuction, WonAuction*
Agentes Correlatos	*TimeredAuctionFinalizer, BookNotifier*
***Peer* Hospedeiro**	Realizador do leilão (Publicador)

Eventos		
Evento	**Mensagem**	**Tratamento**
Criação	-	Notificar os *peers* que oferecem serviços mapeados na tarefa leiloada, através do envio de mensagens **CallForProposals**.
Novo lance	*BidEntry*	Atualizar o livro de ofertas do leilão, e notificar **BookNotifier** com uma mensagem **BookChanged**.
Término do leilão	*AuctionDeadline*	Enviar as mensagens **EndOfAuction** e **WonAuction**, para anunciar o término e o vencedor.

LowestCostAuctioneer	
Responsabilidade	Implementa a estratégia de eleger o lance de menor custo.
Mensagens de Entrada	-
Mensagens de Saída	-
Agentes Correlatos	*BaseAuctioneer*
***Peer* Hospedeiro**	Realizador do leilão (Publicador)

TimeredAuctionFinalizer	
Responsabilidade	Notificar o **BaseAuctioneer** sobre o término de um leilão. É criado pelo próprio BaseAuctioneer.
Mensagens de Entrada	-
Mensagens de Saída	*AuctionDeadline*
Agentes Correlatos	*BaseAuctioneer*
***Peer* Hospedeiro**	Realizador do leilão (Publicador)

BookNotifier	
Responsabilidade	Notificar os *peers* interessados em um leilão, a cada alteração no livro de ofertas. É criado pelo BaseAuctioneer.
Mensagens de Entrada	*OpenBook, CloseBook, BookChanged*
Mensagens de Saída	*BookEntry*
Agentes Correlatos	*BaseAuctioneer, BookListener*
***Peer* Hospedeiro**	Realizador do leilão (Publicador)

Eventos		
Evento	**Mensagem**	**Tratamento**
Abrir livro	*OpenBook*	Adiciona o *peer* como observador do livro de ofertas
Novo lance	*BookChanged*	Notifica todos os observadores do livro de ofertas, através de uma mensagem *BookEntry*.
Fechar Livro	*CloseBook*	Remove o *peer* como observador do livro de ofertas

BookListener	
Responsabilidade	Tratar avisos sobre novo lance no leilão observado.
Mensagens de Entrada	*BookEntry*
Mensagens de Saída	*OpenBook*
Agentes Correlatos	*BookNotifier*
***Peer* Hospedeiro**	*Peer* interessado em um leilão (Executor).

Eventos		
Evento	**Mensagem**	**Tratamento**
Novo lance	*BookEntry*	Atualiza sua cópia do livro de ofertas.
Criação	-	Envia uma mensagem **OpenBook** para o publicador

Término do leilão

Finalizado o leilão, o *peer* publicador já possui uma relação dos *peers* que executarão os serviços necessários. Os agentes que irão tratar o workflow são instanciados e enviados aos seus respectivos *peers*. O processo de execução irá detalhar como são criados os agentes e para onde são movidos.

3.5 - A execução de um workflow

Finalizada a etapa de instanciação de um processo, o workflow está preparado para a sua execução, com seus agentes devidamente alocados nos *peers* envolvidos. A execução de um workflow será detalhada no decorrer desta seção, apresentando a anatomia dos os agentes, como são criados, as suas mensagens trocadas e outros detalhes envolvidos.

3.5.1- Requisitos

No capítulo 2, observamos alguns problemas potenciais das arquiteturas centralizadas para a coordenação de workflows, em geral associados a requisitos não-funcionais. Este trabalho estuda uma arquitetura que proporcione, de forma intrínseca, soluções para os problemas potenciais da arquitetura centralizada. A partir disto, os seguintes requisitos devem ser observados para o projeto da arquitetura:

- Escalabilidade e Desempenho → Um dos problemas potenciais do emprego de uma arquitetura centralizada é a sua limitação em relação ao número de clientes, já que o desempenho do sistema costuma cair com o aumento do número de clientes.
- Robustez → A existência de um único ponto de falha, o servidor central, pode acarretar a queda de todo o sistema;
- Custo → Escalar uma arquitetura centralizada implica em investimentos e manutenção, o que reduz o *uptime* do sistema, causando prejuízos aos seus usuários.

3.5.2 – Construindo uma solução descentralizada

O primeiro passo da abordagem proposta neste trabalho é entender o que expressa o metamodelo do diagrama de atividades da UML, nossa linguagem de modelagem para workflows.

3.5.2.1 - O metamodelo do diagrama de atividades da UML

A UML (*Unified Modeling Language*) constitui numa abordagem que unificou diversas propostas de análise orientada a objetos, para permitir ao engenheiro de software expressar um modelo de análise usando uma notação de modelagem, regulada por um conjunto de regras sintáticas, semânticas e pragmáticas (PRESSMAN, 2002). Os diagramas da UML possibilitam a observação de um sistema, ou parte dele, através do seu comportamento, de sua estrutura, de seu ambiente, da sua implementação, ou até mesmo do ponto de vista do usuário do sistema.

Para que fosse possível definir a notação e as regras para cada diagrama da UML, foi necessária a elaboração de um modelo com capacidade de representar os elementos de cada diagrama, bem como os relacionamentos entre eles e eventuais restrições. Tal modelo é, portanto, um metamodelo, capaz de expressar a estrutura aceitada por um diagrama da UML. Um metamodelo na UML é composto por metaclasses, que representam os elementos do diagrama, e seus relacionamentos.

Um processo importado para Dynaflow deve ser representado internamente de forma adequada, para que seja possível realizar operações, atribuir dados e estabelecer comportamentos de forma adequada. Em outras palavras, é necessário possuir um modelo que represente processos. Uma vez que o sistema aceita processos descritos por diagramas de atividades, o sistema deve ser capaz de representar tais diagramas. Para isto, conta-se com o suporte do metamodelo do diagrama de atividades da UML, apresentado abaixo:

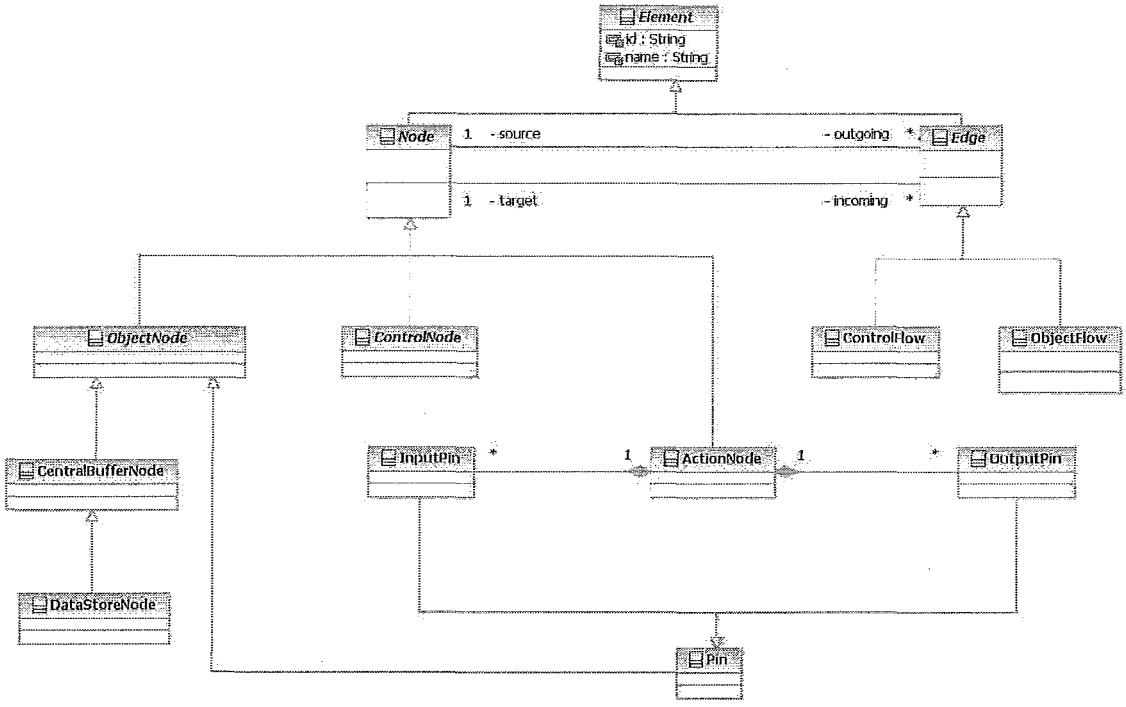


Figura 12 – Metamodelo para o diagrama de atividades

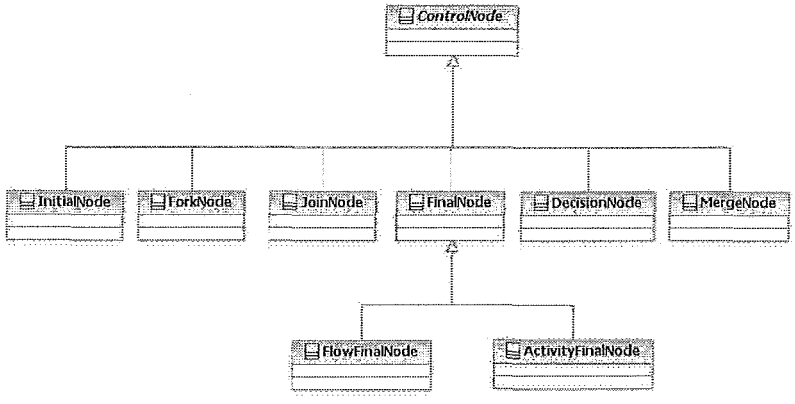


Figura 13 – Visão detalhada dos nós de controle do metamodelo

O metamodelo apresentado na figura 12 (e na figura 13, que detalha os nós de controle suportados) é uma visão simplificada do metamodelo original. Todo processo importado para o sistema é representado através deste metamodelo, que captura os aspectos importantes para o propósito do trabalho.

3.5.2.2 – A semântica do metamodelo

Cada elemento presente no metamodelo representa um elemento de um diagrama de atividades, que possui características e comportamento próprios:

- Edge → Representam um fluxo de dados ou de controle que liga dois nós, identificando o nó fonte e o nó alvo.
- Node → Representa qualquer nó no diagrama. Podem ser: de controle (quando representa uma estrutura de controle), de objeto (quando representam um objeto de dados) ou de ação (quando expressam um comportamento).
  InputPin e OutputPin → Nós de objeto, que representam a entrada ou a saída de objetos de um nó de ação.
- InitialNode → Identifica as tarefas que são iniciadas primeiramente, quando o workflow é iniciado. Todas as tarefas alvo dos fluxos que partem de um nó inicial são iniciadas simultaneamente. Um nó inicial pode ter vários fluxos de saída.
- ForkNode → Divide um fluxo de execução em múltiplos fluxos concorrentes.
- JoinNode → Funciona como uma barreira de sincronização, aguardando pela execução todos os fluxos que chegam até este nó. Após a execução de todos os fluxos, prossegue com o fluxo de execução do workflow.
- DecisionNode → Implementa o comportamento do desvio condicional, computando uma determinada condição global do workflow para decidir qual fluxo que parte deste nó dará prosseguimento à execução do workflow.
- MergeNode → A intercalação apenas identifica o final de um desvio condicional.
- FlowFinalNode → Quando atingido por um fluxo de execução, aborta esta fluxo, mas não interfere em nenhum outro fluxo de execução no workflow.
- ActivityFinalNode → Quando atingido por um fluxo de execução, qualquer outro fluxo de execução do workflow deve ser abortado.

Os elementos *InitialNode*, *ForkNode*, *JoinNode*, *DecisionNode*, *MergeNode*, *FlowFinalNode* e *ActivityFinalNode* são nós do tipo controle, porque possuem papéis relacionados ao controle do fluxo de execução do sistema.

3.5.2.3 – Derivando os agentes da execução

Conforme as premissas para a construção de agentes, o projeto de agentes deve priorizar agentes minimalistas e especializados, dentre outras características desejáveis

apresentadas no capítulo 2. Por outro lado, um processo é definido por elementos, cada um com uma semântica específica.

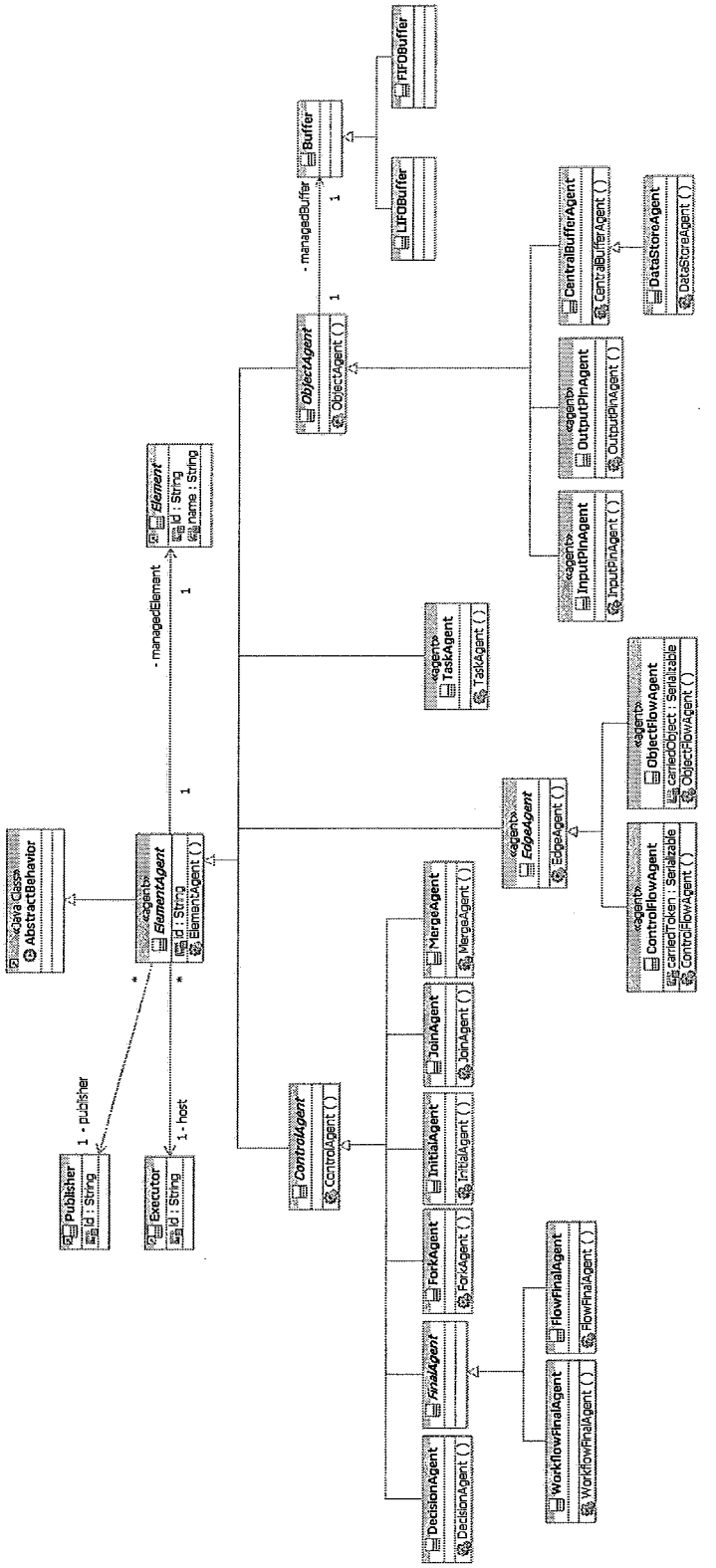


Figura 14 – Hierarquia de agentes que coordenam a execução descentralizada

A arquitetura multi-agentes foi obtida capturando-se o comportamento de cada elemento existente no metamodelo do diagrama de atividades, na forma de um agente autônomo. Ou seja, para cada elemento do metamodelo apresentado na figura 11, existe um agente que captura única e exclusivamente o comportamento daquele elemento, dando origem a hierarquia apresentada na figura 14.

3.5.3 - Pacote de Execução

Este pacote contém os artefatos responsáveis por coordenar, de forma distribuída, a execução de uma instância de um workflow, seguindo a arquitetura proposta.

3.5.3.1 - Agentes Coordenadores de Execução

A coordenação da execução de um workflow é orientada por agentes distribuídos pelos diversos *peers* que colaboram em uma determinada instância.

Conforme os conceitos sobre emergência apresentados, um sistema que apresenta um comportamento emergente possui agentes altamente especializados, que possuem conhecimento apenas do que é necessário ao seu trabalho. Seguindo esta filosofia, os agentes coordenadores de execução possuem uma visão limitada do workflow, atuando no escopo necessário apenas ao seu trabalho. Tais agentes não possuem conhecimento da especificação de um workflow como um todo.

O metamodelo do diagrama de atividades da UML forneceu uma importante contribuição para o projeto dos agentes utilizados na coordenação, já que é do metamodelo que emerge o comportamento capturado pelos agentes descritos a seguir. Para cada elemento do diagrama de atividades presente na especificação de um workflow, é criada uma instância de um agente especializado no comportamento daquele elemento, sendo que esta instância é alimentada com os dados relativos àquele elemento.

Todos os agentes são criados pelo *peer* publicador. Após ser criado, cada agente se move para um *peer* hospedeiro, dependendo da natureza do agente. A seguir, será apresentada a anatomia dos agentes que capturam a semântica do diagrama de atividades.

3.5.3.1.1 - InitialAgent

Agente que representa o comportamento de um nó inicial (*initial node*) de um diagrama de atividades. Monitora a memória compartilhada pela autorização para iniciar um workflow. Dada a autorização, o agente envia um *token* de controle para cada fluxo de saída, disparando a execução de todo o sistema.

InitialAgent	
Responsabilidade	Iniciar a execução das primeiras tarefas do workflow.
Mensagens de Entrada	*ObjectRequest*
Mensagens de Saída	*ObjectResponse*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* publicador

3.5.3.1.2 - TaskAgent

Agente responsável por parte do comportamento de um nó de ação (*action node*) em um diagrama de atividades, e também representa uma tarefa propriamente dita de um workflow. O comportamento é parcial porque este agente não tem responsabilidade de executar o serviço mapeado na tarefa que representa. A execução propriamente dita é delegada a outro agente, que encapsula a infra-estrutura para execução do serviço.

Adiante, veremos que os serviços são aceitos na forma de *web services*. O agente *TaskAgent* delega a execução do serviço para o agente *WebServiceAgent*, que implementa a infra-estrutura necessária para se comunicar com *web services*. Como agentes são fracamente acoplados, *WebServiceAgent* pode ser substituído, sem prejuízos a manutenção do agente *TaskAgent*.

O *TaskAgent* recupera os objetos enviados e recebidos como parâmetros (de entrada e de saída, respectivamente) do serviço. Os parâmetros de entrada são obtidos a partir do *InputPinAgent*. Os parâmetros de saída são fornecidos ao *OutputPinAgent* para armazenamento.

Um nó de ação pode possuir vários fluxos de entrada e de saída. Quando possui vários fluxos em sua entrada, existe uma junção implícita: a tarefa apenas será executada quando todos os fluxos de entrada forem disparados.

TaskAgent	
Responsabilidade	Tratar os parâmetros de entrada e saída de uma tarefa, e delegar a execução de serviços ao agente competente.
Mensagens de Entrada	*ObjectResponse* *ExecutionAuthorization* *EndOfService*
Mensagens de Saída	*StartServiceRequest* *ObjectResponse* *EndOfTask* *NotifyTaskStatus*
Agentes Correlatos	*InputPinAgent, WebServiceAgent, OutputPinAgent*
***Peer* Hospedeiro**	*Peer* vencedor do leilão para a tarefa

3.5.3.1.3 - InputPinAgent

São agentes que capturam o comportamento dos pontos de entrada (*input pin*) de um diagrama de atividades. Os pontos de entrada recebem objetos enviados por outros nós, por meio de fluxos de objetos, e os armazenam até que sejam requisitados para consumo pelo nó de ação associado.

InputPinAgent	
Responsabilidade	Armazenar os objetos que serão consumidos pela tarefa associada.
Mensagens de Entrada	*ObjectResponse, ObjectRequest*
Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	*EdgeAgent, TaskAgent*
***Peer* Hospedeiro**	Mesmo *peer* que hospeda a tarefa associada.

3.5.3.1.4 - OutputPinAgent

São agentes que capturam o comportamento dos pontos de saída (*output pin*) de um diagrama de atividades. Os nós de ação não armazenam seus objetos produzidos, o que fica a cargo dos pontos de saída associados ao nó de ação. Quando solicitado, um ponto de saída entrega um objeto ao fluxo de objetos solicitante.

O agente gerencia um *buffer* dos objetos criados pelo nó de ação correspondente, empregando alguma estratégia de ordenação de dados, conforme especificado no diagrama de atividades (FIFO é a estratégia padrão, caso não seja especificada a política de ordenação).

OutputPinAgent	
Responsabilidade	Armazenar os objetos produzidos pela tarefa associada.
Mensagens de Entrada	*ObjectResponse, ObjectRequest*
Mensagens de Saída	*GuardNotification, ObjectResponse*
Agentes Correlatos	*TaskAgent, EdgeAgent*
***Peer* Hospedeiro**	Mesmo *peer* que hospeda a tarefa associada.

3.5.3.1.5 - EdgeAgent

Agente que captura o comportamento das arestas direcionadas que conectam pares de nós em um diagrama de atividades, realizando o transporte de objetos ou de *tokens* de controle entre esses nós.

Não permanece hospedado em uma única agência, ao contrário dos demais agentes do pacote de execução. Por representar fluxos, este agente se move constantemente entre os *peers* hospedeiros das tarefas antecessora e sucessora, realizando o transporte de objetos ou de *tokens* de controle.

EdgeAgent	
Responsabilidade	Conduzir objetos ou *tokens* de controle de um nó para o sucessor deste.

Mensagens de Entrada	*ObjectResponse, ObjectRequest*
Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	Todos os agentes da coordenação, exceto o *GuardAgent*.
***Peer* Hospedeiro**	Alterna entre o *peer* que hospeda seu elemento fonte e o *peer* que hospeda seu elemento alvo.

3.5.3.1.6 – ForkAgent

Agente que captura o comportamento de uma separação (*fork node*). O agente gera múltiplas cópias do *token* de controle recebido, e as envia para os fluxos de saída, simultaneamente. Esta simples estratégia permite que o workflow continue sendo executado através de múltiplos fluxos de execução concorrentes.

ForkAgent	
Responsabilidade	Divide um fluxo de execução em múltiplos fluxos concorrentes.
Mensagens de Entrada	*ObjectResponse, ObjectRequest*
Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* que hospeda a tarefa antecessora imediata.

3.5.3.1.7 - JoinAgent

Agente que captura o comportamento de uma junção (*join node*), sincronizando os seus múltiplos fluxos de entrada. Para isso, aguarda até que todos os fluxos de entrada forneçam seus respectivos *tokens* de controle, quando então, envia o *token* de controle para o seu único fluxo de saída.

JoinAgent	
Responsabilidade	Sincronizar fluxos de execução.
Mensagens de Entrada	*ObjectResponse, ObjectRequest*

Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* que hospeda a tarefa sucessora imediata.

3.5.3.1.8 - DecisionAgent

Agente que captura o comportamento condicional de uma decisão (*decision node*), realizando desvios em um fluxo de execução. Uma decisão possui um fluxo de entrada e vários fluxos de saída. Quando uma decisão é encontrada, apenas um fluxo de saída pode ser selecionado para realizar o desvio. Para possibilitar a seleção, cada fluxo de saída possui uma sentinela, que é uma expressão booleana criada pelo usuário durante a modelagem do diagrama de atividades. Quando uma decisão é encontrada, a sentinela de cada fluxo é avaliada para determinar qual fluxo de saída deverá ser atravessado. Como apenas um fluxo pode ser escolhido, as sentinelas devem ser mutuamente exclusivas.

Este agente trabalha em conjunto com o *GuardAgent*, que representa o comportamento dos sentinelas. Devido a isso, a explicação sobre o processo de decisão continua a seguir.

DecisionAgent	
Responsabilidade	Realizar desvios condicionais.
Mensagens de Entrada	*ObjectResponse, ObjectRequest, GuardResponse*
Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	*GuardAgent*
***Peer* Hospedeiro**	*Peer* que hospeda a tarefa antecessora imediata.

3.5.3.1.9 – GuardAgent

A expressão booleana especificada por uma sentinela possui variáveis que assumem valores definidos anteriormente em algum ponto do workflow. No Dynaflow,

para simplificar, uma sentinela possui apenas uma variável, que é restrita a nomes de pinos de saída (*output pin*). Desta forma, uma sentinela é uma expressão booleana que considera o valor de algum pino de saída que foi calculado em passos anteriores do workflow.

Se o valor booleano das sentinelas fosse computado apenas quando um nó de decisão fosse encontrado, o agente que representa o nó de decisão (*DecisionAgent*) deveria saber em qual *peer* se encontra o pino de saída considerado por cada uma das sentinelas, e, de alguma forma, recuperar seus valores. Tal abordagem traria complexidade ao agente, que deveria conhecer informações sobre outros *peers* do workflow, e não apenas os *peers* das tarefas sucessoras ou antecessoras imediatas. O problema seria ainda maior se o valor do pino de saída tivesse sido calculado há muito tempo e as estruturas de dados referentes aquele workflow já tivessem sido desalocadas pelo *peer* que hospedou o pino de saída.

Para evitar que o agente *DecisionAgent* conheça informações além de sua responsabilidade ou para se precaver da impossibilidade de obter os valores dos pinos de saída, a avaliação de cada sentinela fica a cargo do agente *GuardAgent*.

Uma instância de *GuardAgent* captura o comportamento de uma sentinela (*guard*) específica, apoiando o trabalho de *DecisionAgent*. Para cada fluxo de saída de uma decisão, existe uma instância de *GuardAgent*. *GuardAgents* são sempre instanciados por *DecisionAgent*.

Cada *GuardAgent* é hospedado no mesmo *peer* onde se encontra o pino de saída que deve monitorar. Quando o pino de saída monitorado recebe um valor, o *GuardAgent* avalia imediatamente a expressão. Se for falsa, o agente não toma nenhuma providência. Se a expressão for verdadeira, o agente notifica o seu *DecisionAgent*, que então seleciona o fluxo de saída correspondente àquela sentinela.

GuardAgent	
Responsabilidade	Atuar como sentinela (*guard*), auxiliando o *DecisionAgent*.
Mensagens de Entrada	*GuardNotification*
Mensagens de Saída	*GuardResponse*
Agentes Correlatos	*DecisionAgent*
***Peer* Hospedeiro**	*Peer* que hospeda o pino de saída vigiado.

3.5.3.1.10 - MergeAgent

Este agente possui o comportamento de um nó de intercalação (*merge node*), apenas permitindo que um *token* de controle, vindo de algum de seus fluxos de entrada, siga adiante para o seu fluxo de saída. O papel de uma intercalação meramente indica que os diversos fluxos de execução resultantes de uma decisão anterior se encontram novamente.

MergeAgent	
Responsabilidade	Atuar como intercalação, identificando o fim de um desvio.
Mensagens de Entrada	*ObjectResponse, ObjectRequest*
Mensagens de Saída	*ObjectResponse, ObjectRequest*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* que hospeda a tarefa sucessora imediata.

3.5.3.1.11 – FlowFinalAgent

Captura o comportamento de um final de fluxo, finalizando uma linha de execução que chega por um de seus fluxos de entrada, sem interferir na execução das demais linhas de execução do workflow. É interessante quando se deseja filtrar informações, destruindo aquelas que não são relevantes.

FlowFinalAgent	
Responsabilidade	Eliminar fluxos de execução, sem finalizar o workflow.
Mensagens de Entrada	*ObjectResponse*
Mensagens de Saída	*ObjectRequest*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* que hospeda a tarefa antecessora imediata.

3.5.3.1.12 – ActivityFinalAgent

Agente que captura o comportamento de um final de atividade, finalizando a execução de todos os fluxos de execução do workflow quando atingido.

ActivityFinalAgent	
Responsabilidade	Finalizar a execução de todo o workflow.
Mensagens de Entrada	*ObjectResponse*
Mensagens de Saída	*EndOfWorkflow, ObjectRequest*
Agentes Correlatos	*EdgeAgent*
***Peer* Hospedeiro**	*Peer* publicador

3.5.3.2 – Anatomia das mensagens dos agentes de execução

A maioria dos agentes que suportam a execução se comunica através de requisição e resposta, e, para isto, são utilizadas apenas duas mensagens, *ObjectRequest* e *ObjectResponse*. Este modelo de comunicação é adotado porque, seguindo a semântica do diagrama de atividades, o objeto produzido por uma tarefa não é propagado imediatamente para a próxima tarefa: o objeto deve ser armazenado até que a tarefa sucessora o requeira. Uma tarefa apenas solicita um objeto quando está ociosa. Cada mensagem *ObjectRequest* ou *ObjectResponse* possui um identificador que possibilita a notificação apenas daqueles agentes interessados naquela instância da mensagem.

A seguir, são detalhadas essas e outras mensagens utilizadas pelos agentes do pacote de execução. Algumas mensagens de *TaskAgent*, referentes a execução de serviços, serão detalhadas ao falarmos sobre a execução de serviços e o seu agente (*WebServiceAgent*).

ObjectRequest	
Objetivo	Representa a disponibilidade de um agente para consumir dados ou *token* de controle.
Atributos	Objeto ou *token* de controle; ID referente ao agente que será

	notificado.
Produtores	Qualquer agente que represente o comportamento de um elemento do metamodelo, exceto InitialAgent.
Consumidores	Qualquer agente que represente o comportamento de um elemento do metamodelo, exceto FlowFinalAgent e ActivityFinalAgent.

ObjectResponse	
Objetivo	Representa a oferta de dados (ou *token* de controle) por um agente.
Atributos	Objeto ou *token* de controle; ID referente ao agente que será notificado.
Produtores	Qualquer agente que represente o comportamento de um elemento do metamodelo, exceto FlowFinalAgent e ActivityFinalAgent.
Consumidores	Qualquer agente que represente o comportamento de um elemento do metamodelo, exceto InitialAgent.

GuardNotification	
Objetivo	Informar sobre a chegada de um objeto a um pino de saída, que pode estar sendo monitorado por alguma sentinela.
Atributos	ID do pino de saída (*output pin*) e ID do fluxo monitorado.
Produtores	*OutputPinAgent*
Consumidores	*GuardAgent*

GuardResponse	
Objetivo	Notificar o *DecisionAgent* sobre uma sentinela avaliada como verdadeira.
Atributos	Guard; ID do nó de decisão;
Produtores	*GuardAgent*
Consumidores	*DecisionAgent*

EndOfWorkflow

Objetivo	Informar sobre o encerramento da execução de um workflow.
Atributos	ID do workflow; Objeto ou *token* de controle.
Produtores	*ActivityFinalAgent*
Consumidores	Publicador

ExecutionAuthorization	
Objetivo	Informar que usuário autorizou a executação de uma tarefa.
Atributos	Tarefa a ser executada.
Produtores	Executor
Consumidores	*TaskAgent*

EndOfTask	
Objetivo	Informar que uma tarefa finalizou.
Atributos	A tarefa e os valores gerados pela execução do serviço.
Produtores	*TaskAgent*
Consumidores	Executor

NotifyTaskStatus	
Objetivo	Notificar o publicador sobre o novo estado de uma tarefa.
Atributos	A tarefa e o seu novo estado.
Produtores	*TaskAgent*
Consumidores	Publicador

3.6 – A execução de serviços no *peer* executor

Depois de apresentada uma arquitetura descentralizada para execução de workflows, resta discutir como os serviços são executados por um *peer* executor. Como já discutido, o agente *TaskAgent*, não executa serviços diretamente, delegando tal responsabilidade para um agente especializado.

Antes de prosseguir, serão apresentados alguns requisitos básicos que devem ser atendidos pela arquitetura do *peer* executor.

3.6.1 – Requisitos para a arquitetura de um *peer* executor

3.6.1.1 - Escalabilidade

Um *peer* pode colaborar executando serviços que atendam a tarefas de diversos workflows simultaneamente. Ao participar de vários workflows, o *peer* se compromete em executar vários serviços simultaneamente. Se os serviços providos pelo *peer* forem executados localmente, poderá haver a sobreposição da execução de tarefas, sob risco de perda de desempenho.

O *peer* deve, portanto, contar com uma arquitetura que o permita ser escalável no que tange aos serviços fornecidos. O *peer* pode optar por distribuir seus serviços por outras máquinas, com quem apenas trocará mensagens para delegar a execução dos serviços.

3.6.1.2 – Baixo acoplamento

A arquitetura do *peer* executor deve permitir que serviços providos sejam adicionados e removidos de forma desacoplada. O objetivo é flexibilizar a aplicação, para que o usuário utilize serviços legados de forma não intrusiva, sem exigir compilação de código e sem impor conhecimento do sistema pelo serviço.

Um serviço deve possuir, ainda, um descritor que apresente informações relevantes, como suas entradas e saídas, sua finalidade e sua localização. As informações descritivas de um serviço são importantes para que o *peer* responda às consultas de outros *peers* (publicadores).

Um mecanismo baseado em mensagens pode ser uma boa forma de obter o desacoplamento entre o serviço e o sistema, principalmente se o mecanismo adotar padrões abertos para suas mensagens.

3.6.1.3 – Suporte assíncrono

Um serviço pode consumir bastante tempo de processamento. É importante que o sistema seja notificado ao fim da execução do serviço, ao invés de aguardar o final da execução. Para isto, o sistema deve ser capaz de executar serviços de forma assíncrona.

3.6.2 – Arquitetura do *peer* executor

Para permitir o suporte à distribuição dos serviços, baixo acoplamento e invocações assíncronas, a arquitetura do *peer* executor baseia-se em *web services* para a comunicação com os seus serviços. A aplicação de *web services* também considera:

- A ampla aceitação como solução para interoperabilidade entre sistemas;
- Que os protocolos e os formatos das mensagens são abertos e baseados em XML;
- O funcionamento sobre o protocolo HTTP (*HyperText Transfer Protocol*), o que o torna hábil a funcionar em máquinas por trás de *proxies* ou protegidas por *firewalls*;
- A convergência de aplicações corporativas para o ambiente da Internet.

Serviços *web* possibilitam a comunicação entre um serviço e o seu cliente de forma desacoplada, através de troca de mensagens. Entretanto, a troca de mensagens entre um serviço e seus clientes respeita certos padrões, o que justifica a adoção de um agente especializado neste tipo de comunicação, o *WebServiceAgent*.

Serviços *web* são descritos por arquivos XML que obedecem ao padrão WSDL (*Web Service Description Language*). Como já discutido, o sistema mantém um índice de descritores, para permitir que um *peer* realize consultas sobre os metadados dos serviços existentes na rede. Na prática, é uma porção relevante dos arquivos WSDL que é indexada.

O agente *TaskAgent* delega a execução de um serviço para outro agente, através da criação de uma mensagem *StartServiceRequest*, sem ter conhecimento do agente ou do ele fará. Caso fosse necessária uma nova forma de execução de serviços, bastaria a criação de um novo agente especializado, em substituição ao *WebServiceAgent*. A figura 14 apresenta um esquema conceitual para execução de serviços por um *peer*.

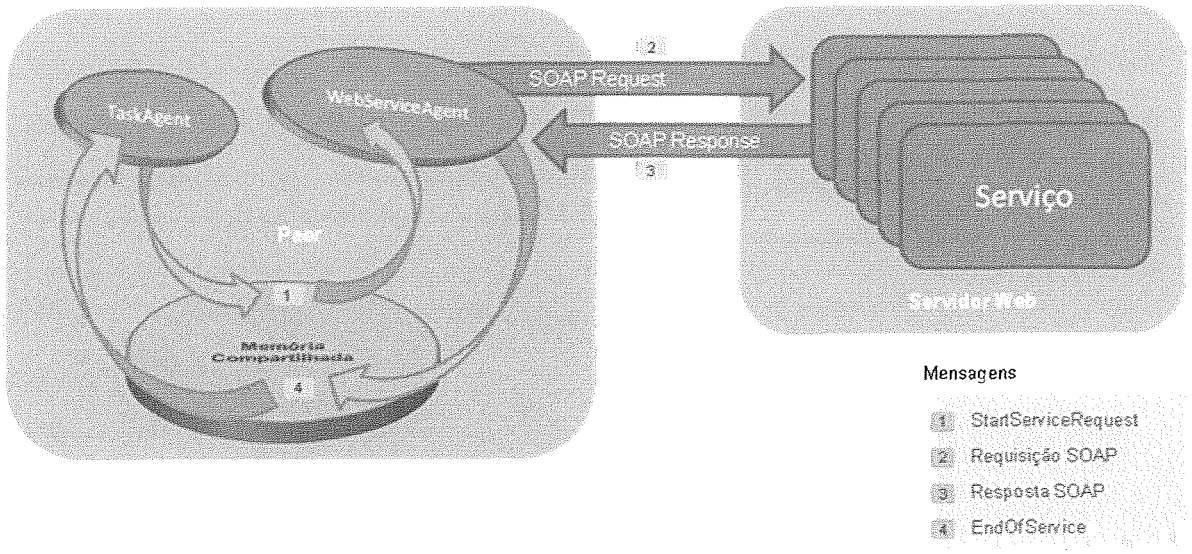


Figura 15 – Tratamento de requisição e resposta pelo *WebServiceAgent*

3.6.3 – O agente *WebServiceAgent*

A troca de mensagens entre um serviço *web* e os seus clientes baseia-se no protocolo SOAP (*Simple Object Access Protocol*). SOAP é um protocolo de comunicação cuja finalidade é facilitar a troca de informações entre sistemas (interoperabilidade). SOAP é baseado em XML, o que significa que uma mensagem SOAP é um documento XML. A principal vantagem de utilizar XML como formato de uma mensagem é obter a independência da plataforma utilizada no cliente e no servidor. Para o transporte de uma mensagem SOAP, o protocolo de aplicação HTTP (*HyperText Transfer Protocol*) é utilizado como protocolo de transporte, o que facilita o acesso a serviços hospedados em servidores por trás de *proxies* ou *firewalls*. A figura 15 representa a troca de mensagens SOAP entre o cliente e o servidor do serviço.

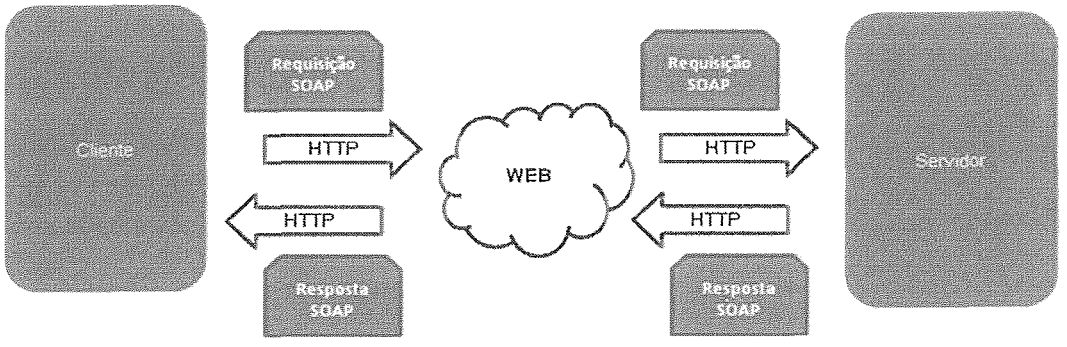


Figura 16 – Troca de mensagens SOAP

A responsabilidade do *WebServiceAgent* é abstrair a comunicação com serviços *web*, se comportando como cliente de serviços *web*. A comunicação é realizada pelo agente tratando requisições e respostas SOAP. Para isto, o agente utiliza uma API que permite manipular as mensagens SOAP em baixo nível, construindo a mensagem incrementalmente. Interação com mensagens em baixo nível é necessária, principalmente, para tratar tipos de dados complexos, como veremos adiante.

O agente é criado durante a inicialização da aplicação e permanece ativo até o encerramento da mesma.

WebServiceAgent	
Responsabilidade	Abstrair a comunicação com serviços *web*.
Mensagens de Entrada	*StartServiceRequest*
Mensagens de Saída	*EndOfService* *FailedExecutionTask*
Agentes Correlatos	*TaskAgent*
***Peer* Hospedeiro**	Qualquer *peer* que forneça algum serviço *web*.

Eventos		
Evento	**Mensagem**	**Tratamento**
Executar serviço *web*	*StartServiceRequest*	Cria uma requisição SOAP e chama o serviço *web*. Ao final do serviço, cria uma mensagem *EndOfService*. Uma mensagem *FailedExecutionTask* é criada em caso de erro.

3.6.4 – Anatomia das mensagens trocadas entre os agentes *TaskAgent* e *WebServiceAgent*

Ao se apresentar as mensagens do pacote de execução, algumas mensagens de *TaskAgent* foram omitidas propositalmente, para que fossem apresentadas dentro do contexto da execução de serviços e do seu agente, *WebServiceAgent*. Tais mensagens são apresentadas a seguir:

StartServiceRequest	
Objetivo	Informar que um serviço deve ser executado.
Atributos	Parâmetros relevantes a chamada do serviço.
Produtores	*TaskAgent*
Consumidores	WebServiceAgent

EndOfService	
Objetivo	Informar que um serviço foi executado.
Atributos	Informação retornada por um serviço.
Produtores	*WebServiceAgent*
Consumidores	*TaskAgent*

FailedExecutionTask	
Objetivo	Informar sobre a falha na execução de um serviço.
Atributos	Informações sobre a falha.
Produtores	*WebServiceAgent*
Consumidores	*TaskAgent*

3.6.5 – Interoperabilidade

Outra razão para a escolha de serviços *web*, como forma para executar serviços, é a interoperabilidade obtida com a definição de tipos de dados através do padrão WSDL.

A primeira abordagem para suporte a serviços, neste trabalho, era através de uma classe abstrata. Serviços deveriam estender tal classe, implementando um método que era chamado quando o serviço fosse inicializado. Para que um objeto consumido ou produzido por um serviço fosse utilizado com outros serviços, a sua classe deveria ser conhecida por estes outros serviços. Para ser transportado, bastaria a serialização do objeto, e sua condução até o destino através do agente *EdgeAgent*.

Esta abordagem impactou na interoperabilidade dos serviços, já que uma mesma definição de uma classe deveria ser conhecida por provedores de serviços que tivessem a intenção de utilizá-la. A definição deveria ser a mesma, correspondendo à mesma estrutura em memória. Por estarem distribuídos ao longo de uma rede P2P, que possui uma natureza heterogênea, não seria aceitável exigir que os *peers* tivessem exatamente a mesma definição de uma classe. Os serviços não poderiam ser orquestrados de forma interoperável.

Uma abordagem mais aceitável, e que possibilita a interoperabilidade, é aquela em que os serviços apenas conheçam a definição do tipo de dado, e não a classe a ser utilizada. Serviços *web* oferecem essa facilidade, por definir um padrão baseado em XML para definição de serviços e seus objetos, o WSDL.

Uma vez solucionado o problema da definição de um mesmo tipo de dado ao longo da rede, surge outro problema: como construir o cliente de um serviço (o *WebServiceAgent*, neste caso)? Em geral, clientes de serviços *web* devem possuir objetos que reflitam cada tipo de dado definidos no descritor do serviço (WSDL). Ao se invocar um serviço, um objeto é transformado em um documento XML que segue a definição do descritor do serviço (WSDL), segundo regras de transformações oriundas de metadados, existentes no cliente, que ditam tal transformação. Ou seja, em geral, os clientes são acoplados ao serviço provido, conhecendo os tipos de dados e a interface do serviço.

Entretanto, o agente *WebServiceAgent* deve ser um cliente genérico de serviços *web*. O método a ser chamado no serviço, e os seus parâmetros, não podem ser definidos de forma programática. Para que o agente trate o serviço de forma transparente, ele edita mensagens SOAP, construindo mensagens para requisições e interceptando respostas para extrair seu conteúdo.

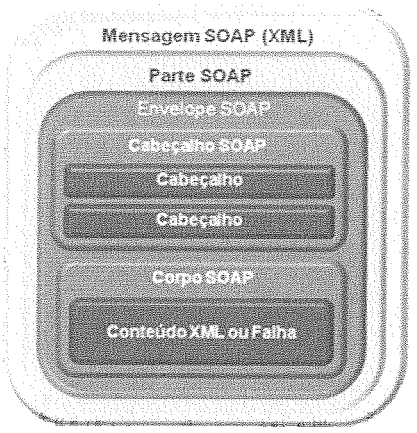


Figura 17 – Estrutura de uma mensagem SOAP (requisição ou resposta)

Para que o agente *WebServiceAgent* obtenha um objeto retornado por um serviço *web* e o envie ao agente *TaskAgent*, ele extrai o documento XML contido no elemento *corpo SOAP* da resposta SOAP, apresentada na figura 17. Desta forma, ele está recuperando o objeto retornado pelo serviço, já na forma de um documento XML. Uma mensagem é enviada ao *TaskAgent*, contendo o documento XML. *TaskAgent* cumpre com a sua responsabilidade, enviando o documento XML a um pino de saída ou para um fluxo de objeto.

O envio de objetos para serviços funciona de forma análoga: uma vez recebido um objeto, representado como um documento XML previamente criado por algum serviço, o *TaskAgent* envia uma mensagem para *WebServiceAgent* contendo tal objeto. *WebServiceAgent* constrói uma mensagem de requisição SOAP, adicionando o documento XML, que representa o objeto, como um atributo do elemento *corpo SOAP* da mensagem. A figura 17 representa a estrutura de uma mensagem SOAP tanto para requisição quanto para resposta.

3.7 - Pacotes do sistema

As dependências entre os pacotes apresentados ao longo deste capítulo, contendo agentes, entidades ou mensagens são apresentadas abaixo. Também constam no diagrama os pacotes de bibliotecas utilizadas para a implementação da ferramenta.

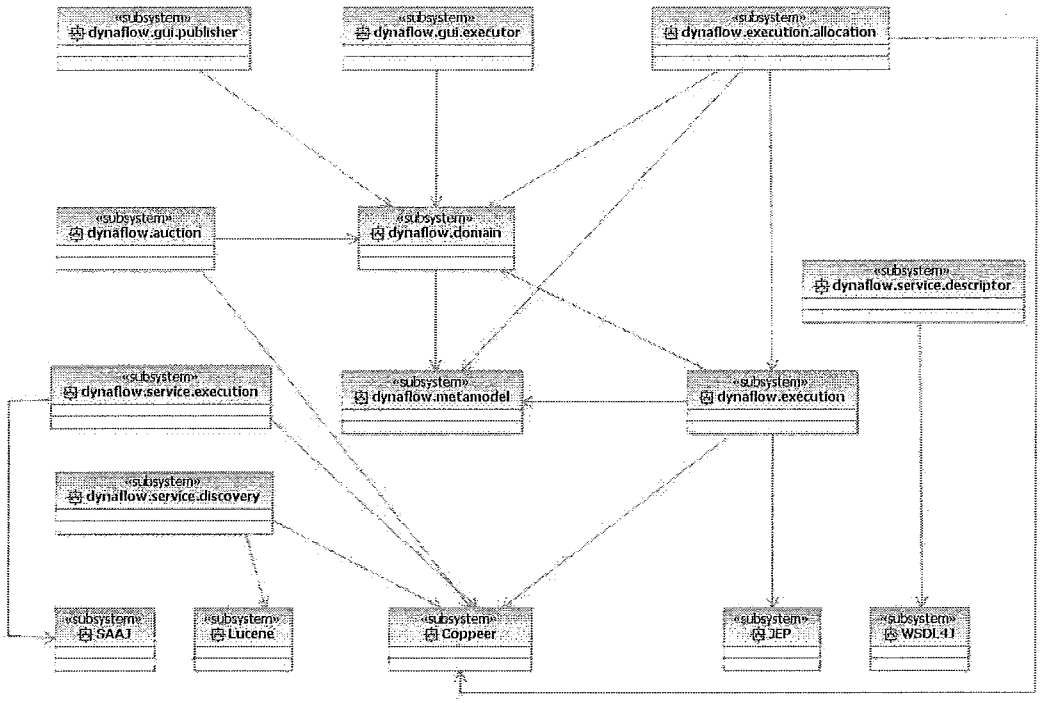


Figura 18 – Visão dos pacotes do sistema

As bibliotecas utilizadas pelo sistema para suportar as suas funcionalidades são:

- COPPEER → *Framework* de serviços P2P que oferece suporte a agentes, já detalhado anteriormente.
- Lucene → *Framework* para que oferece os serviços de busca e recuperação da informação, aplicados na indexação dos descritores dos serviços.
- SAAJ → Biblioteca de manipulação de mensagens SOAP em baixo nível, e invocação de serviços *web*.
- WSDL4J → Biblioteca para *parser* de arquivos descritores de serviços, que obedecem ao padrão WSDL.
- JEP → Biblioteca de suporte a operações matemáticas, utilizada na avaliação das sentinelas presentes em fluxos de saída dos elementos de decisão.

3.8 - Considerações

Uma abordagem ingênua, alternativa àquela apresentada neste capítulo, poderia utilizar um único agente, hospedado no *peer* publicador, para coordenar a execução.

O agente coordenador autorizaria cada *peer* a executar sua tarefa quando as dependências da tarefa estivessem satisfeitas. As autorizações seriam feitas através da troca de mensagens entre o *peer* publicador e o *peer* executor.

Como o agente coordenador seria hospedado no *peer* publicador, ele conheceria a definição de todo o processo instanciado, inclusive informações de execução, como o endereço de todos os *peers* envolvidos. Esta abordagem tem um caráter centralizador apenas em relação às instâncias que o *peer* publicou, já que o *peer* não centraliza a coordenação de instâncias que não são de sua autoria.

Outra abordagem é também utilizar um único agente coordenador, mas com mobilidade: o agente se moveria entre os *peers*, dirigindo o fluxo de execução. O *peer* publicador não teria participação ativa, recebendo apenas informações sobre o andamento do processo.

Em ambas as abordagens, o agente coordenador conheceria todo o workflow, e não apenas uma partição deste. Além disto, o comportamento do agente trataria circunstâncias mais complexas, como a realização de desvios e paralelismo de fluxos.

No caso do agente móvel, o tratamento destas circunstâncias exigiria ainda mais sofisticação em seu comportamento, o que não é coerente com a filosofia minimalista dos agentes em uma arquitetura multi-agentes. O tratamento de exceções também seria complexo: o que fazer se um *peer* se tornar *offline*? Se o agente coordenador estiver em um *peer* no momento de sua desconexão, como recuperar o seu estado?

As abordagens triviais apresentadas possuem caráter centralizador e esbarram em ambas as premissas do projeto de agentes: o agente proposto possuía a visão do workflow como um todo e comportamento complexo, contendo toda a semântica do workflow (expresso pelo diagrama de atividades): desvios, paralelismo, sincronização de fluxos, finalização do workflow, entre outros.

4 – Validação

Este capítulo tem como objetivo validar o protótipo implementado. Serão apresentadas a interface do sistema e as métricas colidas durante a execução.

As funcionalidades do papel de publicador e de executor são integradas numa mesma instância do programa, mas em interfaces distintas. As funcionalidades providas pela interface do publicador são:

- Importar uma definição de processo;
- Buscar serviços;
- Associar serviços a tarefas;
- Configurar e iniciar o leilão de um workflow;
- Visualizar o estado da execução de cada tarefa de um workflow instanciado.

As funcionalidades oferecidas pela interface do executor são:

- Registrar endereços de serviços *web*;
- Visualizar livro de ofertas;
- Oferecer lances em leilões;
- Executar uma tarefa pronta;
- Visualizar o estado de cada tarefa de sua responsabilidade.

4.1 – Configurações Iniciais

4.1.1 – Configuração do Publicador

Para que um *peer* exerça o papel de publicador, a única configuração necessária são os tipos de leilões a serem disponibilizados. Os leilões se diferem pelas suas estratégias e quem as define é o agente leiloeiro. Desta forma, um *peer* publicador deve configurar uma lista de agentes leiloeiros, através do arquivo de configuração do sistema (*dynaflow-beans.xml*).

```
<bean name="lowestCostAuction" class="dynaflow.auction.AuctionConfig">
        <property name="auctioneer">
                <value>dynaflow.auction.auctioneers.LowestCostAuctioneer</value>
        </property>
        <property name="parameters">
                <map>
                        <entry key="duration" value="60"/>
                </map>
        </property>
</bean>
```

Acima, está definido um leiloeiro que determina o vencedor do leilão baseado no menor custo para realização de uma tarefa. O seu único parâmetro é a duração do leilão.

4.1.2 – Configuração do Executor

Quando um *peer* deseja exercer o papel de executor, ele deve configurar os serviços que serão oferecidos. Para isto, ele deve fornecer ao sistema o(s) endereço(s) do(s) serviço(s) *web* que serão utilizados.

O endereço dos serviços podem ser configurados no arquivo de inicialização do sistema (*dynaflow-beans.xml*), ou através da interface do executor, como mostrado na figura 19.

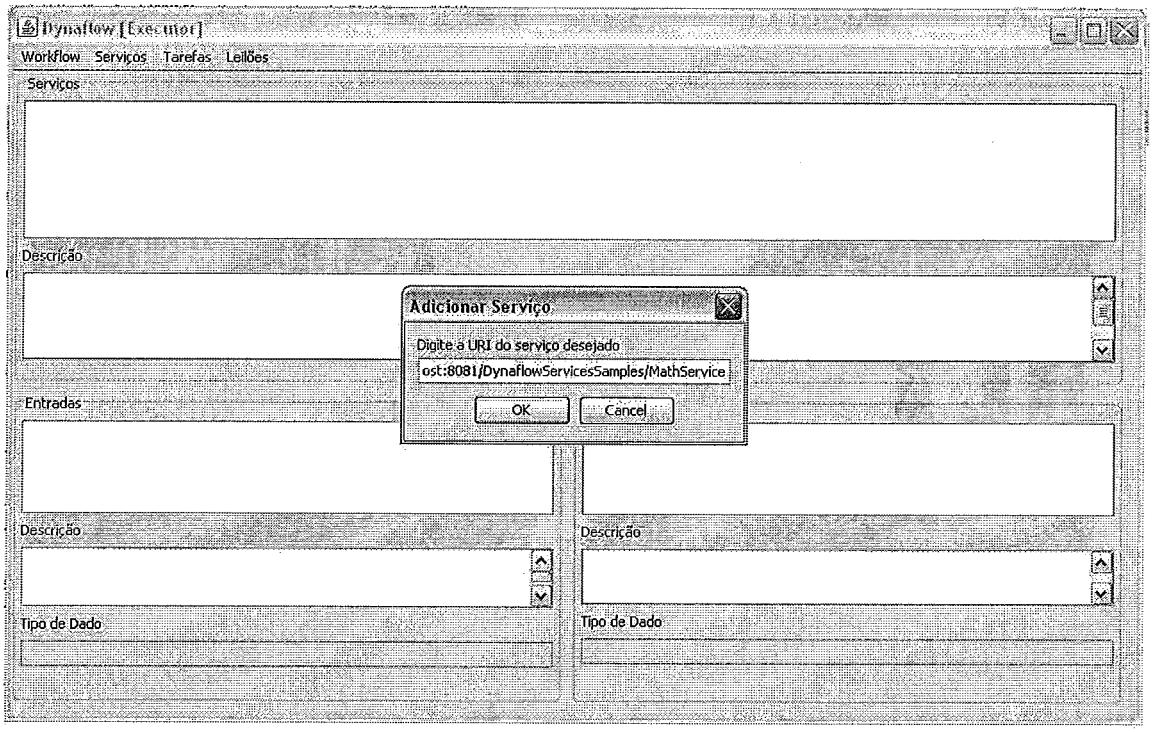


Figura 19 – Registro de serviços *web*

Com ao menos um endereço de serviço, a interface exibirá as operações disponibilizadas pelo serviço, suas documentações e seus parâmetros. Os parâmetros são acompanhados de suas respectivas documentações e seus tipos de dados (figura 20).

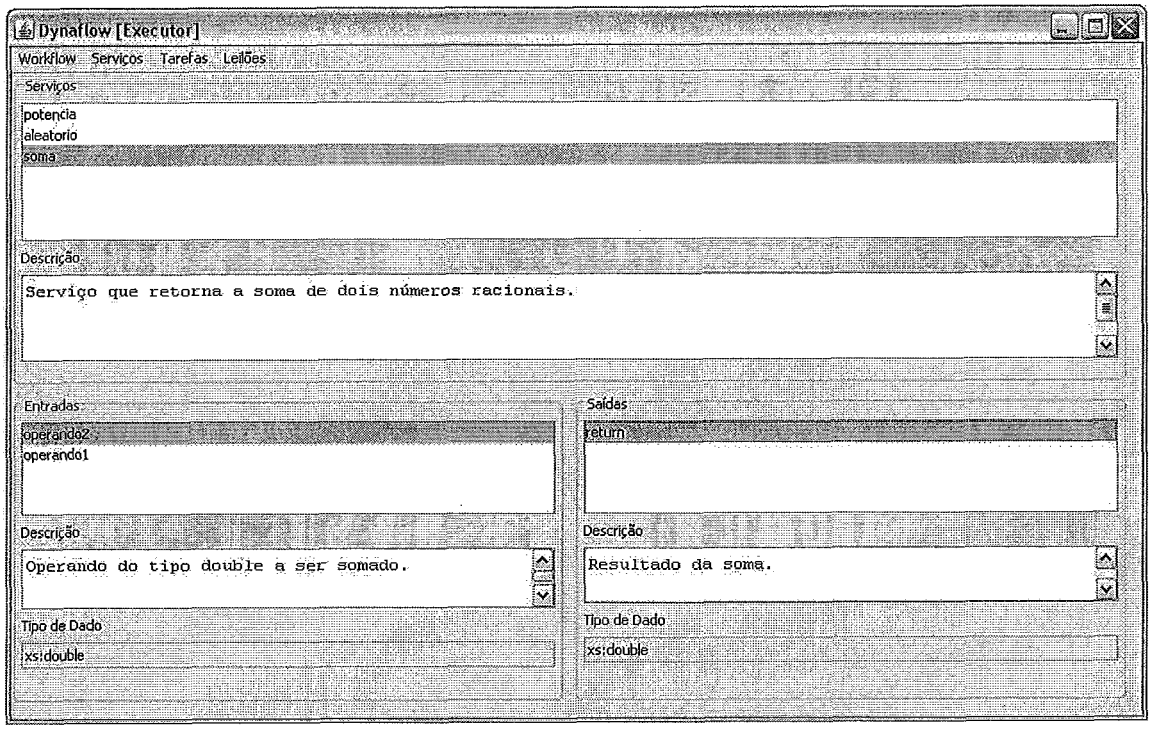


Figura 20 – Parâmetros de entrada e saída para um serviço

4.2 – Instanciação e execução de um workflow

4.2.1 – Publicador: Importar definição de um processo

Como já explicado, o sistema aceita definições de processos descritos por diagrama de atividades, exportados para o formato XMI. Na interface do publicador, o usuário deve selecionar o caminho para o arquivo com a definição do processo. Cada vez que uma definição é importada, uma instância distinta é criada.

Uma vez importado um processo, a interface exibe as tarefas do workflow, como na figura 21. O usuário pode optar por visualizar tarefas de outra instância, através da caixa de seleção "workflows".

Neste exemplo, o processo importado corresponde à equação representada pelo diagrama de atividades da figura 22.

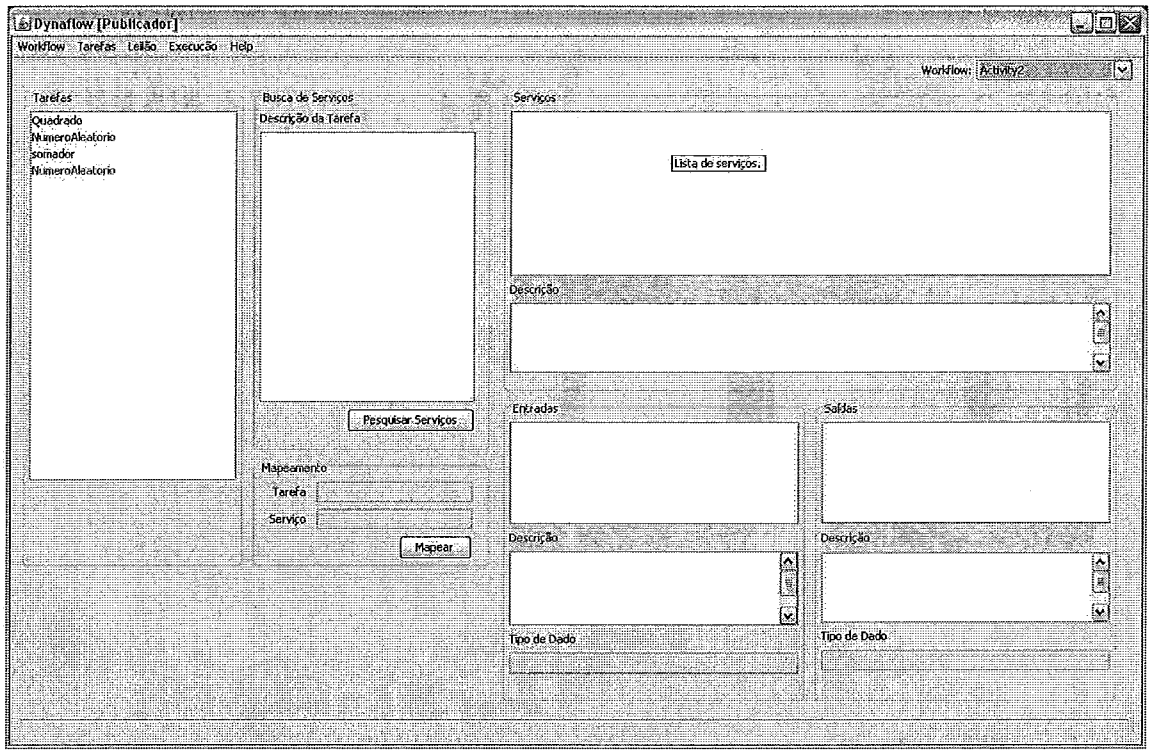


Figura 21 – Tarefas de um processo importado para o sistema

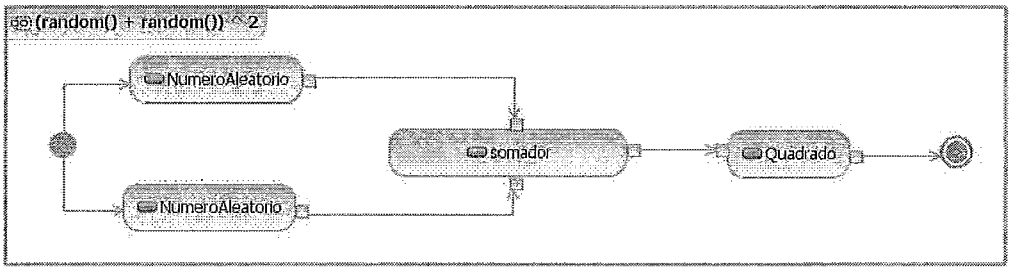


Figura 22 – Diagrama de atividades importado, que descreve uma equação

4.2.2 – Publicador: Buscar serviços

Depois de importada a definição de um processo, o publicador deve buscar serviços que possam ser mapeados em cada tarefa do processo. Como a busca é por similaridade, o usuário publicador informa uma descrição para a tarefa e solicita a busca para o sistema. Cada resposta retornada por um *peer* que possui algum serviço similar é apresentada na listagem de serviços. Caso o usuário deseje mais informações sobre o

serviço, ele pode selecionar o nome ou os parâmetros de entrada e saída para obter a respectiva descrição. Na figura 23, é apresentado um serviço recuperado com descrição similar a "aleatório".

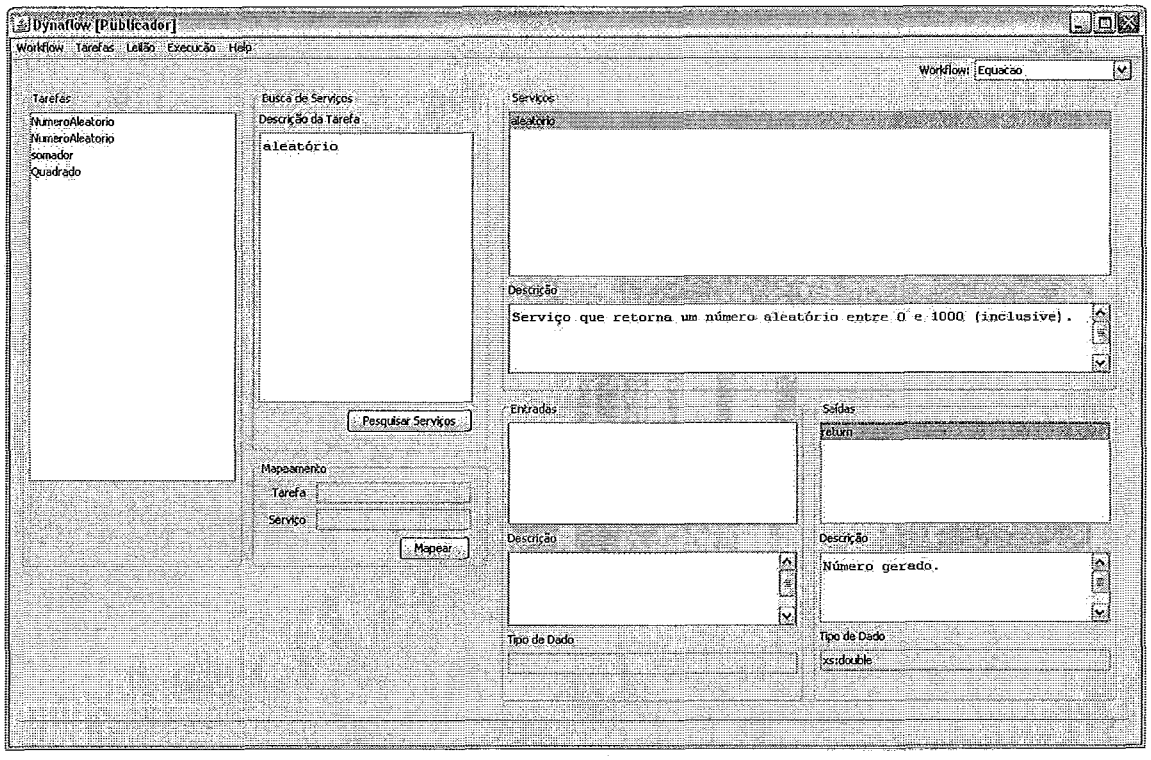


Figura 23 – Busca por serviços

4.2.3 – Publicador: Mapear tarefas

Uma vez recuperado um serviço que atenda a uma determinada tarefa, o usuário realiza o mapeamento. Para isto, ele deve selecionar a tarefa e o serviço de suas respectivas listas (figura 23). Selecionando-se "mapear", o sistema associa a tarefa ao serviço e exibe uma janela para o mapeamento dos parâmetros de entrada e de saída, como apresentado na figura 24. Para cada parâmetro da tarefa, deve-se selecionar o respectivo parâmetro do serviço.

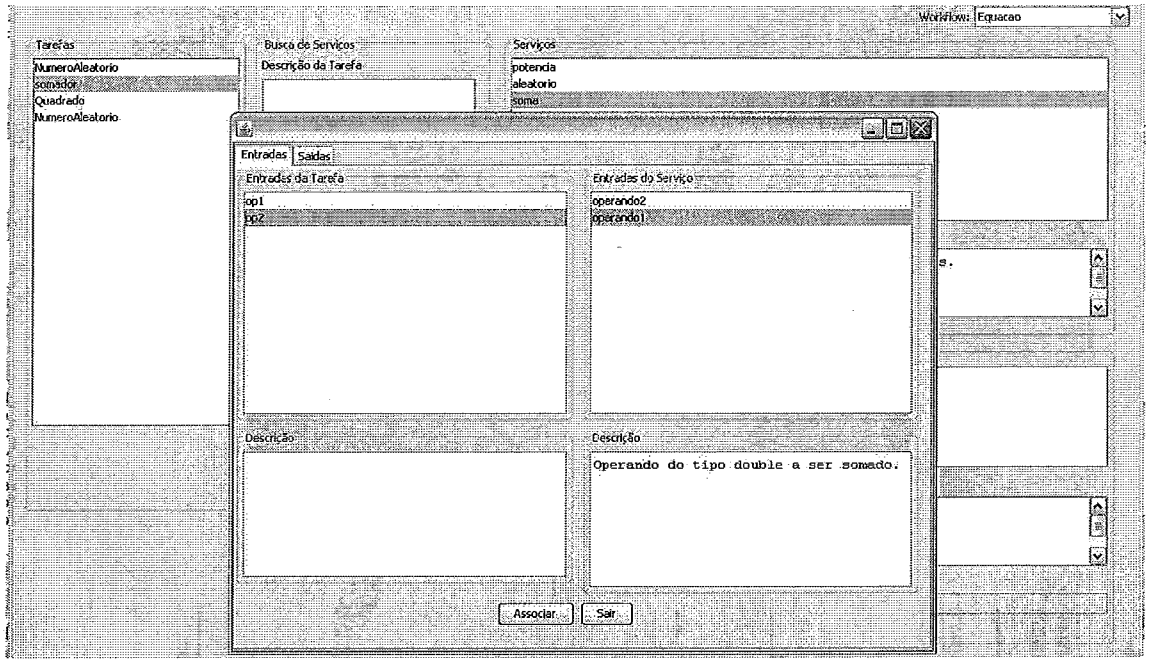


Figura 24 – Associação de parâmetros

O processo de mapeamento somente pode ser dado como finalizado quando todas as tarefas do workflow estiverem mapeadas em algum serviço.

4.2.4 – Publicador: Configurar e iniciar leilão do workflow

Depois que todas as tarefas se encontram mapeadas em serviços, o leilão de cada tarefa deve ser iniciado para que se selecione um único *peer* dentre aqueles que fornecem o respectivo serviço. O leilão é realizado para cada tarefa, mas a configuração é única para todas as tarefas de um mesmo workflow. A figura 25 apresenta a interface para a configuração do leilão.

Figura 25 – Interface de configuração do leilão para as tarefas de um workflow

No exemplo apresentado na figura 25, um leilão está sendo configurado para o workflow "Equação", utilizando-se uma estratégia de menor custo (tipo de leilão) e com duração de 60 segundos.

A tabela "Indicadores da tarefa" apresenta os parâmetros que o agente leiloeiro configurado utilizará para avaliar lances. Ao atribuir um lance, o participante do leilão deve fornecer tais parâmetros, que são obtidos a partir do agente leiloeiro configurado pelo usuário. Neste exemplo, apenas o custo é considerado.

Depois de configurado um leilão, o usuário pode iniciá-lo através da opção no *menu* da interface do publicador..

4.2.5 – Executor: Visualizar livro de ofertas

Depois de iniciado o leilão, os executores podem visualizar o livro de ofertas através da interface apropriada (figura 27).

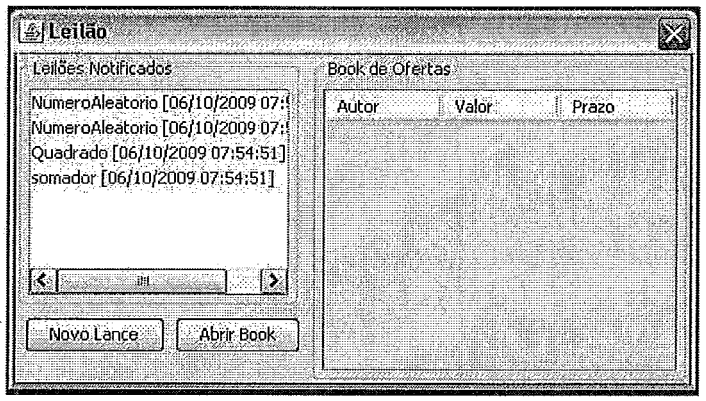


Figura 27 – Interface de configuração do leilão

Na figura 27, é possível visualizar os leilões notificados para o *peer* executor em questão. Neste caso, o executor foi notificado sobre quatro leilões, porque ele oferece os três serviços utilizados pelo publicador durante o mapeamento.

4.2.6 – Executor: Oferecer lance

Ao optar por atribuir um lance, o executor seleciona um leilão e atribui o lance na interface de visualização do livro de ofertas. No entanto, ele deve preencher os indicadores obrigatórios para o lance (figura 28).

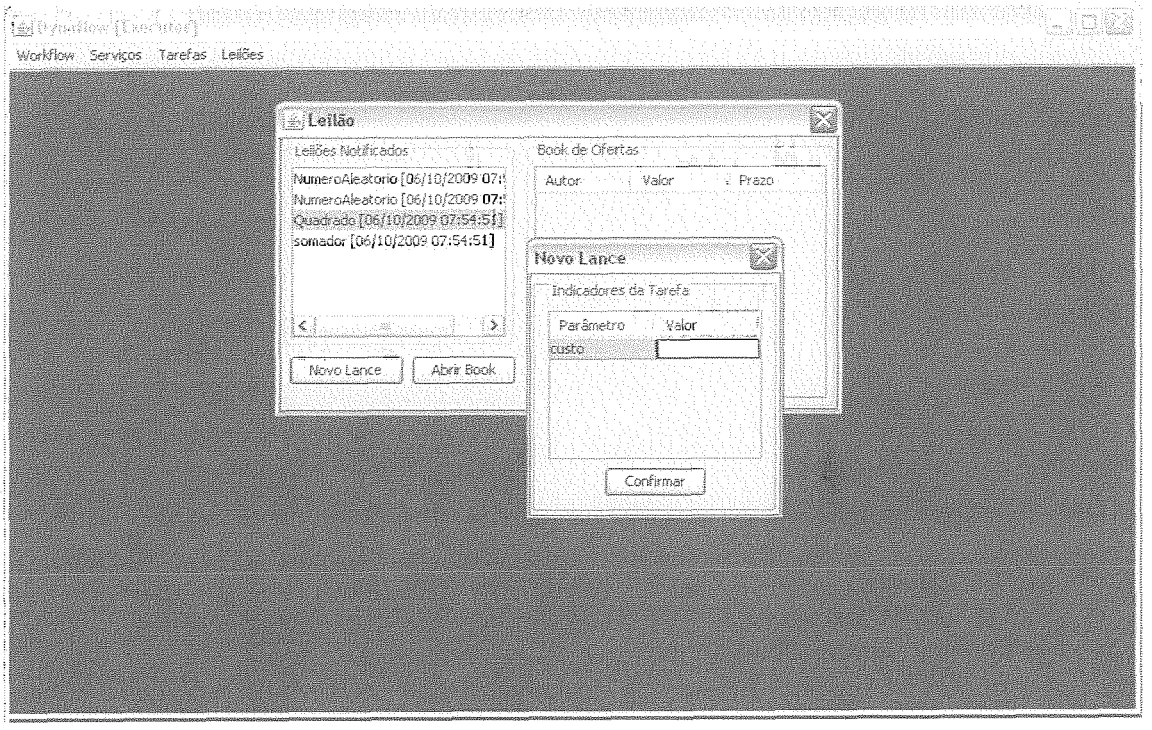


Figura 28 – Lance que requer apenas o custo

4.2.7 – Executor: Executar tarefa

Após o término do leilão das tarefas de um workflow, os executores vencedores de cada leilão recebem os agentes que apóiam a execução do workflow. A única operação que um executor pode realizar é autorizar o início da execução de uma tarefa. Entretanto, apenas as tarefas prontas para execução podem receber este comando.

Na figura 29, é apresentado um painel que mostra atividades em espera, prontas para execução e executadas.

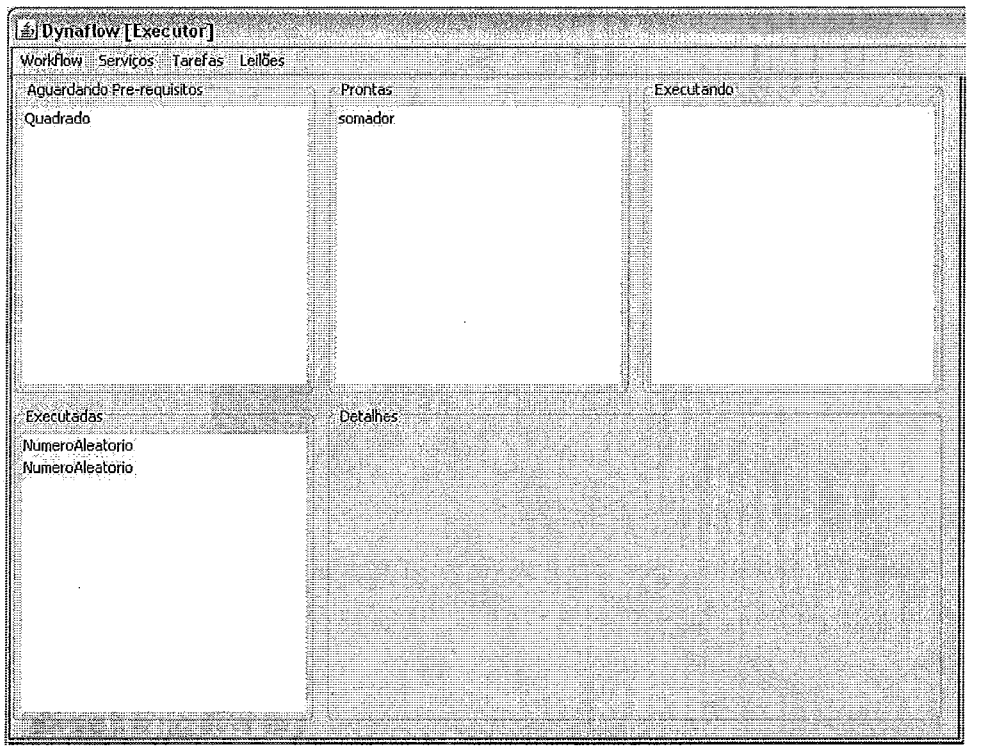


Figura 29 – Painel de execução de tarefas

4.3 – Métricas

Algumas métricas foram estabelecidas para a observação dos atributos relevantes a um sistema distribuído baseado em agentes, em geral associadas às mensagens e aos agentes que as manipulam.

As métricas foram coletadas através da execução dos processos descritos a seguir. Para cada instância de processo, foram utilizados dois *peers* (um *peer* com papel de publicador e outro *peer* com papel de executor). Portanto, as métricas serão apresentadas por *peer*.

As métricas definidas foram:

- Número de mensagens trocadas localmente no *peer*
  - Leituras da memória compartilhada
  - Escritas na memória compartilhada
- Número de mensagens trocadas remotamente com o *peer*
  - Leituras da memória compartilhada
  - Escritas na memória compartilhada
- Número de agentes movidos entre *peers*

- Número de agentes enviados pelo *peer*
- Número de agentes recebidos pelo *peer*

- Total de agentes criados no *peer*

4.3.1 – Processo A: Equação matemática

O processo A (figura 30) representa uma equação matemática simples, onde dois números racionais são gerados e somados. O valor da equação é o quadrado desta soma.

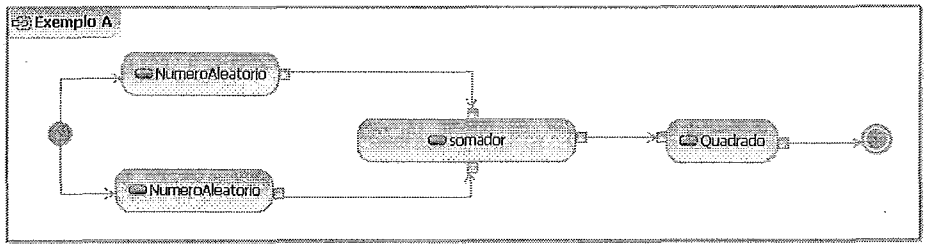


Figura 30 – processo A (equação matemática)

As métricas obtidas para cada *peer*, através da execução desta equação, são apresentadas abaixo:

- *Peer* publicador

Métrica		Valor
Local	Mensagem Leitura	53
	Mensagem Escrita	42
	Mensagens Leitura	21
	Mensagens Escrita	11
Agentes Criados		35
	Agentes Enviados	19
	Agentes Recebidos	3
Agentes Movidos localmente		1

- *Peer* executor

Métrica		Valor
Local	Mensagem Leitura	75
	Mensagem Escrita	66
	Mensagens Leitura	20
	Mensagens Escrita	26
Agentes Criados		7
	Agentes Enviados	3
	Agentes Recebidos	19
Agentes Movidos localmente		6

4.3.2 – Processo B: Monitoramento de temperatura

Este processo ilustra um mecanismo simples para o monitoramento da temperatura de uma caldeira industrial, descrito pelo processo apresentado pela figura 31.

O processo começa com a medição da temperatura da caldeira, exercida pela atividade "Termômetro". Toda temperatura aferida é registrada em um banco de dados, através da execução da atividade "Registrar Temperatura".

O sistema pode disparar um alerta caso a medição indique que a temperatura da caldeira ultrapassou o nível de segurança de 500° Celsius. Caso a temperatura esteja dentro da faixa segura (abaixo dos 500° Celsius), nenhum alarme é emitido e o processo é encerrado.

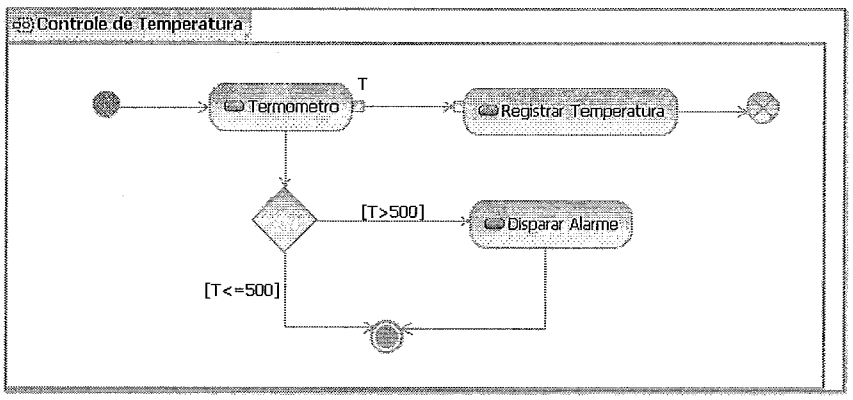


Figura 31 – Processo B (Monitoramento da temperatura de uma caldeira)

As métricas coletadas em cada *peer*, após a execução desta equação, são apresentadas abaixo:

- *Peer* publicador

Métrica		Valor
Local	Mensagem Leitura	40
	Mensagem Escrita	33
	Mensagens Leitura	11
	Mensagens Escrita	14
Agentes Criados		32
	Agentes Enviados	14
	Agentes Recebidos	1
Agentes Movidos localmente		3

- *Peer* executor

Métrica		Valor
Local	Mensagem Leitura	42
	Mensagem Escrita	42
	Mensagens Leitura	12
	Mensagens Escrita	16
Agentes Criados		5
	Agentes Enviados	1
	Agentes Recebidos	14
Agentes Movidos localmente		5

5 – Conclusão

Este é um trabalho, fundamentalmente, de Arquitetura de *Software*. Durante o trabalho de análise, projeto e implementação do protótipo apresentado, algumas boas práticas para o projeto de uma arquitetura multi-agente puderam ser extraídas. Projetar uma arquitetura multi-agentes tem forte relação com a definição de protocolos de comunicação entre os seus agentes. Um das principais preocupações quando se projeta uma funcionalidade em um sistema multi-agente é estabelecer protocolos que guiem a troca de mensagens dentre os agentes criados para atender à funcionalidade. Protocolos elaborados sem o devido cuidado podem levar a complexidades difíceis de serem gerenciadas. Um bom protocolo possibilita que os agentes se comuniquem de forma organizada, compartilhando seus conhecimentos e, assim, colaborem de forma eficiente para atingir ao objetivo do conjunto. Além disto, agentes devem ser projetados com foco no comportamento sistemático, que emerge das suas interações.

Para o estudo de MAS (*multi-agent systems*), uma classe de aplicações foi escolhida: workflows. Neste trabalho, foram apresentados argumentos de autores que defendem que workflows possuem uma natureza descentralizada e distribuída, e que, portanto, não se adéquam a arquiteturas centralizadas quando questões como robustez, desempenho e escalabilidade são relevantes. Vimos como uma arquitetura multi-agentes pode auxiliar na descentralização da coordenação de tarefas em um workflow e que dois requisitos são fundamentais para o sucesso de tal arquitetura: agentes minimalistas e com conhecimento restrito ao relevante para o seu trabalho.

A arquitetura proposta não apresenta pontos de centralização para coordenar a arquitetura. Toda a execução ocorre sem a interferência constante de uma mesma instância de um agente. Nenhum agente possui conhecimento completo do workflow. O *peer* publicador apenas participa ativamente nas etapas anteriores a execução. Depois de iniciada a execução, o *peer* publicador atua passivamente, recebendo apenas o *status* do andamento do processo.

O Dynaflow é um protótipo para coordenação, de forma descentralizada, da execução de processos de negócio, cujas tarefas são serviços computacionais distribuídos dentre participantes de uma rede. O Dynaflow ainda não pode ser definido como um sistema de gestão de workflows (WfMS), pois sua preocupação, por enquanto, não é oferecer funcionalidades para gestão de workflows, e sim apresentar uma proposta

de arquitetura básica, que funcione de forma descentralizada e distribuída, em ambientes puramente descentralizados, como redes *peer-to-peer*.

Enquanto sistemas centralizados possuem limitações de escalabilidade inerentes, os sistemas *peer-to-peer* se beneficiam da chegada de novos nós na rede. No Dynaflow, mais *peers* significa mais serviços *web* disponíveis para consumo. A entrada de novos *peers* enriquece a rede, favorecendo a publicação de mais workflows.

Apesar da ausência de centralização na etapa de execução, pode ser necessária uma participação mais ativa por parte do publicador em algumas situações de exceções. Como exemplo, a detecção e o tratamento das situações em que um *peer* ultrapassa o limite de tempo tolerado para execução de sua tarefa, podem ser mais simples se realizados com o auxílio do *peer* publicador, graças ao seu conhecimento de todo o workflow.

5.1 – Resultados

O sistema funcionou conforme esperado, coordenando workflows como os utilizados para a sua validação. As metas definidas no capítulo 1 foram alcançadas:

- Executar em ambientes *peer-to-peer*, realizando a coordenação de workflows de forma puramente descentralizada;
- Aplicação de conceitos de sistemas multi-agentes, através da aplicação de agentes minimalistas, especializados em tarefas bem definidas e pequenas;
- Validação do *framework* COPPEER;
- Estudou a modelagem orientada a agentes, mostrando como agentes possibilitam o projeto de arquiteturas descentralizadas.

5.2 – Considerações sobre o COPPEER

Como uma das metas deste trabalho foi validar o *framework* COPPEER, foram extraídas algumas considerações durante a sua utilização:

- Durante o ciclo de vida de uma aplicação multi-agentes, alguns agentes podem se tornar inúteis. O *framework* deve oferecer um mecanismo que elimine agentes

obsoletos, para cessar a atuação desses agentes sobre o ambiente e para evitar o desperdício de recursos computacionais, como a memória *heap* do sistema;

- Uma das questões mais importantes no projeto de um sistema multi-agentes baseado em troca de mensagens, como o COPPEER, é a elaboração dos protocolos que regem as interações entre os agentes. Isto envolve a elaboração de diversas mensagens e o cuidado no projeto de cada agente para se inscrever corretamente como observador dos eventos. Para suportar a elaboração dos protocolos, poderia ser criada uma solução baseada em MDA (*model driven architecture*) para a modelagem dos protocolos para a troca de mensagens. Outra solução possível seria um *plugin* gráfico que trouxesse algum grau de abstração para a interação entre os agentes;
- Durante o seu desenvolvimento, as versões de uma aplicação evoluem rapidamente. Em algumas situações, alguns *peers* podem ter uma versão desatualizada da aplicação, o que resulta em problemas durante a execução. Para evitar isso, o sistema poderia notificar o usuário sobre versões incompatíveis;
- O *framework* poderia suportar a atualização automática de aplicações, para evitar o esforço manual no *deploy* de aplicações;
- Agentes poderiam ser implementados em uma linguagem de mais alto nível que Java, que fornecesse um grau de abstração suficiente para projeto rápido e preciso dos agentes.

5.3 – Barramento de serviços

Observando o sistema sob a ótica de SOA (arquitetura orientada a serviços), observa-se que ele possibilita a orquestração de serviços fracamente acoplados e interoperáveis (*web services*) distribuídos pelos diversos *peers* de uma rede P2P, seguindo uma definição formal (o diagrama de atividades). Desta forma, o sistema possui alguma proximidade com o conceito de barramento de serviços e uma vertente desta pesquisa pode ser criada para explorar este aspecto.

5.4 – Trabalhos futuros

Geralmente, trabalhos acadêmicos consideram cenários ideais. No caso deste trabalho, o tratamento de exceções durante a execução de um workflow não fez parte do seu escopo. No entanto, algumas idéias simples podem ser discutidas em trabalhos futuros.

5.4.1 – Tratamento de *peer offline*

Redes P2P não trazem garantias em relação à permanência de seus *peers* na rede. A saída de um *peer* que possui o papel de executor para um workflow requer que o sistema trate a ausência deste *peer*. Uma solução óbvia é a escolha de outro *peer* para cumprir o papel do *peer* que abandonou a rede. Entretanto, esta escolha envolve dois detalhes:

- Deve-se encontrar ao menos um *peer* que ofereça o mesmo serviço perdido com a saída do *peer*;
- Dentre os possíveis *peers* provedores do serviço, o sistema deverá eleger um único *peer*, através de um novo leilão.

5.4.1.1 - Detecção

A detecção da desconexão de um *peer* é realizada pelo próprio *framework* COPPEER, ao tentar realizar qualquer tipo de comunicação com o *peer* desconectado. O *framework* notifica um agente ao tentar realizar algum tipo de operação que envolva uma agência desconectada.

5.4.1.2 - Tratamento

No primeiro leilão realizado para a tarefa, durante a instanciação do workflow, pode-se armazenar a lista de *peers* que possuem o serviço, ordenados por ordem de colocação no leilão, e não apenas o ganhador do leilão. Os demais colocados no leilão seriam contatados caso o *peer* vencedor se desconectasse da rede.

Como apenas o *peer* publicador possui informações sobre todo o workflow, ele deverá participar deste processo, contatando e enviando os agentes necessários para o novo *peer*.

5.4.2 – Tratamento de *timeout* na execução de uma tarefa

Um *peer* executor não deve ultrapassar o limite de tempo destinado a realização da tarefa.

5.4.2.1 – Detecção

O *peer* publicador pode sabe qual *peer* está sendo notificado no momento, pois ele recebe mensagens de status ao início e ao final da execução de cada tarefa.

5.4.2.2 – Tratamento

O *peer* publicador envia uma mensagem ao peer que está executando a tarefa, notificando que ele pode abortar a tarefa, pois ela será delegada a outro *peer*. O tratamento será semelhante ao do *peer offline*: um novo leilão será realizado ou o próximo colocado no leilão será notificado.

6 – Referências Bibliográficas

Alonso, G. et al., 1995. **Exotica/FMQM: A persistent message-based architecture for distributed workflow management.** In *Proceedings of the IFIP WG8*. Trondheim, Norway, Aug. 1995, PP. 1-18

Babaoglu, O., Meling, H. & Montresor, A., 2002. **Anthill: A framework for the development of agent-based peer-to-peer systems.** In *International Conference on Distributed Computing Systems*. IEEE Computer Society; 1999, pp. 15-22.

Berkeley, 1999. **SETI@home**. Disponível em http://setiathome.ssl.berkeley.edu [Acessado em 23 de março de 2009]

Chawathe, Y. et al., 2003. **Making gnutella-like p2p systems scalable.** In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM New York, NY, USA, pp. 407-418.

Clarke, I. *et al.*, 2001. **Freenet: A Distributed Anonymous Information Storage and Retrieval System**. In *Designing Privacy Enhancing Technologies*. pp. 46-66.

Davenport, 1993.

De Wolf, T. & Holvoet, T., 2005. **Emergence Versus Self-Organisation: Different Concepts but Promising When Combined.** In *Engineering Self-Organising Systems*. pp. 1-15. Disponível em: http://dx.doi.org/10.1007/11494676_1 [Acessado 25 de julho de 2009].

Dyson, G., 1997. **Darwin among the machines: The evolution of global intelligence**, Basic Books.

Eder, J., Panagos, E., 1999. **Towards distributed workflow process management.** In *Proceedings of the WACC workshop on Cross-organizational workflows*.

Ellis, C.A., 1999. **Workflow technology.** *Computer Supported Cooperative Work. Wiley, Chichester, UK*, 29–54.

Fakas, G.J., Karakostas, B., 2004. **A peer to peer (P2P) architecture for dynamic workflow management.** *Information and software technology*, 46(6), 423-431.

Fanning, S., Parker, S., 2002. **Napster**. *Los Angeles, CA: Napster, Inc.*

Frankel, J., Pepper, T., 1999. **Gnutella**. Disponível em *www.gnutella.com.*

Gnutella. The *Gnutella Protocol Specification* v0.4. Disponível em *rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.htm* [Acessado em 4 de junho de 2009].

Grundy, J. et al., 1998. **A decentralized architecture for software process modeling and enactment.** *Internet Computing, IEEE*, 2(5), 53-62.

Guessoum, Z. & Briot, J., 1999. **From active objects to autonomous agents.** *Concurrency, IEEE*, 7(3), 68-76.

Heinl, P. et al., 1999. **A comprehensive approach to flexibility in workflow management systems.** *SIGSOFT Softw. Eng. Notes*, 24(2), 79-88.

Hollingsworth, D. & others, 1994. **Workflow management coalition: The workflow reference model.** Document TC00-1003, Workflow Management Coalition, Dec.

Jendrock, E., Ball, J., Carson, D., Evans, I., Fordin, S., Haase, K., 2008. **The Java EE 5 Tutorial.** Disponível em http://java.sun.com/javaee/5/docs/tutorial/doc/sjsaseej2eet.html

Jennings, N.R., 2001. **An agent-based approach for building complex software systems.** *Commun. ACM*, 44(4), 35-41.

Joeris, G., 1999. **Defining flexible workflow execution behaviors.** *Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, 24, 49–55.

Karger, D. et al., 1997. **Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.** In *annual acm symposium on theory of computing*. The assocation for computing, pp. 654-663.

Katz, M.L. & Shapiro, C., 1994. **Systems competition and network effects.** *The Journal of Economic Perspectives*, 93-115.

Kiepuszewski, B., 2002. **Expressiveness and suitability of languages for control flow modelling in workflows.** PhD thesis, Queensland University of Technology, 2002.

Ludascher, B. et al., 2006. **Scientific workflow management and the Kepler system.** *Concurrency and Computation: Practice and Experience*, 18(10), 1039-1065.

Ludascher, B. & Bowers, S., 2005. **Actor-oriented design of scientific workflows.** *Lecture Notes in Computer Science*, 3716, 369.

Milojicic, D.S. et al., 2002. **Peer-to-peer computing**, Technical Report HPL-2002-57, HP Lab, 2002.

Miranda, M., Xexeo, G. & de Souza, J., 2006. **Building Tools for Emergent Design with COPPEER.** In *Computer Supported Cooperative Work in Design, 2006. CSCWD '06. 10th International Conference on.* pp. 1-6.

Muth, P. et al., 1998. **From centralized workflow specification to distributed workflow execution.** *Journal of Intelligent Information Systems*, 10(2), 159–184.

OMG, 2002. **Unified Modeling Language, Superstructure**, v2.2. Disponível em: http://www.omg.org/cgi-bin/doc?formal/09-02-02 [Acessado 22 de Setembro de 2009].

Parunak, H.V., 1997. **"Go to the ant": Engineering principles from natural multi-agent systems.** *Annals of Operations Research*, 75, 69–102.

Portmann, M. et al., 2001. **The cost of peer discovery and searching in the gnutella peer-to-peer file sharing protocol.** In *Ninth IEEE International Conference on Networks, 2001. Proceedings.* pp. 263-268.

Russell, S.J. et al., 1995. **Artificial intelligence: a modern approach**, Prentice hall Englewood Cliffs, NJ.

Stoica, I. et al., 2001. **Chord: A scalable peer-to-peer lookup service for internet applications.** In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.* ACM New York, NY, USA, pp. 149-160.

Sun, 1995. **Java Technology.** Disponível em http://java.sun.com/

Sun, 1999. **JXTA**. Disponível em http://jxta.dev.java.net/

Vieira, P. & Rito-Silva, A., 2005. **Adaptive workflow management in WorkSCo.** In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on.* pp. 640-645.

Yan, J., Yang, Y. & Raikundalia, G.K., 2006. **SwinDeW - A p2p-Based Decentralized Workflow Management System.** *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART A: SYSTEMS AND HUMANS*, 36(5).

Yang, Y, 2000. **Architecture and the related mechanisms for web-based global cooperative teamwork support.** Informatica (Ljubljana). Vol. 24, no. 1, pp. 13-19. Mar. 2000

W3C, 2004. **Web Services Architecture.** Disponível em http://www.w3.org/TR/ws-arch. [Acessado em 22 de janeiro de 2009]

Wang, A.I., Liu, C. & Conradi, R., 1999. **A multi-agent architecture for cooperative software engineering.** In *Proc. of The Eleventh International Conference on Software Engineering and Knowledge Engineering (SEKE'99).* pp. 1–22.

Weiss, G., 2000. **Multiagent systems: a modern approach to distributed artificial intelligence**, The MIT Press.

WfMC – Workflow Management Coalition.
Disponível em: http://www.wfmc.org. [Acessado em 20 de janeiro de 2009.]

WfMC, 1995. **The Workflow Reference Model.** The Workflow Management Coalition Specification. Disponível em: http://www.wfmc.org/reference-model.html [Acessado em 28 de janeiro de 2009].

WfMC, 1999. **Terminology & Glossary.** The Workflow Management Coalition Specification. [Acessado em 27 de janeiro de 2009]. Disponível em: http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

Wooldridge, M., 1997. **Agent-based software engineering.** *Software Engineering. IEE Proceedings- [see also Software, IEE Proceedings]*, 144(1), 26-37.