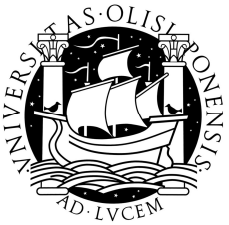
UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática



SUPORTE À RECONFIGURAÇÃO DINÂMICA DE COMPOSIÇÕES DE SERVIÇOS

Pedro Miguel Correia Xavier

MESTRADO EM INFORMÁTICA

2010

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SUPORTE À RECONFIGURAÇÃO DINÂMICA DE COMPOSIÇÕES DE SERVIÇOS

Pedro Miguel Correia Xavier

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor Hugo Alexandre Tavares Miranda

MESTRADO EM INFORMÁTICA

2010

Resumo

Uma das capacidades mais importantes dos sistemas computacionais nos dias que correm, é a possibilidade de estes permitirem a colaboração de diferentes unidades de processamento separadas fisicamente.

Para que isto seja possível, cada um dos nós tem de ser capaz de trocar informação com os outros, tendo para isso que conhecer as regras que definem as características da ligação em si e o formato dos dados.

Dada a complexidade das interacções estabelecidas, torna-se imperativo que sejam disponibilizados mecanismos às aplicações, que reúnam as funcionalidades que lhes permitam de forma facilitada efectuar o envio e recepção de dados.

As plataformas de composição e execução de protocolos permitem a orquestração de protocolos com características bem definidas, disponibilizando às aplicações canais de comunicação com características complexas.

As características da comunicação, variam quer pela mudança do requisitos das aplicações quer por alterações do ambiente de execução que tornam impossível garanti-las fisicamente. As plataformas devem permitir que as composições sejam alteradas dinamicamente com o menor custo para as aplicações, ou seja reduzindo ao máximo o tempo de inactividade ao mesmo tempo que asseguram a coerência das mensagens em trânsito.

O Appia é uma plataforma de composição de protocolos desenvolvida com o propósito de permitir a composição de estruturas complexas e oferece à partida uma flexibilidade nas estruturas das composições, no entanto não permite que estas sejam modificadas em tempo de execução.

O foco deste trabalho está direccionado para o estudo e implementação de um mecanismo de reconfiguração dinâmica que permita colmatar esta lacuna do Appia. Este objectivo foi conseguido pela evolução da plataforma no sentido de processar eventos de reconfiguração em tempo de execução.

Na validação da solução proposta verificou-se que a degradação de desempenho é inferior a 15% e que as propriedades oferecidas pela plataforma não são afectadas.

Palavras-chave: Reconfiguração dinâmica de sistemas, Composição de protocolos, Sistemas distribuídos modulares, Sistemas adaptativos

Abstract

One of the most important capabilities of computer systems these days is the possibility that they allow the collaboration of different processing units physically separate.

To make this possible, each of the nodes must be able to exchange information with the others, and must know which rules define the characteristics of the connection itself and the exchanged data format.

Given the complex set of interactions, it is imperative that some mechanisms be available to applications, comprising the features that enable them to carry out an easier way to send and receive data.

The platforms of composition and implementation of protocols allow the orchestration of well defined protocols, providing the applications with communication channels with complex traits.

The characteristics of the communication required by the application change according to different requirements or by changes in the execution environment. The platforms should allow the compositions to be changed dynamically with the lowest cost for applications, which means, minimizing downtime while ensuring consistency of the messages being exchanged at that time.

Appia is a protocol composition platform developed in order to support complex compositions structures and offers flexibility to match this requirement, but does not allow them to be modified at run time.

The focus of this work is directed towards the study and implementation of a dynamic reconfiguration mechanism to bridge this gap. This was achieved by the extension of the existing platform implementation to handle reconfiguration events.

The validation of the proposed solution includes the verification of the degradation of performance, which was found to be below 15% and also to guarantee that the reconfiguration does not affect the correct execution of Appia.

Keywords: Systems dynamic reconfiguration, Protocols compositions, Modular distributed systems, Adaptative systems

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Algumas classes de aplicações distribuídas necessitam de um conjunto de propriedades que ultrapassa aquelas que são vulgarmente oferecidas pela pilha protocolar TCP/IP. Por exemplo, foi já demonstrado que a coerência forte de réplicas de uma base de dados pode ser obtida através da concretização do paradigma da máquina de estados distribuída. Contudo, a concretização deste paradigma requer a entrega atómica de mensagens. Esta propriedade é caracterizada por garantir a entrega de mensagens a todos os processos correctos ou a nenhum e por entregar as mensagens a todos os processos pela mesma ordem, independentemente da origem.

O desenvolvimento de algoritmos fornecendo estas propriedades não é trivial, sobretudo quando se tem em conta que quer os processos quer o substrato de comunicação podem falhar, existindo inclusive alguns resultados de impossibilidade bem conhecidos. Foram por isso desenvolvidas bibliotecas fornecendo concretizações fiáveis e testadas destes algoritmos. A utilização destas bibliotecas é vantajosa para o programador de aplicações por evitar a pesquisa em domínios altamente especializados e simultaneamente, oferecer concretizações já sobejamente testadas, que reduzem substancialmente o tempo de desenvolvimento e depuração de erros.

Muitos destes algoritmos são construídos incrementalmente a partir de outros que fornecem propriedades complementares, criando desta forma relações de dependência entre propriedades. A disseminação fiável de mensagens por exemplo, pode ser conseguida utilizando algoritmos que apresentem garantia de entrega de mensagens ponto-a-ponto, os quais por sua vez assentam sobre algoritmos de entrega não fiável de mensagens ponto-a-ponto.

As plataformas de composição de protocolos (PCP) são ferramentas de middleware que agrupam implementações destes algoritmos, fornecendo uma interface normalizada para a comunicação entre os algoritmos e entre estes e a aplicação. As composições são usualmente representadas graficamente sob a forma de camadas, à semelhança do modelo OSI ou TCP/IP. Ao isolar a implementação dos algoritmos, as PCP favorecem a

independência entre concretizações dos protocolos e a sua reutilização. Adicionalmente, permitem ao programador incluir na pilha exclusivamente o conjunto de propriedades efectivamente requerido pela aplicação e suas dependências, suportando apenas os custos adicionais de comunicação impostos por estes protocolos e não por outros, que oferecem propriedades desnecessárias ou redundantes.

Os actuais sistemas distribuídos têm de lidar eficientemente com muitas formas de dinamismo que ocorrem no ambiente em que operam. Por exemplo, a capacidade de lidar com as contínuas chegadas e partidas de nós numa rede peer-to-peer, é tão relevante que é considerada um requisito do próprio sistema. Da mesma forma, os sistemas devem adaptar-se às diferentes exigências na utilização dos seus próprios componentes, permitindo que os serviços sejam adicionados ou removidos em tempo de execução. Quando a mobilidade se torna requisito do sistema, como nas redes móveis ad hoc, os sistemas devem ser capazes de oferecer os seus serviços apesar das mudanças frequentes na topologia da rede. Além disso, as aplicações podem ter que se ajustar em função das especificidades da rede onde estão inseridas, com diferentes larguras de banda ou com diferentes requisitos em termos de segurança.

Adicionalmente, algumas propriedades podem não ser necessárias permanentemente. Por exemplo, numa aplicação distribuída com necessidades de funcionamento ininterrupto, o administrador de sistema pode querer activar funções de registo de actividade apenas quando suspeita do comportamento da aplicação. No entanto, esta propriedade são onerosas ao desempenho do sistema e devem por isso ser desactivadas sempre que possível.

Da mesma forma, as PCPs têm de disponibilizar mecanismos que permitam que as composições se adaptem. A reconfiguração em tempo de execução é um aspecto muito importante por permitir adequar o desempenho da composição sem que os serviços que disponibilizam sejam interrompidos. Nem todas as PCPs que têm vindo a ser desenvolvidas apresentam um nível semelhante de dinamismo. Algumas, como o x-Kernel são estáticas, estabelecendo a ligação entre as camadas no momento da compilação da plataforma. As mais recentes ou estabelecem a associação no momento de instanciação da pilha ou permitem a sua reconfiguração em tempo de execução.

O Appia é uma plataforma de suporte à composição desenvolvida na linguagem de programação Java por uma equipa de investigação do Departamento de Informática da FCUL e que apresenta diversas características inovadoras, como o seu modelo de composição, que mistura propriedades do modelo horizontal e vertical. Desta forma, o Appia permite a utilização simultânea de diferentes composições, e que estas partilhem estado entre si. Apesar de apresentar alguma capacidade de adaptação, por exemplo por permitir a utilização alternada de composições, esta não é suficiente por estar limitada

ao conjunto de combinações antecipado pelo programador em tempo de desenvolvimento e por exigir a participação da aplicação. Importa por isso encontrar um modelo que permita uma reconfiguração transparente das composições, idealmente sem forçar à paragem por completo da plataforma e que possa ser desencadeada por monitores (locais ou remotos) que vigiem autonomamente o ambiente de execução.

Existem diversas aplicações para um serviço com estas características. Um monitor que vigie a lista de participantes num grupo poderia por exemplo coordenar a introdução/remoção de protocolos de cifra de mensagens sempre que pelo menos um dos participantes não se encontre na mesma rede local dos restantes; balancear adequadamente velocidade e consumo de energia em ambientes móveis ou activar funções de logging para depuração de erros sempre que necessário.

1.1 Objectivos

As particularidades da plataforma de suporte à composição Appia impedem a reutilização de soluções encontradas por plataformas com outros modelos de composição. Assim, os objectivos do trabalho apresentado neste documento são:

- a identificação do conjunto de passos necessários à extensão das capacidades da plataforma de suporte à composição Appia por forma a permitir a reconfiguração dinâmica das composições, em tempo de execução, e de forma transparente para o programador e utilizador, sem comprometer a correcção dos protocolos em execução.
- a definição de um protocolo de reconfiguração, que permita a execução concertada de operações de reconfiguração por diferentes instâncias da plataforma
- a concretização das alterações no código do núcleo da plataforma
- a validação funcional e de desempenho das extensões realizadas.

1.2 Contribuições

As contribuições deste trabalho são as seguintes:

- um mecanismo de reconfiguração das pilhas protocolares utilizadas pela plataforma de suporte à composição de protocolos Appia em tempo de execução.

- uma aplicação de administração de pilhas protocolares Appia que permite a sua reconfiguração em tempo de execução.

1.3 Organização do documento

Este documento está estruturado da forma descrita em seguida. O capítulo 2 descreve algumas plataformas de suporte à composição, colocando o foco nas diferentes estratégias para a reconfiguração que apresentam. O capítulo 3 faz uma apresentação mais aprofundada do Appia, detalhando os aspectos necessários para a compreensão do trabalho realizado, que é descrito nos capítulos 4 e 5 e avaliado no capítulo 6. Finalmente, o capítulo 7 sumariza as principais conclusões deste trabalho.

Capítulo 2

Trabalho relacionado

Para reduzir a sua complexidade, a maioria dos modelos representando a comunicação entre dois ou mais processos são conceptualmente organizados num conjunto de camadas ou níveis, cada uma construída utilizando os serviços prestados por outras e criando relações de dependência. O número de camadas, o nome que as define, a informação de estado que retêm e a funcionalidade que implementam diferem entre modelos, no entanto, em todos, a função de cada uma das camadas é fornecer serviços a outras camadas, escondendo destas os detalhes de concretização. A implementação de cada serviço é suportada por um "estado local" (o estado da instância do protocolo), por exemplo, uma variável que guarde um número de sequência num protocolo de ordenação, sendo que a união de todos os estados individuais constitui o estado global da composição.

Tipicamente, a composição é idêntica em todos os processos comunicantes. Cada camada de um processo mantém uma conversação com a camada análoga nos processos com que comunica. As regras e convenções usadas nesta conversação são no seu conjunto denominadas como o "protocolo da camada". Sumariamente, um protocolo é um acordo entre intervenientes numa comunicação que define a forma como esta se processa.

A troca de informação entre duas camadas análogas não é efectuada directamente. Cada camada, ao enviar uma mensagem, entrega-a à camada imediatamente abaixo, adicionando aos dados informação de controlo na forma de cabeçalhos ou terminadores. A camada mais abaixo da composição será responsável por introduzir toda a mensagem no canal físico por onde é efectivamente transmitida. Na máquina de destino, a mensagem percorre uma lista de camadas idênticas mas no sentido inverso, sendo que em cada camada é actualizado o estado e são extraídos os cabeçalhos e terminadores introduzidos pela camada análoga na máquina de origem. Exemplos deste modelo de composição são as pilhas TCP/IP e OSI. No caso particular do TCP/IP, a pilha está organizada em 5 camadas (física, ligação de dados, rede, transporte e aplicação), sendo

cada uma das camadas concretizada por um ou mais protocolos, oferecendo qualidades de serviços distintas como a entrega fiável e ordenada ou não fiável, disponibilizadas respectivamente pelo TCP e pelo UDP.

Uma aplicação distribuída utiliza um ou mais canais de comunicação para comunicar com os pares. Cada canal consiste num conjunto de protocolos, um protocolo por camada, que disponibilizam o conjunto de propriedades desejáveis para a comunicação. Exemplos destas propriedades podem ser a garantia de entrega da mensagem, a confidencialidade da informação ou a sua ordenação relativamente a mensagens enviadas por outros participantes. Uma vez que assegurar cada uma destas propriedades tem um custo computacional e de tráfego não negligenciável, o conjunto de propriedades utilizado deve ser tão pequeno quanto possível. Cabe à plataforma de composição assegurar que as propriedades usadas resumem-se apenas às requeridas pela aplicação. Ao conjunto de propriedades oferecidas por um canal dá-se, neste documento, o nome de qualidade de serviço.

Os sistemas operativos convencionais oferecem um conjunto de qualidades de serviço limitadas, tipicamente ditadas pela pilha protocolar TCP/IP. Desta forma, ao programador das aplicações distribuídas é apenas disponibilizada a comunicação ponto-a-ponto entre dois quaisquer dispositivos ligados à Internet, podendo este optar por uma qualidade de serviço com garantia de entrega ordenada das mensagens (fornecida pelo protocolo de transporte TCP) ou por uma qualidade de serviço sem garantia de entrega (fornecida pelo protocolo de transporte UDP). Estas duas qualidades de serviço são manifestamente insuficientes para algumas aplicações distribuídas como a replicação activa de bases de dados, jogos multi-participante em linha ou a partilha de ficheiros entre pares (peer-to-peer). Para colmatar estas limitações, muitas aplicações são obrigadas a concretizar os seus próprios mecanismos de extensão da qualidade de serviço. Em alternativa, têm vindo a ser desenvolvidas plataformas de suporte à composição, que são tipicamente acompanhadas de bibliotecas de protocolos. O programador das aplicações, pode assim associar uma destas plataformas de suporte à composição à sua aplicação e seleccionar a qualidade de serviço pretendida. Nestes casos, a comunicação passa a processar-se através da plataforma de suporte à composição, beneficiando das propriedades adicionais oferecidas e simplificando o desenvolvimento da aplicação.

Ao contrário da composição protocolar TCP/IP, importa oferecer ao programador de aplicações um ambiente onde a qualidade de serviço utilizada possa ser seleccionada de um conjunto largo de opções por forma a que a utilização dos serviços tenha o menor impacto possível no desempenho da aplicação. Este requisito invalida a possibilidade de utilizar soluções monolíticas, onde os pontos de interface entre as camadas são bem

conhecidos e imutáveis. Soluções monolíticas podem ser encontradas por exemplo nas concretizações do TCP/IP no núcleo dos sistemas operativos actuais.

O desconhecimento, no momento da concretização da plataforma, do conjunto de protocolos que será seleccionado pelo programador aplicacional levanta um conjunto adicional de desafios à interacção entre os diferentes protocolos, que terá que ser realizada através de uma interface normalizada pela plataforma e respeitada por todos os protocolos concretizados. A interface definida pela plataforma de composição deve assegurar diferentes objectivos, entre os quais:

- impor um modelo normalizado de comunicação entre camadas de uma mesma composição
- assegurar a comunicação entre instâncias distintas da composição, em processos distintos que podem ser executados em processadores com arquitecturas distintas ou compilados a partir de linguagens de programação distintas.
- assegurar a comunicação entre as camadas análogas em diferentes processos.

Existem diferentes modelos de composição que podem satisfazer estes requisitos, mas que podem ser agrupadas em duas categorias dependendo do modelo conceptual com que as camadas são organizadas, nomeadamente verticais ou horizontais.

As plataformas horizontais caracterizam-se por disponibilizar concorrentemente as mensagens a processar ao conjunto dos protocolos da composição. As mensagens são pois colocadas, num repositório acessível pelos protocolos e o seu processamento é realizado em paralelo por todos estes.

Por outro lado, as plataformas verticais entregam as mensagens a cada uma das camadas pela ordem em que estas surgem na composição, assumindo que a mensagem viaja da aplicação (topo da pilha) para a rede (fundo da pilha) no seu envio e na ordem inversa na recepção. O processamento é realizado por um protocolo de cada vez, e no fim deste a mensagem é entregue ao protocolo seguinte na pilha.

Neste capítulo pretende-se analisar diferentes tipos de plataformas numa perspectiva das dificuldades adicionais que apresentam para a alteração da lista de protocolos em tempo de execução. A análise é necessariamente genérica, sendo discutidos os constrangimentos associados às diferentes arquitecturas de várias plataformas existentes e identificados os mecanismos de adaptação disponibilizados por essas.

2.1 Plataformas horizontais

As plataformas horizontais caracterizam-se por disponibilizar concorrentemente as mensagens a todas as sessões da composição que os pretendam tratar. O Coyote [BHSC98] é uma plataforma que concretiza o modelo de composição horizontal. Esta plataforma refina o conceito de camada, decompondo-o em "micro protocolos", que implementam individualmente algumas das propriedades do serviço.

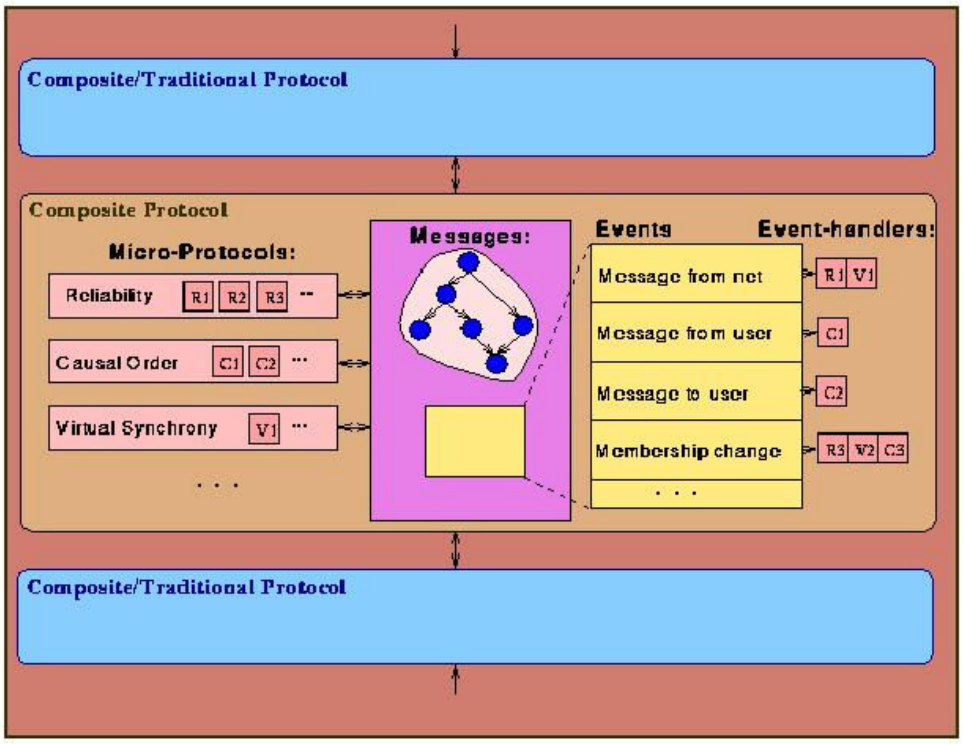


Figura 1 : Arquitectura do Coyote

A arquitectura do Coyote (fig. 1) baseia-se em eventos e em rotinas de tratamento de eventos potenciando a configurabilidade na medida em reduz as dependências entre os módulos. Os eventos podem ser gerados quer pela própria plataforma (por exemplo aquando da recepção de uma nova mensagem ou do disparo de um temporizador) ou pelos micro protocolos. A figura 1 enfatiza a organização não hierárquica (horizontal) do Coyote. Internamente, a plataforma concretiza um paradigma em que cada protocolo regista rotinas de tratamento de eventos, e quando um evento desse tipo chega ao sistema, as rotinas registadas anteriormente são executadas. Conceptualmente, as rotinas de tratamento de eventos são executadas concorrentemente. Se necessário, as rotinas podem especificar uma ordenação associando uma prioridade ao registo, sendo nesse caso garantido que as rotinas com a prioridade mais elevada são executadas antes das

restantes. Esta prioridade é especificada sob a forma de um número inteiro o que obriga à coordenação de valores entre os programadores dos micro protocolos sempre que existam dependências.

A plataforma coloca as mensagens recebidas da aplicação e da rede num saco, acessível por todos os protocolos. Para além disso é disponibilizado a todos os micro protocolos um mecanismo de memória partilhada.

O Coyote foi concebido para definir os serviços complexos com base em módulos existentes em tempo de compilação, no entanto favorece a adaptação dessa configuração em tempo de execução. A capacidade de registar/retirar o registo das rotinas de tratamento dos eventos em tempo de execução permite que micro protocolos que já façam parte da composição sejam activados substituindo ou complementando os já existentes e deste modo alterar dinamicamente a qualidade de serviço utilizada.

Contudo, esta plataforma não disponibiliza um mecanismo em tempo de execução que permita a inclusão na composição de novos protocolos. Esta funcionalidade traria enormes vantagens em termos de suporte à evolução a longo prazo, pois permitiria a adição de novos serviços sem que a operação fosse interrompida, respondendo à variabilidade dos requisitos das aplicações.

Em resumo, o Coyote apresenta-se como uma plataforma cuja filosofia horizontal baseada em partilha de memória, eventos e registo de rotinas de tratamento, permite alguma adaptação das composições dentro de um conjunto limitado de funcionalidades definido na compilação.

2.2 Plataformas verticais

As plataformas verticais caracterizam-se por estruturar hierarquicamente os protocolos, ou seja, cada protocolo fornece um serviço aos protocolos da composição que estão acima e utiliza os serviços dos protocolos dos quais depende. A configuração das composições neste tipo de arquitectura pode ser representadas como um grafo acíclico dirigido, em que os vértices são protocolos e as arestas relações de dependência.

Os eventos são processados por um protocolo de cada vez, no fim do qual é entregue ao protocolo seguinte, dependendo do sentido em que percorrem a composição.

Determinadas plataformas permitem composições com estruturas complexas, por exemplo, na forma de pilha, árvore ou em diamante. A pluralidade dos caminhos possíveis no grafo permite que os eventos percorram caminhos diferentes. Deste modo, torna-se necessário definir o conjunto de sessões que cada evento percorre. A forma

como esse caminho é calculado difere nas várias plataformas analisadas. Outro aspecto importante prende-se com o conhecimento que cada protocolo tem dos adjacentes, e o momento em que essa informação é construída.

Por exemplo, a plataforma x-Kernel [HP91] disponibiliza mecanismos para a construção de grafos de protocolos implementados como um núcleo do sistema operativo. Define três tipos de entidades: protocolos, sessões e mensagens, representadas na figura 2.

Cada protocolo, é um módulo independente que cumpre com as especificações do Interface Uniforme de Protocolos (UPI) definido pela plataforma, e é responsável não só pela criação das suas sessões mas também pelo encaminhamento ascendente das mensagens. As sessões contêm as estruturas de dados e código que suporta cada instância do protocolo. As mensagens são os únicos objectos activos e contêm dados do utilizador e cabeçalhos.

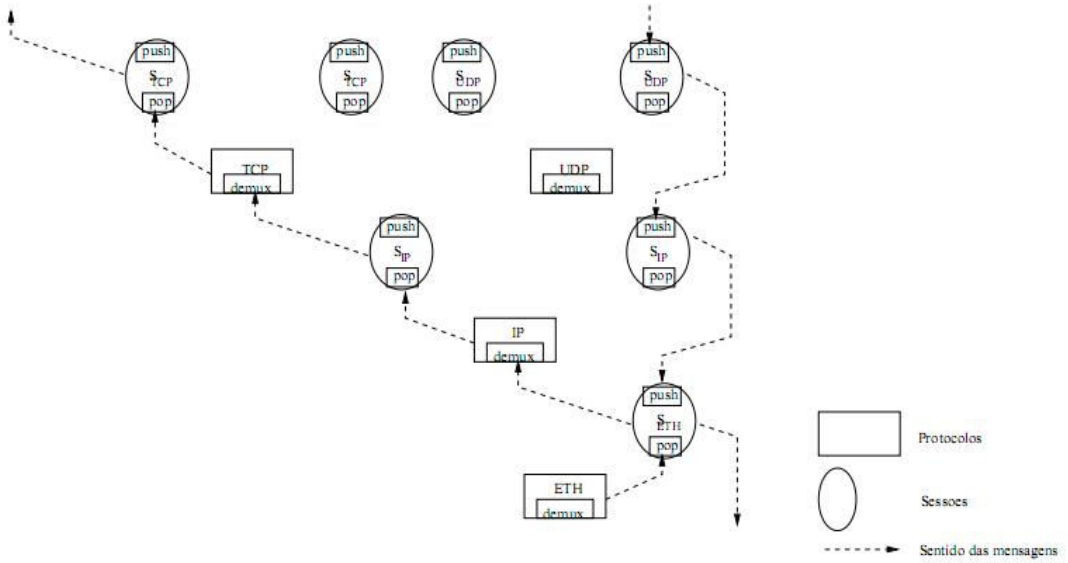


Figura 2 : Encaminhamento de mensagens no x-Kernel

O grafo de protocolos é gerado em tempo de compilação, a partir de uma descrição em ficheiro e não é passível de ser alterado em tempo de execução. Deste modo, a reconfiguração dinâmica das composições é sacrificada à custa de um melhor desempenho conseguida também graças a implementações muito eficientes das operações mais comuns usadas pelos protocolos e que integram a plataforma.

O Horus [RBM96] e o Ensemble [RBHVK98] são representativos de um modelo estritamente vertical, ou seja, em que o conjunto de protocolos atravessado por cada mensagem é constante. Cada protocolo da pilha é um módulo, com a sua responsabilidade bem definida e que na inicialização regista um conjunto de funções de

interacção com as camadas acima e abaixo. Ambas as plataformas suportam a reconfiguração dinâmica das composições criando pontos de mudança, que são coincidentes com as *mudanças de vista*. Uma mudança de vista é um conceito nuclear da comunicação em grupo, paradigma para o qual ambas as plataformas foram desenvolvidas, As mudanças de vista são momentos em que a comunicação entre as aplicações é voluntariamente interrompido para permitir a entrada ou saída de membros e todo o estado dos protocolos pode ser colocado num estado inicial. A reconfiguração é concretizada através da especificação de uma nova pilha para ser utilizada na vista que será instalada após a execução do protocolo de mudança de vista.

2.2.1 BAST

A plataforma de suporte à composição de protocolos BAST [GG98] utiliza o padrão de desenho *Strategy* para separar a interface do serviço da sua concretização. O modelo de composição é também vertical, com o mecanismo de herança, que é comum nas linguagens de programação orientadas a objectos a desempenhar o papel de agregador de interfaces. Este modelo simplifica consideravelmente a reconfiguração dos protocolos, uma vez que a ligação entre as interfaces e as concretizações é dinâmica. Contudo, a reconfiguração das composições é muito dificultada, uma vez que a pilha é definida pelo mecanismo de herança no momento da compilação,

2.2.2 Appia

A plataforma Appia [M01], sobre a qual incide este trabalho é caracterizada por ser orientada aos eventos e por oferecer uma maior flexibilidade na construção de estruturas de protocolos complexas, que a distingue das outras plataformas verticais. Nesta arquitectura, toda a interacção entre camadas da mesma composição, entre camadas em processos diferentes e com o próprio núcleo do Appia é baseada na troca de eventos. Cada camada implementa individualmente algumas das características fornecidas pelo serviço, e o seu estado pode ser partilhado entre pilhas distintas, mecanismo que permite a referida flexibilidade na composição dos protocolos. As composições são criadas com base numa definição estática que especifica os eventos relevantes para o serviço e faz alguma validação sobre a correcção da composição e uma dinâmica que implementa esse serviço, responsável por armazenar o estado do protocolo e por fornecer uma implementação do protocolo por via de uma função de tratamento dos eventos.

Ambas as definições são especificadas em tempo de execução, permitindo por exemplo a criação de novas composições durante a execução da aplicação. Contudo, a estrutura interna da plataforma não favorece a reconfiguração em tempo de execução das existentes, muito por culpa dos mecanismos utilizados para o encaminhamento dos eventos entre camadas e que visam uma optimização do desempenho.

2.2.3 Resumo

As plataformas de suporte à composição têm vindo a mostrar-se ferramentas úteis para simplificar o desenvolvimento de aplicações distribuídas por oferecerem ao programador de aplicações estender o conjunto muito limitado de propriedades oferecidas pelo TCP/IP. Diferentes modelos de composição têm vindo a ser propostos, colocando a ênfase em diferentes aspectos como a facilidade de incorporação de novos protocolos ou o desempenho. Este capítulo comparou alguns dos modelos de composição existentes, focando-se sobretudo nas capacidades de reconfiguração das composições em tempo de execução. Apesar de suportar a criação e eliminação de canais em tempo de execução, a estrutura interna da plataforma de suporte à composição Appia dificulta consideravelmente a reconfiguração das composições existentes. O próximo capítulo apresenta uma descrição mais profunda desta plataforma que facilitará não só compreender a origem das dificuldades na reconfiguração dinâmica mas também a descrição do trabalho realizado.

Capítulo 3

Appia

A plataforma Appia suporta a composição e operação de pilhas de protocolos. Para tal disponibiliza um conjunto de classes que permitem o desenvolvimento de novos protocolos e mecanismos que permitem executar aplicações beneficiando de um serviço de comunicação com propriedades complexas.

A abordagem seguida por esta plataforma é totalmente orientada aos eventos. Ou seja, a comunicação entre camadas está restringida à troca de eventos, uma aproximação que beneficia a independência entre camadas por balizar a partilha de informação a um modelo normalizado e bem conhecido. Sendo uma plataforma vertical agrupa os protocolos em pilhas, criando canais por onde circulam eventos em ambos os sentidos.

O suporte do Appia à operação das pilhas passa pela disponibilização de um mecanismo que é responsável por gerir os eventos no sistema, permitindo às sessões criar, inserir e consumir eventos de um canal. A circulação dos eventos obedece a regras impostas pela plataforma. Essas regras definem a ordem pela qual os eventos são entregues a cada um dos protocolos da pilha de forma a garantir que as relações de dependência entre eventos são asseguradas.

As aplicações por vezes necessitam de mais do que um canal que lhes disponibilize serviços distintos de rede. A plataforma garante esse suporte, gerindo os eventos centralmente e disponibilizando uma forma de os canais trocarem informação entre si. Essa troca é baseada na partilha do estado de um protocolo entre canais. Com esta funcionalidade o Appia oferece uma maior flexibilidade na criação de composições com estruturas complexas, construindo-as com base em pilhas protocolares.

3.1 Protocolos e composições

No Appia, todos os protocolos são concretizados pela extensão de duas classes, a classe *Layer* e a classe *Session*. As duas subclasses criadas têm finalidades distintas.

A classe derivada de Layer vai definir as características imutáveis do protocolo, especificando os eventos que consome, produz e que são fundamentais para uma

execução correcta do protocolo. Esta classe também implementa um método que cria uma instância do protocolo, ou seja, uma instância da classe derivada de *Session*.

A classe derivada de Session instancia o protocolo. A classe concretiza um interface predefinido que fornece rotinas de tratamento para os eventos que o protocolo processa. O processamento dos eventos, por invocação do método *handle* da sessão, pode alterar o estado do protocolo ou modificar os cabeçalhos ou atributos do evento. No processamento de um evento, a sessão pode introduzir outros eventos no canal, ou consumir esse evento não o entregando às sessões seguintes.

Conceptualmente, a composição de protocolos no Appia define uma pilha no topo da qual está a camada que implementa a aplicação e no fundo a camada de interface com a rede. Entre estas são colocadas outras camadas que adicionam propriedades ao canal de comunicação. À semelhança da definição dos protocolos, o Appia utiliza também duas representações das composições.

Uma definição estática representada por um objecto da classe *QoS*, composto por um vector de instâncias das subclasses de Layer dos protocolos a utilizar. A *QoS* define a Qualidade de Serviço do canal e caracteriza-se pelo conjunto de propriedades que são disponibilizadas à aplicação. A composição criada pela *QoS* é validada a partir dos eventos requeridos e fornecidos pelas Layers que a compõe. Essa validação garante que todos os eventos requeridos pelos protocolos são fornecidos por algum dos protocolos da composição.

Os canais são instâncias da classe *Channel* e são criados a partir de um *QoS*. O canal é composto por instanciações das subclasses de *Session* dos protocolos e que suportam o processamento dos eventos.

3.1.1 Criação do canal

A criação do canal consiste em criar um objecto do tipo *Channel*, que concretize uma determinada *QoS*. Quando instanciado a partir da definição desta, é criada uma estrutura capaz de acolher as sessões, objectos que estendem a classe *Session*. Após a instanciação da classe *Channel,* as sessões podem ser atribuídas explicitamente pelo programador ou automaticamente, através do método disponibilizado pela subclasse de *Layer* do protocolo. O mecanismo de atribuição pelo programador permite a associação de uma sessão a mais do que um canal, resultando na partilha do seu estado.

A figura 3 apresenta a relação entre os diferentes actores da criação de um canal.

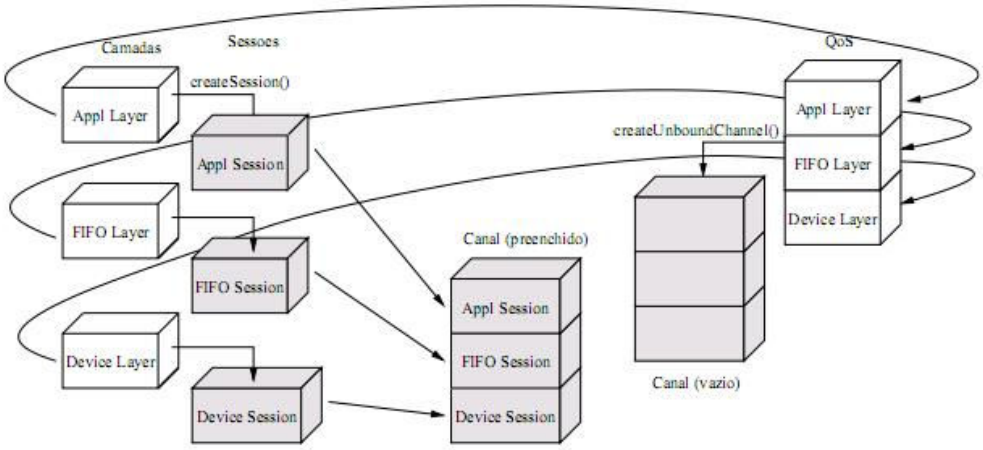


Figura 3 : Instanciação do canal a partir da QoS (Miranda, 2001)

3.1.2 Criação das rotas dos eventos

Para melhorar o desempenho, o Appia optimiza o caminho que cada evento percorre no canal, entregando-o apenas às sessões cuja subclasse *Layer* correspondente declarou o interesse do protocolo no seu processamento. Esta optimização permite uma maior eficiência do fluxo dos eventos no canal mas requer que sejam criados caminhos específico para cada classe de evento.

Os caminhos são determinados em duas fases. Numa primeira fase, a *QoS* cria um conjunto de rotas estáticas na forma de um vector de *Layers*. Para cada evento declarado como sendo produzido por uma *Layer*, é gerado um *QoSEventRoute*. Cada um desses objectos associa o tipo de evento a um vector contendo referências para as *Layer* que declararam o interesse em processar os eventos desse tipo.

Posteriormente, o canal é criado e as rotas projectadas num conjunto de rotas efectivas na forma de vectores de sessões. A partir do conjunto de objectos do tipo *QoSEventRoute* criados anteriormente e do vector de sessões que formam o canal, é criado um conjunto de objectos do tipo *ChannelEventRoute*. Cada objecto *QoSEventRoute* dará origem a um objecto *ChannelEventRoute* no canal. Este associa a cada tipo de evento um vector de sessões a que os eventos desse tipo serão entregues.

De notar que todas estas operações são realizadas uma única vez, antes de o canal entrar em operação, concentrando assim o esforço computacional no arranque da aplicação. Os caminhos são atribuídos às instâncias de eventos no momento em que estes são introduzidos no canal pela primeira vez.

3.2 Eventos

Cada evento é caracterizado pelo canal em que circula, a direcção em que percorre a pilha de sessões, e a sessão que os gerou. As interacções com o canal (por exemplo, o pedido de temporizadores e os anúncios da abertura e encerramento dos canais) implicam a disponibilização de um conjunto de eventos com características bem definidas. Para além desses, o Appia permite que o programador crie os seus próprios eventos. Estendendo as classes de eventos existentes através do mecanismo de herança podem ser adicionados atributos que suportem a lógica dos protocolos desenvolvidos. A extensão dos eventos complementa o mecanismo de desenvolvimento de protocolos, fornecendo um meio de estabelecer as interacções entre as sessões que concretizam os protocolos desenvolvidos. Este mecanismo permite potenciar a reutilização de protocolos ao mesmo tempo que facilita a compatibilidade entre diferentes versões de um protocolo.

3.2.1 Tipos de eventos

Todas as classes de eventos estendem a classe base *Event* que se limita a definir os requisitos básicos de um evento que possibilitam o seu encaminhamento no canal. A classe *Event* suporta a comunicação entre camadas das composições de um mesmo processo. É a classe base de todos os outros eventos e pode ser estendido no sentido de criar eventos de âmbito local.

Para suportar o envio pela rede o Appia disponibiliza a classe *SendableEvent*. Os eventos que estendam esta classe podem ser enviados pela rede. Para tal são adicionados dois atributos que definem o endereço de origem e endereço de destino usados pela camada de acesso à rede. Os atributos definidos nos eventos que estendam esta classe não são automaticamente enviados pela rede. É da responsabilidade do programador garantir que esses são transmitidos, utilizando o mecanismo de adição de cabeçalhos que integra esta classe. Este mecanismo permite a adição e extracção de cabeçalhos do evento. Estes cabeçalhos podem ser de tipos primitivos ou objectos, desde que serializáveis.

Alguns eventos são processados não só pelas camadas mas pelo próprio canal. Esses eventos tipicamente realizam controlo da operação do canal ou interagem com mecanismos geridos por esse. Exemplos desses eventos são o *ChannelInit* e o *ChannelClose* que sinalizam respectivamente o início e terminação da actividade no canal, sendo entregues às sessões e por fim processados pelo canal. Os tipos de eventos *Timer* e *PeriodicTimer* disponibilizam às sessões um mecanismo de temporizadores baseados em eventos. Este serviço evita que a sessão esteja em espera activa, ou que

tenha que recorrer a chamadas ao sistema operativo. Note-se que o canal utiliza uma thread própria para gerir esta funcionalidade.

3.2.2 Eventos assíncronos

Conceptualmente, o Appia utiliza um único fio de execução (*thread*), partilhado por todas as sessões. Contudo, algumas sessões, como aquelas que recebem eventos da rede ou que interagem com o utilizador, podem ter necessidade de executar um fio de execução próprio. A introdução assíncrona de eventos define um ponto de sincronização único que permite a estes fios de execução, que são conceptualmente externos ao canal, a introdução de eventos no canal. Contudo, esta introdução tem características excepcionais, que serão abordadas nos pontos considerados relevantes para a descrição do trabalho realizado.

3.2.3 Caminho do evento

Cada evento é caracterizado pela sessão que o gerou, ponto na pilha a partir do qual é determinada a sequência de sessões a visitar. No processo de encaminhamento, são utilizados os seguintes atributos, declarados na classe *Event*.

- src – sessão que criou o evento
- route – vector de sessões interessadas em processar o evento atribuído pelo canal
- firstSession – índice da primeira sessão à qual o evento é entregue
- currentSession – índice da sessão que processa o evento numa determinada iteração
- dir – a direcção em que percorre o vector de sessões

Quando é instanciado, cabe à sessão preencher os atributos src e dir. Os restantes atributos são preenchidos quando o evento é iniciado por invocação do seu método *init*. A invocação deste método é requerida aquando da inserção do evento no canal.

O atributo currentSession será actualizado a cada iteração, ou seja, quando o escalonador de eventos pretende processá-lo, pede a este que lhe devolva a sessão a que deverá ser entregue. Essa sessão é obtida actualizando o atributo currentSession em função da direcção do evento e da informação sobre a rota do evento. Após isso é usado para devolver a sessão que ocupa essa posição no vector de sessões. Note-se que uma determinada sessão pode inverter a direcção do evento no seu processamento, pelo que o caminho percorrido por este não tem necessariamente de ser apenas num dos sentidos.

Sendo a indexação da posição do evento feita em relação ao seu vector de eventos, que recorde-se será um subconjunto de sessões do vector do canal, essa posição não corresponde necessariamente à posição nesse vector.

3.2.4 Cálculo da primeira sessão

A primeira sessão a que o evento é entregue é determinada quando este é inicializado. O cálculo é feito num método do canal, que utiliza o vector de sessões que define a pilha, a informação que o evento mantêm e a rota para o tipo de evento.

A determinação da primeira sessão é realizada com uma iteração que procura a primeira sessão após a origem em comum entre o vector de sessões do canal e a rota para o tipo de evento. Caso não encontre, o evento é colocado acima do topo da pilha onde pode ser processado pelo canal. Subentenda-se que rota será a sequência de sessões que o evento visita dependente da direcção em que circula.

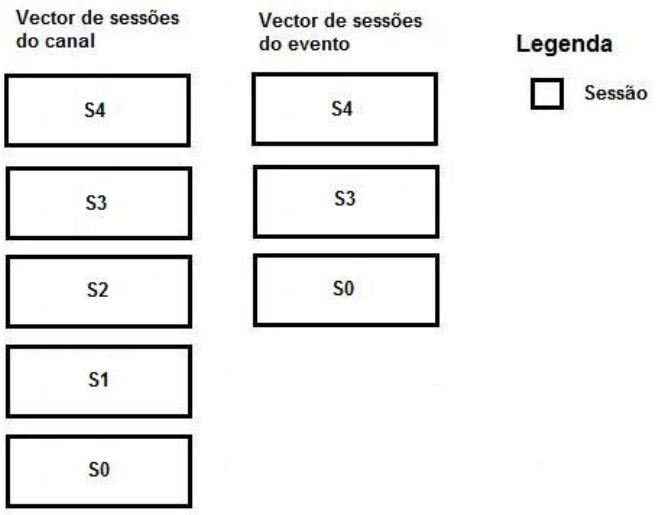


Figura 4 : Cálculo da primeira sessão

Considere-se os vectores do canal e de um determinado tipo de evento descritos na figura 4 Se a sessão S1 inserisse um evento deste tipo com a direcção ascendente, então a primeira sessão que o evento visitaria seria a sessão S3. Outro exemplo, se fosse inserido no canal assincronamente com sentido descendente, a primeira sessão a que seria entregue seria a sessão S4. Isto deve-se ao facto dos eventos inseridos assincronamente não terem uma sessão de origem. Nestes casos a primeira sessão a que são entregues, é a sessão no topo ou na base do seu vector de sessões do evento dependendo da sua direcção.

3.2.1 Indicador de posição

Cada evento é processado por uma sessão de cada vez, percorrendo o vector de sessões na direcção especificada nos atributos do evento. O progresso nesse vector é controlado pelo evento que usa o atributo currentSession como um cursor no vector de sessões e que possibilita determinar qual a sessão a que deve ser entregue em cada iteração. Esse cursor é actualizado cada vez que o evento é entregue a uma sessão, pelo que até esse evento ser processado referencia a posição da pilha na qual reside a última sessão que o processou. Inicialmente ele assume o valor inválido de -1.

A primeira vez que o evento é processado o cursor é inicializado com o índice da primeira sessão a visitar. Quando este índice assume um valor fora do intervalo de 0 ao número de sessões menos um, significa que o evento percorreu todas as sessões. Os eventos terminam o seu fluxo no canal.

3.3 Escalonamento de eventos

O escalonamento de eventos é a tarefa principal da plataforma em tempo de execução. Compreende as actividades de inserção, consumo e invocação do processamento dos eventos que circulam nos canais. No Appia essa tarefa é concretizada por uma instância estática da classe *Appia* que executa num único fio de execução. Nesta instância registam-se um conjunto de objectos da classe *EventScheduler*, e cada um realiza o escalonamento num grupo de canais. A plataforma Appia permite a definição de novos algoritmos de escalonamento através da extensão desta classe e fornece uma única concretização.

O algoritmo concretizado garante uma ordenação que respeita a ordem causal em todos os eventos que circulem nesse canal. Para além disso garante a entrega FIFO dos eventos num dos sentidos do canal. Estas garantias são fundamentais para a correcta execução dos protocolos por assegurarem que os eventos são apresentados a todos os protocolos pela ordem em que foram gerados. A ordenação causal e FIFO são definidas da seguinte forma:

A ordenação FIFO impõe que se a sessão S1 envia o evento E1 e depois envia o evento E2 na mesma direcção, então a camada imediatamente a seguir nessa direcção deverá receber E1 seguido de E2.

A ordem causal dos eventos garante a manutenção da relação causa-efeito entre eventos. Ou seja, que nenhuma sessão poderá receber um evento E2, gerado em consequência de um evento E1 antes de receber E1. A ordenação causal é transitiva. Por exemplo, existindo dois eventos enviados em direcções diferentes na sessão S1 pela ordem E1 e depois E2, se uma sessão S2 inverta o sentido de E2, e E1 e E2 acabarem

por ser entregues a uma sessão S3, então os eventos E1 e E2 devem chegar à sessão S3 pela ordem E1 e depois E2.

3.3.1 Concretização

A instância do Appia itera sobre a lista de EventSchedulers em ciclo infinito, invocando a sua função de tratamento de eventos. Em cada iteração é processado apenas um evento por esse escalonador garantindo que todos são chamados rotativamente. A condição de bloqueio da instância da classe *Appia* é baseada na existência de eventos. Para isso guarda localmente o número total de eventos inseridos em todos os canais. Quando este valor chega a zero, o fio de execução bloqueia. Com a inserção de um novo evento num dos canais, o fio de execução é desbloqueado e retoma o processamento de eventos.

A ordenação de eventos em cada canal é assegurada pelo escalonador através de uma lógica de processar um evento e todos os eventos inseridos na mesma direcção durante o seu processamento. Depois processa os eventos introduzidos durante o seu processamento no sentido inverso esgotando as relações de causalidade do primeiro. Finalmente processa os outros eventos introduzidos no canal após o primeiro evento mas que não tenham relação de causalidade com esse.

3.3.1 Listas de eventos

Os eventos inseridos nos vários canais geridos por um escalonador são mantidos em três listas: a Main, a Reverse e a lista Waiting. Note-se que as listas contêm eventos de todos os canais geridos pelo escalonador. A lista Main contém os eventos que aguardam processamento na mesma direcção do último evento tratado. A lista Reverse, os eventos na direcção contrária. Finalmente, a lista Waiting contêm os eventos introduzidos assincronamente no canal e que aguardam a sua inserção nas listas restantes, o que criará o seu ponto de sincronização.

Consumo de eventos

O consumo dos eventos com garantia de ordenação é bastante simplificado pela forma como esses são inseridos nas listas. Sendo assim em cada iteração do Appia um escalonador invoca o método consumeEvent de um dos seus escalonadores que processam o primeiro evento encontrado nas suas listas. A procura é feita pela ordem: lista Main, lista Reverse e finalmente lista Waiting. Ao ser seleccionado um evento, o escalonador retira-o da lista e invoca o método handle da sessão que o próprio evento determina entregando esse evento à sessão. No caso dos eventos de canal, se não existem mais sessões que o queriam processar, o evento mesmo assim invoca o método handle do canal onde circula.

Inserção de eventos

A inserção de eventos nas listas é determinante na garantia de ordem causal. Os eventos podem ser inseridos de duas formas. Assincronamente se provierem de fora do canal e sincronamente se forem inseridos por uma sessão ao processar um evento. Os eventos inseridos sincronamente são inseridos nas listas Main e Reverse dependendo da sua direcção. Os eventos inseridos assincronamente são sempre inseridos na lista Waiting.

Na lista Main são inseridos à cabeça os eventos que circulam na mesma direcção do evento que está a ser processado, ou seja, quando uma sessão ao tratar um evento S1 causa o envio de um outro evento S2 na mesma direcção que S1.

Os eventos gerados que sejam inseridos na direcção oposta são adicionados na cauda da lista Reverse.

A lista Waiting permite a inserção de eventos externos ao canal. Esta operação tem pois de ser realizada de forma sincronizada com a própria thread do EventScheduler. Os eventos nestas condições são:

- os eventos inseridos por uma thread diferente da thread do EventScheduler, por exemplo inseridos por threads de recepção de dados, por exemplo as que processam a leitura de sockets ou do teclado.
- os eventos introduzidos por uma sessão de outro canal, por exemplo o caso de uma sessão partilhada pelos canais C1 e C2, que no processamento de um evento no canal C1 introduz um evento no canal C2.
- os eventos que o próprio canal insere durante o processamento no topo da pilha de um evento do tipo ChannelEvent, por exemplo o EccoEvent enviado no tratamento de um evento do mesmo tipo EccoEvent.
- os eventos inseridos por uma determinada sessão mas que sejam marcados como inseridos ou criados por outra. Neste caso enquadram-se os eventos que sejam inseridos não respeitando a ordem imposta pelo Appia sendo inseridos num ponto diferente da rota. A ordem causal estaria a ser posta em causa justificando este comportamento.

3.3.2 Exemplo descritivo

Para mais fácil se perceber o mecanismo de inserção nas listas considere-se o cenário de um canal da figura 5 composto pelas sessões S0, S1, S2 e S3. Na figura 5 está representado o processamento pelo canal, representado por uma sessão fictícia. O nome de cada sessão refere o índice correspondente à posição no vector de sessões que

o canal guarda. O fio de execução que insere o evento recebido no canal também está representado por uma caixa para mais fácil representação.

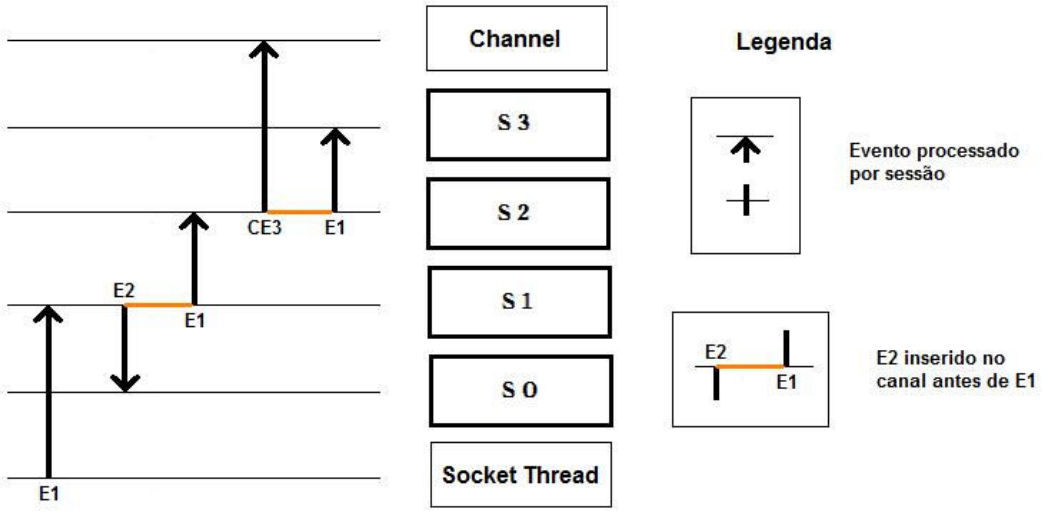


Figura 5 : Inserção e consumo de eventos

A figura 5 mostra um cenário em que:

- o evento E1 é recebido pela rede e percorre todas as sessões da pilha.
- a sessão S1 ao processar o evento E1 insere primeiro o evento E2 no sentido inverso e depois o evento E1 no mesmo sentido.
- a sessão S2 ao processar o evento E1 insere primeiro o evento de canal CE3 no mesmo sentido e depois o evento E1 também no mesmo sentido.
- todos os eventos são processados por todas as sessões do canal

A tabela seguinte mostra o conteúdo das listas que suportam o escalonamento após a inserção no canal e durante o processamento dos vários eventos do cenário da figura 5. A ordem pela qual essas operações são realizadas também é representada, equivalendo a primeira linha à primeira tarefa executada.

Operação	Main	Reverse	Waiting
Socket Thread insere E1			E1
S0 processa E1			

S0 insere E1	E1		
S1 processa E1			
S1 insere E2		E2	
S1 insere E1	E1	E2	
S2 processa E1		E2	
S2 insere CE3	CE3	E2	
S2 insere E1	CE3 : E1	E2	
S3 processa CE3	E1	E2	
S3 insere CE3	CE3 : E1	E2	
Canal processa CE3	E1	E2	
S3 processa E1		E2	
S0 processa E2			

Tabela 1 : Conteúdo das listas de eventos

Capítulo 4

Reconfiguração de protocolos

Em determinadas situações importa conhecer e reconfigurar alguns parâmetros da execução de protocolos. Por exemplo, num protocolo de detecção de faltas, cada processo envia periodicamente uma mensagem sinalizando a sua actividade. Idealmente, o período de transmissão deve ser tão pequeno quanto possível, para que os processos detectem a falta de um processo rapidamente. Contudo, um período excessivamente curto, aumenta o consumo de largura de banda. Importa por isso adaptar o temporizador aos recursos disponíveis em cada instante. Para além do estado do protocolo, outros constrangimentos do ambiente de execução, como sejam as condições de rede ou as preferências do utilizador, podem justificar a necessidade de adaptação.

O dinamismo das alterações das condições operacionais da rede justifica a necessidade de um mecanismo de reconfiguração, que proceda à adaptação dum protocolo em função de políticas predefinidas [RLR06] ou dos parâmetros óptimos do sistema em alternativa a soluções mais simples mas com um desempenho muito aquém do desejado como a interrupção completa da comunicação entre os processos para instalação de uma nova composição protocolar. Para automatizar a reconfiguração, as alterações das condições operacionais devem idealmente ser descobertas e sinalizadas pela própria plataforma de composição, que terá que ter a capacidade de emitir informação sobre o estado dos protocolos. A partir destas, poderá ser observado que o valor de um atributo de uma sessão está fora do intervalo desejável, e proceder à sua reconfiguração automaticamente.

As plataformas de composição de protocolos têm requisitos rígidos em termos de desempenho. Por isso, o mecanismo de monitorização e reconfiguração deverá ser transparente em relação à execução das próprias sessões e minimizar o impacto no desempenho das composições. A transparência é um requisito importante por permitir a utilização de concretizações de protocolos nos quais esta funcionalidade não estava prevista o que estende o tempo de vida das concretizações dos protocolos.

4.1 Modelo

O Appia não disponibiliza mecanismos de monitorização e reconfiguração das sessões que permitam em tempo de execução obter informação sobre o estado dos protocolos e proceder à sua reconfiguração [P01]. Este capítulo apresenta uma solução transparente para os protocolos desta limitação.

A solução proposta disponibiliza dois serviços de leitura dos atributos de sessões: um serviço de leitura pontual e um serviço de leitura periódica, que requer um registo prévio a partir do qual são geradas leituras a intervalos de tempo predefinidos. São também disponibilizados serviços de escrita desses atributos. Os serviços de leitura e escrita são atómicos, na medida em que garantem que numa operação que incida sobre atributos de várias sessões da mesma composição são processados em todas as sessões sem que o canal processe qualquer outro evento que possa alterar o estado de alguma delas, prevenindo desta forma incoerências no estado de um conjunto de sessões.

A arquitectura consiste num componente integrado nas composições da aplicação, o *ContextSensor*, que coordena as operações nas composições de protocolos. O *ContextSensor* é implementado por dois canais Appia (*RemoteInvocationChannel* e *ContextNotificationChannel*), associados ao canal que se pretende monitorizar.

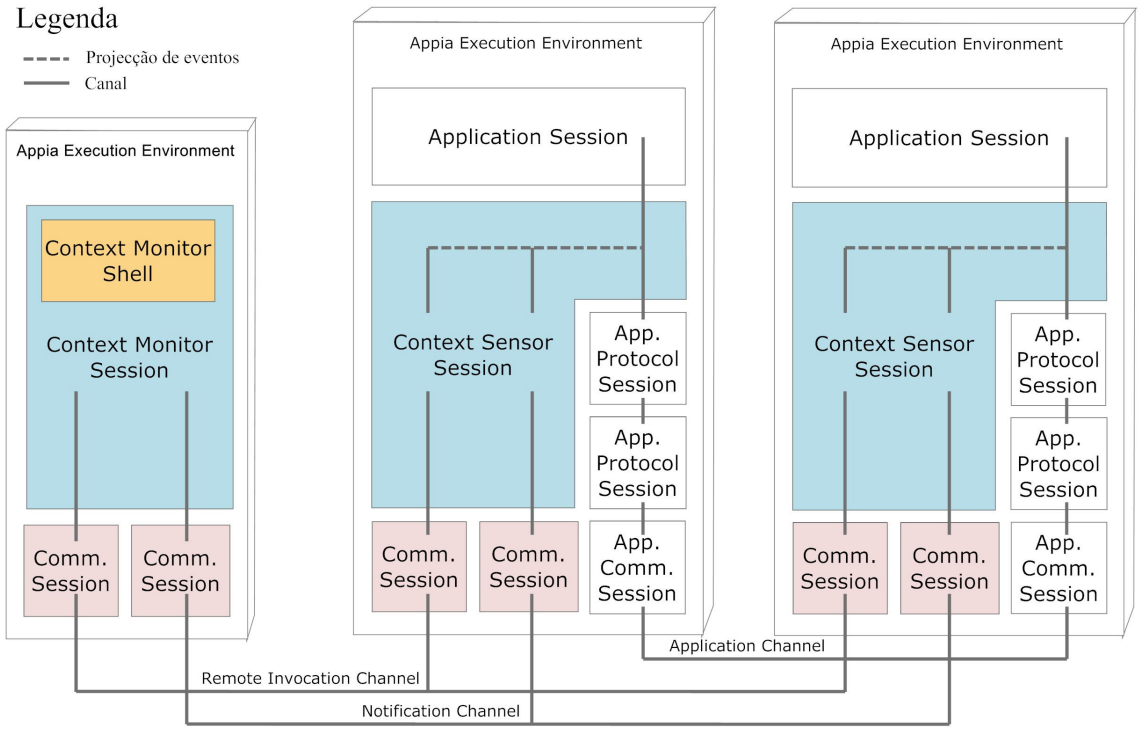


Figura 6 : Arquitectura de Monitorização com consola

O *ContextSensor* respeita a semântica Appia. A interacção com consolas externas é realizada através de eventos que utilizam os novos canais definidos. Algumas operações podem originar a devolução de um evento de resposta, sendo também predefinidos os eventos que as suportam. Estes eventos são considerados eventos externos, pois estabelecem a interacção entre as consolas e o ContextSensor e circulam nos canais de controlo. Outros eventos, os eventos internos, projectam os eventos externos em eventos que circulam nos próprios canais da aplicação e realizam a interacção com as próprias sessões monitorizadas.

A sessão *ContextSensorSession* actua como coordenador das operações e estabelece a ponte entre canais de controlo e os canais da aplicação, traduzindo os eventos externos em eventos internos e vice-versa. A tradução dos eventos consiste em criar um evento num canal a partir de um evento recebido noutro canal. Com a tradução de um evento que invoca uma operação pode haver necessidade de criar estruturas de dados que suportem a tradução inversa, ou seja, do evento de resposta.

Para suportar o processamento dos eventos internos nos protocolos monitorizados, é disponibilizada a classe *MonitorizableSession* que estende a classe *Session* adicionando-lhe métodos de tratamento desses eventos.

Em alternativa ao modelo de monitorização remota do canal, o modelo concretizado suporta também a monitorização local. Nomeadamente, é permitido a qualquer camada (como por exemplo a camada de aplicação) invocar as operações introduzindo ela mesma no canal monitorizado o evento interno que invoca a operação. Neste caso os eventos internos de resposta em vez de serem introduzidos no canal de controlo são entregues à camada que os gerou, utilizando os mecanismos básicos de encaminhamento de eventos do Appia. Este modelo não requer a injecção da camada ContextSensorSession no canal da aplicação, no entanto, apenas suporta as operações de leitura pontual e escrita de atributos.

4.2 Canais de controlo

A comunicação entre o *ContextSensor* e as consolas de monitorização tem de ser suportada por canais de comunicação. De modo a não limitar as propriedades dessa ligação ao protocolo de rede usado no canal da aplicação nem interferir com a aplicação, foram criados canais próprios. São definidos dois canais, o *RemoteInvocationChannel* e o *NotificationChannel*, com semânticas da camada de rede independentes. O *RemoteInvocationChannel* é um canal usado para invocar as operações no *ContextSensor*. Também é usado pelo *ContextSensor* para devolver as respostas de leitura pontual. O NotificationChannel é um canal utilizado apenas para o envio das notificações resultantes das operações de leitura cíclica.

As composições desses canais são simples, consistindo da própria camada *ContextSensorSession*, que é partilhada entre esses e os canais da aplicação, e de uma camada de rede que tem de ser do mesmo tipo que a camada de rede utilizada nesses canais pelas consolas. O Appia disponibiliza concretizações dos protocolos UDP e TCP que podem ser usados como camada de rede. Ao optar pelo protocolo TCP através da camada *TcpComplete*, a ligação irá beneficiar da fiabilidade que caracteriza este protocolo. O protocolo UDP implementado pela camada UdpSimple fornece um serviço de entrega que não garante a entrega dos eventos. Note-se que é da responsabilidade do *ContextSensor* inicializar esses canais.

As operações de leitura pontual e leitura cíclica retornam os valores lidos por canais diferentes. Esta abordagem permite num cenário como o descrito na figura 6 que as leituras pontuais sejam feitas utilizando uma ligação fiável e por outro lado as leituras cíclicas sejam devolvidas através de uma ligação não fiável, possivelmente para múltiplos destinatários utilizando um endereço de difusão selectiva (*multicast*).

Em resumo, a utilização de canais em separado permite a definição de um interface mais flexível entre consolas de monitorização e o *ContextSensor* e o suporte de arquitecturas de rede mais elaboradas. O custo desta solução prende-se com o maior número de recursos utilizados e pela necessidade de impor regras na identificação dos canais de controlo de modo a que possam ser distinguidos.

4.3 Processamento de eventos

O processamento de eventos pode ser dividido em duas funções distintas. Uma função que implementa a lógica do fluxo de eventos associados a cada operação e outra função que consiste no tratamento dos eventos pelas sessões monitorizadas.

O fluxo dos eventos é implementado pela sessão *ContexSensorSession* e pelas sessões monitorizadas ao estenderem a classe *MonitorizableSession*. A primeira coordena o fluxo de eventos de invocação de um serviço e envio das respostas. Resume-se ao tratamento de um evento externo que invoca um serviço, à tradução desse num evento interno e por fim à tradução das respostas em eventos externos. As sessões monitorizadas são responsáveis por gerarem respostas no tratamento dos eventos internos que invocam uma operação. O modelo de dados usado é descrito na subsecção 4.3.1 onde são descritas as estruturas de dados utilizadas e são definidos os eventos utilizados. A subsecção 4.3.2 descreve para cada tipo de operação o fluxo de eventos gerados.

A outra função é realizada pelas próprias sessões monitorizadas, ao invocarem métodos herdados da classe *MonitorizableSession* que manipulam o estado do

protocolo. Esta função é descrita na subsecção 4.3.3 onde é descrito o processo de leitura ou escrita dos atributos de uma sessão.

4.3.1 Tipos de dados

Nesta secção são definidos os tipos e os eventos utilizados nas operações de leitura e escrita de atributos. Cada evento invocando uma operação no ContextSensor possui um identificador que depois é atribuído a cada resposta enviada. Na leitura cíclica quando é utilizado um endereço de difusão selectiva, esse identifica univocamente o pedido que deu origem à resposta.

Parâmetros dos eventos

Os parâmetros de entrada e saída de cada operação são guardados pelos atributos de cada tipo de evento. Quando esses eventos são transmitidos pela rede, esses parâmetros são serializados no campo Message do evento, sendo esta tarefa realizada pela sessão *ContextSensorSession*.

Os tipos de dados definidos são:

- RequestId – identifica um pedido e contêm uma String, um porto e um endereço IP.
- Attribute – contêm uma String que define o nome do atributo de uma sessão
- ValuedAttribute - contêm uma String que define o nome do atributo de uma sessão e um Object que guarda o seu valor.
- SessionAttributesList – contêm uma String que define o nome da sessão e um ArrayList cujos elementos são do tipo Attribute.
- SessionValuedAttributesList - contêm uma String que define o nome da sessão e um ArrayList cujos elementos são do tipo ValuedAttribute.
- ChannelAttributesList - contêm uma string que define o nome do canal e um ArrayList cujos elementos são do tipo SessionAttributesList.

Eventos

Os eventos recebidos pelo *ContextSensor* representam comandos e os enviados por este as respostas a esses comandos. São designados de eventos externos e estendem a classe *SendableEvent* disponibilizada pelo Appia que permite o seu envio pela rede. O Appia disponibiliza uma forma de serializar os seus parâmetros, disponibilizando no atributo Message uma zona de memória para onde esses parâmetros são serializados no envio e de onde são extraídos na recepção.

Os eventos externos são:

- CtxQueryEvent – Invoca a operação de leitura pontual. Contêm um RequestId e uma ChannelAttributesList.
- CtxAnswerEvent – Responde a uma operação de leitura pontual. Contêm um RequestId e uma SessionValuedAttributesList.
- CtxStartMonitorEvent - Invoca a operação de leitura cíclica. Contêm um RequestId, uma ChannelAttributesList, um Object que guarda o endereço de resposta e um Integer que define o intervalo entre cada resposta.
- CtxStopMonitorEvent - Cancela uma operação de leitura cíclica. Contêm um RequestId.
- CtxNotifEvent - Responde a uma operação de leitura cíclica. Contêm um RequestId e uma SessionValuedAttributesList.
- CtxUpdateEvent - Invoca a operação de leitura cíclica. Contêm um RequestId, uma ChannelAttributesList.

Os eventos internos estendem o novo tipo *CtxInternalReconfigEvent*, que estende a classe Event, e circulam nos canais monitorizados. Na recepção de um comando pelo canal de controlo é criado um evento interno inserido no canal monitorizado. O evento interno de resposta, gerado por uma sessão, é depois traduzido num evento externo enviado pelos canais de controlo.

Os eventos internos são:

- CtxIntQueryEvent - Contêm um RequestId e uma ChannelAttributesList.
- CtxIntAnswerEvent - Contêm um RequestId e uma SessionValuedAttributesList.
- CtxIntNotifEvent - Contêm um RequestId e uma SessionValuedAttributesList.

A utilização destes dois tipos de eventos justifica-se pela necessidade de salvaguardar que nenhum evento interno é introduzido por engano na rede. Deste modo os eventos externos estendem a classe SendableEvent que suporta o envio pela rede e os eventos internos estendem a classe Event que suporta troca de eventos entre sessões de um mesmo processo. Para mais detalhes sobre estes eventos, o leitor deve ler a subsecção 3.2.1 que explicados em detalhe os eventos Appia.

4.3.2 Fluxo de eventos

Leitura pontual

Na operação de leitura pontual de atributos é recebido um evento de pedido de leitura, ao qual o *ContextSensor* responde com vários eventos, cada um desses com os valores obtidos com o processamento de pedido por parte de uma das sessões. A leitura de atributos acontece quando o evento de pedido percorre o canal da aplicação e é tratado por uma sessão da qual são requeridos valores. Esta sessão devolve um evento de resposta com os valores dos seus atributos, inserindo-o no canal em sentido inverso. Podendo ser lidos atributos de várias sessões, a sessão verifica se haja mais atributos a ler de outras sessões, e só nesse caso volta introduzir o evento do pedido no canal.

O cenário da figura 7 mostra o fluxo de eventos numa operação de leitura de atributos de duas sessões distintas.

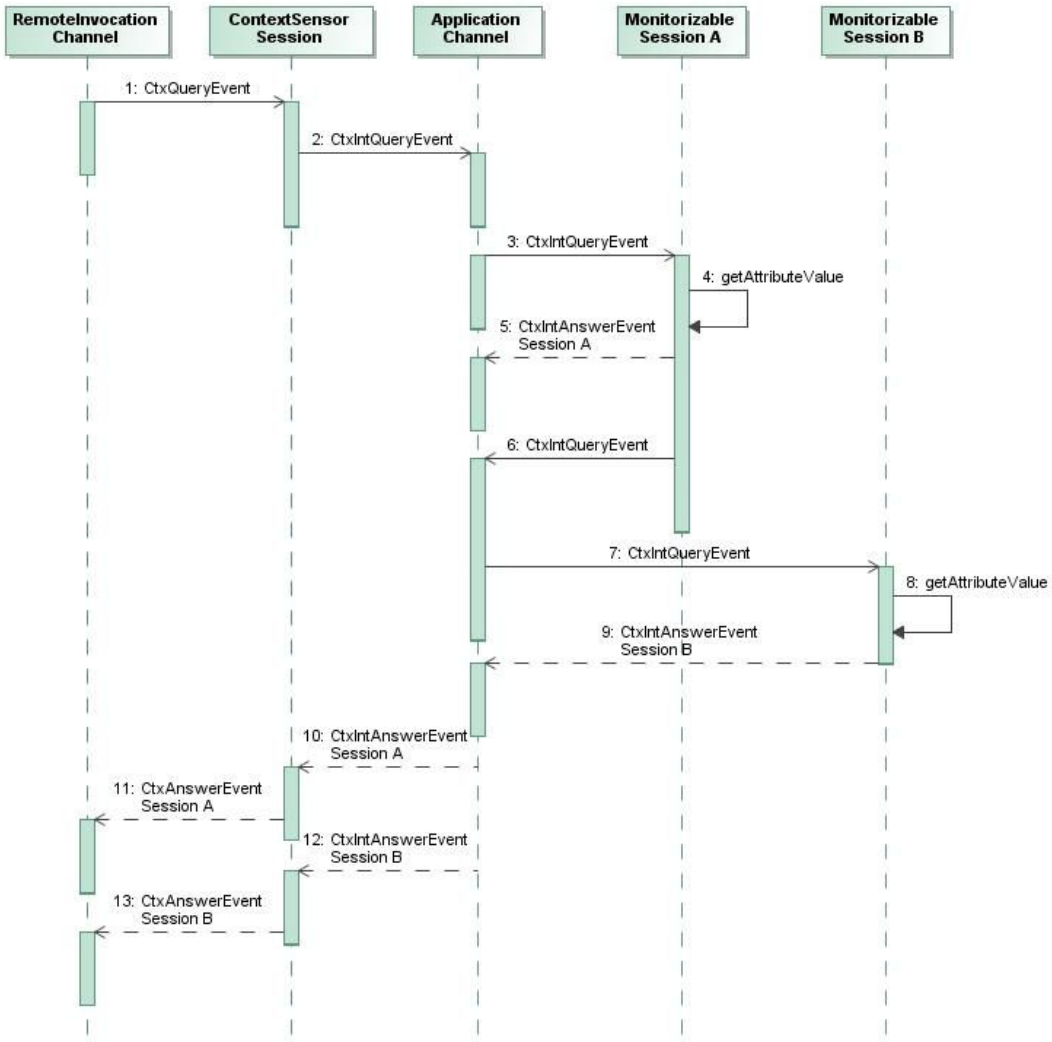


Figura 7 : Leitura pontual de atributos

No fluxo acima é possível identificar algumas particularidades interessantes. O evento de pedido de leitura (passos 3, 6 e 7) visita todas as sessões antes que qualquer das respostas (CtxIntAnswerEvent A e B) seja processada porque foram inseridas no canal no sentido inverso. Essas respostas são processadas pela ordem em que foram inseridas (eventos 10 e 12) e por cada uma é enviada uma resposta à consola pelo canal *RemoteInvocationChannel*. O pedido faz referência a duas sessões, logo cada uma responde com os valores obtidos com o método *getAttributeValue* disponibilizado na classe *MonitorizableSession*.

Leitura cíclica

A operação de leitura cíclica disponibiliza um mecanismo de notificação periódica dos valores de um conjunto de atributos de uma pilha de protocolos. Este mecanismo é iniciado por um evento que define os atributos a monitorizar e o período de tempo que separa cada resposta devolvida. A partir deste é criado um temporizador que estende a classe Appia *PeriodicTimer*. Este evento é criado pela sessão *ContextMonitorSession* e enviado para ser tratado pelo canal. Cada vez que o temporizador for despoletado, o canal envia um evento que é consumido pela sessão *ContextSensorSession*. Esta ao tratá-lo gera um evento de leitura cíclica que é introduzido no canal monitorizado.

O fluxo de eventos é semelhante ao usado na leitura mas são usados eventos diferentes que permitem distinguir as operações e devolver uma resposta à consola por um canal diferente, neste caso pelo *ContextNotificationChannel*. O cenário da figura 8 mostra o fluxo de eventos num pedido de leitura de atributos apenas da Session A, ou seja, no caso em que mesmo suportando a monitorização, não são lidos atributos da Session B.

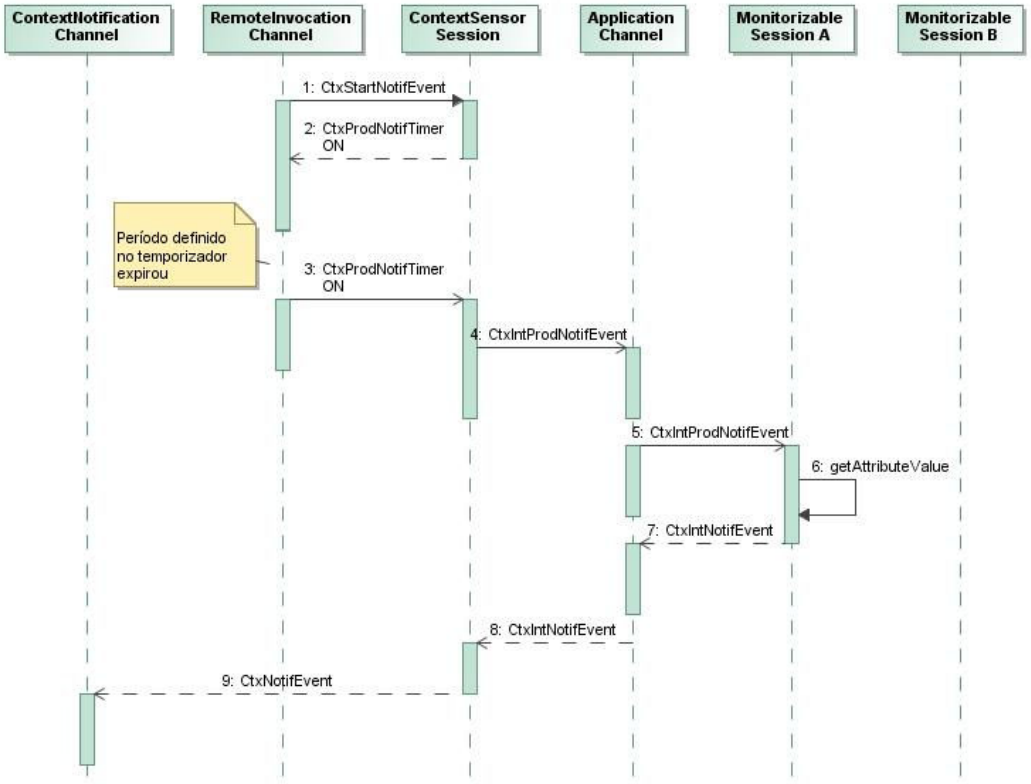


Figura 8 : Leitura cíclica de atributos

Note-se no fluxo o envio da resposta (passo 9) pelo ContextNotificationChannel, ao contrário da leitura pontual que usa o canal *RemoteInvocationChannel*. Observa-se também a optimização feita no tratamento dos eventos internos que invocam operações pois a sessão A verifica que não há mais atributos a ler e não introduzo o evento de pedido de novo no canal.

Após o registo para notificação de atributos serão enviadas respostas até que esse registo seja cancelado. A operação é suportada por evento próprio que quando processado pela *ContextSensorSession* limpa os registos internos associados à operação de registo.

Todavia o canal não é informado da paragem do serviço de notificação pelo que continua a gerar eventos temporizadores periodicamente. O evento temporizador é cancelado quando é recebido o próximo evento a seguir à paragem do serviço. A sessão *ContextSensor* ao recebe-lo, não encontra nenhuma referência a este e envia para o canal o mesmo evento mas desactivado. O canal pára então de enviar o evento periódico desta operação. O fluxo seguinte mostra os eventos associados à paragem do serviço, onde se pode ver que toda esta operação é transparente para os canais monitorizados.

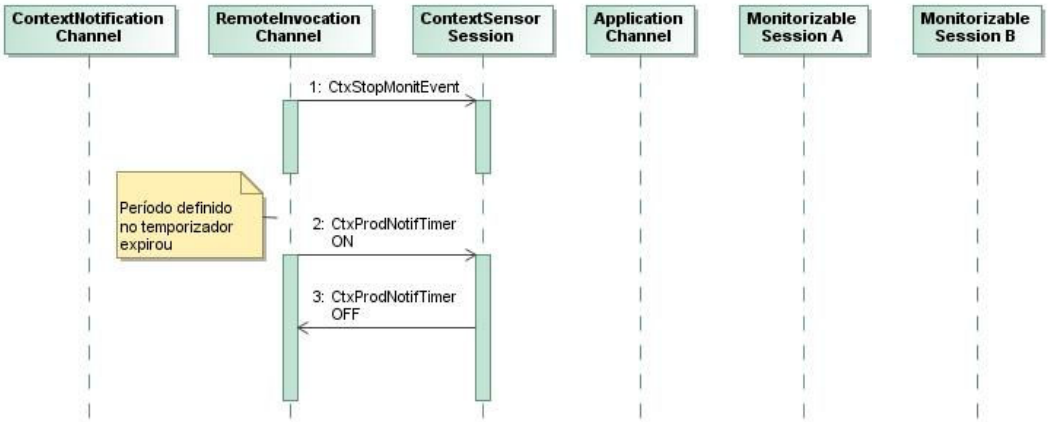


Figura 9 : Desactivação do serviço de leitura cíclica

Este fluxo mostra o processo de desactivação de um evento periódico, neste caso, o passo 2 despoletado pelo canal é desactivado enviando o evento no passo 3.

Escrita de atributos

A operação de escrita de atributos é caracterizada por não devolver resposta. O cenário da figura 10 mostra o fluxo de eventos de um pedido de escrita que apenas incide sobre atributos da Session B. Neste caso o evento é entregue à Session A mas como não existem atributos desta sessão a reconfigurar, a sessão volta a introduzir o evento sem invocar o método setAttributeValue.

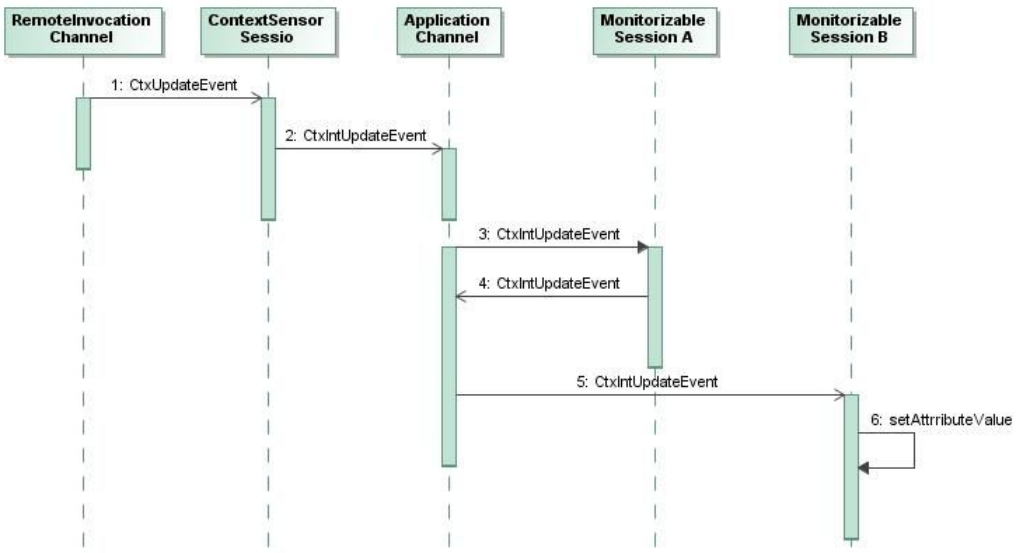


Figura 10 : Escrita de atributos

Um aspecto que se pode observar é visível na descontinuidade no processamento do evento *CtxIntProdNotifEvent* (passos 2 e 3). Quando o evento é inserido no canal assincronamente vai para a lista Waiting acedida concorrentemente. Deste modo não há garantia que qualquer outro processo tenha inserido um outro evento nessa lista, logo não se pode garantir que o canal o entregue à Session A logo que ele tenha sido inserido pela *ContextSensorSession*. Este comportamento é generalizável para todos os eventos que a sessão *ContextSensorSession* insere num canal quando processa um evento de outro canal.

4.3.3 Leitura e escrita de atributos

A leitura e escrita de valores são realizadas quando uma sessão recebe um evento que define um pedido. Qualquer sessão que pretenda suportar a leitura e escrita dos atributos pode estender a classe *MonitorizableSession* ou implementar ela própria os métodos de tratamento dos eventos.

A implementação fornecida processa o evento procurando no seu atributo que define a lista de atributos a operar, uma referência ao seu identificador de sessão. Se não houver significa que esta não terá que devolver qualquer resposta e envia o evento para o canal. Se for requerido que execute a operação sobre um conjunto de atributos seus, itera nesse conjunto de atributos e invoca os métodos de leitura ou escrita, respectivamente *getAttributeValue* e *setAttributeValue*.

Na leitura, à medida que vai lendo os atributos vai adicionando esse atributo e valor num evento de resposta. No fim da iteração envia o evento de resposta se necessário, remove o seu conjunto de atributos da lista do evento e se esta não ficar vazia envia o evento do pedido.

Leitura

A leitura dos valores de atributos assemelha-se nas semânticas de leitura pontual e de notificação. A existência de dois métodos de tratamento de eventos distintos justifica-se por em cada semântica ser usado um evento de pedido e um evento de resposta específico. Esta diferenciação permite que o tratamento da resposta pela sessão *ContextSensorSession* utilize a classe do vector para devolver as respostas pelos dois canais como descrito na secção 4.2. Deste modo os eventos de pedido ou de resposta em ambas as semânticas são constituídos pelos mesmos atributos, respectivamente uma lista de objectos *SessionAttributesList* ou de objectos *SessionValuedAttributesList*.

Sendo a plataforma desenvolvida na linguagem Java, torna possível usar o mecanismo de introspecção dos objectos que possibilita a leitura dos atributos das sessões especificando apenas o seu nome. Deste modo consegue-se reduzir a quantidade

de informação transmitida no pedido e resposta ao mesmo tempo que se simplifica o interface. Outra grande vantagem está na redução de ocorrência de erros por a classe ou tipo do atributo serem mal definidos. A abordagem seguida assenta numa lógica de ler o atributo com o nome especificado e retornar um objecto que a consola de monitorização saberá interpretar.

A leitura do valor de cada atributo é concretizada no método *getAttributeValue* da classe *MonitorizableSession* fazendo uso da reflexividade da linguagem Java. Neste método através do nome do atributo é obtido um objecto da classe Field que permite ler o valor desse atributo, desde que esse seja um atributo público.

A impossibilidade de ler atributos protected e private é aceitável uma vez que se a classe restringe o acesso a esses atributos por outras classes, fará algum sentido que não seja acessível a partir de uma consola.

Nem todos atributos são lidos da mesma maneira, existindo condicionantes dependentes do seu tipo. Por exemplo os atributos de tipos de dados primitivos necessitam de ser encapsulados num objecto compatível que depois é devolvido na resposta.

Da mesma forma as listas e vectores também apresentam algumas condicionantes. Os objectos da Collections Framework do Java são serializáveis, e embora possam ser devolvidos na resposta, pode haver problemas por causa dos efeitos colaterais. Neste caso se o objecto lido é alterado antes de a resposta ser devolvida, o valor retornado é o valor do objecto no momento do envio e não do momento em que o evento de leitura é processado. No caso dos vectores definidos com [ ], como estes não são serializáveis logo não podem ser enviados directamente para a consola.

Para contornar esta questão teria de ser desenvolvido um mecanismo que retornasse esse array encapsulado num objecto como a ArrayList ou proceder ao envio desse do número de elementos seguidos por cada um desses. Considera-se que a implementação deste mecanismo, pela sua complexidade, cai fora do âmbito do trabalho desenvolvido.

Quando invocada uma operação de leitura pode dar-se o caso que o nome de algum atributo tenha sido mal definido. Outra situação possível é ler um atributo cujo valor é indefinido, ou seja, o seu valor seja null. Para que possa ser feita a distinção entre estes dois casos, na resposta devolvida são incluídos apenas os atributos com identificador válido, mesmo que com valor a null.

Escrita

A escrita do valor de cada atributo é concretizada no método *setAttributeValue* da classe *MonitorizableSession* e segue a mesma abordagem da leitura de atributos, ou

seja, utilizando a reflexividade da linguagem Java cria um objecto de manipulação do atributo do tipo Field. Alguns dos constrangimentos identificados na leitura aplicam-se na escrita, um exemplo disso é a possibilidade de definir o valor apenas de atributos públicos.

Uma possível resolução desta limitação passaria por definir métodos públicos de acesso a cada atributo. Deste modo em vez de manipular o valor dos atributos directamente seria chamado o método. Esta é aliás uma abordagem seguida na implementação de objectos Java designados de Beans quando é comum o seu acesso remoto. Com um mecanismo deste género bastaria invocar cada um dos métodos simplificando o processo quer de escrita quer de leitura.

Os atributos redefinidos têm de ser tipos primitivos ou objectos serializáveis, sendo que no primeiro caso o valor recebido no evento é um objecto de encapsulamento que será automaticamente traduzido para o atributo da sessão. Pelos mesmos motivos que na leitura, os atributos do tipo vector definidos com [ ] não são suportados.

Por outro lado, o problema dos efeitos colaterais dos objectos da Collections Framework não se aplica. Na escrita o objecto é membro do evento e não será acedido por outra sessão para além daquela que utiliza o seu valor para alterar um atributo da sua instância.

O retorno de resposta na operação de escrita foi equacionado. No entanto os novos valores podem ser confirmados com um pedido de leitura subsequente. Comparativamente, este pedido implica apenas mais um evento a circular no canal monitorizado (dois em vez de um), em relação à opção de retornar a resposta pelo que se justifica a opção tomada.

4.3.4 Atomicidade das operações

A possibilidade de ler atomicamente atributos de várias sessões do mesmo canal oferece à consola a possibilidade de obter uma vista do estado da composição, possibilitando a recolha de uma informação mais abrangente. Do mesmo modo, redefinir o valor dos atributos de várias sessões, permite realizar uma reconfiguração no âmbito de todo o canal.

A atomicidade das operações de leitura e escrita está fortemente dependente da implementação do algoritmo de escalonamento de eventos implementado pela classe *EventScheduler*. A garantia pode ser dada por este entregar cada evento a todas as sessões do seu caminho antes que qualquer outro, desde que nenhuma sessão ao tratá-lo gere eventos no mesmo sentido desse.

O processamento dos eventos internos pelas sessões de um canal verifica estas condições. Assim sendo com uma única operação consegue-se gerir uma imagem do estado da composição num dado instante, monitorizando não uma mas várias sessões. Esta possibilidade tem vantagens do ponto de vista de utilização da ferramenta um vez que permite obter resultados diferentes da invocação de duas operações consecutivas. Por este motivo os eventos que suportam a invocação das operações foram definidos no sentido de especificar um conjunto de sessões e para cada uma dessas, um conjunto de atributos.

As aplicações que usem vários canais de comunicação, podem com um único *ContextSensor* monitorizar sessões dos vários canais, injectando a sessão partilhada *ContextSensorSession* nas suas pilhas. Seria possível no tratamento de um evento de pedido, introduzir um evento interno por cada canal, invocando a operação em sessões de vários canais. No entanto como a garantia da atomicidade não seria dada optou-se por não disponibilizar um interface que permitisse esta operação uma vez que o resultado será o mesmo da invocação de vários pedidos endereçados a cada um dos canais. Tal não é possível porque os eventos internos que são introduzidos nos canais da aplicação, são colocados pelo *EventScheduler* na lista Waiting. Como é explicado na secção 3.4.2, esta lista é acedida concorrentemente pelo que não existe a garantia que dois eventos internos introduzidos em canais distintos não são intercalados por um evento introduzido por um outro fio de execução.

4.3.5 Geração de respostas

Como já várias vezes foi referido, na leitura de atributos é gerada uma resposta por cada sessão ao processar um evento de invoca uma operação. Esta abordagem implica um maior consumo de recursos na medida em que gera vários eventos. Outra abordagem poderia ter sido seguida. Por exemplo, uma solução em que o evento de pedido recolhesse os valores dos atributos das várias sessões e que quando todos os valores tivessem sido recolhidos fosse gerada uma única resposta. A opção tomada deve-se à necessidade de garantir que a resposta é devolvida. O problema do não envio de resposta, está relacionado com a optimização feita no processamento do evento de pedido em cada sessão, descrita na secção 4.3.2. Este problema pode acontecer por dois motivos: incorrecta definição do pedido ou alteração da composição.

A incorrecta definição do pedido acontece quando na lista de atributos é especificada uma sessão que não existe. Na abordagem de uma resposta por sessão, o evento circula no canal enquanto houver sessões a monitorizar. Se uma sessão especificada não existir, o evento é processado por todas as sessões válidas que geram a sua própria resposta. Quando percorre todo o canal o evento é descartado. A resposta para as sessões inválidas não é gerada porque nenhuma sessão foi identificada com o

pedido. Se apenas uma resposta fosse gerada, teria de ser gerada pela última sessão a preencher os valores, e neste dar-se-ia o caso em que o pedido percorria a composição sem satisfazer todos os pedidos, logo não seria gerada uma resposta.

A alteração da composição também origina o mesmo problema, uma vez que mesmo que o pedido pode referir apenas sessões válidas no momento em é introduzido no canal, o mecanismo de reconfiguração, descrito no capítulo seguinte possibilita que uma dessas sessões seja removida ou substituída antes que o pedido seja tratado por essa. Esta questão impossibilita até a validação do evento no momento em que é introduzido no canal. Mais uma vez a abordagem seguida garante o retorno de resposta para as sessões que após o desbloqueio ainda sejam válidas.

4.4 Adaptação dos protocolos

A plataforma Appia disponibiliza concretizações para vários protocolos, que terão de ser adaptadas para suportar a funcionalidade. A solução proposta teve a preocupação de reduzir ao mínimo o esforço dispendido nessa adaptação.

Numa lógica orientada aos eventos, faz sentido que essa adaptação passasse por adicionar os eventos ao conjunto de eventos processados pela camada e fornecer rotinas de tratamento genéricas dos eventos de suporte à funcionalidade. A disponibilização de uma implementação para essas rotinas tem vantagens na uniformização do algoritmo usado e na reutilização, evitando que cada protocolo tenha a necessidade de implementar o processamento desses eventos. A declaração dos eventos pela camada é um requisito imposto pela implementação da própria plataforma Appia.

Conforme descrito na secção 3.1, a definição dos protocolos no Appia é especificada a dois níveis: um estático por extensão da classe *Layer* e outro por implementação por extensão da classe *Session*. O suporte da funcionalidade tem impactos nas duas definições.

A definição estática de cada protocolo deverá incluir os eventos internos definidos pelo mecanismo de monitorização. Sendo assim, a *Layer* que define o protocolo deve ser alterada, incluindo no conjunto de eventos que a camada pretende processar os eventos *CtxtIntQueryEvent*, *CtxtIntProduceNotifEvent* e *CtxtIntReconfigEvent*. O conjunto de eventos que a camada gera deverá incluir também os eventos *CtxIntAnswerEvent* e *CtxtIntNotifEvent*. Uma possível optimização seria a extensão da classe *Layer* definindo uma nova classe *MonitorizableLayer* que incluísse de raiz esses eventos nos conjuntos referidos. Esta simplificação da adaptação dos protocolos não pode ser aplicada porque a classe de base apenas declara as variáveis que guardam cada conjunto, sendo a instanciação dos vectores realizada pelas *Layers* de cada protocolo, especificando nesse momento o número de eventos que cada conjunto terá.

A concretização de cada protocolo tem de ser alterada para que os eventos de monitorização sejam processados. Esta alteração consiste na alteração da sua classe base, substituindo a classe *Session* pela classe *MonitorizableSession* fornecida. Com esta alteração, o protocolo herda os métodos de tratamento de eventos "*handleContextInternalEvents*", que processa todos os eventos internos. Deve ser acrescentada uma invocação no método "handle" da classe a este, para os eventos do tipo *ContextInternalEvent*, classe que todos os eventos internos estendem e que por sua vez estende a classe Event.

4.5 Resumo

Os eventos de base da plataforma Appia foram utilizados como veículo de passagem de informação entre as consolas de monitorização e a aplicação e mesmo entre sessões da composição. Os eventos suportam os parâmetros de entrada e saída dos serviços de forma estruturada ao mesmo tempo que a sua especialização facilita a projecção de cada evento numa operação ou numa etapa da sua concretização.

A flexibilidade oferecida na composição de protocolos, e a possibilidade de as sessões partilharem o seu estado facilita a definição de um fluxo de eventos que permite abranger a leitura ou escrita de atributos de várias sessões e em vários canais alargando o espectro de atributos acessíveis em cada operação.

O escalonamento de eventos, impondo regras de ordenação bem definidas, possibilitou a implementação de operações que incidissem sobre várias sessões com garantia de atomicidade de execução. Deste modo conseguiu-se um mecanismo que permite a leitura ou escrita no âmbito da composição de protocolos.

O desenvolvimento da plataforma na linguagem Java, simplificou a manipulação dos atributos na medida em que a reflexividade sobre os objectos de sessão permitiu a identificação não tipificada desses atributos, com vantagens na redução de erros na instanciação dos mesmos.

Em resumo, foram concretizadas as semânticas que permitem a monitorização e reconfiguração dos protocolos, respondendo à necessidade de adaptação destes em função do dinamismo das condições de execução. Algumas propriedades da plataforma Appia permitiram enriquecer os serviços disponibilizados no sentido de fornecer um interface de gestão dos protocolos simples mas ao mesmo tempo de âmbito alargado.

Capítulo 5

Reconfiguração de composições

O conjunto de propriedades requeridas por uma aplicação pode variar ao longo do tempo. Os exemplos seguintes apresentam três situações, com requisitos distintos do ponto de vista da plataforma de suporte à composição, de alterações à composição:

- A aplicação pode necessitar de incluir uma camada de depuração, que permita analisar os eventos em circulação num canal. Esta camada é uma propriedade complementar, que disponibiliza um serviço que é utilizado apenas durante um curto espaço de tempo. A possibilidade de adicionar e remover esta camada dinamicamente ao canal, permite que esse seja construído apenas com as propriedades requeridas pela aplicação e que, sem interrupção do serviço, sejam adicionadas outras apenas enquanto forem necessárias.
- Por razões de desempenho, pode ser vantajoso proceder à substituição de uma camada por outra, que concretize o mesmo protocolo mas com um algoritmo distinto. Se esta substituição for realizada sem que haja uma paragem do canal, será uma operação transparente para a aplicação que não vê alterado o fluxo de eventos que circulam no canal.
- Por vezes as alterações da composição incidem não sobre uma mas sobre várias camadas e envolvem uma combinação de operações que não podem ser realizadas apenas com uma invocação. Um exemplo seria a alteração do conjunto de camadas que fornecem entrega fiável ponto-a-ponto de uma combinação utilizando as camadas UDP e FIFO pela camada TCP.

A reconfiguração das composições em execução coloca alguns desafios não negligenciáveis, uma vez que há que garantir a manutenção da coerência do estado dentro de uma mesma instância da composição (por exemplo garantir que apesar da alteração do serviço de fiabilidade nenhuma mensagem é perdida) e entre instâncias, uma vez que as mensagens têm que atravessar no receptor a mesma sequência de camadas que atravessaram no emissor. Este capítulo descreve as alterações realizadas à plataforma Appia para suportar estas alterações e os requisitos que o modelo

apresentado coloca às camadas. O desenvolvimento de camadas satisfazendo estes requisitos sai fora do âmbito deste trabalho e é relegado para trabalho futuro.

A versão original do Appia não dispõe dos mecanismos que permitem que a composição de um canal seja alterada após este iniciar a operação. Este capítulo apresenta uma solução baseada em eventos que permite adicionar, remover e substituir camadas de uma composição em tempo de execução, garantindo que nenhum evento é processado pelas camadas afectadas pela reconfiguração enquanto todas as alterações não são concluídas. Para tal disponibiliza uma forma de impedir que algumas das sessões de um canal processem eventos (bloqueio), colocando as sessões num estado quiescente. A reconfiguração pode afectar uma ou mais camadas, permitindo realizar várias operações sobre mais do que uma sessão com garantia de atomicidade. O trabalho assegura que após a reconfiguração, não existem restrições à circulação de eventos, sendo totalmente respeitada a semântica original.

O sistema de bloqueio tira partido da optimização do caminho percorrido por cada evento, limitando tanto quanto possível o seu impacto às sessões bloqueadas, o que permite que em algumas situações os eventos atravessem a pilha não sendo bloqueados pelas sessões quiescentes. A arquitectura consiste num componente integrado nas composições da aplicação, o *ContextAgent*, que coordena as operações nos canais. Os comandos são transmitidos ao *ContextAgent* por um canal Appia, o *ReconfigurationChannel* designado de canal de controlo.

O canal de controlo e a estrutura a composição protocolar que o forma são caracterizados da mesma forma que na arquitectura da reconfiguração de sessões. A lógica de troca de eventos também é semelhante. Deste modo não será repetida essa descrição neste capítulo.

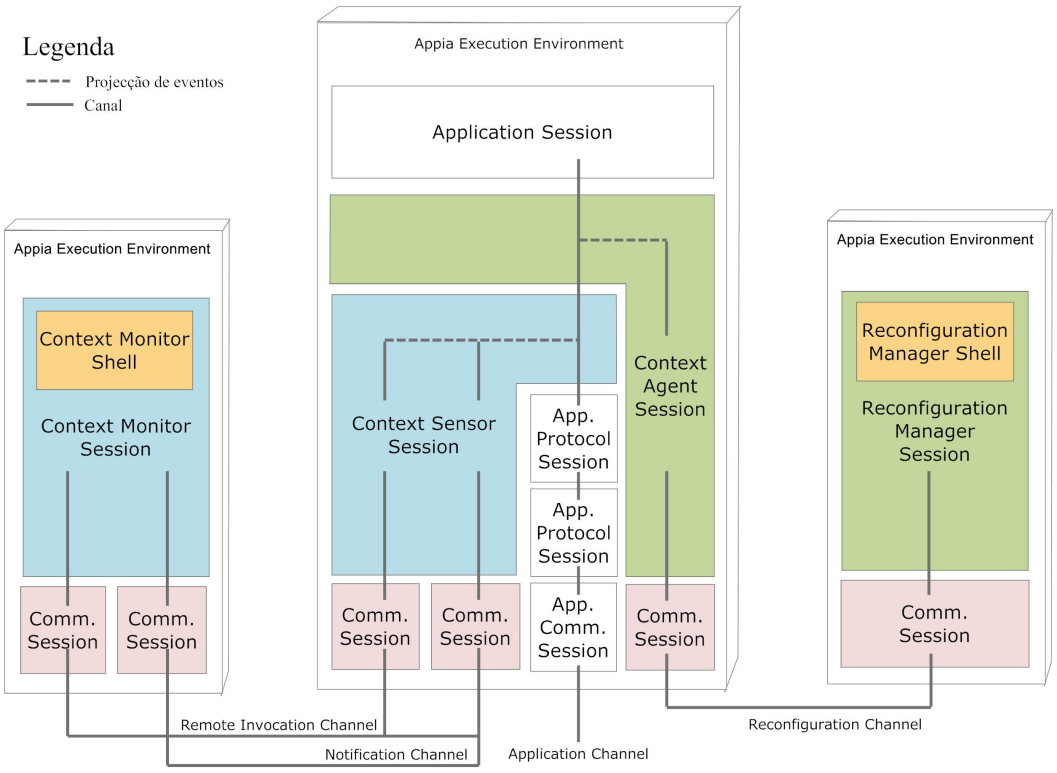


Figura 11 : Arquitectura de Reconfiguração e Monitorização

A reconfiguração é coordenada por um agente, também ele concretizado como uma sessão que é injectada na pilha e que converte os eventos externos em eventos internos. O tratamento desses comandos implicou a extensão da própria plataforma, ao dotar o núcleo da capacidade de alterar as rotas que definem dinamicamente o caminho percorrido pelos eventos.

5.1 Concretização das Operações de reconfiguração

O mecanismo de reconfiguração permite alterar uma determinada pilha alterando o vector de sessões que a constituem. Este disponibiliza operações de remoção, adição e substituição de sessões de um canal.

Para facilitar a recuperação do canal a partir do estado quiescente das sessões, é dada uma garantia de atomicidade das operações de reconfiguração e que permite ao utilizador reconfigurar uma pilha sem que nenhum evento seja processado pelo canal, ou seja, que o evento de reconfiguração seja processado e todas as alterações que daí decorram sejam visíveis antes de qualquer outro evento circule no canal.

Cada uma das operações também requer um tratamento diferenciado. Por exemplo, a remoção de sessões de uma pilha pode ser definida à custa do nome da sessão. Por outro lado, ao adicionar uma sessão tem de ser especificada a posição onde essa é

inserida e ao mesmo tempo o tipo da nova sessão, ou seja, a subclasse de *Layer* que lhe dá origem.

Nos casos de substituição de sessões pode haver interesse em inicializar a nova sessão com alguns dos atributos da sessão que essa substitui. Ao serem removidas da pilha as sessões podem ter interesse em ser notificadas e desse modo proceder a à libertação de alguns recursos. Para tal, as sessões deverão implementar o interface *ReconfigurationHandlingSession* que declara os métodos que serão chamados pelo núcleo antes de processar um determinado comando sobre essa camada. Estes métodos permitem à própria sessão realizar alguma actividade que seja necessária para manter a coerência na lógica do protocolo que implementam, sendo que poderão enviar eventos, desde que de forma assíncrona sob pena de violar a ordem dos eventos.

5.1.1 Processamento no núcleo

Para reconfigurar as composições de serviços do Appia foi utilizada uma estratégia de processamento dos eventos diferente da utilizada para os eventos de monitorização em que os eventos são inseridos no canal e tratados por cada uma das sessões. Na reconfiguração das composições, o processamento dos eventos é realizado pelo núcleo.

Para as diferentes operações foram usadas abordagens diferentes com vista ao tratamento atómico dos eventos.

Por exemplo, os eventos internos de adição, remoção e substituição de uma sessão são processados pelo próprio canal, sendo os eventos internos subclasses da classe *ChannelEvent*. No entanto, o canal só altera a informação que retém sobre a composição, sendo necessário invocar um método de actualização do escalonador de eventos que será responsável por actualizar os eventos que se encontram já a aguardar processamento de acordo com a nova composição do canal.

Já os eventos que gerem o bloqueio dos canais são processados pela sessão *ContextAgentSession*, que cria um evento interno que ela própria processa, invocando depois o método do escalonador de eventos que bloqueia uma sessão. A utilização de uma abordagem centralizada no núcleo prende-se com a necessidade de operar ao nível do canal. Os impactos da reconfiguração vão além da informação manipulada por cada sessão, pelo que o processamento destes eventos apenas pelas sessões seria insuficiente. Ao proceder à reconfiguração no núcleo consegue-se ter o controlo de todas as estruturas de dados que definem o canal ao mesmo tempo que da informação que este guarda sobre o vector de sessões.

Sendo assim, os eventos inseridos nos canais pelo *ReconfigurationAgent* são processados pela classe *Channel*. As operações de adição, remoção ou substituição de

algumas das camadas de uma pilha têm impactos na definição dinâmica e estática das composições. Embora cada um destes comandos tenha especificidades, as alterações podem ser circunscritas às instâncias das classes nucleares *QoS, QoSEventRoute, EventScheduler, Channel, Event e ChannelEventRoute*, sendo de certa forma refeito o cálculo das rotas conforme explicado na secção 3.1.

5.1.2 Actualização do canal

O conceito de composição no Appia é suportado por dois objectos que o representam conforme descrito na secção 3.1. A alteração da composição tem naturalmente impactos sobre essas estruturas de dados que representam em cada momento o conjunto de serviços disponibilizados à aplicação. O processo de alterar essa informação é baseado no próprio processo de criação do canal, no entanto existem algumas diferenças. A abordagem seguida consiste em criar uma nova QoS e alterar o objecto Channel que guarda as sessões do canal. Outra possibilidade seria a criação de um novo canal, mas partindo do princípio que o conceito de reconfiguração dinâmica implica não reiniciar o canal, optou-se por esta via, que simplifica a actualização dos eventos pendentes que fazem referência a esta instância.

A nova QoS é definida à custa da QoS original. As operações de reconfiguração apenas determinam quais as alterações a efectuar sobre a configuração existente. Por exemplo, na remoção de camadas, é definido apenas o índice da posição da camada a remover. Deste modo o novo vector de *Layers* tem de ser criado com base no existente. O vector original é obtido da *QoS* atribuída ao canal na sua criação. A partir deste é criado um novo vector para onde são copiadas as *Layers* que se mantêm após a reconfiguração. Na remoção esse vector será formado por um subconjunto das *Layers* do vector original, ao passo que na adição ou substituição algumas das novas *Layers* são criadas a partir do nome da classe especificado no pedido.

A partir do novo vector de *Layers* é instanciada uma nova *QoS*, que gera o um conjunto de rotas estáticas de acordo com a nova estrutura da composição. Essas rotas são um conjunto de objectos da classe *QoSEventRoute* definindo as *Layers* relevantes para cada tipo de evento. Note-se que alterando as *Layers*, o conjunto dos eventos que circulam no canal pode ser alterado.

O canal como foi dito anteriormente guarda informação sobre o caminho percorrido por cada evento. Com a redefinição da *QoS* essa informação tem de ser refeita. O cálculo desses caminhos utiliza a *QoS* (da qual utiliza as rotas estáticas geradas na sua criação) e o vector de sessões que o compõe. O vector de sessões do canal original é pois substituído por um novo vector, que inclui as sessões que se mantêm e em alguns casos mais algumas sessões. A lógica é semelhante à alteração

feita no vector de *Layers* da *QoS*, sendo que as novas sessões não são instanciadas mas sim obtidas por invocação do método *createSession* da *Layer* correspondente.

Com o vector de sessões e a QoS redefinidas o canal pode recalcular os caminhos dos eventos que substituem os existentes. O vector de sessões dum determinado tipo de evento pode ser alterado por uma das sessões ter deixado de fazer parte do canal ou por ter sido adicionada uma sessão que também o pretende tratar. Essa projecção é refeita invocando o método *makeEventsRoutes* da classe *Channel*, que parametriza com o novo vector de sessões do canal e a nova *QoS*. Este método vai redefinir um novo conjunto de objectos *ChannelEventRoute* que o canal guarda, de forma a atribuir um vector de sessões a cada evento que seja inserido.

Sendo esse vector de sessões atribuído a cada evento quando este é inicializado, a alteração do canal descrita acima, apenas tem impactos nos eventos que venham a ser inseridos após essa reconfiguração. Os eventos pendentes não verão estas alterações e terão que ser actualizados individualmente.

5.1.3 Actualização das listas de eventos

Com a reconfiguração do canal, o vector de sessões que pretendem processar um determinado tipo de evento pode mudar. Essa alteração é reflectida no objecto *ChannelEventRoute* do tipo desse evento que o canal cria ao ser actualizado. Cada evento quando é inicializado obtém desse objecto a sua rota no canal. Da reconfiguração resulta que o mesmo evento, requerendo a sua rota antes ou depois da actualização do canal pode obter rotas diferentes.

Esta questão é problemática, no sentido em que enquanto o evento de reconfiguração é processado poderão haver eventos assíncronos a ser introduzidos na lista *Waiting*, que recorde-se é acedida concorrentemente. Isto implica que os eventos pendentes (que aguardam na lista *Waiting* e que já foram inicializados), cujo vector com que foram iniciados é diferente do vector obtido do canal, têm de der actualizados.

Nem sempre existem diferenças. Por exemplo, se uma sessão for inserida e não pretender tratar o evento, o seu vector de sessões mantêm-se inalterado. A regra aplicável é:

- se a operação de reconfiguração envolver uma sessão que trate um evento inserido no canal, e o seu vector de sessões tiver sido obtido antes da actualização do canal, então o seu vector terá que ser actualizado.

Não é possível determinar o momento em que foi inserido um evento assíncrono, pelo que a necessidade de actualizar esse evento é determinada com a comparação dos dois vectores. Sempre que forem diferentes o evento terá que ser actualizado. Dado isto, o processo de actualização do evento começa por requer ao canal reconfigurado o vector

de sessões que lhe seria atribuído caso fosse inserido nesse instante. O vector de sessões obtido e o vector com o qual o evento foi inicializado são comparados. Se ambos os vectores forem constituídos pelas mesmas sessões esse evento não é afectado pela reconfiguração e não precisa de ser actualizado. Se forem diferentes então o vector obtido do canal é atribuído ao evento e os índices de posicionamento são recalculados.

Os índices de posicionamento são o índice da primeira sessão e o índice da sessão corrente. O índice da primeira sessão é sempre recalculado, no entanto só ficará desactualizado com a alteração do vector, se o evento percorrer a pilha do topo para a base pois nesse caso é inicializado com o número de sessões.

O índice da sessão corrente, apenas é actualizado cada vez que o evento for entregue a uma sessão. Uma vez que os eventos na lista *Waiting* estão pendentes e ainda não foram processados por nenhuma sessão, esse índice não sofre alterações.

Note-se que o algoritmo de escalonamento nos dá a garantia que no momento em que o evento de reconfiguração é processado só existem eventos pendentes, e não existe nenhum evento em circulação, ou seja, todos os eventos estão na lista *Waiting*. Mais, como a lista *Waiting* contêm eventos dos vários canais geridos pelo escalonador, a alteração cinge-se aos eventos do canal reconfigurado.

5.1.4 Transferência de estado

A operação de substituição de uma sessão S1 por outra newS1 tipicamente é realizada com o intuito de substituir uma camada por outra concretização de um serviço. Nestes casos pode ser vantajoso iniciar a nova sessão com alguma informação do estado da sessão substituída, por esses atributos fazerem sentido em ambas as implementações e para tornar a operação mais imperceptível do ponto de vista da composição.

Deste modo a operação de substituição de sessões permite que a nova sessão seja inicializada com alguma informação de estado da sessão existente. Para tal a operação define um conjunto de identificadores de atributos da sessão S1 que são projectados num conjunto de identificadores de atributos da sessão newS1. Esta projecção é directa, ou seja, a cada atributo lido corresponde um atributo ao qual é atribuído esse valor. Assim sendo, ambos os atributos têm de ser de tipos compatíveis.

Para concretizar este mecanismo foram usados os métodos que obtêm e alteram o estado dos atributos de uma sessão, descritos no capítulo 4. Isto implica que ambas as sessões estendam a classe *MonitorizableSession*. O processo de transferência de estado é concretizado no momento em que o vector de sessões do canal é actualizado, pelo que será visível a todos os atributos que venham a ser entregues à nova sessão.

5.1.5 Sessões partilhadas

As operações de reconfiguração do canal envolvendo uma sessão partilhada têm particularidades uma vez que não é possível garantir a atomicidade. Deste modo, para proceder à remoção ou substituição de uma sessão partilhada, esta tem de ser bloqueada. Na adição de uma sessão partilhada devem ser bloqueadas as sessões adjacentes.

No tratamento do evento externo que invoca estas operações a sessão *ContextAgentSession* insere um evento de reconfiguração por cada canal. Estes eventos são inseridos assincronamente nos canais (podendo ser intercalados por outros eventos) mas no entanto como as sessões são bloqueadas apenas os eventos de reconfiguração são processados.

Contudo, não se pode garantir que outro evento de reconfiguração seja introduzido no canal entre os vários eventos gerados pela *ContextAgentSession*. Efectivamente poderia dar-se o caso que um evento fosse introduzido e que fosse processado pela sessão, no entanto tal só seria possível com a invocação de dois eventos quase simultaneamente. A probabilidade de isso acontecer é baixíssima e implicaria que mais do que uma consola de reconfiguração fosse usada, pelo que considera-se um risco aceitável. Resumindo, as operações de reconfiguração de sessões partilhadas não oferecem atomicidade, mas uma quase garantia de atomicidade.

A necessidade de introduzir eventos em cada um dos canais é determinada pelo facto de nenhum objecto do Appia que executa o escalonamento dos eventos (instância Appia, instância *EventScheduler*) não tem conhecimento dos canais que existem.

5.2 Bloqueio de sessões

A possibilidade de uma aplicação utilizar várias pilhas de protocolos, poderá criar a necessidade de proceder à reconfiguração atómica desses vários canais. Por outro lado, poderá ser necessário proceder a uma reconfiguração entre vários processos ou nós que necessite de ser sincronizada. Outra possibilidade é a necessidade de realizar duas operações consecutivas de forma atómica. Por exemplo, adicionar uma sessão e remover uma outra. Estas tarefas não podem ser realizadas com o mecanismo simples de reconfiguração, pois requerem que de alguma forma o fluxo de eventos seja interrompido.

Ao longo deste documento já foi usada a designação de operação atómica. A atomicidade é garantida sempre que uma dada operação é executada, dando garantia ao operador que não é processado qualquer outro evento enquanto todas as tarefas que constituem essa operação são executadas. As operações de reconfiguração da composição de um canal dão essa garantia no âmbito de um canal. Mais, as operações

definidas no capítulo 3 de monitorização e reconfiguração de atributos de sessões também.

A impossibilidade de estender o âmbito de uma operação a mais do que um canal é decorrente da inserção de eventos internos nesses canais. Uma vez que são inseridos assincronamente, não é garantido que esses sejam guardados na lista *Waiting* consecutivamente. Por esse motivo, poderá sempre ser inserido um outro evento pelo meio que altere o estado de algum dos protocolos.

A solução passa por promover os vários canais envolvidos a um estado quiescente. Neste estado, os eventos introduzidos no sistema seriam aceites na mesma mas seria bloqueada a sua circulação no canal ou em algumas das sessões que o constituem. Durante este período poderiam ser realizadas operações de reconfiguração sobre os canais sem que houvesse alteração de estado das sessões ou processamento de eventos, se a sua configuração não estivesse estável. Objectivamente, pretende-se que com o desbloqueio dos canais ou sessões, o processamento dos eventos seja retomado com a nova configuração mas que a ordem desses eventos seja mantida como se esses canais estivessem em operação.

5.2.1 Conceito

O bloqueio de sessões de um canal consiste em impor ao escalonamento de eventos restrições na entrega de eventos a algumas sessões do canal. Essas sessões, designadas de sessões quiescentes são impedidas de processar todos os eventos à excepção dos eventos de reconfiguração. Para tal cada sessão mantêm uma variável de estado de bloqueio que é verificada sempre que o escalonador de eventos tenta entregar um evento a essa sessão.

Este condicionamento no processamento determina que quando um evento deve ser entregue a uma sessão bloqueada o escalonador de eventos entra num modo em que apenas os eventos de reconfiguração são consumidos. Este mecanismo de preempção permite que mesmo com o canal bloqueado os eventos de reconfiguração sejam processados permitindo a reconfiguração e desbloqueio do canal. Note-se que enquanto o canal não está bloqueado os eventos de reconfiguração são tratados como os outros, ou seja, são ordenados nas mesmas listas que os outros eventos. Deste modo o processamento dos eventos não é afectado enquanto não houver necessidade de entregar um evento a uma sessão bloqueada.

Sendo os eventos de reconfiguração guardados nas listas, isso implica que quando o canal bloqueia esses eventos tenham de ser extraídos da lista, independentemente da sua posição. A necessidade deste mecanismo de preempção é decorrente da forma sequencial como os eventos são consumidos das listas.

Durante esse bloqueio, uma vez que a entrega de eventos às sessões do canal é gerida por um escalonador de eventos, todos os canais geridos por esse escalonador serão bloqueados.

5.2.2 Circulação de eventos em canais com sessões bloqueadas

A possibilidade de tornar quiescente apenas algumas das sessões do canal tem vantagens na possibilidade de continuar a processar os eventos que não sejam entregues a essas sessões. Desta forma é possível continuar a processar alguns eventos e garantir que a reconfiguração das sessões bloqueadas é feita de forma atómica. Note-se que esta possibilidade não mete em causa a ordenação dos eventos uma vez que as dependências criadas são decorrentes do processamento do evento por parte de uma sessão. Desta forma, os eventos se não visitam as sessões bloqueadas não criam dependências ao serem tratados por estas ao gerar novos eventos.

Foram identificados três cenários distintos onde mesmo com algumas das sessões quiescentes poderão circular eventos no canal. Considere-se a pilha de protocolos da figura 12.

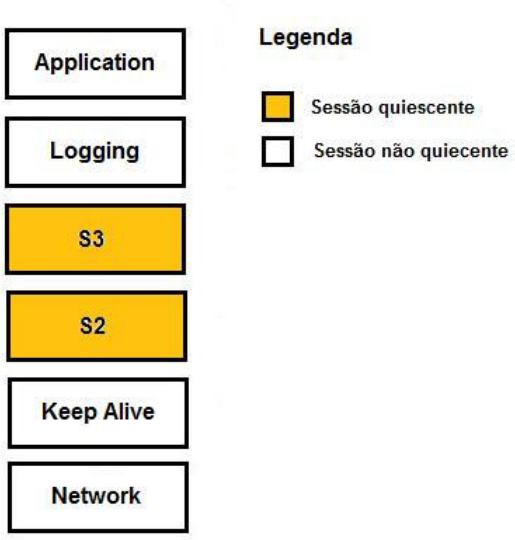


Figura 12 : Pilha com sessões quiescentes

Note-se que sendo o Appia uma plataforma totalmente orientada aos eventos, as alterações de estado dos protocolos apenas decorrem do processamento de eventos, sejam eles inseridos por sessões do canal ou introduzidos no canal por outra thread.

Considere os seguintes tipos de eventos com as rotas definidas do seguinte modo.

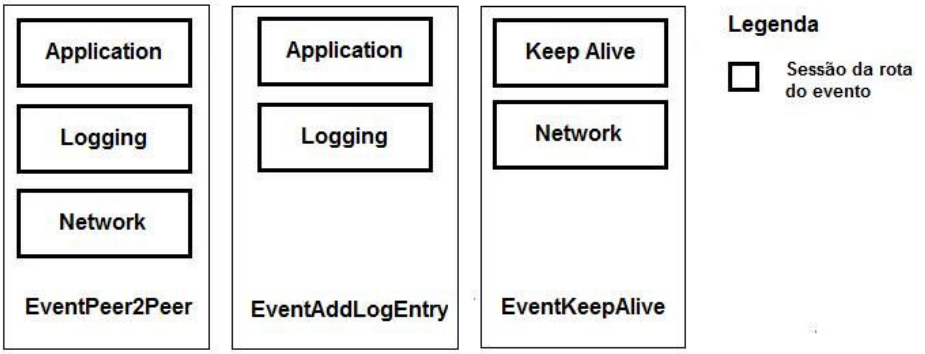


Figura 13 : Vector de sessões dos eventos

No caso do *EventPeer2Peer* suponha-se que os protocolos S2 e S3 implementam uma lógica de comunicação em grupo. O evento mesmo com as sessões quiescentes poderá circular no canal visitando sessões em posições acima e abaixo dessas, utilizando um serviço de entrega ponto a ponto.

No caso do *EventAddLogEntry*, a aplicação poderá usar um serviço de registo de operação que esteja entre a sua aplicação e as sessões quiescentes. Deste modo a aplicação não fica inibida de utilizar um serviço de alto nível mesmo que o serviço disponibilizado pelo canal esteja limitado.

No caso evento do *KeepAlive*, teríamos vantagens se a ligação estabelecida pela camada de rede fosse enriquecida com um serviço que permitisse um controlo do estado da ligação. Este controlo poderia ser efectuado mesmo com o serviço de comunicação em grupo desactivado, permitindo manter a ligação de rede activa.

Note-se que ao tornar uma sessão quiescente, não se impede que essa sessão insira eventos de forma assíncrona. Ou seja, se eventualmente a sessão da camada de rede da figura 12 estivesse bloqueada, mesmo assim a thread de leitura do *socket* poderia inserir os eventos que seriam colocados na lista *Waiting*. Este raciocínio é aplicável à camada de aplicação, que utilizará uma thread para leitura do input do utilizador. Esta funcionalidade é possível porque a verificação do estado quiescente de uma sessão que leve ao bloqueio de um evento é realizado quando o escalonador selecciona o evento e o tenta entregar a uma sessão quiescente.

5.2.3 Concretização

De forma a suportar o bloqueio das sessões, o núcleo teve de ser alterado passando a operar em dois modos, um no qual todos os eventos são processados, e outro, em que apenas os eventos de reconfiguração são processados.

Este modo preemptivo de tratamento destes eventos é necessário por estes eventos estarem inseridos nas listas junto aos outros eventos. A abordagem de não criar uma lista em separado para estes eventos é justificada pela opção de manter a ordenação em conjunto com os demais eventos. O tratamento dos eventos de reconfiguração apenas é preemptivo quando o canal está bloqueado, sendo que quando este é desbloqueado, os restantes eventos de reconfiguração passam a ser tratados pela ordem normal do canal.

Iteração da instância Appia

O mecanismo de bloqueio de sessões que promove um determinado canal a um estado quiescente, implicou a reformulação do método *instanceRun* da classe Appia. Conforme descrito no capítulo 3 este iterava sobre todos os *EventScheduler* enquanto houvesse eventos a tratar. O número de eventos era controlado pela própria instância, numa variável que controlava o número total de eventos em todos os escalonadores.

Com a possibilidade de um bloqueio de um canal o número de eventos deixa de ser suficiente para determinar se um escalonador pretende ou não processar eventos. Deste modo a condição de bloqueia da instância, passa a ser baseada na existência de escalonadores que tenham eventos e que não estejam bloqueados. Essa informação é intrínseca a cada escalonador, pelo que a instância Appia invoca o método *isAbleToConsumeEvent* de cada um dos escalonadores, que permite apurar se esse está bloqueado.

Escalonamento de eventos

Para suportar o bloqueio de sessões, o escalonamento dos eventos passou a consumir os eventos em dois modos: modo normal e um modo preemptivo.

No modo normal os eventos são consumidos das listas sequencialmente, como fazia até então, procurando nas listas (pela ordem *Main*, *Reverse*, *Waiting*) e consumindo o primeiro evento encontrado.

No modo preemptivo, o escalonador processa o primeiro evento de reconfiguração que encontra nas listas procurando pela mesma ordem. Para guardar o estado de bloqueio do escalonador foi definida a *isBlockedByEvent* que indica se esse está bloqueado, sendo actualizada sempre que se tenta consumir um evento.

Se um evento não for consumido, porque bloqueia o canal e existirem um evento de reconfiguração das listas, esse é retirado da lista e processado. Esse evento pode estar em qualquer posição, pelo que é necessário criar uma ligação entre o evento antes e o evento depois na lista. Este mecanismo de preempção dos eventos de reconfiguração permite que a reconfiguração aconteça mesmo com o canal bloqueado.

A inserção dos eventos continua a ser feita mesma forma, sendo que em modo bloqueado os eventos que circulem nas sessões que não estejam quiescentes, podem inserir eventos sincronamente. A inserção de eventos assíncronos pode ser feita do mesmo modo, e até as sessões que estejam bloqueadas os podem inserir, por exemplo, ao receber dados por um *socket* gerido por outra *thread*.

Ao inserir ou consumir um evento, o escalonador contabiliza o número de eventos que estão nas listas distinguindo os eventos de reconfiguração dos restantes. Para tal define as variáveis *nrOfPendingEvents* e *nrOfPendingSpecialEvents*.

A possibilidade de bloqueio do escalonador, determina que esse possa ter eventos mas que não os deseje processar. Deste modo disponibiliza à instância do Appia um método que diz se este está interessado em processar algum dos eventos nas listas. Esse método, *isAbleToConsumeEvent* devolve um estado que é calculado usando as três variáveis criadas:

(NOT isBlockedByEvent AND nrOfPendingEvents>0) OR

(isBlockedByEvent AND nrOfPendingSpecialEvents>0)

Ou seja, um determinado EventScheduler deseja processar eventos se não estiver bloqueado e existirem eventos a processar ou se estiver bloqueado e existirem eventos de reconfiguração a processar.

Pode-se pensar que o mecanismo como está implementado, desvirtua o conceito de bloqueio do canal uma vez que, uma instância de EventScheduler pode gerir eventos de vários canais, o seu bloqueio implica o bloqueio de todos os canais afectos a essa instância. Esta abordagem justifica-se pelo facto do Appia, como é sugerido em [M01], associar por omissão um escalonador de eventos diferente a cada canal. Considera-se que o bloqueio de todos os canais um custo aceitável, na medida em que a partilha do mesmo só fará sentido quando existam sessões partilhadas, por exemplo, caso a aplicação usa mais do que um canal.

5.2.4 Reconfiguração de sessões quiescentes

O bloqueio de apenas algumas sessões de um canal implica um cuidado extra na reconfiguração de sessões. Ao permitir que os eventos circulem até que fiquem bloqueados, possibilita-se que quando a operação de reconfiguração do canal seja executada existam eventos que já tenham sido processados por algumas das sessões, ou seja, que estejam inseridos nas listas *Main* e *Reverse*. Esta situação, é motivada pelo próprio bloqueio, pois sem bloqueio como foi descrito na secção 5.1 não existe este problema porque todos os eventos estão pendentes (na lista *Waiting*).

Em termos práticos, esta questão traduz-se por no momento em que o canal é reconfigurado podem existir eventos nas três listas. Esta situação motiva três regras de ouro associada ao bloqueio:

1) se houver sessões bloqueadas num canal, as operações de reconfiguração devem incidir apenas sobre as sessões bloqueadas. Excepção feita à inserção de uma sessão que pode ser feita nas posições adjacentes ao bloco de sessões bloqueadas.
2) se forem bloqueadas várias sessões de um canal essas sessões devem ser sessões que ocupam posições consecutivas na pilha.
3) se houver sessões bloqueadas num canal, não podem ser bloqueadas mais sessões.

Estas regras garantem a ordem causal no canal e a coerência da informação que os eventos guardam sobre o caminho percorrido.

Por exemplo a regra 1) evita que haja eventos órfãos. Este problema consiste na possibilidade de haver um evento cuja sessão de origem seja removida, impossibilitando o cálculo dos seus índices de posicionamento. Considere-se o seguinte cenário:

Se um evento E1 for recebido pela camada de rede S0 e for processado por S0 e S1, gerando nesta última um evento no sentido inverso E2. Após o tratamento de E1 pela camada S1 o escalonador de eventos vai tentar entregar o evento à sessão S2, no entanto como essa sessão está bloqueada o canal é bloqueado. Neste momento o evento E2 ainda não foi entregue a nenhuma sessão, pelo que o índice da primeira sessão foi calculado mas o de sessão corrente não.

Se a regra 1) definida acima for quebrada e por exemplo seja removida a sessão S1, então o evento E2 não poderá ser actualizado pois a primeira posição não pode voltar a ser recalculada, e logo, a próxima sessão a que deve ser entregue não pode ser determinada. Note-se que o evento E1 também teria que ser actualizado mas neste caso a sessão corrente poderia ser recalculada em função do valor que apresenta.

Resumindo, a circulação de eventos nos canais com sessões bloqueadas implica que as sessões reconfiguradas sejam sessões bloqueadas, garantindo que as alterações são sempre nas sessões que o evento ainda vai percorrer.

Por outro lado, a regra 2) garante que as alterações ao canal incidam apenas sobre sessões que ele ainda não visitou. Se essa regra não for aplicada pode dar-se o caso de serem inseridas ou removidas sessões em posições da pilha por onde o evento já tinha passado. Neste caso seria assumido que o evento já teria sido processado por essa sessão quando o não tinha feito, ou, que teria sido processado por uma sessão que já não faz parte do canal. Considere-se o seguinte cenário:

Existe um canal com cinco sessões S0 a S5 e um evento E1 que visita as sessões S0, S2, S3 e S4. Se a sessão S2 estiver desbloqueada e as sessões S1 e S3 bloqueadas, o evento E1 ao percorrer as sessões no sentido S0 para S5 fica bloqueado na sessão S3. Aplicando a regra 1) que diz que as operações recaem sobre sessões bloqueadas a sessão S1 pode ser substituída por outra S1' que trate o evento. Após a reconfiguração o evento teria no seu caminho uma sessão que o trata, numa posição em que já passou mas que não o tinha tratado.

Deste modo garante-se que ao longo do tempo que o evento circula no canal, este pode ser constituído por diferentes vectores de sessões, no entanto a reconfiguração só incide sobre que o evento ainda não tenha sido entregue, ou em posições por onde não tenha passado.

A regra 3) actua como uma segurança para a regra 2) para que não haja operações que incidam sobre uma sessão que já tenha processado algum dos eventos em circulação. Considere-se o seguinte cenário:

O evento E1 (que visita todas as sessões do canal) percorre um canal com as sessões S0 a S5 nessa mesma direcção. Se a sessão S3 estiver bloqueada, o evento vai bloquear depois de ter sido processado por S0, S1 e S2. Se bloquear S1 por exemplo, e em seguida removermos essa sessão S1, a regra 1) é respeitada mas a regra 2) é quebrada levando a incoerência no caminho percorrido pelo evento.

Em resumo, as regras forçam a que as alterações ao vector de sessões do canal apenas incidem sobre sessões que o evento ainda tenha de visitar.

5.2.5 Actualização do índice de sessão corrente

Esta secção complementa o que foi escrito na secção 5.1.3 que descreve o processo de actualização das listas de eventos quando o canal não está bloqueado. Na verdade, a implementação é a mesma, e nesta secção destaca-se as especificidades de actualizar os eventos que podem existir com o bloqueio do canal nas listas *Main* e *Reverse*. Sendo assim, a actualização é sempre feita às três listas pela ordem *Main*, *Reverse* e *Waiting*.

Os eventos que estejam na lista *Main* ou na *Reverse* são eventos inseridos por uma sessão ao tratar um evento. Deste modo a sessão de origem do evento é sempre definida, e o cursor de sessão corrente também.

A sessão de origem não é alterada, pois numa situação de bloqueio, o evento só pode ter sido inserido por uma sessão não bloqueada. As regras referidas de reconfiguração de canais com sessões bloqueadas dão-nos essa garantia, e foram definidas com vista a actualização dos eventos bloqueados já entraram no canal (nas listas *Main* e *Reverse*). Esta garantia torna também possível redefinir sempre o índice da

primeira sessão. Este índice é recalculado por invocação de mesmo método do canal que é utilizado na inicialização do evento.

O mesmo já não acontece com o índice de posição corrente que é actualizado manualmente. Uma vez que a reconfiguração apenas altera uma sessão do canal de cada vez, o índice será incrementado ou decrementado levando em conta o índice de sessão corrente e o índice da sessão alterada. Note-se que os índices referidos são relativos ao vector de sessões do evento e não do vector de sessões do canal.

Na adição de uma camada, o índice de sessão corrente será incrementado quando o índice da posição da sessão inserida for menor ou igual ao índice de sessão corrente do evento. A condição inclui o caso de os dois índices serem iguais por ser permitido que uma sessão seja adicionada nas posições adjacentes ao bloco de sessões bloqueadas.

Na remoção de uma sessão, o índice é actualizado sempre que o índice da posição da sessão removida seja menor que o índice da sessão corrente.

A substituição de uma sessão é implementada como uma operação composta pela remoção de uma sessão e adição da outra. Sendo assim os índices são recalculados em cada uma das operações. Esta abordagem simplifica a actualização dos eventos pois o impacto no vector de sessões do evento é determinado pelo facto de a sessão fazer parte desse vector. Uma vez que na substituição existem duas sessões envolvidas, o impacto pode ser motivado pela remoção da antiga, pela adição da nova ou por ambas.

As condições referidas que determinam a actualização do índice de sessão corrente dos eventos, definem genericamente que esse índice é actualizado quando os eventos estão bloqueados acima de um bloco de sessões. A reconfiguração altera pois o número de sessões que estão nas posições inferiores à posição do evento. As reconfigurações feitas nas sessões das posições superiores não têm impacto pois não afectam a posição da sessão corrente dos eventos que estão abaixo.

5.2.6 Adaptação de protocolos

Como a reconfiguração é realizada pelo núcleo, as sessões não têm que ser alteradas para suportar a reconfiguração do canal. A alteração é opcional e consiste na implementação dos métodos de pré tratamento das operações. Estes métodos são definidos no interface ReconfigHandlingSession que deve ser implementado pela sessão caso esta queira proceder a algum tipo de operação antes que o canal proceda à alteração do vector de sessões.

5.3 Resumo

A plataforma Appia mesmo não disponibilizando um mecanismo de reconfiguração dos canais, pode ser alterada de modo a suportar a reconfiguração dinâmica das pilhas de protocolos.

Esta tarefa envolve a redefinição de informação de suporte ao escalonamento dos eventos e de representação do canal.

Deste modo foi preciso proceder à actualização do vector de sessões que define o canal, que passa a ser definido estaticamente por uma nova QoS.

Os eventos ao guardarem informação sobre o caminho percorrido, tiveram também que ser alteradas. Estes podem estar pendentes de entrar no canal ou em circulação (que já foi processado por alguma sessão) pelo que a actualização teve de levar em conta as especificidades de cada situação.

O desenvolvimento de um mecanismo de bloqueio foi primordial na garantia de atomicidade em conjuntos de operações. Com este mecanismo conseguiu-se dar a mesma garantia a sequências de operações de reconfiguração que por exemplo podem afectar sessões em mais do que um canal.

Com este mecanismo houve a necessidade de alterar o procedimento de escalonamento de eventos, criando um novo modo de operação do escalonador, que apenas processa eventos de reconfiguração. Neste modo o escalonador de eventos passa a fazer um processamento preemptivo, ou seja, os eventos de reconfiguração são sempre processados alterando a ordem definida com a inserção no canal.

Sumariamente, a implementação permite a actualização do canal, fornecendo os mecanismos necessários para a actualização de toda a informação gerada na criação do canal, e que suporta a circulação dos eventos. As alterações necessárias a essa informação foram em grande parte repetição daquilo que é feito com o canal. No entanto a variação dos índices do vector de sessões quer do canal quer de cada evento levou a que esses tivessem que ser actualizados manualmente de acordo com a operação de reconfiguração em causa.

Capítulo 6

Avaliação

Neste capítulo será feita a verificação das funcionalidades implementadas e avaliado o seu impacto na performance do Appia.

Sendo o mecanismo de reconfiguração de sessões e de composições baseado em eventos foi desenvolvida uma consola do tipo linha de comandos que permite a invocação de cada uma das operações. A secção 6.1 descreve os comandos e os seus parâmetros que são usados na criação dos eventos enviados para a composição Appia cuja estrutura foi descrita nos capítulos 4 e 5.

Na verificação das funcionalidades será usada como base de testes a aplicação Ecco (*Ecco*) disponibilizada pelo Appia. Esta permite a troca de mensagens de texto entre dois terminais utilizando um canal de comunicação que suporta o envio e recepção pela rede dos eventos trocados entre as aplicações (*MyEccoEvent*). A composição de cada uma dessas aplicações é formada por uma camada de aplicação (*EccoLayer*) e uma camada de acesso à rede (*TcpCompleteLayer*).

À estrutura base dessas composições é adicionado um conjunto de sessões dummy, que se destinam apenas a preencher mais algumas camadas da composição, permitindo mais facilmente ver o resultado das várias operações. O resultado de algumas operações de reconfiguração de composições está relacionado com os tipos de eventos processados pelas sessões inseridas ou removidas. Deste modo, as sessões dummy são concretizadas por duas camadas distintas, *CtxDummyLayer* e *CtxDummyNotHandleLayer*, que respectivamente processam ou não o evento *MyEccoEvent*. As sessões que concretizam as camadas dummy ao tratarem os eventos imprimem uma mensagem no output que permitirá confirmar as alterações no caminho dos eventos.

A avaliação de desempenho tem em vista a medição do impacto da solução apresentada, sendo comparada com os resultados da versão original do Appia.

6.1 Consola de monitorização e de reconfiguração

As consolas desenvolvidas estabelecem a interface entre o utilizador do sistema e as aplicações monitorizadas e/ou reconfiguradas. São concretizadas por duas composições Appia, designadas de *ContextMonitor* (invocação de comandos de monitorização) e o *ReconfigurationManager* (invocação de comandos de reconfiguração). Existe a possibilidade de juntar ambas as funcionalidades criando uma terceira variante da composição onde são adicionadas ambas as sessões responsáveis pelo envio dos comandos. A ordem pela qual estas são injectadas não é relevante pois a consola apenas envia um comando de cada vez, sendo a compactação e descompactação dos atributos feita apenas por uma das sessões injectadas na consola ou nas pilhas monitorizadas.

As classes de interface: *ContextMonitorShell*, *ReconfigurationManagerShell* concretizam interfaces do tipo linha de comando, responsáveis por traduzir os comandos do utilizador em eventos enviados pelos canais de controlo *RemoteInvocationChannel* ou *ReconfigurationChannel* para a aplicação gerida.

6.1.1 Comandos de Monitorização

Leitura pontual

O comando de leitura pontual envia pelo canal *RemoteInvocationChannel* um evento do tipo *CtxQueryEvent* que especifica uma lista dos atributos a ler da sessão. Em resposta recebe um evento do tipo *CtxAnswerEvent* que retorna uma lista desses atributos e dos seus valores.

- query –r request_id -h sensor_host -p sensor_port -c channel -s session -a attribute_1_1 -a attribute_1 -a attribute_2 … -a attribute_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente o pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	1	Nome que identifica a sessão a que pertencem os atributos.

-a	attribute_name	n	Nomes dos atributos públicos da sessão.

Tabela 2 : Parâmetros do comando de leitura pontual de atributos

Leitura cíclica

A leitura cíclica é iniciada com o comando monitor que envia pelo canal *RemoteInvocationChannel* um evento do tipo *CtxStartMonitorEvent* com uma lista dos atributos a ler da sessão. Ciclicamente é recebido pelo canal *ContextNotificationChannel* um evento do tipo *CtxNotifEvent* que retorna uma lista desses atributos e dos seus valores.

- monitor –r request_id -h sensor_host -p sensor_port -c channel -s session -a attribute_1 -a attribute_2 … -a attribute_n –t time_interval

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente o pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	1	Nome que identifica a sessão a que pertencem os atributos.
-a	attribute_name	n	Nomes dos atributos públicos da sessão.
-t	time_interval	1	Período de tempo em milésimos de segundo que intercala cada notificação devolvida.

Tabela 3 : Parâmetros do comando de leitura cíclica de atributos

O comando de paragem de leitura cíclica envia pelo canal *RemoteInvocationChannel* um evento do tipo *CtxStopMonitorEvent* que especifica o identificador atribuído no comando que iniciou este serviço.

- stop_monitor –r request_id -h sensor_host -p sensor_port

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente o pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.

Tabela 4 : Parâmetros do comando de paragem da leitura cíclica

Escrita de atributos

A reconfiguração de atributos permite especificar um conjunto de novos valores para as variáveis de estado das sessões que concretizam cada um dos protocolos, sendo enviado um evento do tipo *CtxUpdateEvent* para o *ReconfigurationAgent* onde são definidos um conjunto de binómios nome de variável e novo valor, e o nome que identifica a sessão a que estes pertencem.

- update –r request_id -h sensor_host -p sensor_port –c channel –s session –a attribute_name_1 –v new_attribute_value_1 … -a attribute_ name_n –v new_attribute_value_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	1	Nome que identifica a sessão a que pertencem os atributos.
-a	attribute_name	n	Nome do atributo público da sessão.
-v	new_attribute_value	n	Valor a atribuir ao atributo definido no parâmetro anterior. A concretização do interface de linha de comando suporta apenas valores do tipo texto ou numéricos, não sendo possível definir classes no entanto um interface mais rico permitirá definir objectos como novos valores para um atributo.

Tabela 5 : Parâmetros do comando de escrita de atributos

6.1.2 Comandos de Reconfiguração

Eliminação de uma sessão

O comando de eliminação de uma sessão de um canal envia um evento do tipo *CtxDeleteSessionEvent* que define o nome da sessão a eliminar.

- delete_session –r request_id -h sensor_host -p sensor_port –c channel -s session

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	1	Nome que identifica as sessões a eliminar.

Tabela 6 : Parâmetros do comando de eliminação de sessão

Inserção de uma sessão

O comando de inserção de uma sessão no canal envia um evento do tipo *CtxInsertSessionEvent* que define o nome, tipo (*Layer*) e posição da sessão a inserir.

- insert_session -r request_id -h sensor_host -p sensor_port –c channel_id -s new_session_1 -t new_session_1_layer -i position_1 … -s new_session_n -t new_session_n_layer -i position_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	new_session	1	Nome que irá identificar a sessão inserida.
-t	new_session_layer	1	Nome na forma de um Fully Qualified Name da classe da Layer que irá gerar a sessão inserida.

-i	position	1	Índice da posição na pilha que a nova sessão vai ocupar na pilha resultante de inserir todas as sessões do comando. A base da pilha é a sessão índice 0.

Tabela 7 : Parâmetros do comando de inserção de sessão

Substituição de uma sessão

O comando que substitui uma sessão do canal por outra envia um comando do tipo *CtxReplaceSessionEvent* que define o nome, tipo (*Layer*) e posição da sessão a inserir e o nome da sessão substituída. São também definidos um conjunto de nomes e atributos da sessão substituída mapeados num conjunto de identificadores da sessão que a substitui. Este mapeamento suporta passagem de contexto, ou seja, a inicialização da nova sessão com valores da sessão removida.

- replace_session –r request_id -h sensor_host -p sensor_port –c channel_id -s replacing_session -t replacing_session_layer -i position –o old_session_attribute_1 – p new_session_attr_1 ... –o old_session_attribute_n – p new_session_attr_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	replacing_session	1	Nome que irá identificar a sessão que substitui a existente.
-t	replacing_session_layer	1	Nome na forma de um Fully Qualified Name da classe da Layer que irá gerar a nova sessão.
-i	position	1	Índice da posição no vector de sessões do canal da sessão substituída. A base da pilha é a sessão índice 0.
-o	old_session_attr	n	Nome do atributo de contexto da classe substituída.
-p	new_session_attr	n	Nome do atributo da nova classe que é inicializado.

Tabela 8 : Parâmetros do comando de substituição de sessão

Eliminação de uma sessão partilhada

O comando de eliminação de uma sessão partilhada entre vários canais envia um evento do tipo *CtxDeleteSharedSessionEvent* que define o nome da sessão a eliminar.

- delete_shared_session –r request_id -h sensor_host -p sensor_port –c channel_1 –c channel_2 … –c channel_n -s session

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	n	Nomes dos canais Appia que partilham a sessão.
-s	session	1	Nome que identifica a sessão partilhada a eliminar.

Tabela 9 : Parâmetros do comando de eliminação de sessão partilhada

Inserção de uma sessão partilhada

O comando de inserção de uma sessão partilhada nos canais envia um evento do tipo *CtxInsertSharedSessionEvent* que define o nome, tipo (*Layer*) e posição da sessão a inserir.

- insert_shared_session -r request_id -h sensor_host -p sensor_port –c channel_1 –c channel_2 … –c channel_n -s new_session_1 -t new_session_1_layer -i position_1 … -s new_session_n -t new_session_n_layer -i position_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	n	Nomes dos canais Appia que partilham a sessão.
-s	new_session	1	Nome que irá identificar a sessão inserida.
-t	new_session_layer	1	Nome na forma de um Fully Qualified Name da classe da Layer que irá gerar a sessão inserida.

-i	position	n	Índice da posição nas pilhas que a nova sessão vai ocupar em cada canal.

Tabela 10 : Parâmetros do comando de inserção de sessão partilhada

Substituição de uma sessão partilhada

O comando que substitui uma sessão do canal por outra envia um comando do tipo *CtxReplaceSessionEvent* que define o nome, tipo (*Layer*) e posição da sessão a inserir e o nome da sessão substituída. São também definidos um conjunto de nomes e atributos da sessão substituída mapeados num conjunto de identificadores da sessão que a substitui. Este mapeamento suporta passagem de contexto, ou seja, a inicialização da nova sessão com valores da sessão removida.

- replace_shared_session –r request_id -h sensor_host -p sensor_port –c channel_id_1 –c channel_id_2 … –c channel_id_3 -s replacing_session -t replacing_session_layer -i position –o old_session_attribute_1 – p new_session_attr_1 ... –o old_session_attribute_n – p new_session_attr_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	n	Nomes dos canais Appia que partilham a sessão.
-s	replacing_session	1	Nome que irá identificar a sessão que substitui a existente.
-t	replacing_session_layer	1	Nome na forma de um Fully Qualified Name da classe da Layer que irá gerar a nova sessão.
-i	position	1	Índice da posição no vector de sessões do canal da sessão substituída. A base da pilha é a sessão índice 0.
-o	old_session_attr	n	Nome do atributo de contexto da classe substituida.
-p	new_session_attr	n	Nome do atributo da nova classe que é inicializado.

Tabela 11 : Parâmetros do comando de substituição de sessão partilhada

6.1.3 Comandos de bloqueio de sessões

Bloqueio de sessões

O comando de bloqueio de sessões envia um evento do tipo *CtxMakeSessionsQuicentsEvent* que especifica uma lista de nomes das sessões a bloquear.

- make_quiscents –r request_id -h sensor_host -p sensor_port –c channel_id -s session_1 session_2 … session_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	n	Nomes das sessões a bloquear.

Tabela 12 : Parâmetros do comando de bloqueio de sessões

Desbloqueio de sessões

O comando de bloqueio de sessões envia um evento do tipo *CtxMakeSessionsQuicentsEvent* que especifica uma lista de nomes das sessões a bloquear.

- resume_sessions –r request_id -h sensor_host -p sensor_port –c channel_id -s session_1 session_2 … session_n

Flag	Parâmetro	#	Descrição
-r	request_id	1	Nome que identifica univocamente este pedido.
-h	sensor_host	1	Nome da maquina onde é executado o ContextSensor.
-p	sensor_port	1	Porto da maquina onde é executado o ContextSensor.
-c	channel_id	1	Nome do canal Appia da composição reconfigurada.
-s	session	n	Nomes das sessões a desbloquear.

Tabela 13 : Parâmetros do comando de desbloqueio de sessões

6.2 Validação funcional

A validação das funcionalidades foi realizada com operações sobre a aplicação Ecco1 (*net.sf.appia.demo.xml.Ecco*) que implementa um terminal de troca de mensagens com a aplicação Ecco2. Os cenários mostram a invocação das operações sobre a aplicação Ecco1, sendo enviados ou recebidos eventos que permitem validar o comportamento da aplicação. As figuras apresentadas foram recolhidas do output da consola de monitorização e da própria aplicação Ecco1. Cada operação pode ser rastreada pelo identificador atribuído na consola e mostrado na aplicação.

Os cenários apresentados exemplificam todas as operações, no entanto não cobrem as imensas possibilidades de reconfiguração de uma pilha protocolar. Ao ser usada uma pilha baseada em camadas dummy, não foram validados cenários com protocolos reais, contudo, a implementação do mecanismo é genérica e qualquer protocolo, mesmo que algumas adaptações pode ser reconfigurado.

6.2.1 Reconfiguração de sessões

A validação das operações de leitura e escrita de atributos é feita com a leitura e escrita dum atributo da camada de rede que contabiliza o número de mensagens enviadas.

A funcionalidade é validada analisando o resultado obtido na invocação de cada operação. Por exemplo na leitura do atributo é mostram na consola os valores lidos. Na escrita do valor é a reconfiguração do atributo é verificada no output da própria aplicação.

Leitura cíclica

Neste cenário a leitura cíclica é activada e são geradas três notificações de acordo com o número de mensagens que vão sendo enviadas. A figura 14 permite ver a invocação da operação de notificação, as notificações recebidas e a paragem do serviço a partir da consola de monitorização. Por mim é recebida a leitura pontual do atributo.

Neste cenário são validadas as seguintes funcionalidades:

- ✓ inicio e paragem da leitura cíclica.
- ✓ geração de notificações.

As operações invocadas na consola de monitorização são mostradas na figura 14.

```
monitor -h localhost -p 40001 -c ecco_c -s tcp_s_managed -a nrOfDeliveredEvents -t 20000
MONITOR   192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   localHost/127.0.0.1:50000   20000
NOTIFIC   192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   0
NOTIFIC   192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   2
NOTIFIC   192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   5
stop -h localhost -p 40001 -r 192.168.1.65@40000@14:38:37
STOP MONITOR   192.168.1.65@40000@14:38:37
query -h localhost -p 40001 -c ecco_c -s tcp_s_managed -a nrOfDeliveredEvents
QUERY     192.168.1.65@40000@14:41:06   ecco_c   tcp_s_managed   nrOfDeliveredEvents
```

Figura 14 : Invocação de leitura cíclica

No output do terminal (fig. 15) pode ver-se o envio dos eventos de leitura ciclica que devolve os valor de acordo com as mensagens foram sendo enviadas.

```
SEND NOTIF    192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   0
msg 1
> msg 2
> SEND NOTIF    192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   2
msg 3
> msg 4
> msg 5
> SEND NOTIF    192.168.1.65@40000@14:38:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   5
SEND ANSWER   192.168.1.65@40000@14:41:06   ecco_c   tcp_s_managed   nrOfDeliveredEvents   5
```

Figura 15 : Envio de notificações da aplicação

Leitura pontual e escrita

As operações de leitura pontual e escrita do atributo são validadas em conjunto. Neste caso o valor é lido, redefinido e depois lido outra vez.

Neste cenário são validadas as seguintes funcionalidades:

- ✓ leitura pontual do atributo.
- ✓ escrita do atributo.

As operações invocadas na consola de monitorização são mostradas na figura 16.

```
query -h localhost -p 40001 -c ecco_c -s tcp_s_managed -a nrOfDeliveredEvents
QUERY     192.168.1.65@40000@14:41:06   ecco_c   tcp_s_managed   nrOfDeliveredEvents
ANSWER    192.168.1.65@40000@14:41:06   ecco_c   tcp_s_managed   nrOfDeliveredEvents   5
update -h localhost -p 40001 -c ecco_c -s tcp_s_managed -a nrOfDeliveredEvents -v 0
query -h localhost -p 40001 -c ecco_c -s tcp_s_managed -a nrOfDeliveredEvents
QUERY     192.168.1.65@40000@14:41:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents
ANSWER    192.168.1.65@40000@14:41:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   0
```

Figura 16 : Recepção de notificações na consola

Na figura 17 é possível ver (na aplicação) as leituras do valor do atributo intervaladas pela escrita desse valor no terminal de envio de mensagens.

```
SEND ANSWER   192.168.1.65@40000@14:41:06   ecco_c   tcp_s_managed   nrOfDeliveredEvents   5
UPDAT VALUE   192.168.1.65@40000@14:41:15   ecco_c   tcp_s_managed   nrOfDeliveredEvents   0
SEND ANSWER   192.168.1.65@40000@14:41:37   ecco_c   tcp_s_managed   nrOfDeliveredEvents   0
```

Figura 17 : Leitura e escrita de atributos da aplicação

6.2.2 Reconfiguração de composições

As operações de reconfiguração são validadas procedendo à alteração do vector de sessões do canal e procedendo ao envio de eventos pelo canal da aplicação Ecco. O bloqueio de sessões implica a actualização de eventos já inseridos no canal, sendo que alguns desses eventos são enviados da aplicação Ecco1 e outros recebidos da aplicação Ecco2 (o texto é impresso a null).

As operações invocadas na consola de monitorização são identificadas com o RequestId que é mapeado às alterações do lado da aplicação.

Reconfiguração de um canal sem sessões bloqueadas

Este cenário verifica a realização de operações sobre um canal sem sessões bloqueadas. É verificada a reconfiguração do vector de sessões do canal. A composição do vector de sessões do canal e o caminho percorrido por novos eventos confirmam reconfiguração.

Neste cenário são validadas as seguintes funcionalidades:

- ✓ reconfiguração do canal sem preempção dos eventos.
- ✓ actualização do vector de sessões do canal e do caminho percorrido pelos eventos inseridos no canal.

As operações invocadas na consola de monitorização são mostradas na figura 18

```
create_context_id -n ctx_1 -h localhost -p 40001 -c ecco_c
replace_sessions -n ctx_1 -i 2 -s new_dummy_replacing_2_s -t net.sf.appia.demo.context.
REPLACE 192.168.1.65@44000@01:37:33
delete_sessions -n ctx_1 -s dummy_1_s
DELETE 192.168.1.65@44000@02:02:14
insert_sessions -n ctx_1 -i 1 -s new_dummy_inserted_1_s -t net.sf.appia.demo.context.re
INSERT 192.168.1.65@44000@02:03:15
```

Figura 18 : Reconfiguração de canal na consola

No output da consola da aplicação (fig. 19) é possível ver o caminho percorrido pelo evento "msg 1" que foi enviado antes da reconfiguração. Este visita as três sessões da pilha original ("dummy_1_s", "dummy_2_s", "dummy_3_s").

```
msg 1
> HANDLE EVENT MyEccoEvent msg 1 [1428ea|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 1 [16546ef|CtxDummySession|dummy_2_s]
HANDLE EVENT MyEccoEvent msg 1 [1326484|CtxDummySession|dummy_1_s]
```

Figura 19 : Rota do evento não bloqueado

A primeira operação realizada substitui no canal a sessão "dummy_2_s" por outra do mesmo tipo designada de "new_dummy_replacing_2_s". O vector de sessões é então alterado sendo feito primeiro a remoção e depois a inserção da nova sessão. Na figura

20 é possível ver os impactos da operação de reconfiguração e as sessões que formam o canal antes e depois. Note-se a invocação do método de tratamento prévio da sessão antes de ser substituída (PRE HANDLE REP).

```
RECONFIGURATION 192.168.1.65@44000@01:37:33
SESSIONS PRE REP ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[16546ef|CtxDummySession|dummy_2_s]
[1326484|CtxDummySession|dummy_1_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]

PRE HANDLE REP [16546ef|CtxDummySession|dummy_2_s]
CHANNEL ecco_c UPDATED ON SESSION DELETED
CHANNEL ecco_c UPDATED ON SESSION INSERT
SESSIONS POS INS ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[542529|CtxDummySession|new_dummy_replacing_2_s]
[1326484|CtxDummySession|dummy_1_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]
```

Figura 20 : Substituição de sessão na aplicação

Após a alteração é enviado um evento "msg 2" que assume a nova configuração. Esse caminho é obtido do canal que com a reconfiguração acima recalculou as rotas dos vários tipos de evento. A fig 21 mostra o tratamento do evento pela sessão "new_dummy_replacing_2_s"

```
msg 2
> HANDLE EVENT MyEccoEvent msg 2 [1428ea|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 2 [542529|CtxDummySession|new_dummy_replacing_2_s]
HANDLE EVENT MyEccoEvent msg 2 [1326484|CtxDummySession|dummy_1_s]
```

Figura 21 : Rota do evento com sessão substituída

Após o teste de substituição de uma camada, foi realizada a operação de remoção (fig. 22) de uma camada. Neste caso é retirada da pilha a sessão "dummy_1_s". Note-se que na remoção também é dada a possibilidade da sessão "dummy_1_s" libertar algum tipo de recursos na sua rotina de tratamento prévio da operação.

```
RECONFIGURATION 192.168.1.65@44000@02:02:14
SESSIONS PRE DEL ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[542529|CtxDummySession|new_dummy_replacing_2_s]
[1326484|CtxDummySession|dummy_1_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]

PRE HANDLE DEL [1326484|CtxDummySession|dummy_1_s]
CHANNEL ecco_c UPDATED ON SESSION DELETED
SESSIONS POS DEL ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[542529|CtxDummySession|new_dummy_replacing_2_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]
```

Figura 22 : Remoção de sessão na aplicação

Após esta operação é enviado um evento (fig. 23) que reflecte a nova configuração da pilha.

```
msg 3
> HANDLE EVENT MyEccoEvent msg 3 [1428ea|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 3 [542529|CtxDummySession|new_dummy_replacing_2_s]
```

Figura 23 : Rota do evento com sessão eliminada

Finalmente, a operação de adição (fig. 24) de uma sessão que é inserida na posição 1 da pilha. Como se pode ver neste caso as sessões sobem uma posição na pilha. Repare-se que as sessões que se mantêm numa operação, são na verdade o mesmo objecto (número representa a referência). Note-se que na adição, como faz sentido, não existe método prévio de tratamento pela sessão.

```
RECONFIGURATION 192.168.1.65@44000@02:03:15
SESSIONS PRE INS ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[542529|CtxDummySession|new_dummy_replacing_2_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]

CHANNEL ecco_c UPDATED ON SESSION INSERT
SESSIONS POS INS ecco_c
[1f82982|EccoSession|ecco_s]
[18a49e0|ReconfigAgentSession|reconfiguration_agent_s]
[1428ea|CtxDummySession|dummy_3_s]
[542529|CtxDummySession|new_dummy_replacing_2_s]
[36527f|CtxDummySession|new_dummy_inserted_1_s]
[71dc3d|TcpCompleteSession|tcp_s_managed]
```

Figura 24 : Inserção de sessão na aplicação

Por fim é enviado o evento “msg 4” (fig. 25) que permite verificar que a nova sessão já faz parte da sua rota, sendo processado

```
msg 4
> HANDLE EVENT MyEccoEvent msg 4 [1428ea|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 4 [542529|CtxDummySession|new_dummy_replacing_2_s]
HANDLE EVENT MyEccoEvent msg 4 [36527f|CtxDummySession|new_dummy_inserted_1_s]
```

Figura 25 : Rota do evento com sessão inserida

Com a sequência de operações acima foi mostrado que a alteração do vector dos canais é realizada correctamente. Com este resultado, foi mostrado também que a criação da QoS e o recalculo do caminho dos eventos que o canal guarda são actualizados. O processamento neste cenário representa o modo de operação não preemptivo, ou seja, não existindo bloqueio do canal, os eventos de reconfiguração foram tratados como eventos regulares.

Bloqueio de sessões de um canal

O cenário de bloqueio de sessões de um canal verifica se o processamento dum evento é interrompido, quando a próxima sessão a que é entregue está bloqueada. Verifica também se quando é feito um desbloqueio de apenas algumas sessões, se a próxima sessão a processar o evento é desbloqueada, então esse é processado até que fique bloqueado outra vez.

Neste cenário são validadas as seguintes funcionalidades:

- ✓ bloqueio e desbloqueio de sessões.
- ✓ teste da condições de bloqueio de um evento.

As operações invocadas na consola de monitorização são mostradas na figura 26.

```
create_context_id -n ctx_1 -h localhost -p 40001 -c ecco_c
make_quiscents -n ctx_1 -s dummy_1_s -s dummy_2_s -s dummy_3_s
MAKE QUIESCENT 192.168.1.65@44000@16:33:40
resume_sessions -n ctx_1 -s dummy_2_s
RESUME 192.168.1.65@44000@16:36:26
resume_sessions -n ctx_1 -s dummy_3_s
RESUME 192.168.1.65@44000@16:36:51
resume_sessions -n ctx_1 -s dummy_1_s
RESUME 192.168.1.65@44000@16:37:41
```

Figura 26 : Invocação dos comandos de bloqueio e desbloqueio de sessões

Não havendo nenhuma sessão bloqueada o evento “msg 1” é entregue a todas as sessões (fig. 27).

```
msg 1
> HANDLE EVENT MyEccoEvent msg 1 [9b42e6|CtxDummySession|dummy_5_s]
HANDLE EVENT MyEccoEvent msg 1 [3c9217|CtxDummySession|dummy_4_s]
HANDLE EVENT MyEccoEvent msg 1 [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 1 [e70e30|CtxDummySession|dummy_2_s]
HANDLE EVENT MyEccoEvent msg 1 [16d2633|CtxDummySession|dummy_1_s]
```

Figura 27 : Rota de evento em canal sem sessões bloqueadas

Ao bloquear as sessões “dummy_1_s”,“ dummy_2_s”,“ dummy_3_s”, o evento “msg 2” é processado até que tem de ser entregue à sessão “dummy_3_s” que está bloqueada (fig. 28).

```
RECONFIGURATION 192.168.1.65@44000@16:33:40
PRE HANDLE BLOCK [16d2633|CtxDummySession|dummy_1_s]
PRE HANDLE BLOCK [e70e30|CtxDummySession|dummy_2_s]
PRE HANDLE BLOCK [154864a|CtxDummySession|dummy_3_s]
msg 2
> HANDLE EVENT MyEccoEvent msg 2 [9b42e6|CtxDummySession|dummy_5_s]
HANDLE EVENT MyEccoEvent msg 2 [3c9217|CtxDummySession|dummy_4_s]
```

Figura 28 : Bloqueio de sessões na aplicação

É desbloqueada a sessão “dummy_2_s” mas como evento deve ser entregue à sessão “dummy_3_s” este continua bloqueado. Com o desbloqueio da sessão “dummy_3_s” (fig. 29) o evento é entregue às duas e bloqueia na sessão “dummy1_s”.

```
> RECONFIGURATION 192.168.1.65@44000@16:36:26
PRE HANDLE UNBLOCK [e70e30|CtxDummySession|dummy_2_s]
RECONFIGURATION 192.168.1.65@44000@16:36:51
PRE HANDLE UNBLOCK [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 2 [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 2 [e70e30|CtxDummySession|dummy_2_s]
```

Figura 29 : Desbloqueio de sessões na aplicação

Com o desbloqueio da sessão “dummy_1_s” todas as sessões ficam desbloqueadas, e todos os eventos em circulação (msg 2) ou pendentes (msg 3 e msg 4) são processados. A ordem pela qual foram inseridos no canal é garantida (msg 2, msg 3, msg 4).

```
RECONFIGURATION 192.168.1.65@44000@16:37:41
PRE HANDLE UNBLOCK [16d2633|CtxDummySession|dummy_1_s]
HANDLE EVENT MyEccoEvent msg 2 [16d2633|CtxDummySession|dummy_1_s]
HANDLE EVENT MyEccoEvent null [16d2633|CtxDummySession|dummy_1_s]
HANDLE EVENT MyEccoEvent null [e70e30|CtxDummySession|dummy_2_s]
HANDLE EVENT MyEccoEvent null [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent null [3c9217|CtxDummySession|dummy_4_s]
HANDLE EVENT MyEccoEvent null [9b42e6|CtxDummySession|dummy_5_s]

On [Thu Sep 23 16:37:41 BST 2010] : msg 3
> HANDLE EVENT MyEccoEvent msg 4 [9b42e6|CtxDummySession|dummy_5_s]
HANDLE EVENT MyEccoEvent msg 4 [3c9217|CtxDummySession|dummy_4_s]
HANDLE EVENT MyEccoEvent msg 4 [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent msg 4 [e70e30|CtxDummySession|dummy_2_s]
HANDLE EVENT MyEccoEvent msg 4 [16d2633|CtxDummySession|dummy_1_s]
```

Figura 30 : Rota dos eventos no canal desbloqueado

Reconfiguração de um canal com eventos bloqueados

Neste cenário verificada a funcionalidade da substituição de uma sessão por outra. Esta operação é realizada com um evento (msg 1) bloqueado e outro evento pendente recebido pela rede (msg 2). A troca de sessões implica a redefinição do caminho desses eventos, uma vez que uma sessão que esses não visitam é substituída por outra à qual

esses são entregues. Neste cenário é testado o modo de escalonamento preemptivo dos eventos de reconfiguração.

Neste cenário são validadas as seguintes funcionalidades:

- ✓ substituição de uma sessão que não pertence ao caminho por outra que pertence.
- ✓ bloqueio automático de sessão que substitui sessão bloqueada.
- ✓ actualização do caminho (vector sessões, índices posicionamento) de eventos em circulação e eventos pendentes.

As operações invocadas na consola de monitorização são mostradas na figura 31.

```
create_context_id -n ctx_1 -h localhost -p 40001 -c ecco_c
make_quiscents -n ctx_1 -s dummy_not_handle_4_s -s dummy_3_s
MAKE QUIESCENT 192.168.1.65@44000@02:22:13
replace_sessions -n ctx_1 -i 4 -s new_dummy_replacing_4_s -t net.sf.appia.demo.context.reconfigurati
REPLACE 192.168.1.65@44000@02:24:12
resume_sessions -n ctx_1 -s new_dummy_replacing_4_s -s dummy_3_s
RESUME 192.168.1.65@44000@02:24:38
```

Figura 31 : Reconfiguração de canal com sessões bloqueadas

Com o bloqueio da sessão "dummy_3_s" a que devia ser entregue (fig. 32), o evento "msg 1" apenas é processado pela sessão "dummy_5_s" bloqueando se seguida.

```
RECONFIGURATION 192.168.1.65@44000@02:22:13
PRE HANDLE BLOCK [3c9217|CtxDummyNotHandleSession|dummy_not_handle_4_s]
PRE HANDLE BLOCK [154864a|CtxDummySession|dummy_3_s]
msg 1
> HANDLE EVENT MyEccoEvent msg 1 [9b42e6|CtxDummySession|dummy_5_s]
RECONFIGURATION 192.168.1.65@44000@02:24:12
```

Figura 32 : Evento bloqueado

A substituição (fig. 33) da sessão "dummy_not_handle_4_s" pela sessão "new_dummy_replacing_4_s" adiciona mais uma sessão ao caminho do evento "msg 1". Nesta operação o vector de sessões do evento e os seus índices de posicionamento são actualizados.

```
RECONFIGURATION 192.168.1.65@44000@02:24:12
SESSIONS PRE REP ecco_c
[1742700|EccoSession|ecco_s]
[14520eb|ReconfigAgentSession|reconfiguration_agent_s]
[9b42e6|CtxDummySession|dummy_5_s]
[3c9217|CtxDummyNotHandleSession|dummy_not_handle_4_s]
[154864a|CtxDummySession|dummy_3_s]
[e70e30|CtxDummyNotHandleSession|dummy_not_handle_2_s]
[16d2633|CtxDummyNotHandleSession|dummy_not_handle_1_s]
[18a49e0|TcpCompleteSession|tcp_s_managed]

PRE HANDLE REP [3c9217|CtxDummyNotHandleSession|dummy_not_handle_4_s]
CHANNEL ecco_c UPDATED ON SESSION DELETED
CHANNEL ecco_c UPDATED ON SESSION INSERT
SESSIONS POS INS ecco_c
[1742700|EccoSession|ecco_s]
[14520eb|ReconfigAgentSession|reconfiguration_agent_s]
[9b42e6|CtxDummySession|dummy_5_s]
[29ce8c|CtxDummySession|new_dummy_replacing_4_s]
[154864a|CtxDummySession|dummy_3_s]
[e70e30|CtxDummyNotHandleSession|dummy_not_handle_2_s]
[16d2633|CtxDummyNotHandleSession|dummy_not_handle_1_s]
[18a49e0|TcpCompleteSession|tcp_s_managed]
```

Figura 33 : Substituição de sessão bloqueada

A substituição da sessão "dummy_not_handle_4_s" bloqueada, força o bloqueio da nova sessão "new_dummy_replacing_4_s" que para processar o evento tem de ser desbloqueada (fig. 34).

```
RECONFIGURATION 192.168.1.65@44000@02:24:38
PRE HANDLE UNBLOCK [29ce8c|CtxDummySession|new_dummy_replacing_4_s]
PRE HANDLE UNBLOCK [154864a|CtxDummySession|dummy_3_s]
```

Figura 34 : Desbloqueio de sessão

Com o desbloqueio das sessões, os eventos bloqueados ("msg 1" e "msg 2") são processados por todas as sessões do caminho actualizado a que ainda não tinham sido entregues (fig. 35).

```
HANDLE EVENT MyEccoEvent msg 1 [29ce8c|CtxDummySession|new_dummy_replacing_4_s]
HANDLE EVENT MyEccoEvent msg 1 [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent null [154864a|CtxDummySession|dummy_3_s]
HANDLE EVENT MyEccoEvent null [29ce8c|CtxDummySession|new_dummy_replacing_4_s]
HANDLE EVENT MyEccoEvent null [9b42e6|CtxDummySession|dummy_5_s]

On [Fri Sep 24 02:24:38 BST 2010] : msg 2
```

Figura 35 : Rota de evento em canal desbloqueado

6.3 Análise de desempenho

A análise do desempenho avalia o impacto no desempenho da plataforma Appia durante a operação dos canais. O modelo de invocação das operações baseado em eventos, faz com que o custo de os processar seja insignificante, uma vez que representam em condições normais um número muito reduzido. No entanto, as alterações introduzidas no algoritmo de escalonamento, essencialmente para suportar bloqueio das sessões, implicam que mesmo durante a normal operação do canal, ao

processar os eventos sejam feitas algumas verificações com um custo que importa quantificar.

Ao bloquear uma sessão, que leve ao bloqueio do próprio canal, impede-se que os eventos sejam processados no entanto continua a ser possível que esses sejam introduzidos no sistema e que fiquem a aguardar a entrada no canal. Estes eventos vão-se acumular numa lista, e pelo mecanismo de preempção dos eventos de reconfiguração, essa lista pode ter que ser percorrida para extrair um evento que desbloqueio o canal. O número de eventos acumulados na lista pode fazer variar o tempo que demora a ser extraído esse evento. Em condições normais pode ser possível calcular o número de eventos que entram no canal num período de tempo, pelo que será interessante perceber o custo de retomar a operação do canal.

A reconfiguração do canal, que altere a rota dos eventos implica a actualização de cada evento que esteja no canal ou à espera de ser processado. Estas alterações implicam um cálculo para cada um dos eventos que até então realizado antes do arranque do canal. Importa pois, perceber qual o custo de actualizar as rotas de cada evento, uma vez que a actualização das rotas do canal é feita uma única vez, pelo que tem um custo simbólico.

Os testes foram realizados utilizando a ferramenta *Eclipse* em modo *Debug*. Para medição dos tempos foram impressos no ecrã apenas os valores obtidos, deste modo

Processamento dos eventos

Para medir o impacto das alterações na performance do Appia foi criada uma composição que faz circular dois eventos em ambas as direcções do canal. No topo e na base da pilha, existe uma sessão (*CtxPingPongSession*) que inverte o sentido do evento e o reintroduz no canal. Os eventos percorrem o vector várias vezes sendo contabilizado o tempo que demoram nesse percurso. Para que os valores sejam mais facilmente comparáveis, uma vez que o tempo que demora a ir de uma lado ao outro do vector é insignificante, foi considerado o tempo que demora a percorrer o vector num sentido (uma volta) um milhão de vezes (1000000). Para que a análise dos resultados pudesse ser mais fiável, para cada evento foram recolhidas amostras do tempo dispendido nessa tarefa quinze (15) vezes.

A tabela 14 mostra os parâmetros dos três cenários (C1, C2, C3), que diferem no número de sessões dummy (*CtxDummySession*) posicionadas entre as sessões *CtxPingPongSession*. As sessões dummy processam os eventos (*CtxPingPongEvent*) em circulação mas apenas o reintroduzem no canal no mesmo sentido em que foi recebido.

Parâmetro	C1	C2	C3
Nº total de voltas	15000000	15000000	15000000
Nº de voltas por período de tempo	1000000	1000000	1000000
Nº de sessões dummy	3	6	9
Nº de eventos em circulação	2	2	2

Tabela 14 : Parâmetros dos cenários de teste de desempenho

As tabelas seguintes mostram os resultados obtidos com os três cenários. A chave identifica cada uma das configurações da pilha testada. Em cada configuração foi adicionado entre as sessões *CtxPingPongSession* a sessão ou sessões que são injectadas na composição da aplicação para suportar o mecanismo. Os testes realizados apenas calcularam os tempos de circulação dentro de um canal, para que não houvesse impactos do tempo de latência de rede. Foi também medido o custo de suportar só a monitorização ou reconfiguração, de forma a perceber qual o impacto de suportar apenas uma das funcionalidades, no entanto é provável que sejam sempre ambas incluídas.

Chave	Suporte	Tempo médio (ms)	Degradação (%)
O	Appia Original	4464	
A	Reconfiguração	5035	12,78
B	Monitorização	5055	13,23
C	Monitorização & Reconfiguração	5198	16,44
M	Média (A+B+C)	5096	14,15

Tabela 15 : Resultados do cenário de C1

Chave	Suporte	Tempo médio (ms)	Degradação (%)
O	Appia Original	7127	
A	Reconfiguração	8322	16,77
B	Monitorização	8515	19,47
C	Monitorização & Reconfiguração	8606	20,75
M	Média (A+B+C)	8481	19,00

Tabela 16 : Resultados do cenário de C2

Chave	Suporte	Tempo médio (ms)	Degradação (%)
O	Appia Original	9525	
A	Reconfiguração	11623	22,03
B	Monitorização	11760	23,47
C	Monitorização & Reconfiguração	11809	23,98
M	Média (A+B+C)	11731	23,16

Tabela 17 : Resultados do cenário de C3

A análise dos resultados obtidos, permite concluir que o suporte ao mecanismo de monitorização e reconfiguração tem um custo de desempenho da plataforma. A comparação dos resultados obtidos nos três cenários, permite identificar um decréscimo do desempenho associado ao aumento de sessões na pilha. O valor aproximado observável é de 4% por cada três sessões adicionadas. O número de sessões de uma pilha tipicamente não excede as seis sessões pelo podemos considerar o cenário C1 o mais exemplificativo da degradação de desempenho do Appia. Neste cenário embora haja apenas três sessões *CtxDummySession* na pilha, essa é também constituída pelas sessões injectadas e pelas duas sessões *CtxPingPongSession*, que exemplifica uma pilha com seis sessões. Sendo assim, o valor de referência para o decréscimo de desempenho associado ao suporte da funcionalidade será de aproximadamente 14%.

Sendo a degradação associada a três sessões dummy de 4%, subtraindo ao valor médio obtido em cada teste podemos considerar que os restantes 10% estarão associados às alterações realizadas no controlo da instância Appia. Foi redefinido o

modo como essa instância itera nos escalonadores de eventos invocando nesses o método de tratamento de um evento. Para este controlo, foi preciso definir duas variáveis que são alteradas a cada inserção e remoção de evento do canal. Para além disso, a instância Appia tem agora que invocar um método de cada escalonador para saber se esse pretende processar algum evento. A constatação do decréscimo assinalável de desempenho por este factor, é um indicador que melhores resultados poderia obtida se esse controlo fosse possível de ser realizado de outra maneira.

Uma análise crítica dos resultados obtidos, permite afirmar que o decréscimo de desempenho da plataforma Appia é aceitável uma vez que o benefício de suportar a reconfiguração dinâmica das pilhas é uma mais-valia em termos funcionais. Mais, a reconfiguração

Capítulo 7

Conclusão e Trabalho Futuro

O foco deste trabalho consistiu no desenvolvimento de um mecanismo de reconfiguração dinâmica dos protocolos e composições na plataforma Appia. Para tal foram analisados os algoritmos e modelos que concretizam esta plataforma no sentido de identificar as alterações necessárias à extensão da plataforma para que fornecesse um serviço transparente e que não comprometesse a correcção dos protocolos em execução.

A filosofia da plataforma, motivou o desenvolvimento de uma solução baseada em eventos, que permitiram a interacção com componentes de gestão externos à composição, bem como, suportaram dentro da própria composição a interacção com as sessões e canais reconfigurados. Deste modo foi definido um protocolo de reconfiguração que permite a execução concertada de operações de reconfiguração. Esse protocolo fornece um conjunto de operações básicas, endereçadas a um canal no entanto foi enriquecido com um mecanismo de bloqueio de sessões que permite que sejam orquestradas várias operações obtendo um resultado equivalente ao da execução de uma única operação composta.

O mecanismo de bloqueio desenvolvido tirou partido da optimização feita na definição do caminho percorrido pelos eventos, de forma a permitir que mesmo com um bloqueio parcial do canal, alguns eventos pudessem circular, reduzindo o tempo de inoperacionalidade ao mínimo.

Sendo uma plataforma vertical, o Appia suporta a sua execução na informação sobre o encaminhamento dos eventos na pilha. A solução apresentada teve pois que criar um mecanismo que permitisse alterar essa informação, projectando-a nos eventos que circulam no canal. Esta foi de facto a maior dificuldade, uma vez que essa informação é atribuída em tempo de inicialização, implicando uma convergência do estado do canal e dos eventos em função daquilo que é a estrutura do canal antes e depois da reconfiguração.

A implementação do mecanismo requereu um esforço elevado de análise no entanto saldou-se por pequenas alterações ao nível do código de escalonamento de

eventos, código esse crítico no desempenho da plataforma em tempo de execução. Os resultados obtidos comprovam a eficiência da implementação, conseguindo-se um mecanismo que implica um decréscimo inferior a 15% em pilhas com um número aceitável de protocolos.

Para validação destes resultados e da própria funcionalidade, foram desenvolvidas duas consolas que permitem a invocação das operações remotamente. O protocolo permite também que qualquer sessão possa proceder à leitura e escrita dos atributos de uma outra sessão do seu canal.

Como trabalho futuro, será necessário alterar o mecanismo de gestão dos cabeçalhos dos eventos. Este trabalho não apresenta uma solução para este problema, pois a abordagem lógica será refazer o mecanismo de forma a que os cabeçalhos sejam geridos como um saco de onde cada cabeçalho possa ser extraído individualmente. Neste momento o Appia implementa esse conceito como uma lista, que implica que os cabeçalhos sejam adicionados e removidos pela mesma ordem. Com isto, quando um canal é reconfigurado, os eventos embora sejam actualizados, os seus cabeçalhos permanecem inalterados.

Uma solução de compromisso poderia passar por implementar em cada sessão métodos que permitissem remover os cabeçalhos temporariamente, guardando-os no seu próprio estado, para que pudessem ser adicionados de novo. Um mecanismo deste género permitira iterar no vector de sessões de cada evento, e refazer a lista de cabeçalhos, usando para tal a capacidade de as próprias sessões de os armazenarem.

Dependendo da reconfiguração algumas das sessões não voltariam a inserir os cabeçalhos, resultando na remoção desses do evento. Outro requisito é aplicável nesta situação. As sessões teriam que ser capazes de gerar cabeçalhos que pudessem preencher o espaço num determinado evento, caso fosse adicionada uma sessão desse tipo ao canal.

Referências

[M01] Miranda, Hugo, 2001. Plataforma de suporte ao desenvolvimento e composição de protocolos, Dissertação para obtenção do grau de Mestre de Informática, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[P01] Pinto, Alexandre, 2004. Plataforma de suporte ao desenvolvimento e composição de protocolos, Dissertação para obtenção do grau de Mestre de Informática, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[MPR01] Miranda, Hugo, Pinto, Alexandre, Rodrigues, Luis, 2001. Appia, a Flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, USA, IEEE Computer Society (April 16-19 2001) 707-710.

[P05] Pinto, Alexandre, 2005. Appia Group Communication, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[MPC05] Miranda, Hugo, Pinto, Alexandre, Carvalho, Nuno, Rodrigues, Luis, 2005. Appia Protocol Development Manual v2.1, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[BHSC98] N. Bhatti, M. Hiltunen, R. Schlichting, W. Chiu, 1998. Coyote: A system for constructing fine-grain configurable communication services. ACM Transactions on Computer Systems.

[HP91] Hutchinson e L. Peterson, 1991. The x-kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering, 17(1):64-76.

[RBM96] van Renesse, Robbert, Birman, Ken, Maffeis, Silvano, 1996. Horus: a Flexible group communication system. Communications ACM, 39(4):76-83.

[RBHVK98] van Renesse, Robbert, Birman, Ken, Hayden, Mark, Vaysburd, Alexey, Karr, David, 1998. Building adaptive systems using ensemble. Software Practice Expert, 28(9):963-979.

[GG98] Garbinato, Benoit, Guerraoui, Rachid, 1998. Flexible protocol composition in Bast. In Preoceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18), Amsterdam, The Netherlands, IEEE Computer Science Press (Mai 26-29 1998) 22-29.

[RRL07] Rosa, Liliana, Rodrigues, Luis, Lopes, Antónia, 2007. From Appia to R-Appia: Refactoring a protocol composition framework for dynamic reconfiguration. DI/FCUL TR07-4, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[MRAMRL06] Mocito, José, Rosa, Liliana, Almeida, Nuno, Miranda, Hugo, Rodrigues, Luis, Lopes, Antónia, 2006. Context adaptation of the communication stack. International Journal of Parallel, Emergent and Distributed Systems (IJPEDS), 21(2):169-181.

[RRL04] Rosa, Liliana, Rodrigues, Luis, Lopes, Antónia, 2004. Building Adaptive Systems with Service Composition Frameworks, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[RRL98] Rosa, Liliana, Rodrigues, Luis, Lopes, Antónia, 2008. Modelling Adaptive Services for Distributed Systems, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

[T96] Tanenbaum, Andrew, 1996, Computer Networks, 3rd ed., New Jersey, Prentice-Hall 17-20.

[RLR06] Rosa, Liliana, Lopes, Antónia, Rodrigues, Luis, 2006. Policy-driven adaptation of protocol stacks. In ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems, pages 5-12.

[RBR00] Rodrigues, Luis, R. Baldoni, E Raynal, 2000. Deadline-Constrained Causal Order, In Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing, Newport Beach, California, USA (March 15-17 2000).