Paulo Rogério da Motta Júnior

Sinapse
Um Arcabouço para Suporte a Aplicações *Publish/Subscribe* baseado em Configuração Arquitetural

Dissertação apresentada ao Curso de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre.

Área de concentração: Processamento Paralelo e Distribuído

Orientador: ORLANDO GOMES LOQUES FILHO

NITERÓI

2005

"O processo de preparar programas para um computador digital é especialmente atrativo, não só porque pode ser economicamente e cientificamente recompensador, mas também porque pode ser uma experiência harmoniosa tal como escrever poemas ou compor música"

Donald E. Knuth

The Art of Computer Programming, 1997
Prefácio, volume 1

Aos meus pais que sempre acreditaram em mim,
e iniciaram minha jornada computacional.

Agradecimentos

Este pequeno espaço não me permite listar os nomes de todos os amigos que me ajudaram ao longo dessa jornada que foi o curso de mestrado. Muitas vezes suas contribuições não foram conceitos ou comentários técnicos, mas sim uma mão de apoio, ou um ombro para conseguir fôlego até o fim do caminho. Muitas vezes compreendendo as minhas ausências. A todos vocês eu agradeço imensamente. Não podendo deixar de enumerar algumas pessoas, devo agradecer:

Aos amigos Gabriel Gagliano e Angela Portela, sem seu infinito amor e cuidado eu nunca chegaria até aqui.

Ao meu orientador Orlando Gomes Loques Filho, pelo crescimento que me proporcionou através de suas aulas e reuniões.

À Professora Maria Alice Silveira de Britto, da Universidade do Estado do Rio de Janeiro, por seu incentivo ao ingresso no curso de mestrado e por ter me iniciado à pesquisa científica.

Aos amigos que me ajudaram com inúmeras e valiosas revisões deste texto.

Aos companheiros de curso, com os quais muitas horas de trabalho foram compartilhadas.

Resumo da Tese apresentada à UFF como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação (M.Sc.)

Paulo Rogério da Motta Júnior

Dezembro 2005

Orientador: Orlando Gomes Loques Filho

Programa de Pós-Graduação em Computação

O paradigma *Publish/Subscribe*, implementado pelos chamados Sistemas de Notificação, ocupa um importante papel no contexto de desenvolvimento de sistemas distribuídos, permitindo o desacoplamento entre os participantes graças à comunicação assíncrona e anônima. Embora essa característica garanta uma maior flexibilidade, ela não é transparente. O código da aplicação tende a incluir seções para tratamento da interação com o Sistema de Notificação utilizado, criando dependências. Outra aliada ao desenvolvimento de aplicações distribuídas é a modelagem arquitetural, implementada pelos ambientes de configuração. Neste paradigma, os componentes da aplicação são divididos entre funcionais, relacionados ao domínio da aplicação e, não-funcionais, relacionados aos aspectos de suporte à aplicação, tais como as tarefas de comunicação por exemplo. Propomos neste trabalho uma adaptação do Sistema de Notificação "Rebeca", tornando possível sua execução sobre o ambiente de configuração "AC". Essa adaptação expõe os pontos de configuração do suporte à comunicação. Aproveitando a capacidade de interceptação do ambiente AC, isolamos as interações dos módulos da aplicação, o que diminui as dependências do código em relação ao Sistema de Notificação. Além disso, tornamos transparente o suporte à composição de eventos e mobilidade. O modelo proposto é analisado principalmente sob o ponto de vista qualitativo, evidenciando as facilidades disponibilizadas frente o modelo original do Rebeca.

Abstract of Thesis presented to UFF as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

Paulo Rogério da Motta Júnior

December 2005

Advisors: Orlando Gomes Loques Filho

Department: Computer Science

The Publish/Subscribe paradigm implemented by Notification Systems, plays an important role in the context of distributed systems development, allowing the decoupling of the participants due to its anonymous and asynchronous communication. Although this characteristic assures greater flexibility, it is not transparent. The application code tend to include sections to handle the interaction with the Notification System used, creating dependencies. Distributed systems development can be also simplified by architectural modeling implemented by configuration systems. In this paradigm, the application components are divided between functional, related to the application domain and non-functional, related to application's support aspects, like communication tasks for instance. We propose in this work an adaptation of the Rebeca Notification System, making it possible to execute over the AC configuration system. With this adaptation, communication support configuration points are exposed and, using the interception capabilities of the AC system, we could isolate the interactions among application modules, what decreases code dependencies related to the Notification System. We also could turn event composition and mobility transparently supported. The proposed model is analyzed by the qualitative view, making evident the facilities that becomes available when compared to the original Rebeca system.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Com a grande evolução que as tecnologias de rede sofreram, tornou-se cada vez mais fácil conectar computadores, permitindo a criação de sistemas distribuídos. Podemos entender como sistema distribuído um conjunto de computadores independentes, trabalhando de forma a proporcionar ao usuário a sensação de estar usando um único e coerente sistema [TS2002].

A utilização de sistemas distribuídos em contraste com os sistemas centralizados traz vantagens, tais como: proporciona atender um maior número de usuários; apresenta maior disponibilidade, já que o sistema está distribuído em diversas máquinas; tem menor custo de expansão, pois o aumento de poder de processamento é realizado com a adição de máquinas, utilizando um conjunto de componentes mais simples.

No entanto, a complexidade para desenvolver e manter aplicativos como sistemas distribuídos é maior, uma vez que podem fazer parte do sistema máquinas de diferentes arquiteturas com diferentes sistemas operacionais. De acordo com [Pie2002], sistemas *middleware* vêm sendo pesquisados objetivando disponibilizar para os programadores de aplicações uma camada de abstração que permita simplificar o desenvolvimento. Notadamente, os Sistemas de Notificação têm se apresentado como uma boa solução.

Os Sistemas de Notificação podem trazer grandes benefícios para o desenvolvimento de sistemas distribuídos, pois oferecem mecanismos de abstração da comunicação entre os módulos da aplicação. Essa abstração facilita a manutenção de componentes de forma independente e o desenvolvimento de aplicações distribuídas ocorre de maneira flexível e escalável.

O paradigma *Publish/Subscribe*, implementado pelos Sistemas de Notificação, permite o desacoplamento dos módulos participantes de um dado processamento. Alguns módulos do aplicativo produzem determinados tipos de informação, que são disponibilizados a mecanismos de comunicação anônima e assincronamente. Em contraste, outros módulos são

preparados para consumir informações através desses mesmos mecanismos de comunicação, também anônima e assincronamente. Assim, a troca de informações acontece sem que os participantes tomem conhecimento um do outro.

Um Sistema de Notificação utiliza uma infra-estrutura de suporte para permitir a interação dos módulos de aplicação. Os módulos se registram junto a essa infra-estrutura, indicando o tipo de informação que publicam e/ou que subscrevem. As informações são disponibilizadas por métodos de publicação utilizados pelos produtores. Através de mecanismos de assinatura, os consumidores indicam para a infra-estrutura o tipo de informação que desejam receber. Estes mecanismos de assinatura se assemelham semanticamente às operações de álgebra relacional utilizada em bancos de dados relacionais [Cil2003].

Com a Internet, os sistemas distribuídos ganharam uma nova dimensão, podendo atingir uma escala mundial e ter seus componentes espalhados em diversos países [CRW2000]. Quando são usadas técnicas de chamada remota, tais como RPC e RMI [TS2002], a interação é síncrona, criando uma dependência entre os módulos. Outra dependência desse tipo de técnica é o uso de referências diretas entre os módulos participantes, criando acoplamento. Embora possa ser usado um serviço de nomes para identificar os módulos, diminuindo o acoplamento entre eles, isso restringe os aplicativos quanto à escalabilidade. A característica de desacoplamento dos Sistemas de Notificação permite contornar as dependências oferecendo maior escalabilidade. É importante notar que este desacoplamento é resultado de não haverem referências explícitas entre os produtores/consumidores envolvidos em um processamento.

Uma vantagem do paradigma *Publish/Subscribe*, que é um efeito direto do desacoplamento de produtores e consumidores, é que diversos consumidores podem receber uma mesma informação de forma transparente. Isto porque a própria infra-estrutura de suporte se encarrega da entrega aos diversos destinos. De outro modo, todos os destinos deveriam ser conhecidos.

Percebemos, intuitivamente, que a evolução e manutenção de aplicativos desenvolvidos usando o paradigma *Publish/Subscribe* torna-se mais simples e pode ocorrer em etapas. Diferentes módulos podem ser trocados de maneira dinâmica e transparente, já que a comunicação é assíncrona. Isso permite que versões de diferentes componentes sejam substituídas durante a execução sem que seja necessário parar todo o aplicativo. Portanto, muitas funcionalidades continuam sendo utilizadas pelos usuários do sistema. Devemos observar que a possibilidade de substituir os módulos durante a execução é um resultado da independência entre os módulos, mas isso não garante a consistência dos estados de cada módulo. Para tirar proveito dessa possibilidade, os módulos da aplicação devem contar com um mecanismo que permita desativá-los, para então serem substituídos sem comprometimento da consistência de seu estado.

Uma evolução das capacidades de Sistemas de Notificação está na área conhecida como composição de eventos. Esta funcionalidade possibilita que o consumidor seja aliviado do processamento de informação oriunda de mais de um produtor. A composição de eventos está relacionada à aplicação de operadores lógicos de modo a identificar, por exemplo, a interseção ou disjunção de diversos itens, disponibilizando para o consumidor somente o resultado final. O suporte à composição de eventos permite que consumidores interessados no mesmo tipo de informação não tenham que escrever separadamente as mesmas lógicas de interpretação, diminuindo a complexidade envolvida no desenvolvimento. Conseqüentemente, a composição de eventos contribui para a evolução e manutenção de aplicativos.

Com a grande evolução na área de computação móvel, pesquisas têm sido direcionadas para o desenvolvimento de aplicações que possam se beneficiar da mobilidade. Em paralelo, também têm sido pesquisados métodos para disponibilizar o suporte à mobilidade a aplicações já existentes, ou seja, tornar possível executar em ambiente móvel aplicações que foram originalmente concebidas para uso em um ambiente fixo. Em ambos os casos, o uso de Sistemas de Notificação tem se mostrado muito promissor.

Utilizando a forma assíncrona e anônima de comunicação dos Sistemas de Notificação, aplicações legadas podem ser adaptadas aos ambientes com mobilidade. De maneira semelhante, aplicações que precisam rodar em ambientes móveis podem se beneficiar desse tipo de comunicação para realizar seus processamentos. O modelo de comunicação dos Sistemas de Notificação permite contornar as dificuldades dos ambientes com mobilidade, onde a comunicação entre módulos é fragilizada pela constante desconexão. Uma vez que não existem referências diretas entre módulos, caso aconteça uma desconexão, isso não provoca um erro na aplicação.

Os Sistemas de Notificação podem ser estruturados através de um modelo baseado em componentes. Estes componentes podem ser usados para descrever o sistema em ambientes de configuração. No contexto de ambientes de configuração, os componentes de um sistema são vistos como módulos e/ou conectores e têm suas funcionalidades especificadas de acordo com uma interface bem definida. Assim, os módulos e conectores operam ortogonalmente entre si, de maneira que um conjunto de conectores pode ser encadeado para atingir uma funcionalidade composta de múltiplos passos. Uma característica importante da utilização de ambientes de configuração é que os aplicativos podem ser desenvolvidos de acordo com uma descrição arquitetural de seus módulos. Então, a interconexão entre os módulos é realizada pelo ambiente, e a partir daí conectores podem ser inseridos e removidos.

O estudo de diferentes soluções para Sistemas de Notificação leva a um conjunto mínimo de componentes e/ou funcionalidades necessárias, que definem esse tipo de sistema. Este conjunto mínimo possibilita apresentar um modelo arquitetural capaz de ser descrito em um ambiente de configuração.

O uso de um ambiente de configuração, juntamente com a modelagem de um Sistema de Notificação baseada em conectores, permite criar um modelo que agrupe as características de suporte a mobilidade; composição e filtragem de eventos; e transparência na interação com o Sistema de Notificação.

Empregando o modelo proposto, podemos implementar um arcabouço de *software* para o desenvolvimento de sistemas baseados no paradigma *Publish/Subscribe* que facilite o uso de composição de eventos e mobilidade. Além disso, os recursos de configuração possibilitam uma simplificação no processo de expandir e/ou modificar a aplicação e contornar situações de falha. Para tanto, este arcabouço de *software* necessita dos recursos de um ambiente de configuração.

A partir das características funcionais deste modelo, utilizaremos o nome Sinapse, em referência à forma como são realizadas as transmissões de informação pelos neurônios.

1.1 Objetivos

Pretende-se, utilizando a capacidade de interceptação do Ambiente de Configuração AC [Lis2003], criar um arcabouço de *software* que permita a configuração da infra-estrutura do Sistema de Notificação Rebeca [Fie2003a]. Para tanto, propomos a adaptação dos componentes do Sistema de Notificação para que se tornem conectores especializados. A partir do uso de conectores, podemos isolar o roteamento da informação do processamento de filtros de eventos, e encapsular as interações com módulos externos. Introduz-se, assim, três tipos de conectores especializados nas seguintes tarefas: roteamento de informação, composição e filtragem de eventos, e interação com módulos externos.

Em nosso modelo, teremos um controle de alto nível sobre os componentes que formam a infra-estrutura do Sistema de Notificação. Este controle sobre os componentes, facilitará a resolução de falhas, pois permitirá modificar as conexões entre os conectores de roteamento, proporcionando maior flexibilidade de trabalho.

Considerando a facilidade de interceptação de chamadas de método disponibilizada pelo Ambiente de Configuração AC [Lis2003], os programadores poderão desenvolver aplicações de forma mais organizada. Isto porque, os aspectos relativos ao domínio da aplicação ficam separados dos aspectos necessários à interação com a infra-estrutura.

1.2 Organização da Dissertação

Este trabalho está organizado como a seguir: no capítulo 2, são apresentados os conceitos básicos sobre Sistemas de Notificação, Ambientes de Configuração e a maneira como podem ser usados em conjunto a fim de conseguirmos configurar os componentes do Sistema de Notificação. Apresentamos ainda, as características de duas funcionalidades que vêm sendo gradativamente disponibilizadas no Sistemas de Notificação, são elas : composição de eventos e suporte à mobilidade. No capítulo 3, é apresentado o modelo proposto neste trabalho baseando-se nos conceitos apresentados anteriormente. No capítulo 4, são apresentadas as características de um protótipo de implementação do modelo proposto. No capítulo 5, é apresentada uma avaliação qualitativa do modelo através de comparações entre uma aplicação desenvolvida sobre a implementação do modelo Sinapse e uma aplicação desenvolvida sobre o modelo Rebeca original. No capítulo 6, são apresentados os trabalhos relacionados. E finalmente, o capítulo 7 apresenta a conclusão deste trabalho.

2 Conceitos Básicos

Os Sistemas de Notificação implementam o paradigma *Publish/Subscribe* e disponibilizam a comunicação anônima e assíncrona entre os participantes. Os Ambientes de Configuração permitem configurar os componentes de uma aplicação. Estaremos utilizando o ambiente de configuração AC [Lis2003] que implementa o padrão de projeto *Interceptor* e permite controlar de forma flexível as interconexões entre os módulos de uma aplicação.

2.1 Sistemas de Notificação

Nesta seção, veremos o desmembramento da arquitetura dos Sistemas de Notificação em seus componentes elementares, conduzindo ao entendimento tanto das funcionalidades quanto das responsabilidades de cada componente.

De acordo com [HJ1999], podem ser identificadas diferenças na forma de disseminação de informação entre os sistemas baseados em eventos e os baseados em tecnologia *push*, também chamados de Sistemas *Push*. Nos Sistemas *Push* os produtores difundem a informação através de canais determinados, e os consumidores interessados naquele tipo de informação devem se associar a esses canais. Por outro lado, os sistemas baseados em eventos não usam canais para distribuir os eventos, a entrega de eventos é baseada nas assinaturas dos consumidores. Neste trabalho nosso foco foi o estudo de Sistemas de Notificação baseados em eventos, que serão apresentados em maiores detalhes nas próximas seções.

A estrutura interna desses componentes, entretanto, só será apresentada nos C*apítulos 3* e *4*, onde serão tratados, respectivamente, o modelo proposto e a implementação desse modelo. Apresentamos aqui uma visão geral que permite compreender as inter-relações entre os componentes constituintes dos Sistemas de Notificação.

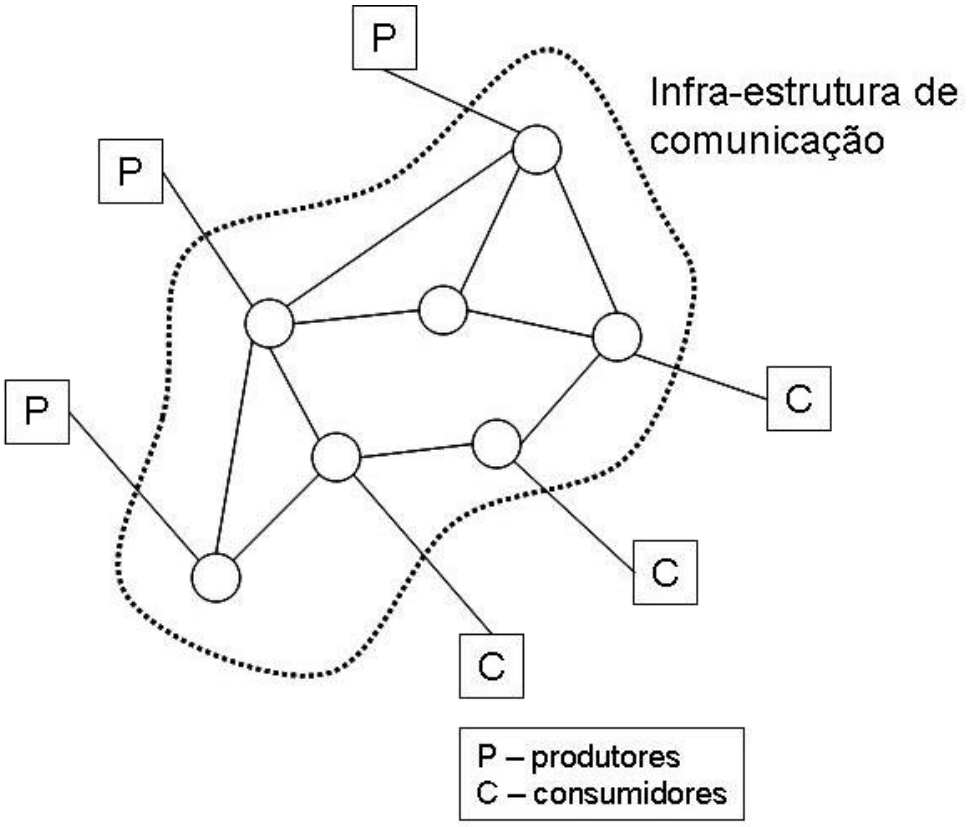


Figura 1 – Exemplo de Sistema de Notificação.

Podemos ver na figura 1, um Sistema de Notificação, relacionando através de uma infra-estrutura de comunicação genérica, produtores e consumidores representados, respectivamente por, "P" e "C". Essa infra-estrutura é formada por componentes especializados, representados na figura por círculos, que formam uma rede permitindo que produtores e consumidores se registrem para trocar informações.

2.1.1 O Paradigma *Publish/Subscribe*

Este paradigma representa uma relação produtor–consumidor, onde determinados módulos produzem um tipo de informação enquanto outros módulos consomem a informação produzida. Através de uma infra-estrutura de comunicação, os módulos da aplicação, interagem de forma anônima e assíncrona. Esta interação entre os módulos acontece através de mecanismos de publicação e assinatura. Por exemplo, aplicações de controle automatizado

de estoque podem registrar interesse em receber informações sobre vendas de produtos e controlar a quantidade em estoque.

Por outro lado, aplicações que demandam uma consulta direta, necessitam de uma base de informação para que possam realizar suas funções. Esse tipo de aplicação não pode tirar proveito do estilo de comunicação utilizado no *Publish/Subscribe*. Aplicações de venda on-line são exemplos desse tipo, onde, em determinado momento, é necessário consultar uma base que informe a quantidade disponível de cada produto comercializado.

No entanto, as informações manipuladas por aplicações que não são baseadas em *Publish/Subscribe* podem gerar eventos para um outro tipo de aplicação que faça monitoramento. Por exemplo, em uma aplicação de controle de vendas, o registro de um item como vendido pode gerar um evento para o estoque. Um módulo de controle de estoque assinaria eventos de venda com o intuito de controlar as baixas no estoque o que poderia levar à criação de uma ordem de compra de um determinado produto.

2.1.2 Eventos

A troca de informação pode ser considerada o ponto central dos Sistemas de Notificação. No entanto, essa informação precisa ser disponibilizada no Sistema de Notificação em algum formato bem definido para que sejam aplicadas operações lógicas sobre seu conteúdo.

Evento é o nome dado ao formato estruturado no qual a informação é disponibilizada nos Sistemas de Notificação. Classificam-se de diferentes formas, e um estudo sobre a sua classificação completa está além do escopo deste trabalho. Podem ser entendidos como tipos abstratos de dados [PZ1996], apresentando um tipo determinado (independentemente dos tipos de suas variáveis), ou como um conjunto de variáveis de tipos definidos, porém sem um tipo em si. Esta representação como um conjunto de variáveis acontece da seguinte forma: cada variável inserida no conjunto tem um nome, tipo e valor. Entretanto, o conjunto como um todo não tem um tipo específico que proporcione uma diferenciação entre eventos distintos.

A seguir, a figura 2 apresenta uma hierarquia de eventos tipados e a sua comparação com um conjunto de eventos não tipados. Através da herança, os eventos tipados podem utilizar atributos definidos nas classes superiores da hierarquia. Um consumidor que expresse interesse em um tipo superior recebe eventos que herdem daquele tipo. Em contraste, os eventos não tipados que precisem de atributos de outro evento precisam redefini-los.

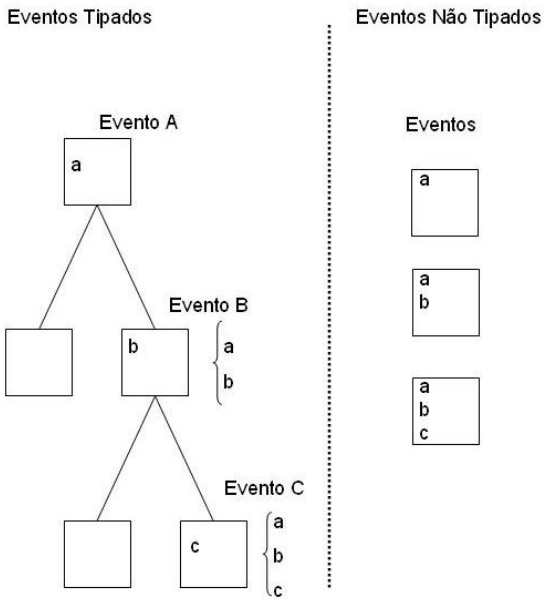


Figura 2 – Comparação entre eventos tipados e não-tipados.

De acordo com os eventos que são disponibilizados pelo Sistema de Notificação, as operações lógicas empregadas para identificar o interesse dos consumidores podem ocorrer por *tópico*, usando nesse caso a verificação do tipo do evento; por *conteúdo*, utilizando as informações contidas no evento; ou ainda de *forma mista*, utilizando ambas as técnicas.

Podemos utilizar os tipos dos eventos definidos pelos usuários para efeito das operações lógicas. Essa facilidade é vantajosa pois possibilita descartar eventos de forma imediata caso o seu tipo não seja compatível com o de interesse do consumidor.

A seguir, apresentamos na *Tabela 1* dois exemplos de código de eventos em contraste. Em (a), um exemplo de evento do Sistema de Notificação Siena [CRW2000], onde os eventos não são tipados, e em (b), um exemplo de evento do Sistema de Notificação Rebeca [Fie2003a], onde os eventos são tipados.

Tabela 1 – Estrutura de eventos em Siena[CRW2000] e Rebeca[Fie2003a]

(a) Ambiente Siena [CRW2000], evento não tipado:	(b) Ambiente Rebeca [Fie2003a], eventos criam uma hierarquia de tipos:

`Notification e =`
`new Notification();`
`e.putAttribute("name, "Siena");`
`e.putAttribute("version", 2);`
`e.putAttribute("lang", "Java");`
	`public class ExEvt extends Event{`
	`ObjectId _id;`
	`String name;`
	`public ExampleEvent(){`
	`this(null);`
	`}`
	`public ExampleEvent(String n){`
	`this.name = n;`
	`_id = new ObjectId();`
	`}`
	`}`

Como podemos ver na *Tabela 1*, um evento não tipado não pode ser distinguido de uma forma global, ou seja, todos os eventos serão da classe *Notification*, diferindo-se apenas em função de suas variáveis internas. Por outro lado, os eventos tipados podem ser distinguidos globalmente.

2.1.3 Filtros

Filtros são conjuntos de operações lógicas que serão aplicadas sobre os eventos a fim de rastrear o interesse de um consumidor. Eles operam sobre os eventos identificando uma relação de cobertura, ou seja, analisando se um dado evento satisfaz às condições lógicas para que um dado consumidor seja notificado. Podemos aplicar as operações lógicas sobre o

tipo do evento, sobre seu conteúdo ou ainda de alguma forma mista. Para que esses procedimentos estejam disponíveis, é necessário que o Sistema de Notificação disponibilize eventos tipados. De acordo com [Fie2003a], a forma mais flexível é a filtragem por conteúdo, uma vez que podemos especificar as operações lógicas em termos dos campos disponíveis no evento. Portanto, um único filtro pode especificar operações sobre cada um dos campos, atingindo assim um nível máximo de detalhamento.

Na figura 3, é apresentado um exemplo de entrega de evento para múltiplos consumidores. Vemos que o interesse pelo evento **E1** é expresso pelos consumidores através do filtro **F1**. Esse filtro é construído por operações lógicas que serão utilizadas para identificar a ocorrência do evento. Os mesmos filtros são ainda utilizados pelos componentes da infra-estrutura para criar as rotas de entrega. Esses componentes encarregados do roteamento estão representados na figura como os círculos numerados de 1 a 5.

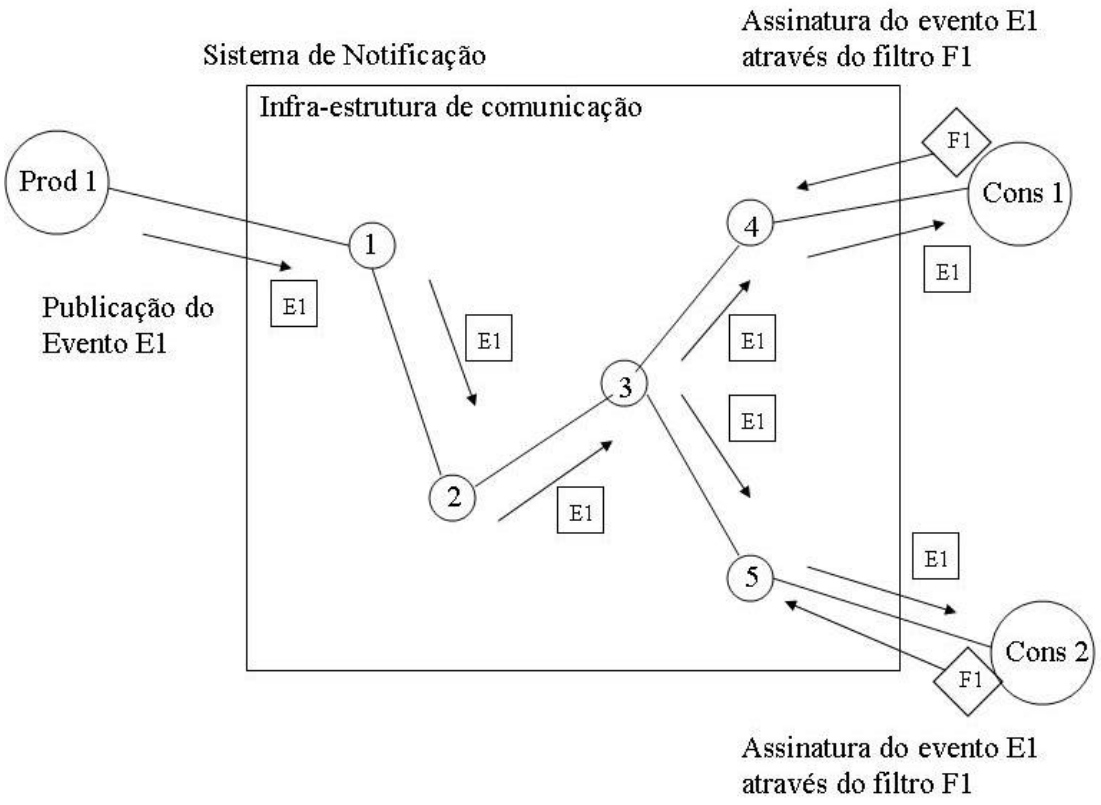


Figura 3 – Representação de um Sistema de Notificação com a entrega de um Evento a múltiplos consumidores.

A seguir, temos um exemplo do código usado para criar um filtro do ambiente Siena[1] [CRW2000]:

```
Filter f = new Filter();
f.addConstraint("name", "Siena"); // name = "Siena"
f.addConstraint("version", Op.GT, 1);// version > 1
```

No exemplo verificamos o mecanismo pelo qual os consumidores expressam seu interesse por eventos. No ambiente Siena [CRW2000], as operações lógicas constituintes dos filtros podem ser aplicadas ao conteúdo do evento. Assim, o filtro expressa quais são as restrições que devem ser cumpridas para que o evento seja de interesse daquele consumidor. Considerando o filtro apresentado no exemplo, devemos seguir o seguinte fluxo de modo a identificar seu interesse:

i. Determinar se o evento tem todas as variáveis que foram definidas: isso é importante porque, se o evento não tiver uma das variáveis, ele não tem como satisfazer às condições do filtro.

ii. Uma vez identificadas as variáveis necessárias, o filtro é aplicado ao evento de sorte a identificar se os valores são satisfatórios para as restrições. Observe que a relação que existe entre as restrições de um filtro é uma relação de "E Lógico". Se a *variável 1* do evento satisfizer à *condição 1* do filtro ^ a *variável 2* do evento satisfizer à *condição 2* do filtro ^ ... ^ a *variável n* do evento satisfizer à *condição n* do filtro, então, o evento satisfará o filtro. No exemplo apresentado, a variável "name" deve ter o valor "Siena" e a variável "version" deve ser maior que 1. Assim, os eventos

[1] Foram estudados durante esse trabalho quatro Sistemas de Notificação distintos que serão apresentados em maiores detalhes mais adiante. Devido a simplicidade de representação de filtros no Sistema Siena [CRW2000], este foi escolhido para exemplificar a criação de um filtro.

["Siena", 1], ["Teste", 1] e ["Teste", 2] não serão aceitos enquanto os eventos ["Siena", 2] e ["Siena", 3] serão aceitos.

Como uma última observação, devemos ressaltar que a utilização de filtros não modifica o conteúdo dos eventos, apenas identifica se um determinado evento satisfaz o filtro e portanto deve ser entregue aos consumidores que tenham usado aquele filtro. Caso um consumidor deseje receber qualquer tipo de evento deve fazer uso de filtros especiais, que aceitem eventos sem restrição. Em Rebeca [Fie2003a], este tipo de filtro, é fornecido como `TrueFilter`, e retorna verdadeiro para qualquer evento filtrado. Caso um consumidor utilize esse filtro, receberá todos os eventos publicados por todos os produtores.

2.1.4 *Brokers*, Tabelas de Rotas e Tabelas de *Call Back*

Conforme apresentado no *Capítulo 1*, os Sistemas de Notificação têm se apresentado como boa solução para diminuir a complexidade envolvida no desenvolvimento de sistemas distribuídos. Considerando a aplicação de Sistemas de Notificação ao desenvolvimento de sistemas distribuídos, consideramos que o ambiente de execução será composto por um conjunto de máquinas interligadas através de uma rede. Assim, o Sistema de Notificação utilizado deverá prover recursos para permitir que produtores e consumidores que estejam executando em máquinas distintas possam trocar informações através da infra-estrutura de comunicação. Os Sistemas de Notificação contam com componentes especializados, capazes de se conectarem entre si estabelecendo a infra-estrutura de comunicação. Para que a informação atinja os pontos de destino onde os consumidores estão conectados devem ser estabelecidas rotas de entrega de eventos. Assim, os filtros criados pelos consumidores são utilizados por esses componentes para manter e atualizar as tabelas internas, permitindo a entrega de eventos. Estes componentes são chamados *brokers*.

Como apresentado na figura 1, os *brokers* formam a infra-estrutura de comunicação, agindo também como pontos de acesso para produtores e consumidores. São especializados na criação de caminhos para a entrega de eventos aos consumidores, denominados por *rotas de*

entrega. Quando um *broker* recebe um evento, precisa determinar quais as rotas de entrega às quais este evento deve ser encaminhado. Para a devida escolha de rota, os *brokers* aplicam os filtros aos eventos. Um *broker* normalmente está conectado a vários outros, constituindo assim uma rede lógica, também chamada rede *overlay* [Eug2003, PB2002, Fie2003a].

Em ambientes com suporte à mobilidade (que serão discutidos detidamente na *seção 2.3*), distinguem-se dois tipos de *brokers*, a saber: internos e de borda [Fie2003a]. Os *brokers* de borda funcionam como pontos de acesso para produtores e consumidores, enquanto os internos conectam-se apenas com outros *brokers*. Esta classificação é unicamente de caráter lógico, uma vez que o código dos *brokers* é idêntico em ambos os casos. Não podemos assumir que uma dada topologia inicial permanecerá imutável durante o ciclo de vida da aplicação. Ou seja, deve ser possível para um *broker* realizar as tarefas necessárias para atender a ambos os papéis. Assim, um dado *broker*, que inicialmente desempenhava um papel interno, pode vir a receber conexões de módulos produtores e/ou consumidores e deve, para tanto, ser capaz de lidar com essa situação, executando funções como um *broker* de borda. Essa possibilidade de mudar de papel é importante para o suporte da mobilidade dos módulos produtores e consumidores que podem se mover de um ponto a outro.

Em um ambiente distribuído, as rotas de entrega de eventos são estabelecidas sobre as interconexões entre os *brokers*. Para que um *broker* decida se um evento deve ou não ser enviado a um vizinho, ele consulta uma tabela que mantêm as informações sobre quais filtros estão associados a cada vizinho. Dessa maneira, quando um evento satisfaz um filtro, o *broker* vizinho associado àquele filtro é eleito como um dos caminhos a receber a propagação do evento. A tabela que mantém esta associação entre *brokers* vizinhos e filtros é chamada *tabela de rotas*.

Um evento é propagado pela infra-estrutura de comunicação até que atinja um determinado *broker* de borda no qual existam um ou mais consumidores registrados para receber aquele

evento. Neste ponto, o evento torna-se apto a ser consumido e para tanto dois modelos podem ser usados pelo consumidor:

i. Modelo *push*: neste modelo o *broker* precisa notificar o consumidor da chegada do evento. Este procedimento é chamado *callback*, pois quando o consumidor se registra junto ao *broker*, uma referência ao consumidor é mantida no *broker* a fim de liberar o consumidor da tarefa de verificar a chegada de eventos. Assim, feito o registro, o *broker* já pode realizar uma *chamada de volta* ao consumidor para entregar o evento.

ii. Modelo *pull*: neste modelo o *broker* mantém o evento armazenado até que o consumidor execute uma chamada para recuperar os eventos de seu interesse. Neste modelo não é necessário que o *broker* mantenha uma referência ao consumidor pois a comunicação é iniciada a partir do próprio consumidor.

Nos Sistemas de Notificação estudados o modelo predominante para comunicação foi o *push*. De modo a viabilizar a chamada *callback* na qual o modelo *push* se baseia, deve existir uma referência para cada consumidor que está registrado no *broker*. Essas referências são armazenadas em uma tabela, e ficam associadas aos filtros, de maneira que quando um evento satisfaz um filtro, a referência ao consumidor é identificada. Esta tabela é chamada *tabela de call back*.

Considerando que a infra-estrutura de comunicação pode passar por situações de indisponibilidade dos *brokers*, é importante delinear uma estratégia para tolerância a falhas. Uma alternativa simples e direta para contornar problemas, é modificar a tabela de rotas dos *brokers* de forma a incluir, para cada endereço de *broker* vizinho, um segundo *broker* alternativo. Assim, surgindo um problema de conectividade, o *broker* alternativo pode ser utilizado para completar a rota de entrega. Esta solução baseia-se na própria tabela de rotas, o que facilita a implementação. Embora seja um bom exemplo para a compreensão do

problema de tratamento de falha de conectividade, esta solução usa uma estratégia muito limitada, não permitindo alterar as conexões dinamicamente. Se um *broker* alternativo também apresentar problemas, voltamos à situação inicial.

Uma outra opção é a reconfiguração dinâmica, que consiste em identificar um *broker* alternativo durante a execução. Essa identificação é feita por uma entidade que conheça a disposição dos *brokers* a fim de identificar rotas alternativas. Assim, se um *broker* estiver inacessível, a infra-estrutura de comunicação do Sistema de Notificação será capaz de ser configurada por uma entidade externa, evitando uma ruptura em duas ou mais redes.

Embora ambas opções sejam equivalentes, podemos considerar a reconfiguração dinâmica mais vantajosa por permitir escolher as rotas alternativas de acordo com as rotas disponíveis.

2.1.5 Funcionamento Básico

Através da infra-estrutura de comunicação dos Sistemas de Notificação os módulos da aplicação distribuída vão poder trocar informações no formato de eventos. Devemos ressaltar que todas as ações em Sistemas de Notificação têm como foco principal a informação que deve ser trocada.

Os módulos da aplicação indicam o papel que estarão desempenhando junto do Sistema de Notificação, seja de produtor ou consumidor. A partir daí passam a ter disponíveis para uso métodos de publicação e assinatura de eventos respectivamente. Vale notar que um mesmo módulo pode se apresentar como produtor de um determinado tipo de informação e como consumidor de uma outra.

Durante a execução da aplicação existem momentos em que os módulos atuam isoladamente e momentos em que a comunicação com outros módulos é necessária. No caso em que estão atuando de maneira isolada, os módulos estão realizando algum processamento interno independente de outros módulos da aplicação. No caso em que é necessário interagir com

outros módulos, o Sistema de Notificação entra em ação, proporcionando uma interação transparente. Na figura 4, temos um diagrama de seqüência que mostra uma interação entre um módulo produtor e um módulo consumidor. Para manter a simplicidade e clareza, vamos tomar uma situação onde uma dada informação é produzida, disponibilizada no Sistema de Notificação, transportada até o consumidor e finalmente entregue através de uma chamada *callback*.

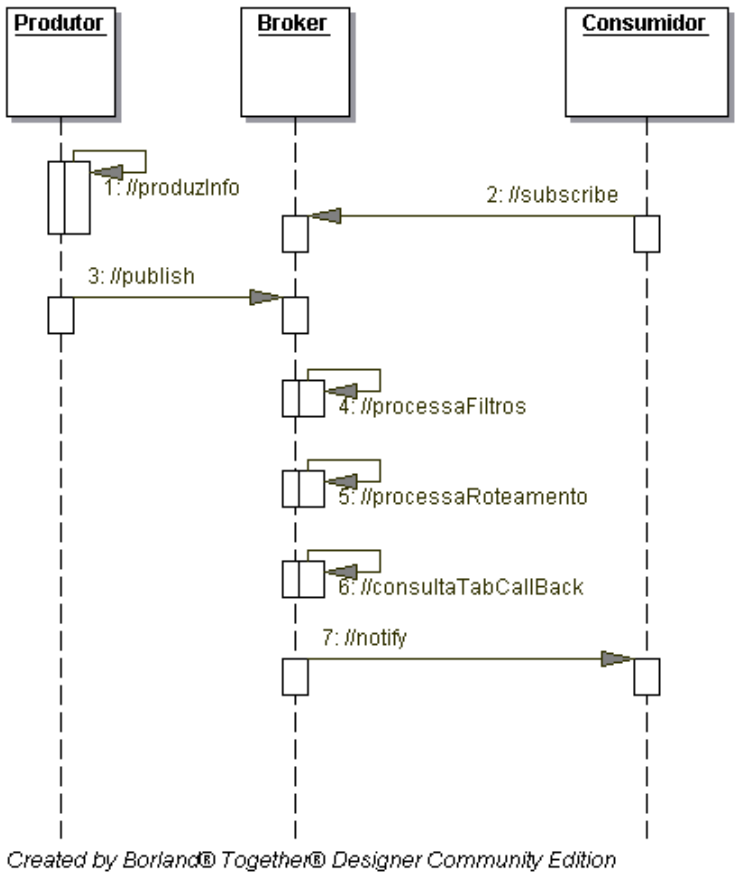


Figura 4 – Diagrama de seqüência de uma interação em um Sistema de Notificação

2.1.6 Sistemas de Notificação Estudados

Contamos com diferentes implementações de Sistemas de Notificação e podemos classificá-los de diferentes formas. Nosso ponto de interesse está, tal como em [CCW2003], na disponibilidade do software, no seu suporte à mobilidade e o suporte, ou possibilidade de extensão, para lidar com composição de eventos.

A seguir a *tabela 2* apresenta os Sistemas de Notificação que foram estudados neste trabalho. Estes sistemas foram escolhidos por serem propostas atuais e contarem com algumas das características necessárias para o desenvolvimento de nossa proposta.

Tabela 2 – Sistemas de Notificação estudados neste trabalho

Sistema	Código disponível	Suporte à mobilidade	Suporte à composição de eventos
Hermes	Não	Não	Externo
LimeLite	Não	Sim	Indiretamente
Siena	Sim	Sim	Sim
Rebeca	Sim	Sim	Sim

2.1.7 Hermes

De acordo com [Pie2002], Hermes é um Sistema de Eventos, e como tal deve agregar suporte a verificação de tipos de invocações, controle de acesso e suporte a transações. O roteamento dos eventos é baseado em uma rede *overlay*, o que pode levar a uma menor taxa de falhas, pois a rede pode se reajustar de forma independente. Os eventos são filtrados e direcionados através da rota de uma forma distribuída.

Os filtros também são baseados em conteúdo, mas permitem tipagem de eventos. Para tanto, existem nós especiais que garantem a verificação de tipo. Essa verificação acontece de forma distribuída de maneira a não comprometer a escalabilidade do sistema.

O suporte a composição não está disponível na arquitetura, mas pode ser conseguido utilizando o CEA [PSB2003] (Cambridge Event Architecture), o qual teve suas pesquisas e desenvolvimento iniciais baseados em Hermes. O CEA permite utilizar a composição de eventos de forma isolada do Sistema de Notificação, mesmo que este não disponibilize suporte a eventos compostos. É introduzida uma camada de detecção de eventos compostos pela qual a aplicação interage com o Sistema de Notificação, além disso uma linguagem própria para a descrição das composições torna mais simples o desenvolvimento de aplicações que utilizam esse recurso.

2.1.8 LimeLite

LimeLite [PMR1999] é um Sistema de Coordenação com suporte para agentes móveis. Embora não se trate propriamente de um Sistema de Notificação, o suporte à mobilidade e a um espaço de tuplas distribuído, apresentam material de interesse para a discussão de suporte à mobilidade. Diferentemente dos Sistemas de Notificação, a mobilidade tratada em LimeLite está relacionada com a movimentação do código dos agentes. Por outro lado, a capacidade das máquinas clientes se moverem de um ponto a outro, influencia diretamente o suporte ao espaço de tuplas compartilhado. Esse espaço de tuplas permite que agentes interessados em trocar informações indiquem que uma tupla (ou cópia dela) seja deixada no espaço compartilhado mesmo que o agente se mova para outro ponto.

2.1.9 Siena

Siena é um Sistema de Notificação com sua versão mais recente portada para Java e utiliza uma arquitetura hierárquica para conexão dos *brokers*. Permite o roteamento de eventos simples e a criação de filtros e "padrões" que são a junção de vários filtros em uma única estrutura (implementado como um array). O uso de "padrões" permite notificar o consumidor somente quando todos os filtros que compõem o "padrão" sejam satisfeitos pelo mesmo evento. Com isso temos um suporte incipiente à composição de eventos, o cliente deve criar diversos filtros e depois inseri-los no padrão.

O suporte à mobilidade trata somente a mobilidade física, discutida na *seção 2.4.1*, e segue um modelo de primitivas de explícitas para desconexão e conexão semelhante ao utilizando no IP móvel. Podemos considerar isso como uma implementação experimental, mas que não modela a realidade uma vez que os consumidores não sabem de antemão que serão desconectados. Durante a desconexão, um proxy recebe os eventos que cabem àquele consumidor. Quando acontece a reconexão do consumidor em outro ponto de acesso, os eventos, filtros e padrões são encaminhados para a nova localização física.

2.1.10 Rebeca

Os autores de Rebeca [Fie2003a] apresentam uma discussão ampla sobre mobilidade, cobrindo diferentes aspectos conceituais. A proposta de Rebeca é utilizar um algoritmo capaz de recriar as rotas de entrega de eventos a partir da movimentação dos módulos externos. Assim, o ambiente se torna reativo à movimentação do módulos, sem que estes precisem indicar explicitamente que irão se desconectar. Em Siena[CRW2000], os módulos devem indicar que vão se desconectar do sistema através de uma chamada *move-out*. Quanto à composição de eventos, Rebeca apresenta uma implementação de suporte simplificada, que permite especificar operações lógicas entre os filtros de maneira semelhante à de Siena.

Rebeca é um Sistema de Notificação desenvolvido em Java, projetado de forma altamente modular. Os eventos têm tipo definido e o ambiente permite que sejam criados filtros baseados nesses tipos. No entanto, também está disponível a filtragem baseada em conteúdo. Além dos filtros simples, podem ser criados *multifiltros,* que são uma forma simplificada de suporte à composição de eventos. Quando usamos os multifiltros, estamos limitados a uma execução em que o evento satisfaça todos os filtros que fazem parte do conjunto, para ser considerado válido. Esta característica dificulta a representação de situações de composição de eventos reais.

A versão de Rebeca utilizada na implementação deste trabalho não contempla o suporte à mobilidade[2], uma vez que dentre os algoritmos de roteamento implementados no Rebeca, o utilizado no protótipo desenvolvido foi o Simple Routing, que baseia suas rotas nos filtros dos consumidores. Portanto, quando os consumidores fazem uma assinatura, os filtros são encaminhados através da infra-estrutura de comunicação a fim de permitir o roteamento dos eventos. O processo de armazenamento de eventos para posterior reenvio também não está disponível no *broker* padrão.

[2] O suporte à mobilidade do modelo Sinapse se dá através dos Conectores de Interação que serão explicados em detalhes na seção 3.3.1. A explicação do suporte à mobilidade no modelo Sinapse é apresentada na seção 3.4.

Neste trabalho, o Sistema de Notificação escolhido para adaptação ao ambiente de configuração foi o Rebeca [Fie2003a]. Esta escolha se deve a dois fatores: i)disponibilidade de código fonte e ii) possibilidade de extensão de eventos e filtros.

2.2 Composição de Eventos

A composição de eventos está relacionada com uma questão lógica, onde o interesse do consumidor está em receber informação semanticamente mais densa, originada a partir de informações mais simples. Para que seja possível trabalhar com informação mais densa, os consumidores devem processar *eventos simples* a fim de inferir informações compostas. No entanto, isto sobrecarrega o módulo consumidor, que precisa receber e processar um grande número de eventos até que seja possível satisfazer as regras de inferência.

O objetivo da composição é permitir a utilização de *metaeventos*, que carreguem um valor semântico maior do que os eventos utilizados na sua criação. Com isso os consumidores ficam liberados do processamento extra, necessário para deduzir a informação composta. Para tanto, devem ser disponibilizados mecanismos que permitam especificar, criar e reconhecer *metaeventos*.

De acordo com [PSB2003], o tratamento de composições de eventos foi inicialmente trabalhado na área de bancos de dados ativos. Nestes sistemas, o funcionamento é centralizado, assim, não existem problemas com queda de desempenho por causa do uso de rede. Além disso, a identificação de composições é mais simples por haver uma centralização do processamento, ou seja, todas as informações necessárias estão disponíveis em uma memória comum. A adaptação da composição de eventos para sistemas distribuídos trouxe a tona as questões relativas a como lidar com a descrição e interpretação das composições em um ambiente não centralizado. Isto porque não temos todas as informações relativas aos consumidores disponíveis todo o tempo.

A composição de eventos é um assunto ainda em pesquisa. Em Siena [CRW2000], a composição é representada por um vetor de filtros, já em Rebeca [Fie2003a], podem ser

criados filtros contendo um vetor de filtros, e operações lógicas podem ser especificadas de maneira ainda muito simples. Em Hermes [PSB2003], é tratada à parte do Sistema de Notificação, a partir de módulos que avaliam os eventos para identificar se satisfazem alguma composição.

Podem ser identificadas duas estratégias para disponibilizar a composição de eventos: a primeira, integrada ao Sistema de Notificação, e a segunda, isolada. Ambas têm características de implementação distintas. Na abordagem integrada, o Sistema de Notificação conhece a estratégia de composição e interpretação usada, enquanto na abordagem isolada o processamento acontece de forma independente. O modelo Sinapse emprega a segunda estratégia e isola o tratamento de filtros e composições de eventos no Conector de Composição e Filtragem, discutido na *seção 3.3.2*.

2.3 Mobilidade em Sistemas de Notificação

As redes móveis têm recebido grande atenção há algum tempo, e o paradigma *Publish/Subscribe* parece se adaptar adequadamente a esse ambiente, facilitando o desenvolvimento de aplicações. Diversos estudos [CRW2000, PSB2003, Fie2003a] têm sido conduzidos para adaptar os Sistemas de Notificação existentes às redes móveis, de maneira que aplicações pré-existentes possam se beneficiar dessas redes. Assim, os módulos da aplicação, instalados em clientes móveis, estarão interagindo com o Sistema de Notificação da maneira como faziam no ambiente fixo, sem tomar conhecimento da mobilidade. O Sistema de Notificação, por outro lado, deve usar mecanismos de adaptação para suportar as redes móveis e prover o suporte necessário aos clientes.

A mobilidade abre espaço para um novo conjunto de aplicações que, utilizando essa característica, tiram proveito da informação de localização. Com isso, torna-se possível utilizar notificações baseadas em contexto de localidade. Um exemplo simples, seria um sistema de navegação para carros que possa buscar vagas em uma rua. Somente os locais de

estacionamento livres dentro de um certo limite de distância seriam considerados válidos. Nossa proposta não estará tratando este novo cenário de aplicações.

Redes móveis têm uma série de limitações, tais como, largura de banda restrita, conexões de baixa qualidade e perda inesperada de conexão por parte dos clientes. Neste último caso, um cliente móvel pode perder notificações ou deixar de publicá-las, semelhantemente a um caso de falha de máquina em uma rede fixa. No entanto, esforços estão sendo empregados no tratamento dessas situações. Por exemplo, em [Fie2003a, ZF2003], a idéia original do IP Móvel deu origem a uma forma de adaptação dinâmica que permite uma mudança de estação base, sem perda de notificações, e que garante a ordem de entrega das mesmas.

É interessante notar que os sistemas Siena [CRW2000] e Rebeca [Fie2003a] foram desenvolvidos considerando redes sem fio estruturadas. No contexto destas redes, os computadores móveis acessam a rede a partir de nós fixos bem definidos, chamados estações base. Os pontos de acesso usados pelos produtores e consumidores (móveis) dos Sistemas de Notificação citados, são mapeados a essas estações base aproveitando a estrutura da rede de comunicação. Isto facilita a adaptação dos Sistemas de Notificação ao contexto de mobilidade, considerando que restringe a complexidade das questões a serem trabalhadas. Em nosso trabalho consideramos o mesmo contexto de redes sem fio estruturadas utilizado nos sistemas Siena e Rebeca.

2.3.1 Classificação dos Tipos de Mobilidade

O conceito de mobilidade no contexto de Sistemas de Notificação, diz respeito à capacidade de um dispositivo (Produtor/Consumidor) passar por um período de desconexão onde não seja possível encontrá-lo. Posteriormente, esse dispositivo pode vir a se conectar em um ponto de identificação, possivelmente diferente do original. Considerando a arquitetura dos Sistemas de Notificação, podemos instalar os componentes de acesso nas estações base de uma rede infra-estruturada e distribuir componentes de roteamento por toda a rede fixa.

Desse modo os clientes móveis estabelecem a conexão de rede, a fim de que os módulos da aplicação possam publicar e assinar eventos junto ao Sistema de Notificação.

Em [Fie2003a], é apresentada uma abordagem para adaptar uma arquitetura projetada para rede fixa a um ambiente móvel. São levantadas as questões referentes a tipos de mobilidade que podem ser classificados em:

i. Mobilidade física: referindo-se à mobilidade de um cliente do ponto de vista de desconectar-se de uma estação base e reconectar-se a outra estação base. Como podemos ver na figura 5, o cliente sai da área coberta pelo ponto de acesso "a" e se locomove para área coberta pelo ponto de acesso "b".

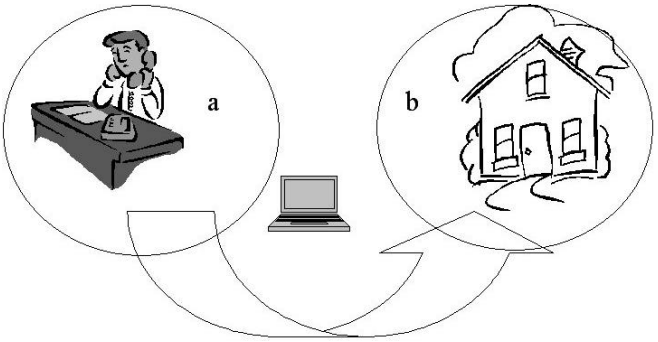


Figura 5 – Mobilidade física

ii. Mobilidade lógica: referindo-se a questão de contexto, onde o cliente não passa por uma desconexão de rede. Ao invés disso, o cliente muda de ambiente dentro de uma mesma área de cobertura, afetando o contexto em que a aplicação está executando e pode afetar o conjunto de assinaturas. Como podemos ver na figura 6, o cliente não sai da área de cobertura da estação base a qual está conectado. Ao invés disso ele troca de contexto quando sai de uma sala e passa para a outra.

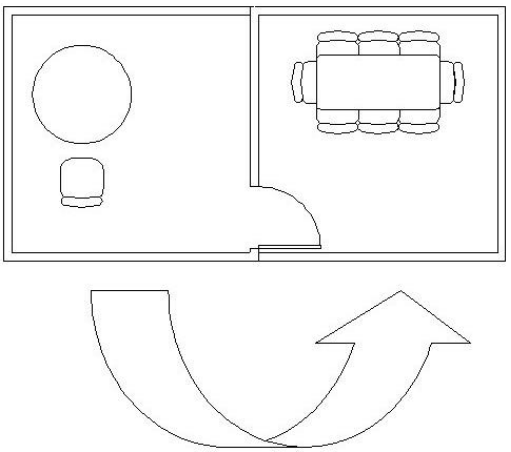


Figura 6 – Mobilidade lógica

A importância desta classificação dos tipos de mobilidade, está relacionada a separação do tipos de problemas que podemos encontrar em cada caso. Quando tratamos a mobilidade física, encontramos principalmente problemas de perda, atraso e entrega desordenada de notificações. Por outro lado, a mobilidade lógica dificulta prover suporte à composição de eventos por estar relacionada a *context-aware* [Dey2001], e isso interfere no controle de assinaturas e filtros.

A proposta para mobilidade física apresentada em Rebeca [Fie2003a], suporta a conexão/desconexão de clientes de forma automática e transparente, isso é feito a partir do armazenamento de eventos até que o consumidor se torne novamente acessível. Esse armazenamento deve ser feito em *buffers* localizados nos *brokers*. A estratégia proposta em [Fie2003a], para o suporte a mobilidade lógica, consiste em calcular com base em um grafo, os possíveis caminhos que podem ser tomados por um dado cliente móvel. Dessa maneira, são enviadas cópias de eventos correspondentes aos possíveis contextos que o cliente pode vir a ter. Isso impede que o cliente perca notificações. A perda seria causada pelo atraso correspondente à reorganização que o *middleware* precisaria fazer em suas rotas em função da mudança de contexto do cliente.

2.4 Ambientes de Configuração

Os Ambientes de Configuração utilizam o paradigma de Arquitetura de Software para permitir a descrição de um sistema em termos de seus componentes e conexões, permitindo fazer uma distinção das características funcionais e não-funcionais do aplicativo [Loq2004]. Entendemos por características funcionais tudo aquilo que está relacionado com o problema que o aplicativo se dispõe a resolver. As características não-funcionais estão relacionadas a todo o suporte e infra-estrutura necessários para que o aplicativo atinja o seu objetivo.

Essa separação entre funcional e não-funcional, dá margem a que o desenvolvimento do aplicativo seja feito de modo mais concentrado, e que as soluções de suporte e infra-estrutura sejam reutilizadas mais facilmente.

Os Ambientes de Configuração disponibilizam maneiras de descrever os componentes do aplicativo e a forma como estes se interconectam. Isso é feito por meio de uma linguagem de descrição de arquiteturas. Adicionalmente, os Ambientes de Configuração disponibilizam um modelo de componentes, que possibilita descrever os componentes do aplicativo.

2.4.1 Arquitetura de Ambientes de Configuração

Nas próximas seções, veremos quais são os elementos básicos de um Ambiente de Configuração, bem como a forma de atuação de cada um, para que seja possível descrever a aplicação em termos de seus componentes e suas interconexões. Apresentaremos, ainda, em maiores detalhes, como são representadas as características funcionais e não-funcionais da aplicação.

2.4.2 Linguagem de Descrição de Arquitetura

É conveniente que exista uma linguagem própria e bem definida para descrever os componentes da aplicação e da topologia de suas interconexões. Essa tipo de linguagem já foi extensamente estudado [Med1996, Szt1999] e uma classificação mais profunda está além do escopo deste trabalho.

A função de uma linguagem de descrição de arquitetura é permitir que os componentes da aplicação sejam descritos junto com sua topologia, para que seja possível carregar a aplicação no ambiente. Está descrição, acrescida de um módulo Configurador, a ser apresentado mais adiante, substitui um módulo central da aplicação que estaria carregando e interconectando os diversos componentes.

2.4.3 Componentes de uma Aplicação baseada em Configuração

Uma aplicação que é desenvolvida baseada em um Ambiente de Configuração tem módulos que representarão cada aspecto da aplicação.

Estes componentes são:

i. módulos: usados para representar os aspectos funcionais da aplicação

ii. conectores: utilizados para representar os aspectos não-funcionais da aplicação e

iii. portas: que representam os pontos utilizados para conectar módulos e conectores.

É importante ressaltar que os módulos da aplicação passam a ser desenvolvidos com essa modelagem em mente. Com isso temos um ganho imediato na organização dos aspectos da aplicação, que passam a ser classificados quanto a sua relação com o domínio da aplicação.

2.4.4 Módulo Configurador

A principal função desse módulo é carregar os componentes de aplicações especificados na descrição da aplicação. Além disso, também permite a interação com a topologia dos componentes da aplicação, disponibilizando funções que permitem realizar intervenções na conexões entre os componentes.

O módulo Configurador age como um módulo gerente da aplicação. Realiza o carregamento e a ligação entre os módulos e conectores que juntos representam todos os aspectos da aplicação, tanto funcionais quanto não-funcionais. Tendo explicitado essa questão, podemos fazer uma comparação entre uma aplicação tradicional com um módulo central e uma aplicação baseada em um Ambiente de Configuração, ressaltando a importância do módulo Configurador, bem como da descrição arquitetural.

Em uma aplicação comum utilizamos um módulo central responsável por carregar os outros módulos e por iniciar as estruturas que serão utilizadas para manipulação de dados. Dessa forma, quando precisamos incluir novas funcionalidades ou modificar as existentes temos que verificar o módulo central para identificar possíveis questões de acoplamento.

Em contraste a essa situação, quando usamos um Ambiente de Configuração podemos modificar a aplicação de forma dinâmica e organizada, diminuindo o impacto necessário para manutenção e evolução das aplicações. Isto porque, conforme apresentado, podemos alterar as ligações entre módulos e conectores, o que nos permite alterar os caminhos de processamento, incluindo novas etapas ou modificando os destinatários.

2.4.5 Funcionamento Básico

Quando utilizamos um Ambiente de Configuração no desenvolvimento de um aplicativo, devemos seguir um raciocínio de separar as atividades funcionais das não-funcionais. Essa divisão está relacionada com as tarefas de cada módulo, ou seja, se for necessário aplicar algum tipo de conversão de dados entre um cliente e um servidor, essa é considerada uma tarefa não-funcional. No entanto, os processamentos realizados tanto pelo cliente quanto pelo servidor são considerados funcionais por estarem diretamente relacionados ao objetivo final do sistema.

Na figura 7, podemos ver um cliente e um servidor interagindo através de uma chamada. Essa chamada é interceptada por um conector que pode realizar algum tipo de processamento

sobre os dados tal como aplicar uma conversão nos dados enviados ao servidor. A lógica para converter os dados, considerada uma tarefa operacional, não fica misturada ao código do cliente nem do servidor.

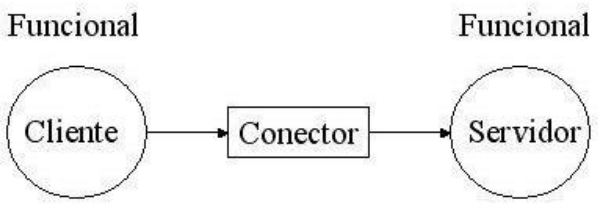


Figura 7 – Módulos funcionais e a interceptação por um conector que realiza uma tarefa não funcional

Podemos perceber que o aplicativo não conta com um módulo central responsável por carregar e ligar os módulos. Está tarefa passa a ser responsabilidade do Ambiente de Configuração. Quando ocorre o carregamento da aplicação, o ambiente carrega os componentes e realiza a interconexão, obedecendo as descrições. A partir desse momento, a aplicação começa a funcionar da mesma forma que ocorreria caso fosse usado um módulo central.

Uma outra característica importante dos Ambientes de Configuração é a capacidade de interagir com a arquitetura da aplicação de forma dinâmica em tempo de execução. Isto torna possível mudar a seqüência de conectores, ou mudar as ligações entre módulos e conectores através de primitivas disponibilizadas pelo ambiente.

2.4.6 Ambiente de Configuração – AC

Este ambiente [Car2001, Lis2003], implementa o padrão de projetos *Interceptor*, permitindo a configuração de aplicações através de uma linguagem simples de descrição de arquitetura, com isso, o aplicativo passa a ser carregado e interligado pelo ambiente. Como as ligações entre os componentes são feitas pelo ambiente, podem ser introduzidos conectores entre essas ligações.

O ambiente AC foi desenvolvido em Java e está baseado nas funcionalidades de RMI para distribuição, contando com três módulos básicos:

i. Coordinator: responsável por coordenar as interações entre todos os módulos *Executor* que estiverem fazendo parte do sistema.

ii. Executor: responsável por instanciar os componentes da aplicação em uma determinada máquina, podem haver *n* módulos *Executor* dando suporte a uma aplicação.

iii. CommandLine: console de comandos utilizado pelo administrador da aplicação para carregar o arquivo de descrição. Permite iniciar a execução, pará-la, interagir com a topologia dos componentes.

São disponibilizadas estruturas de dados que permitem representar uma aplicação descrevendo seus componentes em função de suas características arquiteturais. Assim, temos como representar Portas, Módulos, Conectores e Instâncias e descrever as interações entre eles através de *links*, que são especificados no arquivo descritor da aplicação.

Nesse trabalho propomos adaptar o Sistema de Notificação Rebeca, modelando seus componentes em conectores de uma descrição arquitetural, de modo que seja possível intervir nas conexões desses componentes. A importância do uso de um ambiente de configuração está na facilidade que estes ambientes fornecem recursos para realizar esta intervenção sobre os componentes. Sem o uso de um ambiente de configuração, todas as intervenções teriam que ser feitas de forma manual.

Uma característica importante do ambiente AC é disponibilizar a capacidade de interceptação de métodos, o que permite, a partir de uma chamada de método específica, transferir o controle para outro módulo ou conector. Essa capacidade é fundamental para nossa proposta, pois é a base para o funcionamento dos conectores especializados nos quais os

componentes do Sistema de Notificação Rebeca são mapeados. O funcionamento de cada conector de nossa proposta será discutido em detalhes nos *Capítulos 3* e *4*.

2.5 Aplicando os Recursos de Configuração a um Sistema de Notificação

Embora os Sistemas de Notificação tenham se tornado cada vez mais fáceis de usar, ainda é necessário lidar com todos os aspectos relacionados à utilização dos recursos de publicação e assinatura. Além disso, é interessante separar os aspectos relacionados às interações entre a aplicação distribuída e o Sistema de Notificação. Isso pode ser feito através de mecanismos de interceptação de métodos como o encontrado no Ambientes de Configuração AC.

Dessa maneira, o aplicativo pode ser desenvolvido utilizando uma visão simples de chamadas a métodos. Assim, cada módulo produtor só se preocupa com as suas próprias mudanças de estado, e estas são interceptadas por conectores especializados de modo a publicar as informações. Já os módulos consumidores recebem o resultado do processamento de assinaturas em uma estrutura de dados simples, tal como uma lista ou uma tabela *hash*, em um retorno de método. Quando eventos são recebidos através de chamadas *callback*, o conector os processa e os armazena em uma estrutura de dados. Quando o módulo consumidor do aplicativo executa seu método de consulta, esta estrutura é disponibilizada com o resultado mais recente de interação com o Sistema de Notificação.

Assim, podemos identificar, até aqui, dois tipos de conectores independentes entre si:

i. conector para interceptar as chamadas de método para mudanças de estado dos módulos da aplicação, responsável por permitir a interação com o Sistema de Notificação em si.

ii. conector responsável por tratar as composições e filtragem de eventos.

Os *brokers* (ver *seção 2.1.4*) são responsáveis por criar a rede de entrega de eventos e permitir a conexão dos módulos externos ao Sistema de Notificação. Esses componentes

representam uma característica não-funcional da aplicação, sendo dedicados a criar interconexões de rede. O mapeamento de *brokers* em conectores será explicado em detalhes nos *Capítulos 3 e 4*, referentes à descrição do modelo proposto e de sua implementação. Este mapeamento acrescenta mais um conector ao modelo, o Conector de Roteamento.

A seguir, na figura 8, apresentamos uma visão de alto nível de como funciona a lógica de interceptação do Ambiente de Configuração AC. Em "a", temos uma chamada direta entre um cliente e um servidor, em "b", usando o recurso de interceptação, podemos transferir o controle para um conector. Ao final do processamento do conector o controle segue para o servidor.

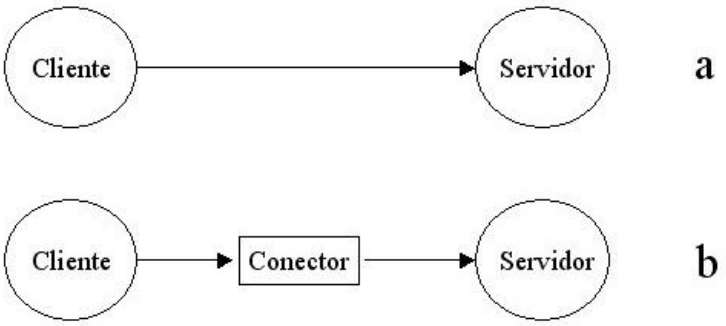


Figura 8 – Chamadas de métodos interceptadas

Utilizando essa facilidade, podemos interceptar chamadas de método do módulos externos, transferindo o controle para o conector responsável pela interação com o Sistema de Notificação. No caso da interceptação ter sido de um módulo produtor, o conector publica um evento; já no caso do módulo consumidor, o conector armazena os eventos recebidos, a fim de retorná-los oportunamente. É através desses conectores que a aplicação é ligada ao Sistema de Notificação.

Percebemos então que os Conectores de Interação, discutidos na *seção 3.3.1*, tornam mais simples o interfaceamento com a aplicação e com diferentes APIs de Sistemas de Notificação já existentes. Ou seja, além da comunicação entre módulos ser anônima, a própria interação

com a infra-estrutura de comunicação passa a ser encapsulada. Isto torna os módulos externos auto contidos em relação ao domínio da aplicação, sem mistura entre os aspectos funcionais e não-funcionais. Dessa forma, identificamos dois casos de aplicação para esses conectores:

i. O primeiro caso é o de uma aplicação que já utiliza um Sistema de Notificação existente, podemos utilizar os Conectores de Interação para adaptar essa aplicação ao modelo Sinapse. Pode-se introduzir os conectores entre os módulos e o modelo Sinapse respeitando a interface que a aplicação já utilizava, tornando assim, a migração para o modelo Sinapse mais simples não sendo necessário modificar os módulos existentes.

ii. O segundo caso é o de uma aplicação que não tem uma dependência prévia de um determinado Sistema de Notificação, pode-se considerar uma aplicação que esta sendo desenvolvida desde o início. Neste caso, os conectores podem ser usados para encapsular as interações com o modelo Sinapse, de forma que não exista na aplicação código para lidar com as interações com o modelo Sinapse. No entanto, é importante ressaltar que os módulos consumidores têm uma dependência de dados em relação ao Conector de Interação. Isto porque, mesmo que os eventos recebidos sejam processados pelo conector, deve existir uma interface comum compartilhada entre o consumidor e o conector para que a informação possa ser trocada.

2.6 Conclusão Parcial

Podemos notar que, tanto os Ambientes de Configuração, quanto os Sistemas de Notificação, permitem o desenvolvimento de aplicações de baixo acoplamento graças às suas respectivas características. Nos Sistemas de Notificação, temos a comunicação assíncrona e anônima, ocorrendo sem que os módulos participantes tenham conhecimento uns dos outros. Por outro lado, nos Ambientes de Configuração, o baixo acoplamento é conseguido graças à especificação das interconexões dos componentes baseados em um modelo descritivo da

topologia desses componentes. Desse modo, os componentes não se referenciam diretamente e são interconectados pelo módulo configurador do ambiente.

A inclusão da composição de eventos e o suporte à mobilidade permitem que os Sistemas de Notificação sejam utilizados de maneira mais eficiente e ampla. Um novo conjunto de aplicações pode ser atendido pelas funcionalidades disponibilizadas. Aplicações que precisem de informação com mais detalhes podem fazer uso da composição de eventos, não só para simplificar seu desenvolvimento, como também para reutilizar a lógica necessária para inferir esta informação. Além disso, passa a ser possível atender aplicações que precisam de suporte à mobilidade.

3 Modelo Proposto

O capítulo anterior apresentou os conceitos necessários para a elaboração do modelo que será proposto neste capítulo. Vamos agora aplicar esses conceitos de forma consolidada e desenvolver um modelo que permita intervenções em sua configuração. Serão introduzidos os conceitos apresentados, de modo a separar o tratamento de composição e filtragem de eventos dos *brokers*, e auxiliar o suporte à mobilidade a partir de mudanças na topologia da infra-estrutura de comunicação. Além disso, as interações da aplicação com o Sistema de Notificação passam a ser encapsuladas em conectores especializados.

3.1 Introdução

Nossa proposta é mapear as funcionalidades de um Sistema de Notificação em um modelo arquitetural, utilizando para isso conectores especializados, para em seguida, implementar um protótipo que permita avaliar qualitativamente o modelo. Para tanto, são necessários três tipos de conectores:

i. Conector de Interação: responsável por permitir a interação de módulos externos (produtores e consumidores) com o Sistema de Notificação.

ii. Conector de Composição e Filtragem: responsável por tratar e resolver as composições e filtros de eventos, isentando os *brokers* de aplicar filtros aos eventos. Quando uma composição tem todas as condições satisfeitas, o próprio Conector de Composição e Filtragem se encarrega de enviar o *metaevento* gerado.

iii. Conector de Roteamento: mapeamento das funções de roteamento dos *brokers*, permitindo o gerenciamento de mais alto nível.

São usados os seguintes mnemônicos para identificar os conectores do modelo Sinapse:

i. CI – Conector de Interação

ii. CCF – Conector de Composição e Filtragem

iii. CR – Conector de Roteamento

Os *brokers* usados em Sistemas de Notificação (ver *seção 2.1.4*) têm suas funções divididas entre Conectores de Roteamento e Conectores de Composição e Filtragem. Isto porque, no modelo tradicional usado no Rebeca [Fie2003a], os *brokers* são responsáveis pelo tratamento de rotas de entrega e processamento de filtros de eventos. Com essa divisão de funcionalidades em dois conectores obtemos maior flexibilidade para configurar os Conectores de Roteamento. Assim, é possível disponibilizar uma maneira de intervir na arquitetura para mudar as rotas de entrega, reagindo à movimentação dos módulos externos [Cas2003]. O processamento de filtros de eventos é delegado aos Conectores de Composição e Filtragem, através da interceptação que acontece no processamento dos Conectores de Roteamento.

Uma outra necessidade é conhecer os tipos de eventos que fazem parte da aplicação que será desenvolvida. Assim, podemos fornecer um meio de criação de filtros, e conseqüentemente de composições, baseado nesses filtros. Com isso, passamos a ter um módulo responsável por manter uma base de tipos de eventos que fazem parte da aplicação, tal como em sistemas de *middleware* CORBA[OMG2004]. Essa idéia de gerenciamento de tipos apresenta uma relação forte e direta com a aplicação que está sendo desenvolvida, ou seja, os eventos que serão manipulados pelos módulos produtores e consumidores.

3.2 Motivação para o Modelo Sinapse

Conforme já apresentado, o paradigma *Publish/Subscribe* pode trazer grandes benefícios para o desenvolvimento de sistemas distribuídos. No entanto, sua utilização, de maneira

isolada, pode criar uma dependência por parte da aplicação em relação ao Sistema de Notificação específico que está sendo usado. Isto porque o código necessário para interagir com o Sistema de Notificação fica mesclado ao código da aplicação. Este ponto pode ser trabalhado a partir da utilização de recursos de interceptação de chamadas de métodos, possibilitando assim uma maior abstração.

A utilização de um modelo arquitetural para descrever o Sistema de Notificação permite trabalhar de forma conjunta as soluções para o suporte à mobilidade e composição de eventos. Uma vez que cada conceito fica descrito em um conector especializado, as expansões de funcionalidades de um Sistema de Notificação, passam a ser implementadas mais flexivelmente. A inclusão de uma nova funcionalidade ou a evolução de algoritmos já existentes, tal como o usado no suporte à composição de eventos, passam a ser tratados isoladamente por estarem concentrados em conectores.

3.3 Estrutura de Componentes do Modelo

Nosso modelo propõe que as funcionalidades de um Sistema de Notificação sejam mapeadas em um modelo arquitetural através do uso de conectores. A seguir, apresentamos na figura 9, as interações entre os conectores propostos.

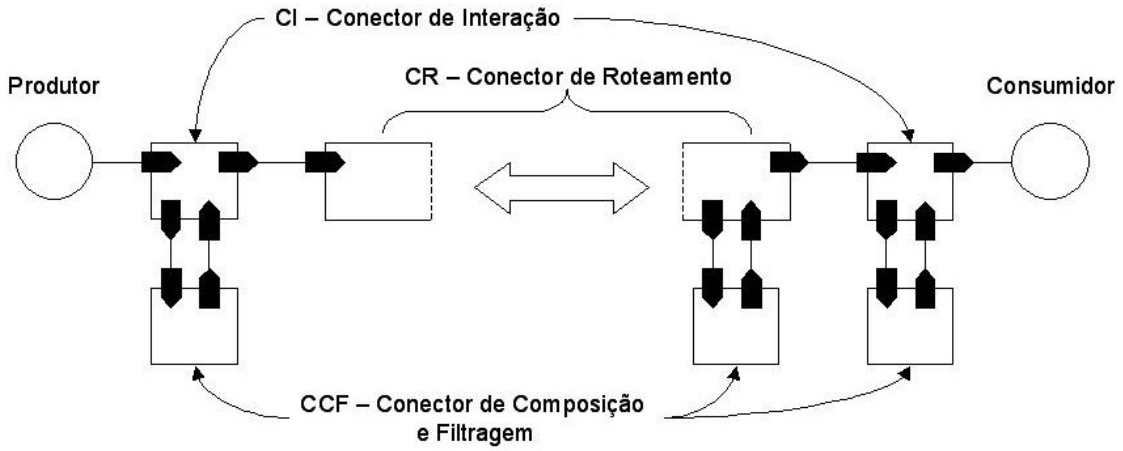


Figura 9 – Representação dos conectores propostos. Os módulos produtor e consumidor são representados por círculos. Os três conectores do modelo são representados por retângulos.

Os Conectores de Composição e Filtragem atuam de maneira independente, executando suas funções sem interferência por parte dos conectores aos quais estão associados. O Conector de Roteamento é representado em duas partes, por encapsular um meio de transporte, por exemplo TCP/IP. O estabelecimento do canal de comunicação é um aspecto não-funcional em relação ao modelo, podendo ser resolvido de diferentes formas. A questão relevante para o modelo é que exista o canal de comunicação. Já o Conector de Interação é o responsável pelas interações dos módulos da aplicação com o Sistema de Notificação.

A proposta tem como base a utilização de um Sistema de Notificação e um Ambiente de Configuração. Portanto, os serviços oferecidos por esses sistemas são considerados utilizados no modelo proposto. O Sistema de Notificação utilizado deve ser modificado para expor os elementos ao Ambiente de Configuração, permitindo a intervenção arquitetural. Em nosso trabalho modificamos o Sistema de Notificação Rebeca [Fie2003a] para trabalhar com o Ambiente de Configuração AC [Lis2003, Car2001]. Os conectores que propomos em nosso modelo devem mapear os componentes e funcionalidades do Sistema de Notificação. Essas modificações serão discutidas em detalhes no *Capítulo 4*, referente a implementação do modelo proposto.

No modelo Sinapse, as composições de eventos são definidas como um conector de filtro que contém um conjunto de filtros e operações lógicas. Para o Sistema de Notificação, a percepção é de um fltro comum, o que garante que as composições de eventos sejam resolvidas em um metanível, não sendo fisicamente introduzidas no Sistema de Notificação.

A seguir, detalharemos os três tipos de conectores que compõem o modelo Sinapse, vamos apresentá-los de forma isolada e detalhada, permitindo que no próximo capítulo sejam discutidas questões referentes à suas implementações.

3.3.1 Conector de Interação – CI

O Conector de Interação tem como objetivo encapsular as interações com o Sistema de Notificação, tornando-o transparente para a aplicação. Esse resultado é obtido através de uma relação direta com os módulos produtores e consumidores. Uma aplicação pode ter vários módulos assumindo os papéis de produtores e consumidores. Por questões de simplicidade, vamos apresentar um cenário envolvendo um produtor e um consumidor independentes.

Vamos analisar primeiro o cenário onde o Conector de Interação encapsula as funcionalidades relativas ao Sistema de Notificação do módulo produtor. A idéia é que, em determinado momento, um módulo passa por uma mudança de estado e essa informação deve ser disponibilizada através de um evento. Consideramos que essa mudança de estado será feita através de uma método, que quando executado é interceptado pelo Ambiente de Configuração AC. O fluxo de execução é transferido para o Conector de Interação, que cria e publica um novo evento com as informações da mudança de estado. Todas essas etapas acontecem sem que o módulo produtor tome conhecimento, permitindo isolar os processamentos necessários para mudar o estado do módulo e os necessários para criar e publicar um novo evento.

Por outro lado, o cenário do lado do módulo consumidor é um pouco diferente. Neste caso, estamos interessados em receber um evento para usar em algum tipo de processamento. Para poder receber as informações relacionadas ao evento, é preciso que o módulo faça uma chamada de método para permitir que o Ambiente de Configuração transfira o controle para o Conector de Interação. Podemos considerar uma chamada de nome padrão definida arbitrariamente como `getInfo()`.

Da mesma forma que acontece com o módulo produtor, essa chamada é interceptada pelo Ambiente de Configuração AC, permitindo a ação do Conector de Interação. A primeira parte da atividade do conector é verificar se já foi feita uma assinatura para receber os

eventos de interesse do consumidor. Para tanto, é usada a API disponibilizada pelo Sistema de Notificação Rebeca. Após realizar a assinatura de eventos, o conector entra num estado de espera até que chegue um evento.

Quando um evento é recebido, o Conector de Roteamento (apresentado na *seção 3.3.3*) notifica, através de uma chamada *callback*, o Conector de Interação do consumidor que está interessado, passando o evento. Esse conector processa o evento e extrai suas informações, preenchendo uma estrutura de dados pré-estabelecida a ser retornada ao consumidor. Esta estrutura de dados conterá as informações do evento num formato próprio para a aplicação, evitando que o consumidor precise trabalhar com eventos.

Uma consideração precisa ser feita sobre a forma de interação do consumidor. Por princípio, a comunicação nos Sistemas de Notificação se faz de forma anônima e assíncrona, conforme já explicado. Portanto, deve existir uma forma pela qual o Conector de Interação notifique o módulo consumidor, da mesma forma que o Conector de Interação é notificado pelo Sistema de Notificação. Temos então duas possibilidades de interação:

i. O consumidor pode buscar as informações através da chamada `getInfo()`, e, com isso, acessa os eventos recebidos até o momento pelo conector. Esse mecanismo funciona como um mecanismo cliente/servidor, onde o consumidor atua como um cliente do Conector de Interação.

ii. O Sistema de Notificação realiza uma chamada *callback* a fim de entregar eventos aos consumidores interessados. Em nosso modelo essa interação é responsabilidade do Conector de Interação e portanto este conector é o destino da chamada. Especificamente no ambiente Rebeca [Fie2003a], esse mecanismo é realizado através de um método especial chamado `process(Event e)`. Quando recebe essa chamada, o conector poderia propagá-la ao consumidor, para isso, deve ser possível

referenciar o módulo consumidor, e isso deve ser especificado na descrição da aplicação.

Estas duas soluções serão mais detalhadamente explicadas no capítulo 5, por se tratar de considerações relacionadas à implementação.

Uma vez executadas as tarefas específicas de interação com o Sistema de Notificação, o controle é retornado aos módulos consumidores/produtores após a interceptação de seus métodos `getInfo()` e `setInfo()`, respectivamente.

3.3.2 Conector de Composição e Filtragem – CCF

O Conector de Composição e Filtragem armazena e aplica as composições e filtros permitindo que os eventos sejam processados de maneira independente dos mecanismos de roteamento.

Os Conectores de Composição e Filtragem são ativados a partir da interceptação dos mecanismos de roteamento de evento, estes mecanismos serão explicados na próxima seção que trata dos Conectores de Roteamento. Quando isso acontece, estes conectores verificam se o evento em questão satisfaz a algum filtro e/ou composição, e em seguida preparam uma lista de identificadores de destinos que precisam receber esse evento.

Temos então duas situações :

i. A primeira, quando o evento satisfaz a um filtro e deve ser encaminhando para um ou mais consumidores. Neste caso, o Conector de Composição e Filtragem deve disponibilizar para o Conector de Roteamento uma lista de identificadores dos próximos nós. Ao chegar ao final da rota essa lista de identificadores conterá referências para os consumidores que estão interessados naquele evento.

ii. A segunda, quando o evento faz parte de uma composição. No modelo Sinapse, as composições de eventos são tratadas a partir de uma máquina de estados. Quando atingimos um estado final onde todas as restrições foram satisfeitas, é gerado um *metaevento* com um identificador. Os *metaeventos* são enviados através dos Sistema de Notificação pelos próprios Conectores de Composição e Filtragem e são tratados como eventos comuns.

3.3.3 Conector de Roteamento – CR

O Conector de Roteamento é responsável pelas funções de roteamento de eventos e por funcionar como ponto de acesso para Conectores de Interação. Além disso notificam os Conectores de Interação dos consumidores através das chamadas *callback*.

Os filtros utilizados pelos consumidores para expressar seu interesse por eventos, são utilizados pelos Conectores de Roteamento para preencher as tabelas de rota. A partir da tabela de rotas a resolução de rotas passa a ser endereçada por identificações dos consumidores, que são disponibilizadas pelos Conectores de Composição e Filtragem após o processamento de filtros e composições. Desta maneira, a complexidade relacionada ao gerenciamento das tabelas de rotas diminui, pois passa a ser trabalhada com endereçamento de nível mais alto. A única verificação necessária é, uma vez obtida a lista de destinos, a identificação para qual dos Conectores de Roteamento devemos encaminhar o evento. Isto será repetido por cada Conector de Roteamento participante da rota até que o último identifique que, ao invés de encaminhar o evento, deve-se notificar o Conector de Interação do consumidor.

3.4 Considerações sobre Mobilidade no Modelo Sinapse

Nossa proposta considera o suporte ao trabalho com redes móveis infra-estruturadas. Consideramos que os Conectores de Roteamento estão localizados na rede fixa, com maior poder de processamento, e os Conectores de Interação funcionam como pontos de acesso para os módulos externos.

O uso do Ambiente de Configuração, juntamente com um ambiente com suporte à mobilidade *ad hoc*, apresenta diversas questões ainda não resolvidas. Contudo, do ponto de vista exclusivamente arquitetural, o tipo de rede móvel utilizado não influencia o mapeamento do modelo sobre o Sistema de Notificação Rebeca. Consideramos que só existe mobilidade no ambiente externo, o que nos leva a dois casos distintos :

i. O consumidor de um evento se move para outro ponto de acesso e passa a utilizar outro Conector de Interação. Enquanto o consumidor estiver em trânsito os eventos continuam sendo entregues no Conector de Interação de origem e devem ser armazenados no *buffer* do Conector de Roteamento. Quando o consumidor se conecta novamente, o novo Conector de Interação deve enviar uma mensagem ao Conector de Interação de origem indicando a nova localização do consumidor. Assim, os eventos armazenados podem ser encaminhados e as rotas de entrega alteradas para a nova localização de acordo com a assinatura do consumidor.

ii. O produtor de um evento se move para outro ponto de acesso e passa a utilizar outro Conector de Interação. Esta situação requer um cuidado especial, uma vez que implica em uma reavaliação dos filtros e composições associados àquele produtor. Completado o processo de migração, o Conector de Interação deve enviar uma mensagem notificando os Conectores de Roteamento da mudança. As composições devem ser então reavaliadas para que a nova posição do produtor seja considerada. Essa questão requer uma reavaliação da rota de acordo com um algoritmo que está fora do escopo deste trabalho. Assumimos que uma entidade externa indica a melhor posição para a instalação do Conector de Composição e Filtragem. Utilizando a capacidade de gerenciamento de alto nível do Ambiente de Configuração, podemos desconectar o Conector de Composição e Filtragem de um Conector de Roteamento e reconectá-lo a outro. É importante notar que é utilizado um único Conector de Composição e Filtragem para cada Conector de Roteamento, num caso de reconfiguração, os Conectores de Composição e Filtragem seriam trocados.

Durante o processo de movimentação dos módulos externos, o que acontece com os eventos pode ser comparado com o efeito de uma quebra de rede. Durante a relocação de conectores, os eventos vão, eventualmente, atingir um Conector de Roteamento a partir do qual não podem prosseguir (rota inválida). Dessa maneira, serão acumulados no *buffer* do Conector de Roteamento e serão encaminhandos novamente, tão logo exista uma rota válida para isso.

3.5 Funcionamento

Até aqui apresentamos uma visão geral do modelo Sinapse e um detalhamento de seus componentes de forma isolada. Agora, vamos reunir esses componentes de maneira a consolidar nossa proposta.

O modelo Sinapse é composto por três entidades. A primeira entidade é um Ambiente de Configuração, responsável por carregar e iniciar os componentes das outras entidades. A segunda entidade é um Sistema de Notificação, descrito em função da primeira, o qual tem por objetivo disponibilizar os recursos de comunicação anônima e assíncrona. A terceira e última entidade é a aplicação, também descrita em termos da primeira, sendo carregada e interligada pelo Ambiente de Configuração. Através das interceptações dos Conectores de Interação, os módulos da aplicação utilizam os recursos do Sistema de Notificação para realizar a comunicação anônima e assíncrona entre seus módulos.

Considerando o desenvolvimento baseado em Sistemas de Notificação, o código necessário para interagir com o sistema fica mesclado ao código da aplicação, conforme apresentado na *seção 3.2*. Isto aumenta a complexidade de desenvolvimento e diminui a legibilidade, além de comprometer a evolução da aplicação caso seja necessário trocar o Sistema de Notificação. Embora haja um desacoplamento entre os módulos produtores e consumidores, estes módulos apresentam um alto acoplamento com o Sistema de Notificação por causa da mesclagem de código. O uso de técnicas de interceptação juntamente com um Sistema de Notificação pode evitar essa mesclagem de código contornando assim o problema de

acoplamento. Em nosso modelo, esse mecanismo é fornecido através do Ambiente de Configuração AC.

O desenvolvimento baseado no modelo Sinapse está diretamente ligado ao uso da interceptação e portanto permite isolar em camadas distintas e bem definidas os aspectos de cada etapa de desenvolvimento. Dessa forma, a aplicação é desenvolvida utilizando uma abstração maior e ficando independente do Sistema de Notificação. Os aspectos relativos as interações com o Sistema de Notificação ficam isolados na camada de conectores. Em caso de haver necessidade de troca do Sistema de Notificação, somente a camada referente aos conectores precisa ser modificada.

Além dos componentes apresentados da seção anterior, também ficam definidos os seguintes papéis :

i. Sistema de Notificação (SN): o Sistema de Notificação no qual o modelo esta aplicado. Após ser adaptado para o uso com o Ambiente de Configuração, não toma conhecimento da operação do modelo. Portanto, as operações são executadas normalmente; no entanto, torna-se possível intervir nas conexões de seus componentes.

ii. Repositório de Filtros: este componente é responsável por manter os filtros e as composições de evento do sistema. É utilizado pelos Conectores de Interação para referenciar filtros e composições de eventos através de um identificador.

iii. Produtor: representação dos módulos da aplicação que produzem a informação que será publicada.

iv. Consumidor: representação dos módulos da aplicação que consomem a informação que será publicada.

v. Oráculo: representação de um módulo capaz de indicar o melhor posicionamento, na rede física, dos Conectores de Roteamento e da disposição dos Conectores de Composição e Filtragem a eles associados em relação aos outros conectores do modelo.

3.5.1 Interação dos Componentes do Modelo

A seguir, apresentamos, na figura 10, a representação de uma interação dos componentes do modelo proposto através de um diagrama de seqüência.

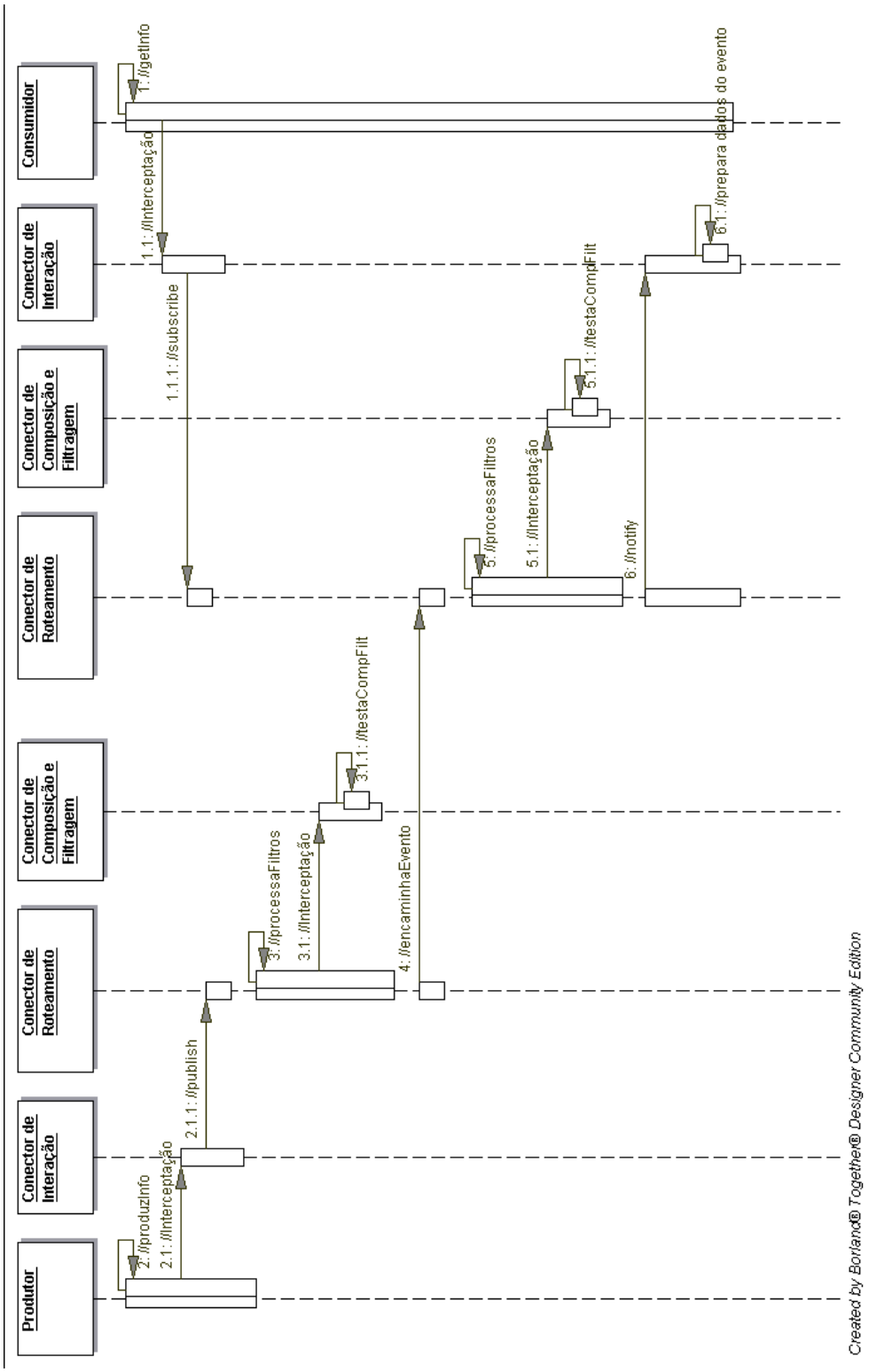


Figura 10 – Diagrama de seqüência de uma interação do modelo proposto

Por questões de simplicidade do diagrama, são usados apenas um produtor, um consumidor, dois Conectores de Roteamento com Conectores de Composição e Filtragem associados.

São representadas as interceptações em relação aos conectores realizadas pelo Ambiente de Configuração. Essas interceptações são apresentadas no diagrama apenas para esclarecer o funcionamento, mas não devem ser interpretadas como chamadas que aconteçam de forma explícita entre os módulos.

Vamos considerar um cenário onde as assinaturas são realizadas antes que qualquer informação seja produzida, isso garante que não haverá perda de eventos.

O primeiro passo da seqüência é a assinatura por parte do consumidor, isso acontece a partir da interceptação do método `getInfo()`. Neste ponto, o Ambiente de Configuração transfere o controle para o Conector de Interação correspondente, que realiza a assinatura de eventos de interesse do consumidor, e passa a esperar pela chegada dos eventos.

O segundo passo consiste em publicar eventos e isso tem início com a interceptação do método que modifica o estado do produtor, `produzInfo()`. De forma semelhante ao anterior, o controle passa para o Conector de Interação que vai criar e publicar o evento com a informação sobre a mudança de estado do produtor.

Com a publicação, o evento atinge a infra-estrutura de comunicação e passa a ser processado pelos Conectores de Roteamento a fim de encontrar a sua rota de entrega. Este processamento tem uma parte delegada ao Conector de Composição e Filtragem, que é o processamento para identificação de quais filtros e composições são satisfeitas pelo evento. Para que isso ocorra, uma nova interceptação, `processaFiltros()`, acontece de maneira que o controle é transferido. Ao final do processamento de filtros e composições, é retornada uma lista de destinos ao Conector de Roteamento, que utiliza esta lista para encaminhar o evento através do método `encaminhaEvento()`. Este processamento dos Conectores de Roteamento e Conectores de Composição e Filtragem se repete através de toda a infra-estrutura.

O último Conector de Roteamento da rota executa uma chamada *callback* para notificar o Conector de Interação do consumidor interessado no evento. Uma vez recebido o evento, o Conector de Interação processa a informação para ser disponibilizada para o módulo cliente em um formato pré-estabelecido. Os eventos ficam armazenados em um *array* até que o consumidor execute uma chamada ao método `getInfo()` para receber a lista de eventos recebidos até o momento. É importante notar que caso não existam eventos disponíveis, o método `getInfo()` retorna imediatamente, e o consumidor precisa executá-lo novamente em um momento posterior.

3.6 Conclusão Parcial

Como foi apresentado neste capítulo, o modelo Sinapse fornece uma abstração para o desenvolvimento de aplicações distribuídas. Isto porque são unidas, em um único modelo, as abordagens de desacoplamento tanto dos Sistemas de Notificação quanto do Ambiente de Configuração AC. Além disso, a utilização de uma descrição arquitetural, como a sugerida, permite que as responsabilidades sejam isoladas de forma flexível, garantindo a evolução de cada uma independente e organizadamente.

Conforme apresentado, para aplicar o modelo Sinapse a um Sistema de Notificação, este deve passar por um conjunto de modificações, de modo a possibilitar as interações com o Ambiente de Configuração. Essa interação entre os dois ambientes permite configurar o Sistema de Notificação durante a execução. Essas modificações serão apresentadas em maiores detalhes no *Capítulo 4*.

Neste capítulo, o modelo Sinapse foi apresentado de forma conceitual visando uma compreensão inicial da proposta. O próximo capítulo vai complementar essa visão, apresentando as questões relativas à implementação deste modelo. Além do relacionamento de classes, serão apresentadas as questões adicionais, relevantes ao total entendimento de como o modelo Sinapse atinge os objetivos definidos.

4 Implementação do Modelo

No capítulo 3, foi introduzido o modelo Sinapse com base nos conceitos apresentados anteriormente. Assim, apresentamos uma implementação de referência, que tem como objetivo mostrar a viabilidade de utilização do modelo. Além disso, identificamos os pontos que podem ser aperfeiçoados a fim de automatizar tarefas, tais como a geração dos conectores.

Uma vez apresentada a viabilidade da utilização dos conceitos conjuntamente, podemos passar a desenvolver alternativas que permitam automatizar cada uma das etapas do processo simplificando a utilização do modelo.

4.1 Ambientes de Suporte ao Modelo

Para a implementação do protótipo do modelo Sinapse, foram utilizados o Ambiente de Configuração AC e o Sistema de Notificação Rebeca. A seguir, apresentamos os requisitos de suporte que cada ambiente deve disponibilizar para que seja possível implementar o modelo Sinapse.

4.1.1 Requisitos de Suporte do Ambiente de Configuração

Como suporte ao modelo, são necessárias algumas características por parte do Ambiente de Configuração, relacionadas abaixo:

i. Suporte à conexão de uma porta ao próprio objeto: deve ser possível que um objeto execute uma chamada de métodos para si próprio através dos mecanismos de interceptação de maneira que seja possível transferir o controle para um conector. Isto é devido a necessidade de interceptar os métodos de mudança de estado dos produtores (`setInfo()`) e os métodos de recebimento de eventos dos consumidores (`getInfo()`).

ii. Deve ser possível passar parâmetros de configuração quando os módulos são instanciados pelo ambiente. Isto porque, quando um Conector de Roteamento inicia a execução, ele precisa saber qual endereço de rede usará para receber conexões. Já os Conectores de Interação devem receber as informações que especificam em que endereço e porta devem se conectar.

iii. Deve ser possível substituir as ligações entre as portas dos módulos sem que seja necessário interromper a execução. Isto permite a reconfiguração dinâmica das conexões entre os Conectores de Roteamento.

4.1.2 Requisitos de Suporte do Sistema de Notificação

O Sistema de Notificação deve ser modificado para que seja possível, através do Ambiente de Configuração, intervir nos módulos durante a execução. As modificações serão explicadas ao longo deste capítulo na descrição das classes de implementação do protótipo do modelo.

A seguir, na figura 11, é apresentada essa relação entre Sistemas de Notificação, a Aplicação e o Recurso de Interceptação, em função dessas necessidades de adaptação.

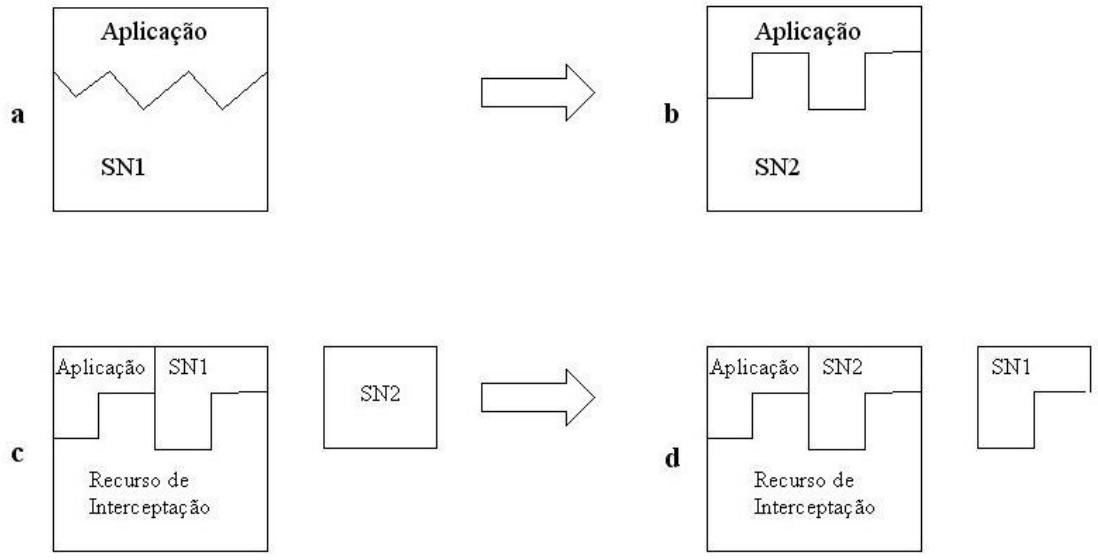


Figura 11 – Relação entre Sistemas de Notificação, Aplicação e o Recurso de Interceptação

Na parte superior da figura, vemos uma mesma aplicação que passa pela modificação necessária para que seja trocado o Sistema de Notificação de SN1 para SN2. Em *a*, temos a representação gráfica da dependência do código entre a aplicação e o Sistema de Notificação SN1, em *b*, a mesma aplicação tem que sofrer modificações para poder utilizar os serviços do novo Sistema de Notificação SN2. Na parte inferior da figura, temos a representação gráfica do ganho atingido pelo uso do modelo Sinapse. A aplicação é desenvolvida sem um relacionamento explícito com o Sistema de Notificação SN1, como podemos ver em *c*. Quando surge a necessidade de troca do Sistema de Notificação, somente o Sistema de Notificação SN2 passa por modificações, a fim de tornar compatível com a aplicação. Assim, em *d*, vemos o novo Sistema de Notificação SN2 já adaptado ao ambiente e entrando no lugar no Sistema de Notificação SN1.

Espera-se que o Sistema de Notificação apresente suporte à mobilidade; no entanto, o suporte à Composição de Eventos é oferecido pelo modelo Sinapse através de uma solução independente e autônoma, encapsuladas no Conector de Composição e Filtragem.

Caso o Sistema de Notificação não suporte a mobilidade, o uso dos Conectores de Interação pode facilitar a inclusão desta capacidade. Conforme apresentado na *seção 3.4*, quando os módulos produtores e consumidores mudam de ponto de acesso, seus novos Conectores de Interação enviam uma mensagem até os pontos de acesso anteriores, invalidando as rotas anteriores e criando novas rotas. Se for utilizado um Sistema de Notificação que não tenha suporte à mobilidade nativo, isso passa a ser possível graças à separação de papéis fornecida pelo modelo.

4.2 Implementação dos Componentes

Nesta seção vamos apresentar os detalhes sobre a implementação dos Componentes que constituem o modelo Sinapse. Nesta primeira etapa, é necessário analisar as características de cada Componente do ponto de vista da implementação, de acordo com as funcionalidades

que precisam ser disponibilizadas. Em seguida, passamos para a descrição do funcionamento desses componentes.

4.2.1 Conector de Interação

As aplicações baseadas no paradigma *Publish/Subscribe* apresentam dois papéis importantes no relacionamento de seus módulos. Estes papéis são os de produtor e consumidor. Assim, temos dois papéis para os Conectores de Interação, um para o suporte ao produtor e outro para o suporte ao consumidor. Devemos observar que esses papéis são exercidos em relação a determinadas portas do módulo, cujas chamadas estão sendo interceptadas. Sendo assim, podemos ter um módulo que exerça os dois papéis em momentos diferentes, através de portas distintas. Esta situação deve ser resolvida com o uso de dois Conectores de Interação, cada qual tratando um papel.

i. No papel de produtor, o Conector de Interação é uma classe derivada de `rebeca.DefaultEventProducer`. Esta classe contém os mecanismos necessários para que um módulo se registre junto ao Sistema de Notificação Rebeca, a fim de publicar eventos. As funções do Conector de Interação relacionadas ao produtor são descritas a seguir:

   a. Estabelecer a conexão com um ponto de acesso. O endereço de rede para conexão é fornecido como parâmetro de configuração para o Conector de Interação de cada produtor.

   b. Executar o código associado a interceptação do método de mudança de estado do produtor de forma a criar e publicar um evento contendo a informação recebida.

ii. No papel de consumidor, o Conector de Interação é uma classe que implementa a interface `rebeca.EventProcessor`. Essa classe deve implementar o método `process(Event e)`, usado pelo Sistema de Notificação como *método gancho*

das chamadas *callback*, ou seja, esse é o método executado para entregar o evento. Dessa forma, quando um evento é recebido pelo ponto de acesso de um dado consumidor, o seu método `process(Event e)` correspondente é executado. O processamento resume-se a pré-processar os eventos recebidos, e armazená-los em um *array* interno, até que o módulo consumidor execute o seu método `getInfo()` correspondente. Podemos relacionar as funções do Conector de Interação para o consumidor como a seguir :

a. Estabelecer a conexão com um ponto de acesso. O endereço de rede para conexão é fornecido como parâmetro de configuração para o Conector de Interação de cada consumidor.

b. Realizar a assinatura dos eventos de interesse do consumidor. A utilização de filtros para assinatura passa a ser programada no Conector de Interação. A assinatura toma como base os filtros que são recebidos do repositório de filtros do sistema através de uma identificação única. Essa identificação é uma *string* atribuída ao filtro pelo programador no momento de sua criação e funciona como chave de identificação no repositório.

c. Processar os eventos recebidos a partir das chamadas *callback* realizadas pelos Conectores de Roteamento. Nesse processamento, os eventos são preparados para serem disponibilizados ao consumidor num formato pré-estabelecido pelo programador da aplicação. Os eventos são acessados pela aplicação através de um método `getInfo()`, quando interceptado esse método transfere o controle para o Conector de Interação que retorna um *array* de eventos recebidos até o momento.

d. Disponibilizar um método que retorne para o consumidor uma cópia da lista de eventos recebidos até o momento da chamada. Este método tem por objetivo permitir que o consumidor acesse os eventos recebidos no momento que for mais conveniente.

4.2.2 Conector de Composição e Filtragem

Este conector provê um mecanismo para gerar e enviar *metaeventos* a partir da identificação das condições expressas em uma composição de eventos. O mecanismo de composição de eventos é implementado através de herança da classe de filtros do Rebeca, que permite manter um conjunto de filtros e operações lógicas entre esses filtros. Fazendo com que a composição seja uma subclasse da classe de filtros, permite garantir a transparência em relação à infra-estrutura de comunicação, que não diferencia filtros comuns de filtros que representem composições.

O processamento de uma composição é realizado pelo Conector de Composição e Filtragem e consiste em executar o método de verificação de eventos de cada um dos filtros para cada evento recebido. Quando um evento satisfaz os critérios de algum filtro, é armazenado para que possa ser usado na etapa de geração do *metaevento*. Ao final da verificação de um evento recebido, o conector testa as condições lógicas da composição, se o resultado for positivo, passa-se para a etapa de geração do *metaevento*. Com os eventos armazenados, o conector cria o *metaevento* e atribui a uma variável especial do *metaevento* o identificador da composição que foi satisfeita. Este identificador é o mesmo utilizado junto do repositório de filtros do sistema. O *metaevento* recém criado é armazenado até que o processamento de todos os filtros e composições termine. Ao final, a lista de *metaeventos* gerados é percorrida e cada *metaevento* é enviado a todos os destinos identificados. O processamento de composição de eventos será visto em maiores detalhes na seção referente às classes de implementação.

4.2.3 Conector de Roteamento

Devido as características encontradas no Ambiente AC, o Conector de Roteamento tem sua implementação baseada em um módulo e não em um conector real. Isto porque esse componente representa a funcionalidade dos *brokers* responsáveis pelo roteamento. Para tanto, torna-se necessário iniciar sua execução de forma isolada a fim de construir a infra-

estrutura de comunicação e isso não seria possível se fosse implementado em um conector. Essa característica é uma particularidade do Ambiente de Configuração AC.

O Conector de Roteamento é composto da junção de duas classes do Ambiente Rebeca original, cujo código fonte foi reunido em uma única classe. Isso permite contornar problemas que impossibilitavam a interceptação correta dos métodos por conta da hierarquia de herança.

Um conector especial de transporte, acessório ao modelo, foi desenvolvido para permitir a interconexão dos Conectores de Roteamento. Este componente é o Conector de Transporte e sua função é estabelecer conexões com outros Conectores de Roteamento. Quando o Conector de Roteamento precisa encaminhar um evento, ele executa uma chamada de método que é interceptada, transferindo o controle para o Conector de Transporte. Dessa maneira, a conexão a ser usada é retornada ao Conector de Roteamento dinamicamente. Através dos recursos de configuração, pode-se substituir os Conectores de Transporte entre Conectores de Roteamento, procedendo assim a alteração dinâmica da topologia. Este conector especial é utilizado por facilitar a adaptação do sistema Rebeca que conta com uma classe especial para encapsular a camada de transporte. Isolando o estabelecimento da conexão de rede nesse conector, os Conectores de Roteamento podem trabalhar de forma semelhante ao sistema Rebeca original, pois a troca de conexões é realizada através da troca de Conectores de Transporte.

Este recurso de configuração que permite a substituição de Conectores de Transporte é a chave para o tratamento de falhas nessa implementação do modelo Sinapse. Na arquitetura Rebeca original, para trocar uma conexão entre dois *R-brokers*[3] é necessário que o *R-broker* responsável pelo estabelecimento da conexão seja encerrado. Um outro aspecto é que cada *R-broker* só realiza uma conexão, ou seja, embora seja possível receber conexões de *n* módulos e *R-brokers*, cada *R-broker* só estabelece uma conexão como cliente. Assim,

[3] Usamos o termo *R-broker* como uma identificação dos *brokers* do Sistema de Notificação Rebeca apenas como forma de não confundi-los com os Conectores de Roteamento do modelo Sinapse.

quando um *R-broker* é encerrado, todos os outros módulos que estão conectados a ele têm suas conexões invalidadas, precisando ser encerrados. O que acontece, então, é um efeito cascata em toda a infra-estrutura de comunicação com o término das atividades do ponto de falha para trás. Este efeito não acontece no modelo Sinapse porque, quando uma conexão se torna inválida, podemos intervir e conectar o Conector de Roteamento a outro através de um novo Conector de Transporte.

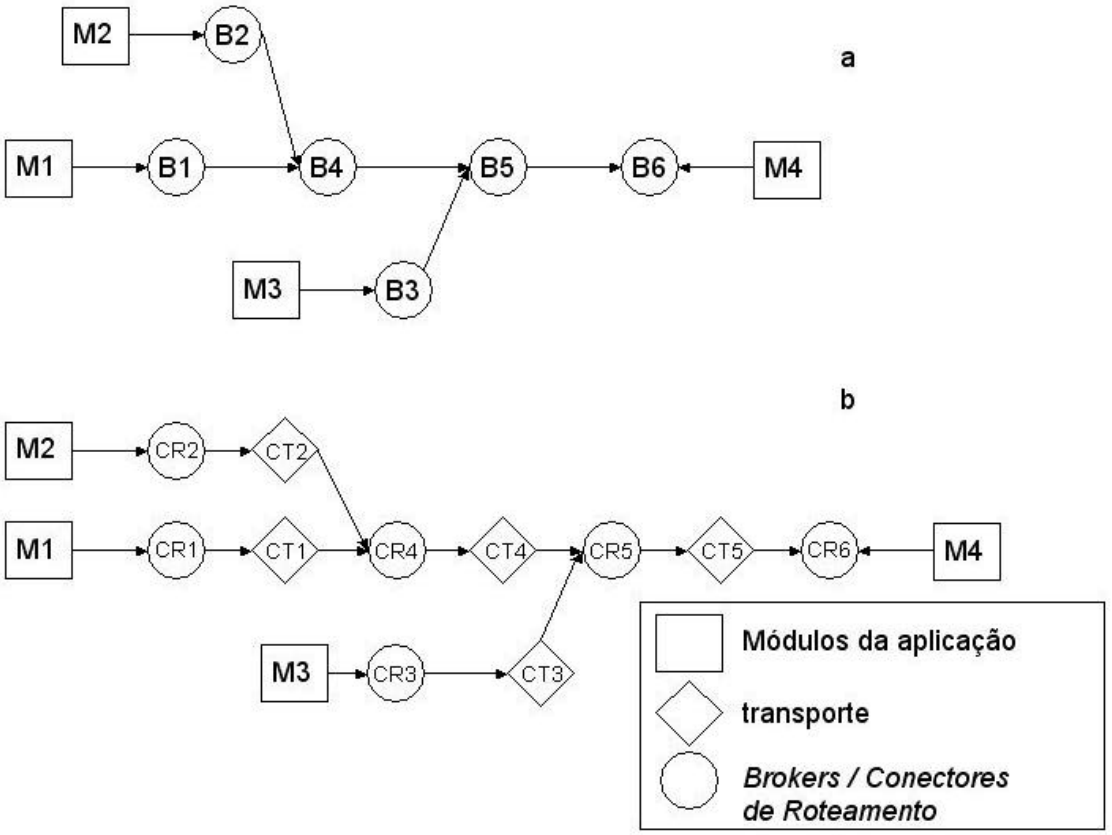


Figura 12 – Comparação entre o modelo Rebeca (a) e a implementação do modelo Sinapse (b)

Na figura 12, vemos duas configurações. Em (a), temos um exemplo de infra-estrutura de comunicação utilizando a arquitetura Rebeca tradicional. Cada *R-broker* se conecta a um outro *R-broker* e cada um pode receber *n* conexões. Assim, se ocorrer uma falha de comunicação entre os *R-brokers* B5 e B6, por exemplo, é necessário encerrar o *R-broker* B5 para estabelecer uma nova conexão, e este efeito se propaga pelo resto da infra-estrutura.

Em (b), temos o mesmo exemplo utilizando a implementação do modelo Sinapse. São usados Conectores de Transporte para realizar a conexão entre os Conectores de Roteamento. Dessa maneira, se for necessário alterar qualquer dessas conexões, devemos executar dois passos :

i. Criar um novo Conector de Transporte que estabeleça a comunicação com o Conector de Roteamento desejado

ii. Reconfigurar a topologia da aplicação através dos recursos do ambiente AC.

A seguir, a figura 13 apresenta as etapas de reconfiguração da topologia. A configuração inicial representada em "a" permite que os módulos *M1* e *M4* troquem informações. Em "b", acontece uma falha na conexão com o Conector de Roteamento *CR6* e a comunicação entre os módulos é interrompida. Em "c", é criado o Conector de Transporte *CT6*, que estabelece comunicação com o Conector de Roteamento *CR6*. Em seguida, o Conector de Roteamento *CR5* passa a se comunicar com o novo Conector de Transporte *CT6*, restabelecendo a comunicação entres os módulos *M1* e *M4*.

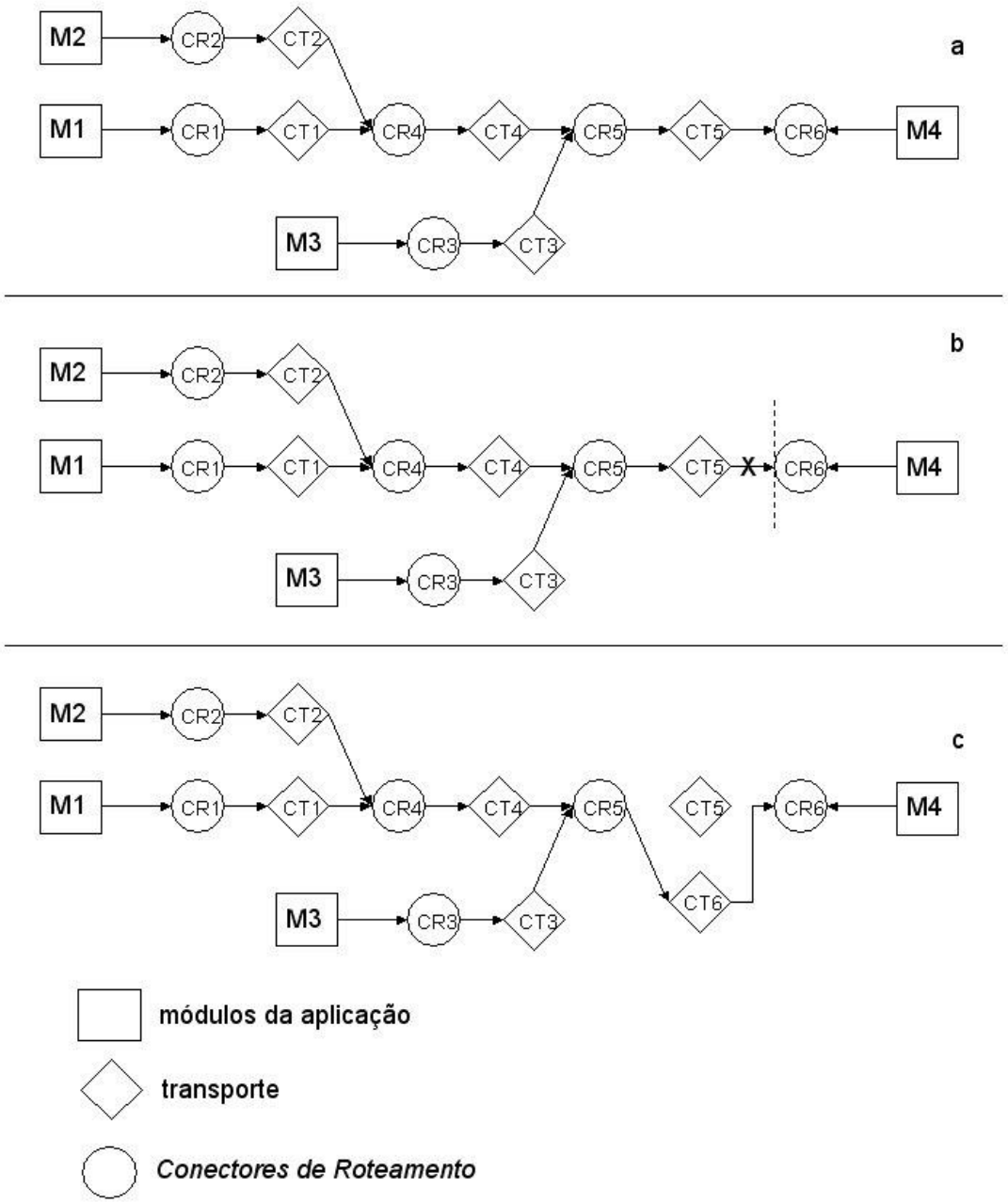


Figura 13 – Reconfiguração da topologia utilizando os Conectores de Transporte

4.3 Interação entre os Componentes

Uma vez definidos os papéis individuais de cada componente do modelo, passamos a descrever como esses componentes interagem entre si, a fim de obter as facilidades descritas.

Considerando que o modelo proposto utiliza um ambiente de configuração como estrutura base, emprega-se a linguagem de descrição do ambiente para expressar as conexões entre os componentes do modelo. Para facilitar a compreensão, o arquivo de descrição declara tanto os módulos da aplicação, quanto a topologia dos componentes do modelo.

A seguir, a figura 14 apresenta a primeira parte do arquivo de descrição em uma ordem lógica, onde são definidas as portas que serão utilizadas, os módulos e conectores juntamente com suas instâncias.

```
port getEventTransport
port getRepository
port getRoutingTable
port checkFiltersAndComposition
port setInfo
port getInfo

module EventRouterAC
{
    inport getEventTransport
    outport getEventTransport
    outport getRoutingTable
} aR1 aR2 aR3

module RoutingTableAC
{
    inport getRoutingTable
    inport checkFiltersAndComposition
    outport checkFiltersAndComposition
} aart1 aart2 aart3

connector CCF
{
    inport checkFiltersAndComposition
    outport checkFiltersAndComposition
} ccf1 ccf2 ccf3

module Produtor
{
    inport setInfo
    outport setInfo
} prod

module Consumidor
{
    inport getInfo
    outport getInfo
    outport getRepository
} cons
```

```
connector CIProdutor
{
    inport setInfo
    outport setInfo
} ciProd

module FilterRepository
{
    inport getRepository
} repository

connector CIConsumidor
{
    inport getInfo
    outport getInfo
    inport getRepository
    outport getRepository
} ciCons

connector RouterConnector
{
    inport getEventTransport
    outport getEventTransport

} zCon1 zCon2
```

Figura 14 – Descrição de portas e módulos

A seguir, a figura 15 apresenta a definição das máquinas onde cada instância será executada, e, por fim, quais as ligações entre as portas de módulos e conectores. São essas ligações entre portas que podem ser manipuladas durante a execução, a fim de alterar dinamicamente a topologia da aplicação.

```
instantiate aR1 localhost 8000
instantiate aR2 localhost 9000
instantiate aR3 localhost 9600

instantiate aart1 localhost
instantiate aart2 localhost
instantiate aart3 localhost

instantiate ccf1 localhost
instantiate ccf2 localhost
instantiate ccf3 localhost

instantiate prod localhost
instantiate cons localhost
```

```
instantiate zCon1 localhost localhost 9000
instantiate zCon2 localhost localhost 9600

instantiate repository localhost

instantiate ciProd localhost -host localhost -port 8000
instantiate ciCons localhost -host localhost -port 9600

link aR1 getRoutingTable aart1 getRoutingTable
link aR3 getRoutingTable aart3 getRoutingTable
link aR2 getRoutingTable aart2 getRoutingTable

link aart1 checkFiltersAndComposition ccf1 checkFiltersAndComposition
link ccf1 checkFiltersAndComposition aart1 checkFiltersAndComposition

link aart2 checkFiltersAndComposition ccf2 checkFiltersAndComposition
link ccf2 checkFiltersAndComposition aart2 checkFiltersAndComposition

link aart3 checkFiltersAndComposition ccf3 checkFiltersAndComposition
link ccf3 checkFiltersAndComposition aart3 checkFiltersAndComposition


link aR1 getEventTransport zCon1 getEventTransport
link zCon1 getEventTransport aR1 getEventTransport

link aR2 getEventTransport aR2 getEventTransport

link aR3 getEventTransport aR3 getEventTransport

link prod setInfo ciProd setInfo
link ciProd setInfo prod setInfo

link cons getInfo ciCons getInfo
link ciCons getInfo cons getInfo

link cons getRepository ciCons getRepository

link ciCons getRepository repository getRepository
```

Figura 15 – Descrição de instâncias e ligações

Para podermos interagir com as portas de um determinado módulo ou conector, é necessário que este seja exposto ao Ambiente de Configuração. Para o modelo Sinapse são necessárias as seguintes portas :

i. getInfo: esta porta é definida na Interface implementada pelos módulos que vão atuar como consumidores de informação. Isso permite que essa porta seja exposta ao

ambiente. Assim, podemos interceptá-la, permitindo que o Conector de Interação do consumidor disponibilize as informações recebidas.

ii. setInfo: esta porta é definida na Interface implementada pelos módulos que vão atuar como produtores de informação. Da mesma forma que no caso anterior, esta porta é interceptada, porém pelo Conector de Interação do produtor, que vai usar as informações passadas pela chamada para publicar o Evento.

iii. getEventTransport: esta porta é definida na Interface implementada pelos Conectores de Roteamento e deve ser interceptada pelos Conectores de Transporte. O objetivo disso é eliminar uma referência direta ao nível de transporte permitindo que alterações sejam feitas de forma transparente através da substituição do Conector de Transporte utilizado.

iv. getRepository: esta porta é definida na Interface implementada pelos módulos consumidores. Embora estes módulos não usem a referência ao Repositório de Filtros, esta porta é utilizada para que o Conector de Interação do consumidor possa conseguir uma referência ao Repositório de Filtros. Isto acontece porque os conectores do ambiente AC ainda não podem realizar chamadas a uma porta de forma independente.

v. getRoutingTable: esta porta é definida na Interface implementada pelas tabelas de rota. Embora as tabelas de rota não tenham sido alteradas, precisam ser expostas ao Ambiente de Configuração, pois os Conectores de Composição e Filtragem ficam associados a elas. Uma vez recebida a referência da tabela de rotas associada, o Conector de Roteamento a armazena e passa a usá-la da forma tradicional, sem a intervenção do Ambiente de Configuração.

vi. checkFiltersAndComposition: esta porta é definida na Interface implementada pelas tabelas de rota. O objetivo dessa porta é permitir que o processamento de eventos aconteça no Conector de Composição e Filtragem que fica associado à tabela de rotas. Com isso, o conector tem a liberdade para verificar todas as condições de composição de eventos e geração de *metaeventos* de modo transparente. A tabela de rotas recebe como retorno do processamento do evento uma lista de destinos para aquele evento. O envio dos *metaeventos* é realizado no próprio conector, tornando-o independente de recursos do Sistema de Notificação utilizado.

4.4 Casos de Testes e Verificações da Implementação de Referência

O objetivo dessa seção é apresentar a estratégia de testes e verificação que foi utilizada ao longo do desenvolvimento da implementação. Não foram realizados testes de carga, pois o objetivo era apenas verificar a viabilidade e o funcionamento da Implementação de Referência. No próximo capítulo, são apresentados outros testes que têm foco na verificação qualitativa da Implementação de Referência.

Durante o desenvolvimento da implementação, foram realizados testes para cada uma das funcionalidades que iam sendo construídas. Assim, os testes foram realizados de forma incremental: cada novo teste utilizava os recursos disponibilizados na etapa anterior, até atingir o pleno funcionamento da implementação.

Os casos de testes contemplam as seguintes funcionalidades :

i. Transparência das interações: o primeiro aspecto a ser implementado e testado foi a capacidade de permitir que os módulos da aplicação interagissem com a infra-estrutura de comunicação de maneira transparente através dos Conectores de Interação.

ii. Tolerância a falhas: a segunda etapa foi construir o suporte a reconfiguração dinâmica da infra-estrutura de comunicação. Neste ponto, foram introduzidos os Conectores de Transporte. O ambiente AC também passou por uma modificação para incluir uma funcionalidade específica para permitir a substituição de conectores. A partir da substituição de Conectores de Transporte torna-se possível contornar situações de falha.

iii. Transparência no processamento de eventos: a terceira etapa consistiu em permitir que o processamento de eventos acontecesse no Conector de Composição e Filtragem. Para tanto, foi necessário criar os pontos de interceptação. Isso foi feito sem que fosse introduzida nessa etapa o processamento de composições de eventos. Essa divisão de trabalho permitiu resolver primeiro a interceptação para posteriormente centralizar os esforços no tratamento das composições. Esses pontos de interceptação foram identificados a partir do estudo do código do sistema Rebeca original e das funcionalidades de roteamento de eventos. A partir da identificação do ponto onde os filtros são aplicados aos eventos foi possível isolar as chamadas a fim de determinar os pontos de interceptação. A mesma estratégia poderia ser usada para guiar o trabalho de adaptação de outros Sistemas de Notificação para disponibilizar os recursos de composição de eventos do modelo Sinapse.

iv. Composição de eventos: a quarta e última etapa constituiu em permitir expressar as composições de eventos. Para tanto, foram criadas classes de evento, filtro e composição e toda a lógica para o reconhecimento e criação de *metaeventos*. Essa etapa aconteceu de forma isolada, uma vez que, na etapa anterior, todos as questões referentes à identificação dos pontos de interceptação para os Conectores de Composição e Filtragem foram resolvidas. Uma questão que assumimos que é resolvida pelo oráculo que indica o melhor posicionamento dos conectores.

Foi construída uma aplicação que serviu de apoio para testes. A aplicação foi alterada ao longo do desenvolvimento, de forma a expressar a utilização de cada nova característica, até que, ao final, todos os recursos haviam sido testados e verificados.

4.5 Restrições e Condicionantes da Implementação

Nesta seção vamos tratar de cada item que, de alguma forma, restringiu ou condicionou as decisões de desenvolvimento do protótipo baseado no modelo Sinapse. As restrições estão separadas em duas categorias: Restrições Tecnológicas, por conta do nível de desenvolvimento das tecnologias usadas, e Restrições de Suporte, por conta das características dos ambientes de suporte AC e Rebeca.

Por se tratar de um protótipo, nem todos os aspectos do modelo puderam ser resolvidos. Alguns itens dependem de suporte tecnológico ainda em evolução.

4.5.1 Restrições Tecnológicas

Nesta seção, vamos apresentar os itens tecnológicos que de alguma forma influenciaram no desenvolvimento da implementação. São eles:

i. Suporte à mobilidade: embora o suporte à mobilidade seja discutido em [Fie2003a], a implementação de um exemplo funcional depende de tecnologias ainda em desenvolvimento. Como exemplo, podemos citar as restrições que a plataforma J2ME apresenta, o que dificulta a implementação de um protótipo. Para tanto, toda uma biblioteca de comunicação móvel precisaria ser desenvolvida, o que estaria fora do escopo deste trabalho. Tanto para o uso dos Conectores de Interação quanto para o uso do Sistema de Notificação, de forma direta, ainda não há disponível o suporte em Java para isso. Isto porque não esta disponível o suporte total de *Remote Method Invocation* para essa plataforma.

ii. Movimentação de módulos de aplicação: os módulos da aplicação têm plena liberdade para se mover, independente de serem módulos produtores ou consumidores. Esta movimentação pode acarretar em mudança de ponto de acesso à rede, o que pode implicar em reorganização das rotas de entrega de eventos. No protótipo, utilizamos um mecanismo manual onde os módulos consumidores refazem suas assinaturas. Esse mecanismo permite manter as rotas de entrega de eventos de maneira simples. Podemos considerar que quando um módulo se move de um ponto a outro ele pode abandonar o Conector de Interação que estava sendo usado e, através de recursos do Ambiente de Configuração, se ligar a um novo conector.

4.5.2 Restrições de Suporte

i. Relocação de filtro: ocorre quando, por um motivo de reorganização da topologia, um Conector de Composição e Filtragem deixa de cumprir sua tarefa por não estar mais num ponto de interseção de rotas das fontes de eventos relevantes. Neste caso, o conector poderia ser movido, pois as informações de filtros estão sendo tratadas de forma independente. No entanto, o algoritmo de roteamento usado necessita que os Conectores de Interação refaçam as assinaturas quando um módulo se move. Assim, quando as assinaturas são refeitas os Conectores de Composição e Filtragem são atualizados e as rotas são atualizadas.

ii. Propagação de *callback* do Conector de Interação ao consumidor: este item depende de um suporte por parte do Ambiente de Configuração, de modo que um conector possa atuar como um módulo cliente. Na versão atual, quando um conector é empregado, não é possível utilizar outras portas que não aquelas que estão sendo interceptadas. Se o Ambiente de Configuração oferecer esse recurso, podemos criar uma porta no módulo consumidor e ligá-la ao Conector de Interação. Assim, quando uma chamada *callback* é recebida, o conector pode propagá-la imediatamente para o consumidor.

4.5.3 Condições Assumidas para o Desenvolvimento do Protótipo

i. Alocação inicial de Conectores de Roteamento: seguirá o modelo dos Sistemas de Notificação tradicionais, ou seja, pode ser definida uma topologia inicial baseada em uma distribuição determinada pelo programador.

ii. Relocação de Conectores de Roteamento: esta situação consiste em mover um determinado Conector de Roteamento de um ponto a outro na rede física. Este tipo de intervenção só tem sentindo quando o componente que será movido mantém algum tipo de estado interno relevante. No caso dos Conectores de Roteamento do modelo Sinapse, isto não acontece, pois o processamento de composições e filtros é delegado a outro componente e a tabela de rotas é recriada. Dessa maneira, para mover um Conector de Roteamento, podemos abandoná-lo e instanciar um novo conector na máquina destino desejada. Em seguida, a topologia é reconfigurada para utilizar o conector novo. Em relação aos outros conectores não são feitas considerações sobre movimentação. Conforme apresentado na seção anterior, os Conectores de Composição e Filtragem não sofrem movimentação. Já os Conectores de Interação são criados para acesso dos módulos externos, se um módulo externo se move, o conector ao com o qual estava interagindo é descartado.

iii. Suporte à tolerância a falhas: graças à utilização do Ambiente de Configuração, torna-se possível ter uma visão global da aplicação. A partir dessa visão global, pode-se acompanhar o estado dos conectores e módulos externos ligados ao sistema de gerenciamento da arquitetura. Esse acompanhamento pode identificar situações de falha e, através da interface disponível, é possível interagir com os módulos e conectores alterando as conexões existentes entre estes. Com isso, situações de falha podem ser contornadas a partir do console de comandos. A automação do tratamento de falhas é um dos pontos ainda a serem desenvolvidos na implementação atual.

4.6 Conclusão Parcial

Como apresentado neste capítulo, foi desenvolvida uma implementação de referência. Para isso, foram feitas modificações no Ambiente de Configuração AC e adaptações no Sistema de Notificação Rebeca, a fim de poderem trabalhar de forma conjunta. Foram apresentados os detalhes de cada classe que foi criada ou adaptada para o funcionamento do modelo.

Devido a restrições por parte da tecnologia ou dos ambientes de suporte, algumas características do modelo não puderam ser plenamente desenvolvidas. Nos casos em que isso ocorreu, foram apresentados os detalhes dessas restrições bem, como as soluções necessárias para contornar os problemas identificados.

Neste capítulo, foi apresentada a implementação desenvolvida como parte do modelo Sinapse. O próximo capítulo apresenta uma comparação entre o modelo de desenvolvimento, baseado no Sistema de Notificação Rebeca, e o desenvolvimento baseado no modelo Sinapse. Além das questões referentes ao desenvolvimento de aplicações distribuídas, também serão apresentados itens relativos ao ganho qualitativo em relação ao sistema Rebeca.

5 Avaliação Qualitativa do Modelo Sinapse

No capítulo 4, foi apresentada a Implementação de Referência do modelo Sinapse. Esta implementação torna possível a experimentação prática e concreta dos conceitos propostos no modelo. Com isso, podemos comparar os aspectos do interfaceamento de aplicações baseadas no modelo Sinapse com as aplicações desenvolvidas no modelo do Sistema de Notificação Rebeca.

Vamos apresentar de maneira detalhada os procedimentos para o desenvolvimento de uma aplicação *Publish/Subscribe* tradicional. Em seguida, essa aplicação é alterada para se adequar ao modelo Sinapse, a fim de apresentar as modificações necessárias e os ganhos qualitativos. Para o melhor entendimento das comparações que serão apresentadas, é

importante que não seja utilizada uma aplicação complexa, pois isso desviaria o foco da discussão. Utilizamos uma aplicação que implementa um par produtor-consumidor de maneira a tornar mais direta a percepção das diferenças entre o modelo tradicional e o modelo Sinapse.

Por fim, são apresentados de maneira, cenários de aplicações que podem utilizar o paradigma *Publish/Subscribe*, e conseqüentemente podem ser desenvolvidas no modelo Sinapse.

5.1 Critérios de Avaliação

A aplicação utilizará os recursos disponibilizados pelo modelo Sinapse a fim de permitir a comparação com o modelo tradicional. A seguir são listados os quatro critérios que serão utilizados para avaliar qualitativamente os benefícios atingidos pelo uso do modelo Sinapse:

i. Interação com a infra-estrutura de comunicação
ii. Assinatura de eventos simples
iii. Composição de Eventos
iv. Continuidade em presença de falhas na infra-estrutura de comunicação

5.2 Descrição Conceitual da Aplicação

A aplicação experimental será composta por um módulo produtor e um módulo consumidor. Estes módulos utilizarão os recursos de comunicação disponíveis nos dois modelos, Rebeca e Sinapse, para realizar a troca de informação. O objetivo está em comparar as características de desenvolvimento.

Para o modelo Sinapse, será considerada uma infra-estrutura, composta de cinco Conectores de Roteamento, ligados em série, executando na mesma máquina. Para o modelo tradicional, será considerada uma infra-estrutura de cinco *brokers* também em série, executando na mesma máquina.

Os eventos contém um atributo *"nome"* de tipo *string*. São definidos três valores distintos que são publicados a fim de satisfazer os filtros. As composições de eventos deverão considerar atributos diferentes de três eventos distintos. Será considerada a operação *"E lógico"* entre três filtros distintos para identificação da ocorrência desejada. Os atributos a serem considerados também serão *strings*, sendo que, neste caso, cada evento contém apenas um atributo. Serão disparados eventos que satisfaçam as condições de cada filtro de forma a completar a composição de eventos.

5.3 Modelo Rebeca *Publish/Subscribe*

Nesta seção, apresentamos os detalhes da implementação da aplicação experimental no modelo tradicional. Foram desenvolvidas duas versões distintas para trabalhar com eventos simples e com eventos compostos. Ambas utilizaram a mesma configuração da infra-estrutura de comunicação para troca de informação.

Para processar essa classe de eventos foi criado um filtro que verifica a igualdade do atributo do evento com o valor definido no próprio filtro. O método de processamento do evento é apresentado na figura 16, a seguir.

```
public boolean match(Event e) {
    if (e instanceof EventoSimples)   {
        EventoSimples tmp = (EventoSimples) e;
        return nome.equals(tmp.getNome());
    }
    return false;
}
```

Figura 16 – Método de processamento de evento simples

Para poder publicar eventos, o produtor deve estabelecer uma conexão com a infra-estrutura de comunicação. O evento pode ser configurado passando-se um valor no próprio construtor.

Dessa maneira, o produtor pode criar o evento juntamente com a chamada ao método para publicação, apresentado na figura 17, a seguir. O primeiro passo é associar-se ao broker e em seguida um evento é instanciado e publicado.

```
_broker = DefaultEventBroker.getEventBroker();
super.setEventBroker(_broker);
…
publish(new EventoSimples("uff"));
```

Figura 17 – Estabelecimento da comunicação e método de publicação de evento simples

Para receber os eventos, o consumidor deve conectar-se a infra-estrutura de comunicação e, em seguida, assinar através de um filtro, o evento de seu interesse. Para isso, deve usar o código apresentado a seguir, na figura 18. Para o exemplo foi criada uma classe `FiltroNome` que é capaz de tratar eventos com o atributo *"nome"*.

```
_broker = DefaultEventBroker.getEventBroker();
try {
    _sub = new Subscription(new FiltroNome("uff"));
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
 broker.subscribe( sub, this);
```

Figura 18 – Estabelecimento da comunicação e assinatura de evento simples

A partir do ponto em que é estabelecida a comunicação e feita a assinatura de eventos, o consumidor pode ser notificado através do método `process(Event e)`, descrito na *seção 4.2.1*. Este método contém a lógica de processamento dos eventos recebidos. Neste exemplo, esse processamento consiste apenas em apresentar as informações recebidas.

Devido às limitações para tratar a composição de eventos apresentadas pelo ambiente Rebeca[Fie2003a], foi necessário criar um evento especial que contivesse três atributos diferentes. Assim teremos os atributos *"nome1"*, *"nome2"* e *"nome3"*. Quando utilizamos os recursos disponíveis em Rebeca para expressar composições, um único evento deve satisfazer *todos* os filtros da composição para que seja considerado válido. Ou seja, não é possível construir um *metaevento* a partir da ocorrência de diferentes eventos que satisfaçam os filtros de uma composição.

Para a versão da aplicação que trabalha com eventos compostos, o produtor não tem diferenças em sua estrutura, mas apenas o evento criado recebe três valores ao invés de somente um. No entanto, o consumidor apresenta uma pequena diferença no código referente à assinatura, pois é necessário criar os filtros em etapas e juntá-los através de um filtro que aplica um *"E lógico"*. Essa diferença é apresentada na figura 19, a seguir.

```
_broker = DefaultEventBroker.getEventBroker();

try {
    FiltroNome1 f1 = new FiltroNome1("uff");
    FiltroNome2 f2 = new FiltroNome2("mestrado");
    FiltroNome3 f3 = new FiltroNome3("filtro3");

    ANDFilter and1 = new ANDFilter(f1,f2);
    ANDFilter and2 = new ANDFilter(and1,f3);
    _sub = new Subscription(and2);

} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
 broker.subscribe( sub, this);
```

Figura 19 – Assinatura de evento composto

5.4 Modelo Sinapse

Também foram criadas duas versões da aplicação experimental para o modelo Sinapse. Cada uma usou um arquivo de descrição da arquitetura semelhante, trocando apenas as referências para os módulos produtores e consumidores, que são diferentes.

Os filtros são criados no Repositório de Filtros e a composição é criada em função destes. A seguir, a figura 20 apresenta esta estratégia. Especificamente na parte de criação da composição, na figura 19 temos a criação de filtros AND intermediários, enquanto na figura 20 é criada uma composição que recebe todos os filtros necessários.

```
FiltroAC f = new FiltroAC();
f.addConstraint("nome","==","uff");
f.setName("filtroF1");
```

```
FiltroAC f2 = new FiltroAC();
f2.addConstraint("nome","==","mestrado");
f2.setName("filtroF2");

FiltroAC f3 = new FiltroAC();
f3.addConstraint("nome","==","filtro3");
f3.setName("filtroF3");

ComposicaoAC c = new ComposicaoAC();
c.addFilter(f,"AND");
c.addFilter(f2,"AND");
c.addFilter(f3);
c.setName("composicaoC1");
```

Figura 20 – Criação de filtros e composição no Repositório de Filtros

O produtor executa o método `setInfo()`, que altera o seu estado, e o Conector de Interação publica o valor. Este método é definido aqui como um exemplo, mas qualquer método poderia ser interceptado para gerar uma publicação de eventos, inclusive poderíamos ter diferentes métodos sendo interceptados e gerando publicações de eventos distintos. A diferença entre o produtor de cada uma das versões da aplicação no modelo Sinapse é que, para a gerar a composição de eventos, o produtor deve alterar seu estado entre valores distintos. A seguir, a figura 21 apresenta o código do produtor para trocar o estado e, logo em seguida, o código do Conector de Interação para publicar o evento. O mesmo Conector de Interação é usado em ambas versões da aplicação de testes.

Chamada de método para alteração de estado do produtor interceptada:

```
getProdutorACInt().setInfo( "1" );
```

Método do Conector de Interação executado após o método `setInfo()`:

```
public void Pos_setInfo(String obj) {
    EventoAC evento = new EventoAC();
    if(obj.equals("3"))    evento.addField ("nome","mestrado");
    else          if(obj.equals("5"))          evento.addField
("nome","filtro3");
    else if(obj.equals("1")) evento.addField ("nome","uff");
    publish(evento);
}
```

Figura 21 – Alteração de estado interceptada e publicação de evento

Para o papel de consumidor temos uma única classe, sendo que a diferenciação entre eventos simples ou compostos fica resolvida em duas classes de Conectores de Interação. Entre as classes de conectores, a diferença está na requisição feita ao repositório do sistema: um filtro, para eventos simples, ou uma composição, para eventos compostos. A seguir, a figura 22 apresenta o código do módulo consumidor.

Estabelece a referência ao repositório de filtros:

```
if (getRepositoryACInt() != null && !repository){
    FilterRepository fr[] = new FilterRepository[1];
    getRepositoryACInt().getRepository(fr);
    repository = true;
}
```

Executa o método `getInfo()`:

```
getConsumidorACInt().getInfo(result);
```

Figura 22 – Consumidor para o modelo Sinapse

O método `getInfo()` recebe como parâmetro um ArrayList, onde serão retornados os eventos recebidos pelo Conector de Interação.

Para os conectores, a diferença está no método de assinatura que referencia o repositório do sistema. O método `Pre_getInfo(Object[] obj)` é o responsável por retornar ao consumidor os eventos recebidos até o momento. Um método `process(Event e)` é disponibilizado para permitir que a infra-estrutura de comunicação realize as chamadas *callback*, iniciando o processamento e armazenamento dos eventos. A figura 23, a seguir, apresenta o código para os Conectores de Interação para os módulos consumidores.

Método assinar para eventos simples:

```
private void assinar() {
    if (_broker!=null) {
        filtro = repositorio.getFilter("filtroF1");
        _broker.subscribe(new      Subscription(filtro),
this);
    }
}
```

Método assinar para eventos compostos:

```
private void assinar() {
    if (_broker!=null) {
        comp = repositorio.getFilter("composicaoC1");
        _broker.subscribe(new Subscription(comp), this);
    }
}
```

Método `Pre_getInfo(Object[] obj)`:

```
public void Pre_getInfo(Object[] obj) {
    obj[0]=events.clone();
```

```
    if (events.isEmpty()) assinar();
    synchronized (events) {events.clear();}
}
```

Método `process(Event e)`:

```
public void process(Event e) {
    EventoAC tmp = (EventoAC) e;
    String evento = "";
    if(!tmp.isMetaeventComplete()) {
     Hashtable a = tmp.getEventFields();
     EventField ef = null;
     Enumeration enum = a.elements();
     while(enum.hasMoreElements())  {
          ef = (EventField) enum.nextElement();
          evento += ef.toString()+" - ";
     }
    } else {
        evento += tmp.toString();
    }
    events.add(evento);
}
```

Figura 23 – Consumidor para o modelo Sinapse

O método `process(Event e)` é chamado pelo Conector de Roteamento quando um evento precisa ser entregue ao Conector de Interação do consumidor interessado. Neste exemplo, o método pode tratar tanto eventos compostos quanto eventos simples. A primeira etapa é converter o evento recebido para o tipo `EventoAC`, esta classe pode transportar um *metaevento*. Assim, o segundo passo é verificar se o evento é um *metaevento* e isso é feito através do método `isMetaEventComplete()`. Caso o evento recebido seja um *metaevento* ele é processado a fim de recuperar os valores dos eventos que foram usados

em sua geração. Isso é feito percorrendo a coleção de `EventFields` do metaevento. Caso o evento não seja um *metaevento*, ele é processado sendo convertido para um formato string. Por fim o evento recebido é adicionado à lista de eventos do Conector de Interação de modo a ser retornado para o consumidor no momento oportuno.

5.5 Comparações Qualitativas

O objetivo principal do modelo Sinapse é oferecer uma contribuição qualitativa para o desenvolvimento de aplicações baseadas em *Publish/Subscribe*. Assim, o modelo foi concebido com o intuito de incluir facilidades quer permitissem simplificar o desenvolvimento baseado nesse paradigma.

Vamos comparar os modelos de desenvolvimento a fim de apresentar os benefícios da utilização do modelo Sinapse. Analisaremos, a seguir, cada um dos quatro quesitos escolhidos para comparação.

5.5.1 Interação com a infra-estrutura de comunicação

O modelo Rebeca não tem uma diretriz clara de como desenvolver o código para interação com a infra-estrutura de comunicação. Isto pode levar a uma desorganização onde o código da aplicação fica mesclado aos aspectos não-funcionais de comunicação. Essa mesclagem de código pode ser vista nas figuras 17 e 18 onde são apresentados trechos de código utilizados pelas aplicações para estabelecer a comunicação com o Sistema de Notificação Rebeca.

O modelo Sinapse inclui um mecanismo que permite que as interações sejam feitas através de Conectores de Interação especializados, tornando as interações transparentes sem interferir no código da aplicação. Com isso, os aspectos não funcionais referentes à interação com a infra-estrutura de comunicação ficam concentrados nos Conectores de Interação de maneira bem definida e determinada. Como podemos ver nas figuras 21 e 23 onde são apresentados trechos de código utilizados pelas aplicações para estabelecer a comunicação com o Sistema de Notificação no modelo Sinapse. Esse código fica nos Conectores de Interação.

5.5.2 Assinatura de eventos

O modelo Rebeca não apresenta nenhuma diretriz quanto ao lugar mais apropriado para a declaração de filtros e realização de assinatura de eventos. Assim, o programador pode declarar os filtros juntamente com o código necessário para realizar a assinatura. Dependendo da complexidade do filtro isso pode tornar o código difícil de manter, além de diminuir consideravelmente a legibilidade. Na figura 18 apresentamos um trecho de código utilizado para realizar uma assinatura, nele uma instância da classe `FiltroNome` é passada como filtro a ser usado. Embora a definição do filtro seja realizada numa classe à parte do consumidor, sua instanciação faz parte do código de um módulo da aplicação.

No modelo Sinapse, foi adotada a idéia de repositório de filtros. Utilizando esse componente, os filtros são criados dentro deste repositório e são acessados pelos Conectores de Interação através dos recursos de configuração do ambiente. As referências são feitas através do nome dos filtros, tornando o código dos módulos consumidores mais claro. Como os nomes dos filtros são *strings*, podem ser determinados de forma a representar o seu objetivo diminuindo a necessidade do programador ter que conhecer os filtros através de seus detalhes. Conforme apresentado na figura 20 os filtros e composições do sistema são instanciados e armazenados no repositório de filtros, passando a ser referenciados por seu identificador único. Na figura 23 são apresentadas duas versões do método `assinar()` responsável por realizar a assinatura dos eventos. Este método faz parte do Conector de Interação, isentando o consumidor da responsabilidade de realizar a assinatura.

5.5.3 Composição de Eventos

A composição de eventos tem um suporte simples no ambiente Rebeca[Fie2003a], através de um mecanismo chamado multifiltro onde, é possível agrupar filtros que vão funcionar de forma conjunta. Um evento deve satisfazer a todos os filtros para ser considerado válido. Contudo, a expressão de composições de eventos através de multifiltros com condições mais sofisticadas pode tornar-se muito extensa, e isto pode dificultar a utilização desse recurso. Na figura 19 é apresentado um trecho de código usado para criar um multifiltro com três

condições, são necessários três filtros que são relacionados através de outros dois filtros especiais chamados `ANDFilter` usados para criar uma interseção entre dois filtros.

No modelo Sinapse, o suporte a composição de eventos foi isolado nos Conectores de Composição e Filtragem e o seu processamento acontece de forma independente. A expressão de composições é feita em função dos filtros definidos no repositório de filtros do sistema, e cada filtro pode ser satisfeito de forma independente dos demais. Quando um evento é recebido para ser processado, o mecanismo de composição verifica e armazena o evento junto de cada filtro que foi satisfeito. No momento em que as condições lógicas entre os filtros da composição torna-se válida, um *metaevento* é gerado e enviado para os destinos.

Esta abordagem adotada no modelo Sinapse permite grande flexibilidade na expressão de composições de eventos, além de permitir a geração de um *metaevento* contendo todos os eventos que foram usados. O processamento, sendo isolado do ambiente, permite aperfeiçoar o algoritmo usado no reconhecimento de composição de forma independente. Na figura 20 uma composição é criada no repositório de filtros do sistema em função dos filtros existentes. As operações lógicas entre os filtros são expressas juntamente com o filtro correspondente.

5.5.4 Continuidade em presença de falhas na infra-estrutura de comunicação

O ambiente Rebeca [Fie2003a] não permite reconfigurar de forma dinâmica a infra-estrutura de comunicação, ou seja, na presença de uma falha na rede, acontece um particionamento que não pode ser contornado. Para corrigir o problema torna-se necessário parar todo o sistema encerrando a execução de todos os *brokers* e recriando a rede. O modelo Sinapse, por outro lado, conta com o suporte adequado para reconfigurar a infra-estrutura de comunicação dinamicamente, uma vez que a topologia do sistema é configurada e mantida através do Ambiente de Configuração. Este suporte foi disponibilizado, mas ainda não há o gerenciamento automatizado para contornar as situações de falha.

5.6 Cenários de Aplicação de Composição de Eventos

Na *seção 2.3*, foram apresentados os detalhes relativos à Composição de Eventos, suas características e os mecanismos necessários para o suporte dessa funcionalidade. Nesta seção, serão discutidos dois cenários de aplicação que fazem o uso da Composição de Eventos para atingir um determinado objetivo de forma mais simples. Estes cenários não foram implementados. O objetivo, aqui, é a apresentar a utilização dos conceitos do modelo Sinapse para resolver situações reais.

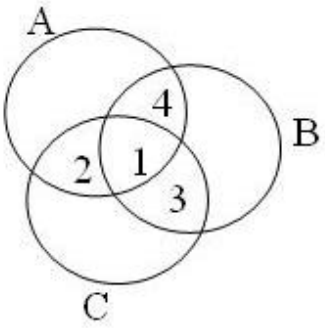
A Composição de Eventos é uma ferramenta que permite ao programador delegar a lógica de processamento de eventos a um tipo especial de filtro. Esse mecanismo permite reconhecer determinadas situações que são expressas em função de operações lógicas sobre os atributos dos eventos e entre os filtros que fazem parte da Composição de Eventos.

5.6.1 Sistema de Informação

Considera-se agora o contexto de um sistema de informação que controla as vendas de uma dada empresa, e que essa empresa trabalhe com vendas em filiais em outros estados. Assim, as informações sobre vendas em outros estados, vendas acima de R$ 10.000,00 e vendas para clientes novos, são publicadas como eventos simples. Para que haja uma correlação, estes eventos devem ser processados pelo consumidor interessado em todas as vendas de compradores novos de outros estados acima de R$ 10.000,00. A quantidade de eventos descartados pode ser muito maior que a quantidade de eventos que no final estarão compondo a informação desejada.

Podemos então criar um filtro que só notifica o consumidor quando as três condições acontecerem conjuntamente e, assim, o processamento será executado quando realmente tiver as informações necessárias para tal. Uma outra vantagem, é que se quisermos mudar a regra de negócio, só precisamos mexer em um componente isolado, responsável por esse reconhecimento, não sendo necessário alterar o consumidor. Isto seria muito útil no caso de incluir mais uma restrição ou modificar o valor de interesse.

Na figura 24, apresentamos um diagrama para visualizar as diversas combinações. Somente a área 1 é de interesse.



A – Compras maiores que R$ 10.000,00

B – Compras fora do estado

C – Compradores novos

1 – Compradores novos, de fora do estado e acima de R$ 10.000,00

1 + 2 – Compradores novos e acima de R$ 10.000,00

1 + 3 – Compradores novos fora do estado

1 + 4 – Compradores fora do estado e acima de R$ 10.000,00

Figura 24 – Exemplo interseção de informação

5.6.2 Bolsa de Valores

Este exemplo tem dois objetivos i) mostrar as três possibilidades de composição de eventos resolvidas pela proposta e ii) servir como base de entendimento, uma vez que esse exemplo é apresentado como referência na literatura do assunto.

O exemplo é composto por dois produtores, *P1* e *P2*, responsáveis por publicar eventos de empresas distintas: *Xtech*, *Ytech* e *Ztech*. Além disso, contamos com três consumidores, *C1*, *C2* e *C3*, cada um interessado num tipo de informação que pode ser composta ou simples.

As *tabelas 3* e *4* apresentam a distribuição de filtros e composições por consumidor:

Tabela 3 – Distribuição de filtros

Consumidor que utiliza o filtro	Filtros sobre ações das empresas		
	F1	F2	F3

1	Xtech>0.03		
		Ytech=0.02	
			Ztech<0.1

Tabela 4 – Distribuição de composições[4]

Consumidor que utiliza a composição	Composições baseadas nos filtros	
	C1	C2
2	(F1 ^ F2)	
3		(F1 ^ F3)

A seguir, apresentamos três figuras que mostram os eventos publicados e a resolução das situações de composição e filtragem de eventos. Esses três casos vão apresentar as possibilidades que ocorrem em uma aplicação baseada no paradigma *Publish/Subscribe*. A partir desses três casos, podem ser geradas todas as combinações possíveis de trabalho com eventos e composição de eventos. Por questões de simplicidade nas figuras são apresentados apenas os Conectores de Composição e Filtragem (CCF) responsáveis por resolver cada um dos três casos.

[4] Cabe ressaltar que as composições de eventos podem contar outros operadores lógicos além da operação "^" – "E". Neste exemplo foi utilizado apenas o operador "^" por questões de simplicidade.

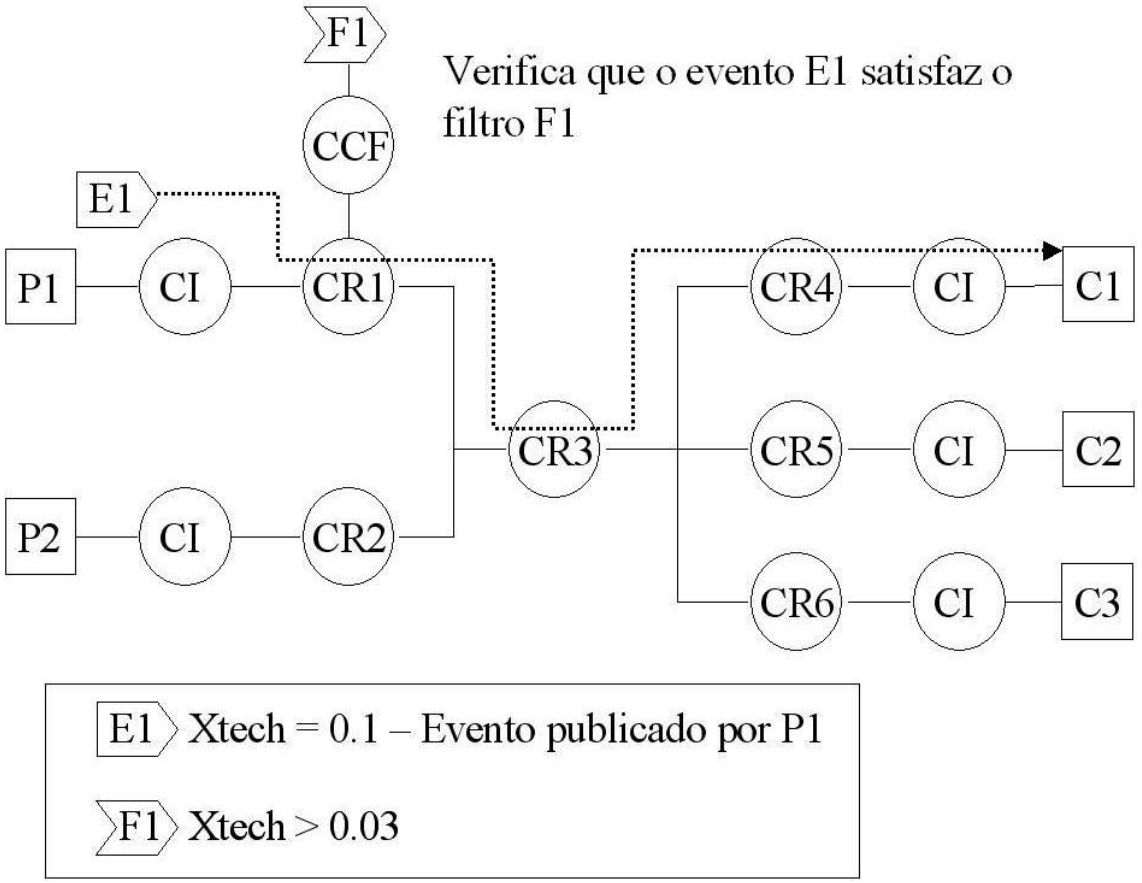


Figura 25 – Evento simples entregue diretamente

A figura 25 apresenta o primeiro caso, onde um evento simples satisfaz um filtro e é entregue diretamente através da infra-estrutura de comunicação ao consumidor interessado. Quando o Conector de Interação do consumidor *C1* realiza a assinatura de eventos, indica que estará usando o filtro *F1*, esse filtro é usado para estabelecer as rotas de entrega pela infra-estrutura de comunicação. Dessa maneira, quando o evento *E1* é publicado pelo Conector de Interação do produtor *P1*, a entrega acontece através da rota criada. De acordo com o modelo Sinapse, os filtros são mantidos nos Conectores de Composição e Filtragem e são processados da mesma forma que no ambiente Rebeca original.

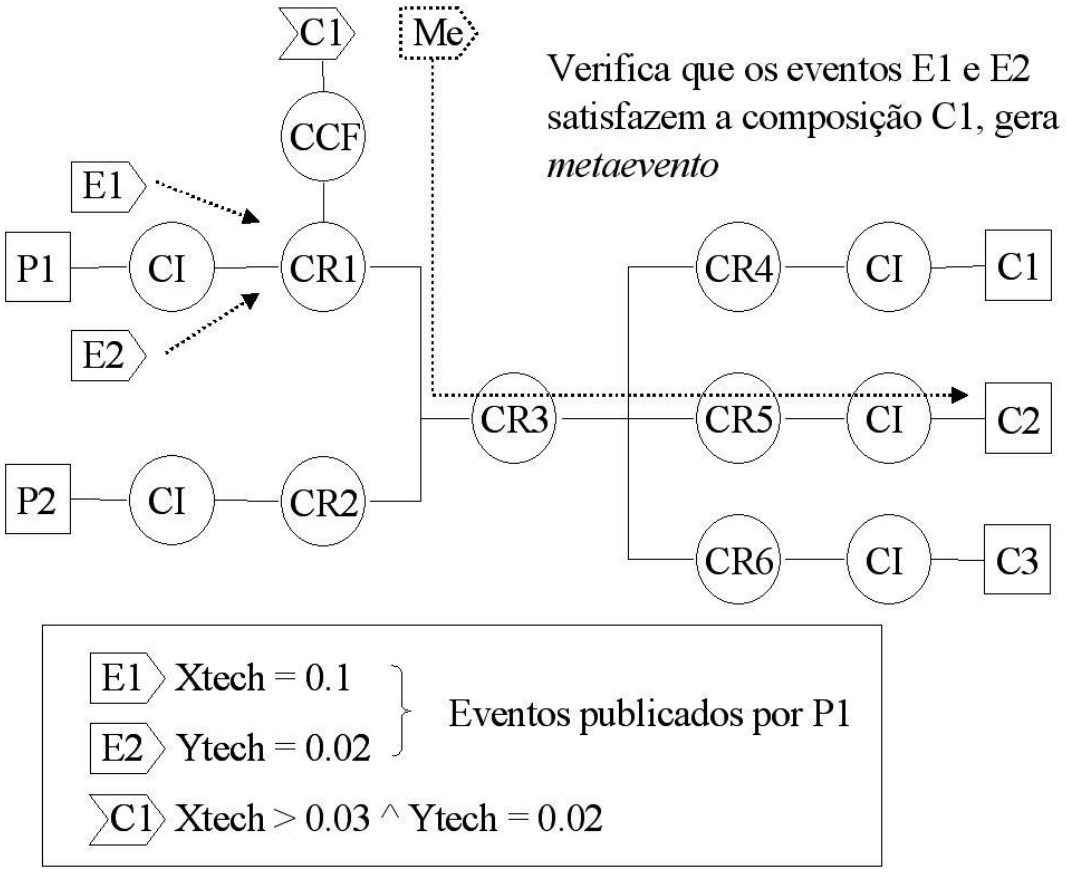


Figura 26 – Composição de eventos da mesma fonte

Na figura 26, vemos o segundo caso, onde uma composição de eventos é usada na assinatura do consumidor *C2*. Depois que o Conector de Composição e Filtragem identifica que os eventos *E1* e *E2* satisfazem às condições para a composição *C1*, é gerado um *metaevento* que segue para ser entregue. Neste caso, os eventos que satisfazem às condições são originários da mesma fonte, que é o produtor *P1*.

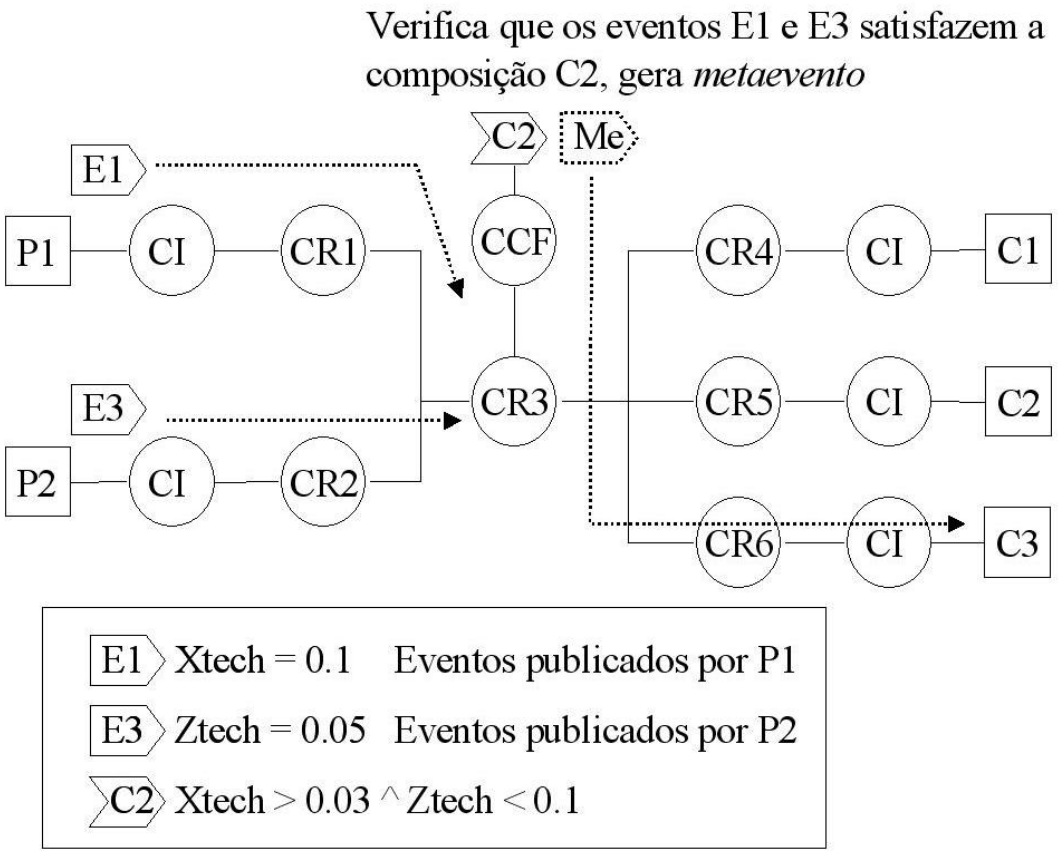


Figura 27 – Evento simples entregue diretamente

Por fim, a figura 27 apresenta o terceiro e último caso. Aqui os eventos *E1* e *E3* são originados em fontes distintas. A composição de eventos pode ser satisfeita por eventos simples ou por outras composições anteriores. Com isso esse caso permite identificar a grande utilidade de descrever as composições em termos dos filtros. A lógica já utilizada para descrever uma composição anterior pode ser reaproveitada para criar uma nova composição mais sofisticada.

Utilizando este mesmo raciocínio, em um cenário diferente, poderíamos criar uma nova composição *C3* a partir das composições já existentes *C1* e *C2*. Dessa maneira as operações lógicas presentes nas duas composições são herdadas, não sendo necessário redefini-las.

De acordo com as figuras, podemos perceber os seguintes *casos primordiais de composição*:

i. O evento é entregue diretamente: o cliente *C1* está interessado somente no evento *E1* e não precisa realizar nenhuma composição;

ii. O evento de interesse é uma composição de dois eventos de um mesmo produtor: dessa maneira, o Conector de Composição e Filtragem associado ao Conector de Interação do produtor resolve a composição e despacha o *metaevento* para o consumidor *C2*;

iii. O evento de interesse é uma composição de fontes distintas. Neste caso, o Conector de Composição e Filtragem que o resolve está associado a um Conector de Roteamento situado num ponto de interseção na rota de entrega para o consumidor *C3*. É importante notar que esta composição pode ser baseada em eventos simples e/ou outras composições pré-existentes.

Não foram ilustrados aqui os casos onde mais de um consumidor estaria interessado em um mesmo evento ou *metaevento*. Isto porque no caso de múltiplos destinos a única diferença estaria que a própria infra-estrutura de comunicação se encarregaria de entregar os eventos (ou metaeventos) em múltiplos destinos.

5.7 Conclusão Parcial

Foram apresentados critérios para avaliação qualitativa das funcionalidades disponibilizadas pelo modelo Sinapse. Para realizar esta avaliação, o modelo Sinapse foi comparado ao modelo tradicional através da implementação de duas aplicações de teste.

Também foram apresentados cenários de aplicações que podem ser implementados usando o paradigma *Publish/Subscribe*. Foram discutidas e apresentadas as vantagens de usar as facilidades do modelo Sinapse como suporte para as necessidades das aplicações.

6 Trabalhos Relacionados

No capítulo 5, foram apresentados alguns cenários de aplicações que servem de referência para casos que se beneficiam do uso do paradigma *Publish/Subscribe*.

Para um melhor entendimento de como foram usados os conceitos referentes ao modelo Sinapse, foi feita uma separação em seções distintas. Assim, os trabalhos que foram usados como base para cada característica desenvolvida na proposta, ficam classificados de acordo com a idéia central à qual se relacionam. Dessa forma, torna-se clara a influência que os conceitos referenciados tiveram sobre a evolução do modelo Sinapse.

6.1 Mobilidade

A mobilidade foi apresentada como uma característica importante para a qual os Sistemas de Notificação devem oferecer o suporte próprio. A proposta do nosso modelo, em relação à mobilidade, é utilizar as características de interceptação do modelo Sinapse para alcançar o suporte à mobilidade, mas essa facilidade não se encontra disponível no protótipo.

Foram comparados Sistemas de Notificação com suporte à mobilidade com diferentes graus de maturidade, tendo sido a arquitetura Rebeca [Fie2003a] a que mais se destacou.

Em [CCW2003], é apresentada a solução considerada para o ambiente Siena, que se baseia em primitivas de conexão e desconexão explícitas. Assim, a aplicação precisa indicar que irá se desconectar do sistema. Essa abordagem, embora também seja usada em alguns outros sistemas, não é uma abordagem que possa ser usada amplamente fora de um ambiente controlado de testes. Com a utilização de primitivas explícitas, a aplicação passa a ser responsável por uma ação sobre a qual não tem controle. Não é possível prever quando ocorrerá uma desconexão por falta de sinal, por exemplo. Assim, esse tipo de abordagem pode ser considerado como experimental.

Já a solução apresentada em [Fie2003a] para o ambiente Rebeca, baseia-se em um modelo reativo. Quando um consumidor se desconecta de um ponto de acesso, seus eventos vão sendo acumulados em um *buffer*, até que, num dado momento posterior, aconteça a reconexão. Neste momento, em que o dispositivo se torna acessível, os eventos acumulados são encaminhados para a nova posição e a rota de acesso é reconstruída.

A proposta do modelo Sinapse se assemelha à encontrada em Rebeca, a partir do uso de *buffers* para armazenar os eventos que não podem ser entregues. Além disso, o novo Conector de Interação refaz as assinaturas do consumidor, o que inválida as rotas que levavam até a localização anterior.

6.2 Composição de Eventos

A Composição de Eventos representa uma evolução para os Sistemas de Notificação, porque permite a reutilização de regras de filtragem mais sofisticadas sem que seja necessário reescreve-las. Contudo, os ambientes estudados apresentam um suporte muito limitado a essa funcionalidade.

Em [CRW2000, PSB2003, Fie2003a] são apresentadas propostas para a Composição de Eventos que funcionam de forma distinta. Em Siena [CRW2000] e em Rebeca[Fie2003a], a composição é representada por uma estrutura que agrupa filtros simples, enquanto em Hermes [PSB2003] é tratada por módulos especiais implantados nos *brokers*. Estes módulos implantados nos *brokers* avaliam as composições e notificam os consumidores interessados nos *metaeventos*. A abordagem utilizada em Hermes motivou o desenvolvimento, em nossa proposta, de um tratamento independente dos *brokers*.

Um ponto de convergência entre os ambientes estudados, está no fato de que a entrega de eventos não está baseada em um *metaevento*, ou seja, não existe a geração de um novo objeto que represente explicitamente a resolução de todas as restrições declaradas na composição. Esta capacidade de gerar um *metaevento* foi introduzida em nosso modelo

através do isolamento do processamento de composições nos Conectores de Composição e Filtragem. Estes conectores são especializados no processamento de composições e filtros, de modo que quando uma composição é reconhecida, é realizada a geração e envio de um *metaevento* que vai ser entregue aos consumidores interessados. Isso acontece de forma transparente para o Sistema de Notificação, que realiza a entrega baseado em seus mecanismos tradicionais.

6.3 Modelos de Arquitetura

O último ponto importante do modelo Sinapse é a utilização de um modelo de arquitetura para expressar os componentes do sistema distribuído. Utilizando um Ambiente de Configuração para controlar a execução, tornou-se possível utilizar recursos de interceptação, permitindo inserir os pontos de transparência apresentados no trabalho.

Uma proposta baseada na arquitetura Siena [CRW2000] permite de maneira dinâmica, trocar as conexões entre *brokers* de modo a reconstruir a rede lógica de entrega de eventos. Isso é feito por um conjunto de módulos que agem diretamente sobre os *brokers*, a fim de mudar as suas conexões. Assim, se o *broker* se torna inacessível, a reconexão pode ser realizada criando um caminho alternativo através de um outro *broker*. Essa abordagem, porém não apresenta um modelo do Sistema de Notificação, usando-o apenas como um exemplo na utilização dos módulos de monitoração. Nossa proposta é conceitualmente semelhante, entretanto vai um passo além, utilizando um mapeamento do Sistema de Notificação em termos de conectores. Com isso, torna-se possível expor outros pontos do ambiente, explorando os recursos de transparência e ortogonalidade.

Um tópico de interesse conceitual é o de *Active Networks*, onde temos uma evolução sobre as redes atuais. Em [Ten1997], a idéia é que os pacotes de rede sejam substituídos por cápsulas que funcionam como mini programas. Essas cápsulas são transmitidas e podem ser executadas em cada nó da rede. Essa proposta implica em mudanças evolutivas também no *hardware*.

Uma vez atingido um determinado ponto da rede, uma cápsula pode utilizar as funcionalidades presentes ou instalar novas funcionalidades. Essa idéia difere dos Sistemas de Notificação, onde os eventos não têm capacidade de executar qualquer funcionalidade. No entanto, o transporte executado pela infra-estrutura de comunicação pode ser comparado a esse transporte ativo, já que os *brokers* e filtros executam certos procedimentos baseados nas informações contidas nos eventos.

Podemos considerar que em um dado momento de evolução das redes ativas, poderíamos substituir a rede *overlay* de *brokers*, por funcionalidades que seriam instaladas diretamente em *hardware*, embora essa possibilidade ainda esteja longe. Essas funcionalidades ainda poderiam ser controladas a partir do modelo que propomos, se considerarmos módulos instalados nos roteadores de rede, o que permitiria seu mapeamento em conectores. Essa possibilidade reforça o poder de abstração do modelo Sinapse.

Outra idéia que foi utilizada em nosso modelo é a de transparência na interação entre os módulos externos da aplicação e a infra-estrutura de comunicação. Essa facilidade é disponibilizada através do uso dos Conectores de Interação que agem de forma semelhante à capacidade de *cross-cutting* de *Aspect Oriented Programming*. Em [Cil2003], é apresentado o contexto evolutivo das relações entre AOP e bancos de dados ativos. Essa idéia vai ao encontro a conceitos que foram utilizados em nosso modelo, que são o de usar a interceptação para conseguir isolar os aspectos funcionais e não-funcionais do sistema. Assim, foi possível chegar a uma implementação onde conseguimos não só a transparência do ponto de vista externo, entre a aplicação e o sistema de infra-estrutura, como também dentro do próprio sistema de infra-estrutura, quando tratamos as composições de eventos.

6.4 Conclusão Parcial

Neste capítulo foram associados os conceitos referentes ao modelo Sinapse às características de outras propostas, mostrando as origens de algumas idéias. Foram discutidas e apresentadas as idéias que deram origem ao modelo e como o trabalho desenvolvido se relaciona com outras propostas na área.

As propostas foram classificadas de acordo com o tipo de funcionalidade do modelo Sinapse com o qual estavam relacionadas. Essa organização permitiu uma apresentação mais específica dos detalhes de cada item.

7 Conclusão

Neste trabalho foi apresentado um modelo para Sistemas de Notificação que tem por objetivo simplificar o desenvolvimento de aplicações distribuídas baseadas no paradigma *Publish/Subscribe*.

7.1 Contribuições do Modelo Sinapse

Nesta seção vamos apresentar as contribuições que o modelo proposto oferece de acordo com os objetivos apresentados inicialmente. As facilidades incluídas no modelo são a base para os tópicos que são apresentados aqui. As contribuições desse trabalho são apresentadas a seguir :

i. Arquitetura configurável para um Sistema de Notificação. A configuração da infra-estrutura de comunicação se dá através dos mecanismos de interceptação. A adaptação dos componentes do Sistema de Notificação Rebeca ao Ambiente de Configuração AC possibilita intervir sobre os componentes da infra-estrutura de comunicação. Facilita assim, o suporte a tolerância a falhas através da reconfiguração dinâmica, onde a topologia de componentes pode ser alterada para contornar problemas de conexão.

ii. Uso de técnicas de interceptação (programação em meta nível) para tornar viável o interfaceamento com o Sistema de Notificação por parte dos módulos produtores e consumidores. Esta característica permite isolar o código dos módulos da aplicação, do código necessários para realizar a comunicação entre esses módulos. Muitos dos detalhes referentes as interações são isolados num componente especializado e bem definido. Assim, as manutenções tornam-se mais fáceis de gerenciar do ponto de vista do acoplamento.

iii. Disponibilizar um mecanismo de reconhecimento de composições de eventos com a capacidade de gerar *metaeventos*, separando essa tarefa do roteamento.

Tradicionalmente os componentes responsáveis por manter as rotas de comunicação também aplicam os filtros aos eventos recebidos. Utilizando os recursos de interceptação, o processamento de composições e filtros foi isolado no Conector de Composição e Filtragem. Com isso, o processamento de composições de eventos torna-se bem definido e flexível, incluindo recursos que facilitam a expressão das composições, além de gerar e entregar de *metaeventos*. Com o mecanismo de composição de eventos independente do Sistema de Notificação usado passa a ser possível a modificação e evolução dos Conectores de Composição e Filtragem de forma isolada. Potencialmente isso pode facilitar estender Sistemas de Notificação existentes que não tenham suporte à composição de eventos.

iv. O modelo Sinapse pode ser considerado mais robusto que o modelo centralizado de desenvolvimento por permitir que o sistema escale para maior poder de processamento a partir da inclusão de novos componentes. Graças às facilidades de administração fornecidas pelo Ambiente de Configuração, torna-se mais simples incluir novos nós de processamento à infra-estrutura de comunicação do sistema distribuído. Com isso, aumentar o poder de processamento da aplicação torna-se mais simples e barato, pois pode ser feito a partir da inclusão de mais máquinas baratas. Em contraste com o modelo centralizado que depende do poder de processamento de uma máquina mais poderosa.

7.2 Benefícios do Uso do Modelo Sinapse

Lista-se, a seguir, os benefícios do uso do modelo Sinapse:

i. Melhor organização no desenvolvimento de aplicações isolando os aspectos funcionais dos não-funcionais.

ii. Diminuição da dependência do Sistema de Notificação por parte da aplicação.

iii. Facilidade na expansão da aplicação através da inclusão de novos módulos dinamicamente.

iv. Facilidade de intervenção nas ligações entre os componentes da aplicação, permitindo alterá-las.

v. Tratamento de eventos simplificado a partir do processamento realizado pelos Conectores de Interação. O módulo consumidor recebe as informações do evento já processadas, num formato pré-definido. Isto evita que o consumidor precise lidar com objetos que não fazem parte do domínio da aplicação.

7.3 Restrições do Uso do Modelo Sinapse

A seguir, apresentamos as restrições relacionadas ao uso do modelo Sinapse:

i. Devido a abstração das interações com a infra-estrutura de comunicação através de conectores, o desenvolvimento de aplicações passa a ser feito a partir de uma visão arquitetural. Isso muda a maneira como o programador escreve os módulos da aplicação, pois a aplicação passa a ser programada em um meta-nível.

ii. Embora o custo predominante seja o de utilização da rede, usar mecanismos de interceptação introduz um custo local de processamento em relação às interceptações. No entanto, este custo tende a ser pequeno considerando o aperfeiçoamento dos mecanismos de interceptação.

7.4 Problemas Identificados

Durante a concepção do modelo Sinapse, foram encontrados diversos tipos de questões que não foram completamente resolvidas. Algumas delas estão dentro do escopo deste trabalho, enquanto outras, devem ser resolvidos em outros contextos de trabalho.

i. Distribuição dos *brokers* na rede física: um problema que não é discutido na literatura é a disposição dos componentes de roteamento na rede física. Embora os Sistemas de Notificação disponibilizem os recursos necessários para iniciar esses componentes, não existe uma diretriz sobre como a distribuição desses componentes pode influenciar no desempenho do sistema. Para nosso modelo, consideramos que uma posição inicial é conhecida.

ii. Armazenamento de eventos que não podem ser entregues por falta de rota válidas: em situações de particionamento da infra-estrutura de comunicação ou de não haver consumidores interessados em seu conteúdo, os eventos publicados acabam por serem descartados quando atingem um ponto de onde não podem prosseguir. Na versão do Sistema de Notificação Rebeca utilizada no desenvolvimento do protótipo, quando os eventos não são entregues a algum consumidor, acabam sendo descartados. Isto acontece por não haver um mecanismos que permita persistir os eventos. Este problema pode ser corrigido com a utilização de *buffers*, conforme proposto em [Fie2003a].

iii. Ordenação e atraso na entrega de eventos: este problema está relacionado com uma característica dos sistemas distribuídos, que é a falta de sincronização de relógios. Cada máquina participante pode sofrer pequenas variações em seu relógio interno, o que gera uma imprecisão quanto a utilização de um rótulo de tempo global. Isso dificulta determinação de que evento foi gerado primeiro para poder ordená-los no ponto de entrega. Esse tipo de problema deve ser tratado pelo Sistema de Notificação.

7.5 Trabalhos Futuros

Nesta seção apresentamos as direções de aperfeiçoamento do modelo Sinapse. Algumas linhas de trabalho podem ser derivadas a partir da lista de questões que foram identificadas e

ainda não foram tratadas. Basicamente, essas questões estão num segundo nível, sendo necessário completar a base do modelo para poder, então, passar aos aperfeiçoamentos.

i. Avaliação de desempenho da implementação de referência do modelo Sinapse. Para melhorar o desempenho do modelo é necessário trabalhar na forma de execução do Ambiente AC e incluir otimizações, isso vai permitir diminuir a sobrecarga gerada e melhorar o desempenho final.

ii. Adaptação de outros Sistemas de Notificação já existentes a fim de comprovar o potencial do modelo Sinapse de fornecer capacidades semelhantes as atingidas no protótipo desenvolvido com o sistema Rebeca.

iii. Desenvolvimento de um Sistema de Notificação *from scratch* com a inclusão do suporte as características do modelo Sinapse desde o início. Isso permitiria atingir uma implementação mais eficiente por não precisar lidar com as características de um Sistema de Notificação já existente.

iv. Disponibilizar a propagação de *callback* do Conector de Interação ao consumidor. Para tanto, o ambiente AC deve ser modificado para permitir que um conector acesse uma porta de um módulo.

v. Disponibilizar *buffers* de armazenamento de eventos de forma transparente através de mecanismos de interceptação, semelhantemente ao usado para os Conectores de Composição e Filtragem.

vi. O repositório de filtros da implementação de referência é centralizado, e isso diminui a escalabilidade do modelo. Deve existir um mecanismo capaz de replicar o repositório de filtros mantendo sua integridade, permitindo escalar para aplicações maiores.

vii. Automatizar o processo de tratamento de falhas através dos mecanismos de configuração. Na versão atual, a alteração na topologia é realizada manualmente, através do console de comandos. Essa funcionalidade deve ser delegada a um módulo capaz de identificar uma falha e, utilizando os mecanismos do modelo Sinapse, contornar o problema.

7.6 Comentários Finais

Neste capítulo foi apresentado um resumo sobre o trabalho que foi desenvolvido, uma lista de problemas que foram encontrados durante o desenvolvimento e uma lista de trabalhos futuros, apresentando os possíveis caminhos de aperfeiçoamento do modelo Sinapse.

Além disso, foram apresentadas as contribuições do modelo, com um detalhamento de como permitem uma melhoria do ponto de vista qualitativo, em relação ao modelo tradicional. O desenvolvimento, bem sucedido da implementação de referência, atinge o objetivo da apresentar a viabilidade do modelo Sinapse, provando assim a hipótese sugerida inicialmente.

Assim, o modelo Sinapse apresenta-se como uma alternativa ao modelo tradicional para o desenvolvimento de aplicações baseadas no paradigma *Pulbish/Suscribe*, tornando esse tipo de desenvolvimento mais rápido, simples e eficiente. Ainda são necessárias, no entanto, melhorias em termos de desempenho para permitir a utilização deste modelo num ambiente que não seja o laboratorial.

8 Referências Bibliográficas

[Car2001] CARVALHO, S. ***Um Design Pattern Para a Configuração de Arquiteturas de Software***. Dissertação de Mestrado. Niterói, RJ : UFF, 2001

[Cas2003] CASTALDI, M. et al. ***A Lightweight Infrastructure for Reconfiguring Applications***. SCM 2001/2003, Portland, Oregon : Springer-Verlag, 2003, LNCS v. 2049, p. 231-244

[CCW2003] CAPORUSCIO, M; CARZANIGA, A.; WOLF, A.L. ***Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications***. IEEE Transactions on Software Engineering, Dezembro 2003, 29 v em 12 p. 1059-1071.

[Cil2003] CILIA, M. et al. ***The convergence of AOP and active databases: towards reactive middleware***. Proceedings of the second international conference on Generative programming and component engineering, New York, NY, USA : Springer-Verlag, 2003, LNCS v. 2830, p.169-188

[CRW2000] CARZANIGA, A.; ROSENBLUM, D.S.; WOLF, A.L. ***Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service***. Proceedings of the Nineteenth Annual Symposium on Principles of Distributed Computing, Portland, Oregon : [s.n.], 2000, p. 219-227.

[Dey2001] DEY,A.K. ***Understanding and Using Context***. Personal and Ubiquitous Computing, London, UK : Springer-Verlag, 2001, v. 5 em 1, p. 4-7

[EGD2001] EUGSTER, P.TH.; GUERRAOUI, R.; DAMM, C.H. ***On objects and events***. Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY : ACM Press, 2001, p 254-269.

[Eug2003] EUGSTER, P.Th. et al. ***The many faces of publish/subscribe***. ACM Computing Surveys (CSUR), New York, NY : ACM Press, 2003, v. 35 em 2, p. 114-131

[Fie2003a] FIEGE, L. et al. ***Supporting Mobility in Content-Based Publish/Subscribe Middleware***. Proc. of the 4th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware '03), Rio de Janeiro : Springer, 2003, p. 103-122.

[Fie2003b] FIEGE, L. et al. ***Dealing with Uncertainty in Mobile Publish/Subscribe Middleware***. 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing, [S.I.]:[s.n.], 2003. Disponível em: <http://lpdwww.epfl.ch/upload/documents/publications/neg--566851379MPAC.pdf>. Acesso em: jan. 2005.

[FRH2004] FOK, C.-L.; ROMAN, G.-C; HACKMANN, G. ***A Lightweight Coordination Middleware for Mobile Computing***. Proceedings Coordination Models and Languages, 6th International Conference, Pisa, Itália:Springer, 2004, LNCS v. 2949, p.135-151

[GH2002] GUERRAOUI, R.; HARI, C. ***On the consistency problem in mobile distributed computing***. Proceedings of the second ACM international workshop on Principles of mobile computing, New York, NY, USA : ACM Press, 2002, p. 51-57

[Har2002] HARTER, A. et al. ***The Anatomy of a Context-Aware Application***. Wireless Networks, [S.I.]:[s.n.], 2002, v. 8 em 2-3, p. 187-197

[HJ1999] HAUSWIRTH, M.; JAZAYERI, M. ***A Component and Communication Model for Push Systems***. Londres, UK : Springer-Verlag, 1999, 18 p.

[HJ2003] HAWICK, K.A.; JAMES, H.A. ***Middleware for context sensitive mobile applications***. Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003, Darlinghurst, Australia : Australian Computer Society, 2003, v. 21, p. 133-141

[JS2003] JIN, Y.; STROM, R. ***Relational subscription middleware for Internet-scale publish-subscribe***. Proceedings of the 2nd international workshop on Distributed event-based systems, New York, NY, USA : ACM Press, 2003, p. 1-8

[Lis2003] LISBÔA, J.C. ***Utilização do Design Pattern Architecture Configurator em um Ambiente de Suporte à Configuração de Arquiteturas***. Exame de Tese de Mestrado. Niterói, RJ : UFF, 2003

[Loq2004] LOQUES, O. et al. ***A contract-based approach to describe and deploy non-functional adaptations in software architectures***. Journal of the Brazilian Computer Society, [S.I.]:[s.n.], 2004, v.10 em 1, p. 5-18

[Med1996] MEDVIDOVIC, N. ***ADLs and dynamic architecture changes***. Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, New York, NY, USA : ACM Press, 1996, p. 24-27

[Mit2000] MITCHELL, S. et al. ***Context-aware multimedia computing in the intelligent hospital***. Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system, New York, NY, USA : ACM Press, 2000, p. 13-18

[MR2003] MEYER, S.; RAKOTONIRAINY, A. ***A survey of research on context-aware homes***. Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003, Australia : Australia Computer Society, 2003, v. 21, p.159-168

[MRV2003] MURPHY, A. L.; ROMAN, G.-C.; VARGHESE, G. ***Dependable Message Delivery to Mobile Units***. Chapter to appear in Handbook of Mobile Computing,

[S.I.]:[s.n.], 2003. Disponível em: http://www.cse.wustl.edu/mobilab/Publications/ Acessado em: dez 2003.

[OMG2004] OMG. ***Common Object Request Broker Architecture: Core Specification***. 2004. Disponível em: http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf Acessado em: jan. 2005

[PB2002] PIETZUCH, P.R.; BACON, J.M. ***Hermes: A Distributed Event-Based Middleware Architecture***. Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02), Vienna, Austria : [s.n.], 2002, p. 611-618.

[Pie2002] PIETZUCH, P.R. ***Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems?***. 6th CaberNet Radicals Workshop, Funchal, Madeira Island, Portugal : [s.n.], 2002.

[PMR1999] PICCO, G.P.; MURPHY, A.L.; ROMAN G.C. ***LIME: Linda Meets Mobility***. Proceedings of the 21st International Conference on Software Engineering. [S.I.]:[s.n.], 1999, p. 368-377

[PSB2003] PIETZUCH, P.R.; SHAND, B.; BACON, J. ***A Framework for Event Composition in Distributed Systems***. Proc. of the 4th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware '03), Rio de Janeiro : Springer, 2003, p. 62-82.

[PZ1996] PRATT, T.W.; ZELKOWITZ, M.V. ***Programming Languages Design and Implementation***. 3 ed., Upper Saddle River, New Jersey : Prentice Hall, 1996, p.641

[RPM2000] ROMAN, G.-C.; PICCO, G.P.; MURPHY, A.L. ***Software Engineering for Mobility: A Roadmap***. The Future of Software Engineering, [S.I.]: ACM Press, 2000, p. 241-258.

[Sat1996] SATYANARAYANAN, M. ***Fundamental challenges in mobile computing***. Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA : ACM Press, 1996, p.1-7

[Szt1999] SZTAJNBERG, A. ***Flexibilidade e Separação de Interesses para Concepção e Evolução de Sistemas Distribuídos***. Exame de Tese de Doutorado. Rio de Janeiro, RJ : COPPE-UFRJ, 1999

[Szt2004] SZTAJNBERG, A. et al. ***Suportando Contratos de QoS no Nível Arquitetural***. Anais do VI Workshop de Tempo Real (WTR 2004), Gramado, RS, Brasil : Inst. Informatica UFRGS, 2004, p.123-130

[Ten1997] TENNENHOUSE, D.L. et al. ***A Survey of Active Network Research***. IEEE Communications Magazine, [S.L.]:[s.n.], 1997, v. 35 em 1, p.80-86

[TS2002] TANENBAUM, A.; STEEN, M. ***Distributed Systems Principles and Paradigms***. Upper Saddle River, New Jersey : Prentice Hall, 2002, p.785

[VGP2003] VOGT, H.; GÄRTNER, F.C.; PAGNIA, H. ***Supporting fair exchange in mobile environments***. Mobile Networks and Applications, Hingham, MA, USA : Kluwer Academic Publishers, 2003, v. 8 em 2, p. 127-136

[Wei1999] WEISER, M. ***The computer for the 21st century***. ACM SIGMOBILE Mobile Computing and Communications Review, [S.I.]:[s.n.], 1999, v.3 em 3, p. 3-11.

[ZF2003] ZEIDLER, A.; FIEGE, L. ***Mobility Support with REBECA***. Proceedings of the 23rd International Conference on Distributed Computing Systems, Washington, DC, USA : IEEE Computer Society, 2003, p 354

9 Apêndice

Nesta seção, vamos apresentar as classes necessárias à implementação do modelo proposto. Além dos componentes que formam o modelo, outras classes que desempenham papéis de suporte foram necessárias para o desenvolvimento de um protótipo funcional.

A implementação passou por duas etapas: a primeira foi o desenvolvimento de funcionalidades do ambiente de configuração AC para que fosse possível realizar as tarefas necessárias ao modelo. A segunda foi a adaptação do Sistema de Notificação Rebeca ao novo ambiente AC. Em paralelo a essas duas etapas, foi construída uma aplicação simples com o objetivo de testar cada uma das novas funcionalidades que foram incluídas.

A primeira etapa de desenvolvimento foi mais direta por se tratar de incluir o suporte para tarefas que já estavam bem determinadas. Essas tarefas foram definidas pelas necessidades do modelo, tais como a passagem de parâmetros para as instâncias e a capacidade de um módulo atuar como cliente e servidor de uma porta própria. Já a segunda etapa, embora menor, exigiu um maior grau de análise, por se tratar da adaptação de um Sistema de Notificação para trabalhar a partir das intervenções de um Ambiente de Configuração.

Apresentamos, a seguir, os detalhes referentes à implementação e adaptação das classes que são utilizadas junto aos ambientes de suporte a fim de criar o modelo Sinapse.

9.1 EventoAC

É uma classe derivada de `rebeca.Event`, com a diferença que a implementação utilizada no modelo tem por objetivo representar eventos de forma genérica. Isso é feito utilizando uma classe utilitária `EventField`, que é uma estrutura para conter os campos de um evento. Essa representação nos garante um evento que contém um número de campos inseridos

dinamicamente pelo Conector de Interação do produtor de acordo com a quantidade de informação a ser publicada.

A classe `EventField` é capaz de armazenar o nome do campo, seu tipo de dado e um valor de acordo com a informação produzida. Já a classe `EventoAC` pode armazenar um conjunto de objetos `EventField` ou um conjunto de objetos `EventoAC`. Isto acontece porque, quando criamos um evento comum, adicionamos campos para representar os atributos de evento. Porém, quando criamos um *metaevento*, devemos armazenar os eventos que o compõem. Além disso, um metavento carrega a informação de qual composição o gerou e um campo lógico, que é utilizado para indicar que o *metaevento* está completo.

Utilizando a classe `EventoAC`, conseguimos determinar um processamento uniforme tanto para filtros quanto para composições de eventos. Além disso, os *metaeventos* são roteados pelo sistema de forma idêntica aos eventos comuns, isto porque são encapsulados como um evento simples. Com isso, conseguimos utilizar os algoritmos de roteamento já presentes para entrega de *metaeventos*. A seguir apresentamos na figura 28 as classes descritas.

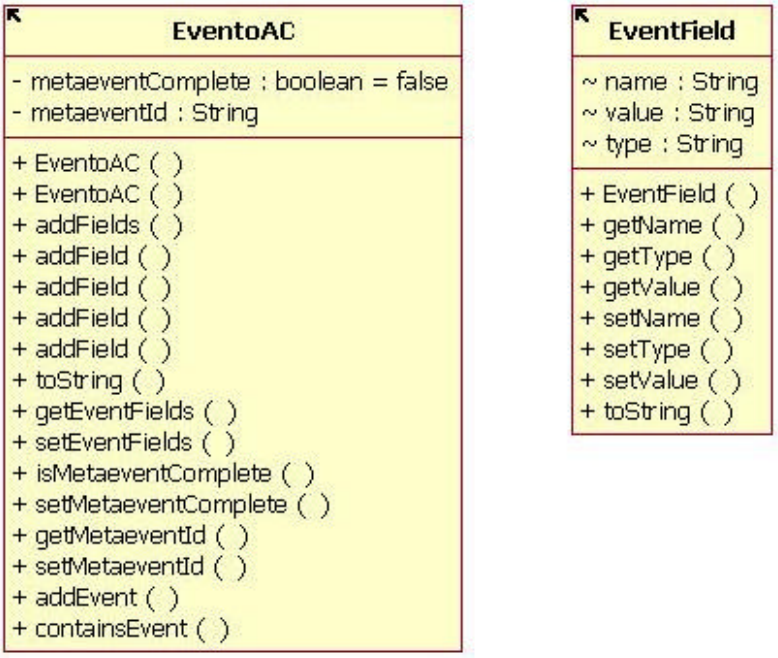


Figura 28 – Classes de eventos

9.2 FiltroAC

A classe `FiltroAC` é derivada da classe `rebeca.Filter`, e de forma semelhante à classe `EventoAC`, também contém um conjunto de objetos, porém estes são da classe `FilterConstraint`. Podemos considerar o `FilterConstraint` como uma classe simétrica à classe `EventField`. A classe `FilterConstraint` armazena um nome de campo, um valor e uma operação lógica que será utilizada para determinar se o valor recebido satisfaz à restrição.

Da mesma forma que um `EventoAC` pode conter vários campos, um `FiltroAC` também pode conter várias restrições. A primeira regra utilizada no processamento de eventos é verificar se o evento contém todos os campos que estão definidos nas restrições do filtro. Se

o filtro definir uma restrição que deve ser aplicada a um campo, mas o evento não contiver esse campo, então este evento não poderá satisfazer todas as restrições. A partir daí, o processamento se dá com base nas operações lógicas definidas em cada `FilterConstraint`, em relação ao valor presente no `EventField` de mesmo nome. A seguir, apresentamos na figura 29 as classes descritas.

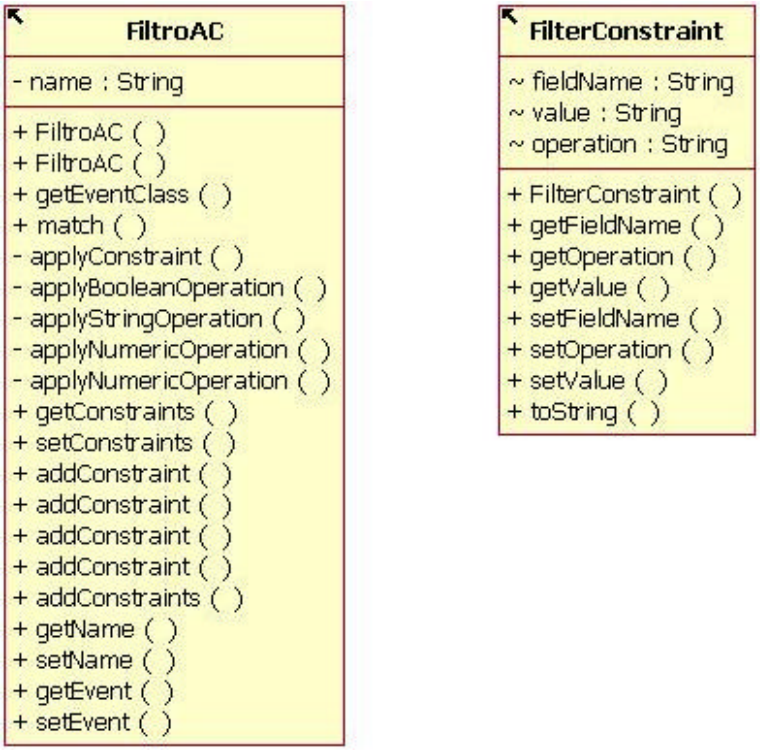


Figura 29 – Classes de filtros

9.3 ComposicaoAC

A classe `ComposicaoAC` também é derivada de classe `rebeca.Filter` e mantém um conjunto de objetos `CompositionField`. Cada um desses objetos mantém um filtro e uma operação lógica. Além disso, inclui uma referência para um objeto `EventoAC`. Esta referência é preenchida com o último evento recebido que satisfaça as restrições do filtro.

Desse modo, podemos encarar o mecanismo de composição de eventos como sendo uma máquina de estados. Cada evento que completa um filtro vai mudando o estado até que, em um dado momento, as operações lógicas determinadas entre os filtros são satisfeitas, atingindo-se um estado final.

Tendo desenvolvido o mecanismo de composição com base nos princípios de filtragem existentes no Sistema Rebeca, é possível utilizar todos os recursos de difusão de filtros já disponíveis. Esses recursos de difusão estão disponíveis no Sistema Rebeca e permitem enviar os filtros através da infra-estrutura de comunicação. Com isso, torna-se transparente a instalação de composições de eventos no Sistema de Notificação. As composições oferecem a mesma assinatura para o tratamento de eventos que os filtros comuns; contudo, o Conector de Composição e Filtragem realiza um passo extra que é a verificação e envio de *metaeventos*.

A concepção do suporte à composição de eventos baseado em módulos independentes se assemelha a idéia desenvolvida em Hermes[PSB2003]. No entanto, o modelo Sinapse difere de Hermes no sentido de que não há necessidade de uma camada entre o consumidor e o Sistema de Notificação. Em Hermes, essa camada é utilizada pelo consumidor com o objetivo de expressar as composições de eventos. No modelo Sinapse os filtros e composições são programados em um repositório de filtros que será discutido na *seção 9.9* e são referenciados pelos Conectores de Interação através de um identificador único no sistema. A localização do processamento nos Conectores de Composição e Filtragem permite um isolamento que mantém os mecanismos de composição transparentes para o Sistema de Notificação.

É importante ressaltar que, mesmo quando as assinaturas são realizadas, o Conector de Interação não tem uma percepção sintática de que está realizando uma assinatura de composição. Isto porque a solução para composição de eventos desenvolvida no modelo

Sinapse fica encapsulada como um filtro simples. Mantendo a mesma sintaxe do mecanismo original para filtros, atingimos o grau de transparência desejado. Potencialmente, essa estratégia poderia ser replicada a outros Sistema de Notificação.

A seguir, as classes de implementação descritas acima são apresentadas na figura 30.

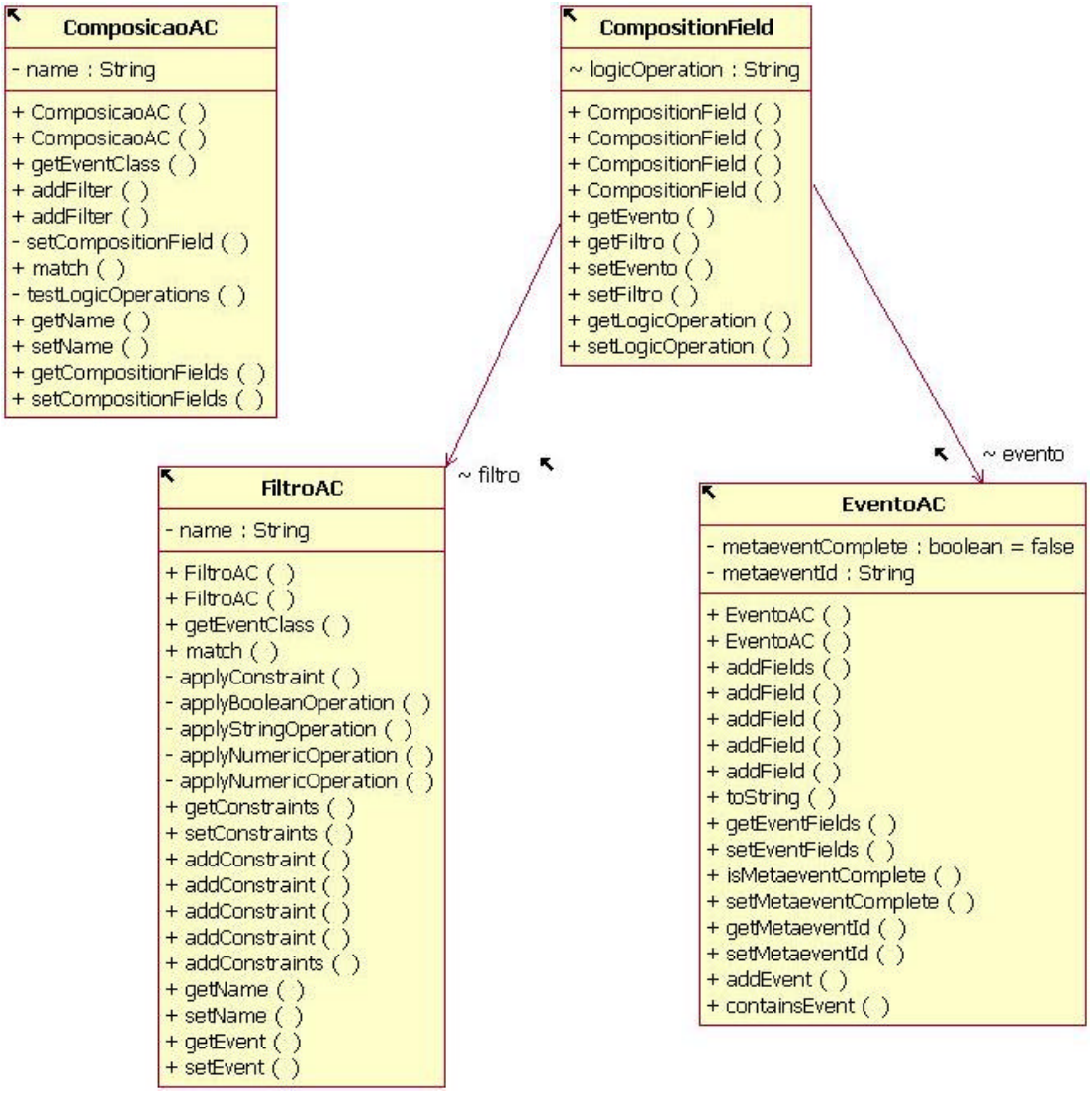


Figura 30 – Classes de composição

9.4 EventRouterAC

A classe `EventRouterAC` é uma classe nova que une as funcionalidades de duas classes do ambiente Rebeca original: `rebeca.network.EventRouter` e `rebeca.network.EventRouterConnection`. Esta nova classe, desenvolvida para permitir a adaptação do Rebeca ao AC, é uma representação da função de roteamento dos *brokers* e permite simplificar a descrição arquitetural. A separação entre *broker* (`EventRouter`) e conexão (`EventRouterConnection`) aumentava a complexidade sem qualquer ganho real para a descrição da arquitetura. Quando foram compostas em uma única classe, `EventRouterAC`, a descrição em relação ao Ambiente de Configuração se tornou mais simples sendo feita de forma direta em um módulo. Os objetos dessa classe são responsáveis por montar a infra-estrutura de comunicação e permitir o envio e entrega de eventos. Esta classe é apresentada a seguir na figura 31.

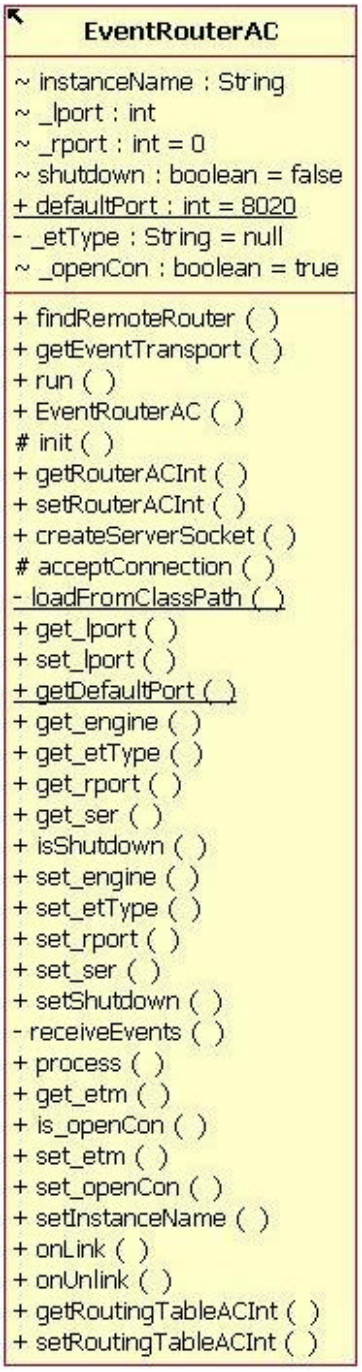


Figura 31 – Classe do Conector de Roteamento

Junto da classe `EventRouterAC` que é a implementação concreta dos Conectores de Roteamento, temos mais duas classes trabalhando de forma coordenada para prover todo o

suporte necessário para a construção do modelo. Essas classes são `RouterConnector` – que é a implementação do Conector de Transporte e `RoutingTableAC` – que é a adaptação da tabela de rotas para o Ambiente AC.

9.5 RouterConnector – (Conector de Transporte)

Conforme explicado anteriormente, para atingir o objetivo da reconfiguração dinâmica, foi necessário utilizar uma classe especial para realizar as conexões entre os Conectores de Roteamento. O Conector de Transporte é implementado pela classe `RouterConnector`. Esta classe é responsável por estabelecer uma conexão com um Conector de Roteamento específico, ficando então em espera. Um Conector de Roteamento é capaz de se comunicar com seu Conector de Transporte através de sua porta `getEventTranport` a fim de receber o canal de rede para a comunicação.

A seguir, a figura 32 apresenta esta classe.

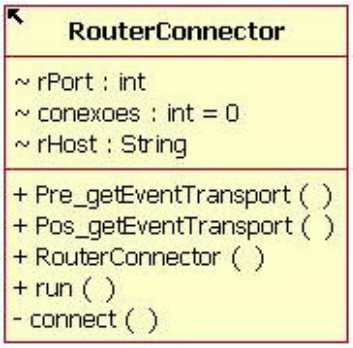


Figura 32 – Classe do Conector de Transporte

9.6 RoutingTableAC

Esta classe é derivada da classe `rebeca.routing.RoutingTable` e fica associada ao Conector de Roteamento. A única alteração sobre a classe original é a modificação do método `getDestinations`. Este método é o ponto onde ocorre a verificação de quais filtros são satisfeitos por um dado evento, ou seja, seu objetivo é retornar todos os destinos que devem receber aquele evento. É exatamente nesse ponto que vamos inserir a ligação com o Conector de Composição e Filtragem, delegando o processamento de filtros ao conector.

Assim, A classe `RountingTableAC` não tem nenhuma função além de expor para o Ambiente de Configuração o ponto de interceptação para o tratamento de composições. Dessa forma, foi alterada a maneira como a tabela de rotas era originalmente criada. Em Rebeca, a tabela de rotas é criada pela classe responsável por implementar o algoritmo de roteamento. Pode haver diversas implementações de algoritmos de roteamento distintas, sendo todas derivadas da classe `rebeca.routing.RountingEngine`, que fornece os mecanismos básicos.

A seguir, a figura 33 apresenta esta classe.

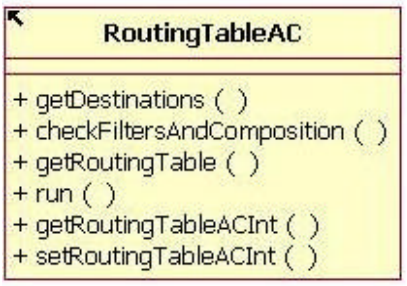


Figura 33 – Classe da Tabela de Rotas adaptada ao AC

9.7 CIConsumidor e CIProdutor

O Conector de Interação poderia conter códigos para tratar as interações tanto com módulos produtores quanto consumidores, mas por questões de simplicidade, foi implementado em duas classes distintas, de acordo com o papel dos módulos que interceptam. O funcionamento das classes é diferente por estarem lidando com papéis diferentes em relação a aplicação. Por parte do consumidor, a classe `CIConsumidor` deve ser capaz de se registrar junto a um Conector de Roteamento e fornecer o ponto de acesso para que seja feita a chamada *callback* de notificação. Por parte do produtor, a classe `CIProdutor` deve ser capaz de se registrar junto a um Conector de Roteamento e publicar as informações interceptadas do módulo produtor na forma de eventos.

A classe `CIProdutor` apresenta uma implementação mais simples, vez que a publicação não requer nenhum processamento complicado, sendo executada diretamente através da API fornecida pelo sistema Rebeca. Já a classe `CIConsumidor` deve fornecer alguns mecanismos extras para permitir seu correto funcionamento. É necessário que haja uma declaração de assinatura expressando o interesse por eventos. Para isso, o conector deve, antes, consultar o repositório de filtros do sistema para conseguir uma referência ao filtro desejado, nesta versão do protótipo os filtros são determinados no código. Uma vez realizada a assinatura, o conector entra num estado de espera; seu método de processamento de eventos contém a lógica necessária para tratar os eventos e/ou *metaeventos* recebidos. Através da interceptação à porta `getInfo()` do módulo consumidor, o `CIConsumidor` deve disponibilizar os eventos recebidos até o momento.

A seguir, são apresentadas, na figura 34, as classes do Conector de Interação.

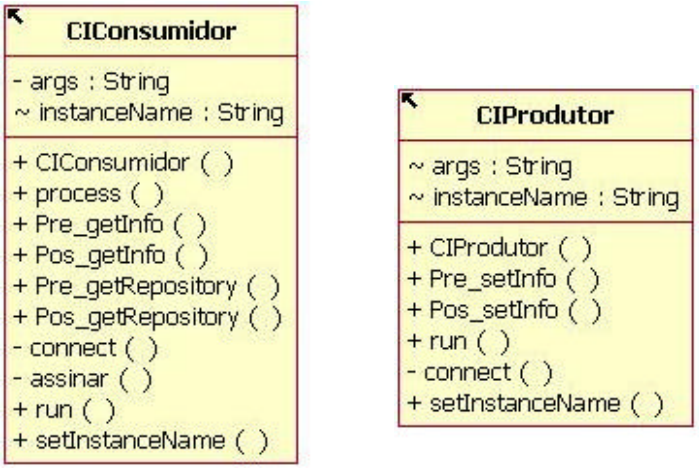


Figura 34 – Classe do Conector de Interação do Consumidor e do Produtor

9.8 CCF – (Conector de Composição e Filtragem)

Todo o processamento de filtros e composições de eventos foi delegado ao Conector de Composição e Filtragem. Este conector fica associado à tabela de rotas e é executado quando é necessário identificar as que filtros são satisfeitos por um evento.

No processamento dos eventos recebidos, a primeira etapa é verificar se estamos utilizando um filtro ou composição. Esta diferenciação é importante porque, se for uma composição e o processamento resultar em uma verificação válida, temos que gerar um *metaevento*. Uma vez identificado o objeto que estará processando o evento, então, executamos o método `match()` desse objeto. Se o evento for processado por uma composição e retornar verdadeiro, então o Conector de Composição e Filtragem passa para a etapa de geração de um *metaevento.*

Os *metaeventos* gerados vão sendo armazenados em uma fila para processamento. Ao final da execução do conector, se essa fila não estiver vazia, é iniciado o processo de envio de *metaeventos* para todos os destinos identificados. Assim, usando a mesma lógica do

Conector de Roteamento, o Conector de Composição e Filtragem envia os *metaeventos*. Isso garante a transparência em relação aos Conectores de Roteamento, pois o processamento de eventos não foi originalmente concebido para retornar um *metaevento*. A seguir, esta classe é apresentada na figura 35.

Figura 35 – Classe do Conector de Composição e Filtragem

9.9 FilterRepository

A última classe a ser apresentada é `FilterRepository`. Esta classe é responsável por armazenar todos os filtros e composições que serão utilizados na aplicação. Dessa maneira, o código para expressar o interesse por eventos fica isolado do código da aplicação. Com essa separação de código, torna-se possível reutilizar as lógicas para filtragem e composição de eventos entre os módulos consumidores. Além disso passa a ser possível expressar composições mais sofisticadas a partir de filtros e outras composições já disponíveis no sistema.

Para a Implementação de Referência, os filtros e composições foram descritos dentro da própria classe do repositório. Uma possibilidade de aperfeiçoamento é exatamente o estudo de uma forma de expressar a semântica de filtros e composições através de um mecanismo simples entre os consumidores e o repositório. Poderia ser desenvolvido o suporte para uma

linguagem que permitisse expressar operações lógicas sobre eventos e entre filtros o que permitiria criar dinamicamente os filtros e composições para atribui-los ao repositório de filtros do sistema.

A seguir, esta classe é apresentada na figura 36.

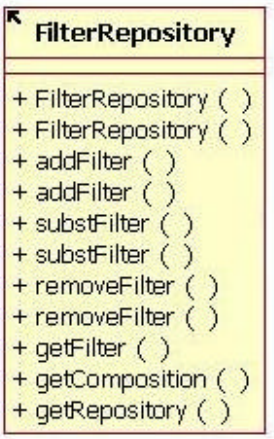


Figura 36 – Classe e interface do repositório de filtros

Os métodos apresentados na classe foram implementados no protótipo, no entanto nem todos foram utilizados por ainda não haver o suporte necessário para especificar os filtros e composições de forma dinâmica.