Instalação do *Appia*

Protocolos em Redes de Dados
FCUL 2000-2001

Hugo Miranda Alexandre Pinto Luís Rodrigues

1 Introdução

O *Appia* [4, 3] é uma plataforma de suporte à composição de protocolos desenvolvida em Java. Nas secções seguintes são descritos os passos necessários à instalação desta plataforma. Da secção 5 em diante, são apresentadas algumas noções fundamentais para a utilização dos protocolos de suporte à comunicação em grupo.

2 Requisitos Mínimos

O *Appia* foi desenvolvido utilizando o Java Development Kit versão 1.2.2. A execução das classes disponibilizadas foi testada utilizando a Java Virtual Machine que acompanha essa distribuição em Linux Red Hat 6.1 e Windows NT.

Espera-se que a plataforma continue a executar em versões posteriores.

3 Instruções de instalação

As instruções seguintes assumem que os ficheiros distribuídos (appia_v1_0.zip e appia_prdv1_0.zip) se encontram numa directoria que será denominada por #APPIA#. Por convenção, sempre que são apresentados caminhos de directorias (*paths*) é utilizado o separador "/". Os utilizadores de Windows deverão naturalmente substituir este separador pelo utilizado nesse sistema operativo: \.

3.1 Instalação da plataforma

A plataforma e alguns protocolos estão contidos no ficheiro comprimido appia_v1_0.zip. Não é necessário descomprimir este ficheiro bastando garantir

que ele está referido na variável de ambiente CLASSPATH.

3.2 Instalação do código fornecido para o projecto

O código específico para o projecto está disponível em appia_prdv1_0.zip, que contem os seguintes ficheiros:

prd.java Classe que lança a aplicação de testes do projecto. A sua utilização é descrita nas secções 3.4 e 4;

appia/protocols/totalTemplate/TotalTemplLayer.java Template para a classe Layer do protocolo;

appia/protocols/totalTemplate/TotalTemplSession.java Template para a classe Session do protocolo;

No Linux este ficheiro é descomprimido pelo comando

```
$ unzip appia_prdv1_0.zip
```

No Windows deve ser utilizado o utilitário WinZip. Chama-se a atenção que estas instruções assumem que a directoria onde os ficheiros são descomprimidos é a mesma para onde ambos os ficheiros de instalação foram inicialmente salvaguardados.

Uma listagem da directoria #APPIA# deverá agora conter:

```
$ ls
appia_v1_0.zip appia/ prd.java
```

3.3 Verificação da instalação

Após a execução dos passos anteriores, deverá já ser possível verificar se a plataforma foi correctamente instalada. O primeiro passo é definir a variável de ambiente CLASSPATH.

No Linux o comando a executar é:

```
$ export CLASSPATH=$CLASSPATH:#APPIA#/appia_v1_0.zip:#APPIA#/.
```

Nas versões do Windows o comando a executar é:

```
$ set CLASSPATH=%CLASSPATH%;#APPIA#\appia_v1_0.zip;#APPIA#\.
```

Uma vez mais se chama a atenção que a expressão #APPIA# deve ser substituida pela directoria onde os ficheiros compactados foram guardados e onde o comando unzip (ou winzip) foi executado.

Neste momento, a aplicação pode ser testada. Os procedimentos para o lançamento da aplicação estão detalhados na secção 4.1.

3.4 Personalização da instalação

Importa agora personalizar a instalação de forma a que reflita o algoritmo a concretizar e a identificação do grupo.

Identificação do protocolo O protocolo deverá ser entregue numa directoria com um nome único gerado a partir do algoritmo a concretizar e da identificação do grupo.

À palavra Total concatena-se (sem espaços):

Token Se o algoritmo a concretizar for baseado num token;

Seq Se o algoritmo a concretizar utilizar um sequenciador;

Causal Se o algoritmo a concretizar utilizar a ordenação causal;

Em seguide concatene o número de grupo.

Exemplo: se o algoritmo a concretizar pelo grupo 53 utilizasse um sequenciador, o nome do protocolo seria: TotalSeq53.

O primeiro T deverá ser maíusculo quando se referir a classes (ficheiros) e minúsculo quando se referir a packages (directorias).

Realize agora os seguintes passos:

1. Crie uma directoria com esse nome abaixo de #APPIA#/appia/protocols. Este será também o nome do *package* a utilizar. Por convenção, as directorias são iniciadas por uma letra minúscula.

2. Copie o conteúdo da directoria #APPIA#/appia/protocols/totalTemplate para a directoria recem criada. Não se esqueça de alterar os nomes dos ficheiros por forma a reflectir a nova designação do protocolo.

3. Edite os ficheiros que copiou por forma a que reflitam a nova designação, nomeadamente, altere o nome das classes e dos construtores.

4. Compile o novo protocolo recorrendo ao comando javac. Da compilação não deve resultar qualquer erro.

5. Altere o ficheiro #APPIA#/prd.java por forma a reflectir a utilização do novo protocolo substituindo todas as referências a TotalTempl pelo novo nome. Ao longo de todo o ficheiro existem duas referências que é necessário alterar, ambas demarcadas por comentários. A primeira na secção das cláusulas import e a segunda na definição do vector qos (a classe TotalTemplLayer está na posição 7 do vector).

6. Compile o ficheiro. Se da compilação resultar algum erro, verifique se tem a variável de ambiente CLASSPATH definida.

7. Volte a executar a aplicação, assegurando-se de que não ocorrem erros durante a inicialização.

4 Utilização da aplicação *prd*

4.1 Lançamento

A aplicação é lançada pela seguinte linha de comando:

java prd -porto <porto UDP> -rank <posicao na vista> -vista <descricao da vista>

onde

porto UDP é o número do porto que deverá ser utilizado pelo processo;

posicao na vista é a posição que o processo ocupa na vista descrita no argumento seguinte. A numeração começa no 0.

descricao da vista é uma sequência de pares da forma IP:porto, separados por espaços, cada uma delas identificando um processo participante no grupo. A descrição da vista tem que ser identica para todos os processos, em particular tem que ter a mesma ordem em todos os processos.

Exemplo: As linhas de comando seguintes mostram o lançamento de um grupo com 3 processos.

java prd -porto 5000 -rank 0 -vista 194.117.21.145:5000 194.117.21.145:5010 194.117.21.145:5020

java prd -porto 5010 -rank 1 -vista 194.117.21.145:5000 194.117.21.145:5010 194.117.21.145:5020

java prd -porto 5020 -rank 2 -vista 194.117.21.145:5000 194.117.21.145:5010 194.117.21.145:5020

Os processos devem ser lançados simultaneamente. Caso contrário, ao fim de alguns segundos a aplicação começa a notificar o utilizador de que os restantes falharam. Esses processos, mesmo que lançados posteriormente, já não serão aceites no grupo entretanto composto e destabilizam a execução dos que tinham já formado uma vista.

4.2 Interface

O interface da aplicação é extremamente reduzido, permitindo apenas as operações necessárias para que os grupos possam avaliar os seus trabalhos.

Após o lançamento, todos os processos notificam o utilizador sobre os membros que compõem o grupo. No caso do exemplo anterior são apresentadas as seguintes linhas:

```
Aberto canal com o nome canalPRD
Instalada nova vista:
Lista de membros (por IP:porto):
{194.117.21.145:5000}
{194.117.21.145:5010}
{194.117.21.145:5020}
Sou o coordenador do grupo
```

A última linha deste texto só é apresentada pelo processo lançado com *rank* 0. Os restantes declaram não ser coordenadores do grupo.

A aplicação aceita 3 comandos:

help que fornece informação sobre os dois comandos restantes;

cast r i m que envia r vezes a mensagem m. Cada reenvio é intervalado i décimas de segundo.

send x r i m que envia r vezes a mensagem m ponto-a-ponto para o membro de índice x. Cada reenvio é intervalado i décimas de segundo.

debug {start|stop|now} [**outfile**] que envia um evento de debug para o canal de comunicação. Este evento pode ser utilizado pelos grupos para obterem informação sobre o seu protocolo. A informação de depuração gerada será enviada para o ficheiro outfile ou, por omissão deste, para o *stdin*. Para mais informação sobre a utilização deste evento, os grupos devem consultar a API da plataforma.

A fim de facilitar a verificação da ordenação total, o número da repetição é adicionado a todas as mensagens transmitidas e impresso no écran conjuntamente com estas.

A linha de comando mantém-se activa enquanto um processo está a enviar uma sequência de mensagens. Em resultado, cada processo pode estar a gerar, simultaneamente, várias sequências de mensagens.

Os processos suspendem o envio de mensagens quando o grupo se bloqueia, e retomam-no quando a nova vista é instalada.

5 *Eventos* para *Comunicação em Grupo*

Para utilizar os mecanismos de *Comunicação em Grupo* existe um conjunto de Eventos que qualquer camada tem que processar e compreender.

Em primeiro lugar vamos descrever um dos modelos mais importantes para a *Comunicação em Grupo*, a *sincronia virtual* [1, 2]. Logo de seguida explicaremos a sucessão de Eventos que corresponde à concretização do modelo. Depois vamos descrever de forma sumaria outros Eventos secundários que são gerados e que poderão ser úteis para certas concretizações. Por fim vamos especificar como colocar cabeçalhos na mensagem/evento.

5.1 Sincronia Virtual

Informalmente, a *sincronia virtual* disponibiliza informação sobre a constituição de um grupo, sob a forma de *vistas*, e garante que todos os processos que instalem 2 vistas consecutivas, entregam o mesmo conjunto de mensagens entre essas vistas.

Esta definição impõem *ordem total* entre as mudanças de vistas e as mensagens enviadas, mas não impõem nenhuma ordenação entre as mensagens entregues numa determinada vista. Na nossa concretização garantimos ainda que todas as mensagens recebidas são entregues na vista em que foram enviadas.

A concretização da *sincronia virtual* necessita do uso de um *detector de falhas* e da execução um protocolo de *escoamento* que garanta que todas as mensagens recebidas por algum processo, são entregues em todos os processos antes da nova vista ser instalada. Para garantir que o protocolo de *escoamento* termina, existe a necessidade de parar temporariamente o tráfego de mensagens dentro do grupo. Isto é executado por um protocolo de *bloqueio*. Embora este bloqueio possa levar a uma degradação do desempenho do sistema durante o processo de mudança de vista, simplifica o desenho da aplicação porque permite entregar qualquer mensagem assim que é recebida.

6 Funcionamento: Eventos

O funcionamento pode ser dividido em 2 períodos básicos: normal e mudança de vista.

Durante a **execução normal** do sistema mensagens podem ser enviadas para outros elementos do grupo. Estas mensagens são representadas, e encapsuladas, pelo evento *GroupSendableEvent* e suas subclasses. Eventos que herdem de *GroupSendableEvent* são por omissão do tipo *cast*, ou

seja, destinam-se a todos os elementos do grupo. Ver secção 6.1 para saber como enviar mensagens ponto-a-ponto. Assim qualquer camada desenvolvida com o intuito de ordenar as mensagens deve ordenar todos os eventos *GroupSendableEvent* ou suas subclasses. Igualmente qualquer camada que pretenda enviar um mensagem para o grupo, de um determinado tipo especifico, deve criar um evento que seja subclasse de *GroupSendableEvent*.

Durante a **execução do protocolo de mudança de vista**, que garante a *sincronia virtual*, são gerados um conjunto de eventos que qualquer camada deve estar a par. Assim para bloquear os processos é gerado o evento *BlockOk* que percorre a pilha de cima para baixo. Cada camada ao receber este evento é informada da necessidade de bloquear o envio de mensagens. Ela pode capturar temporariamente este evento, para por exemplo enviar mensagens pendentes, mas quando o liberta, o reenvia, compromete-se a não enviar mais mensagens até a nova vista ser entregue.

O fim da execução do protocolo de mudança de vista, e a informação da nova vista, são indicados pelo evento *View*. Este evento contém o estado actual do grupo[1] (membros, endereços, ...) e um estado local a cada processo[2] (sou o coordenador?, o meu índice na vista, ...). A partir da recepção deste evento as camadas podem voltar a transmitir mensagens normalmente. O coordenador do grupo é sempre o elemento com o índice mais baixo que ainda não falhou.

Para concluir resta-nos relembrar que:

> **Todas** as camadas que geram eventos devem subscrever e processar os eventos *BlockOk* e *View*. Como foi referido anteriormente, depois de propagar o evento *BlockOk* a camada não deve enviar mais mensagens enquanto não receber o evento *View*.

6.1 Outros eventos

Agora vamos fornecer um breve lista, com uma breve descrição do seu significado, de outros eventos que podem ser processados:

Fail indica a falha de um, ou mais, dos elementos do grupo. Contém um vector de booleanos que indica os elementos que falharam. Trata-se de um evento descendente.

Stable ocorre durante o processo de mudança de vista, ou seja depois do *BlockOk*, e indica que o elemento já recebeu todas as mensagens da vista. Trata-se de um evento descendente.

[1] classe ViewState
[2] classe LocalState

ChannelInit este evento não é um evento de grupo, simplesmente indica que o canal está aberto, ou seja, que já se podem enviar e receber eventos. Este deve ser o primeiro evento a ser recebido por qualquer camada. De notar que os mecanismos de *comunicação em grupo* só ficam prontos quando é recebido o primeiro evento *View*. Trata-se de um evento ascendente.

ChannelClose este evento não é um evento de grupo, simplesmente indica que o canal está fechado, ou seja, que não se podem enviar ou receber eventos. Trata-se de um evento descendente.

Existe ainda uma classe[3] especial que serve como marcador de mensagens ponto-a-ponto, ou seja, qualquer subclasse de *GroupSendableEvent* que herde igualmente dela não é enviada em *cast* mas sim ponto-a-ponto. Os destinatários são definidos como um array dos índices na vista destes, colocado no atributo "dest" do evento. A classe chama-se *Send*. A aplicação de teste ilustra o envio de mensagens ponto a ponto.

6.2 Cabeçalhos e Atributos

Os cabeçalhos são colocados numa classe de nome *ObjectsMessage* que, como o próprio nome indica, concretiza uma mensagem composta por um conjunto de objectos. O conjunto de objectos é concretizado como uma pilha, ou seja, existem duas primitivas *push* e *pop* para respectivamente adicionar ou retirar um objecto. Não esquecer que uma pilha concretiza uma politica LIFO[4]. Assim uma camada deve primeiro definir os objectos que pretende enviar/receber como cabeçalho. Depois basta fazer push/pop de modo a colocar/obter esses objectos.

> ***Atenção:*** os atributos de um evento não são enviados na comunicação entre processos, ou seja, para passar um qualquer valor entre os processos deve ser utilizado o mecanismo de cabeçalhos. Os atributos de um evento são locais ao processo, e à pilha de protocolos.

O interface da *ObjectsMessage*, em termos práticos, é composto por dois métodos:

- "void push(Object obj)" que coloca um objecto como cabeçalho da mensagem.

[3] na realidade trata-se de um "interface"

[4] Last In First Out

- "Object pop()" que retira (segundo a politica LIFO) um objecto, cabeçalho, da mensagem.

Para obter o *ObjectsMessage* de um *GroupSendableEvent* invoca-se o método "ObjectsMessage getObjectsMessage()" do evento.

Quanto aos atributos, um evento do tipo *GroupSendableEvent* tem vários:

- **group** contém um objecto da classe *Group* que representa o grupo ao qual o evento/mensagem pertence.
- **view_id** contém o identificador da vista na qual o evento/mensagem foi enviado.
- **orig** só é valido para eventos ascendentes[5] e contém o índice na vista do membro que enviou o evento/mensagem.
- **source** só é valido para eventos ascendentes, e contém o identificador do membro (classe *Endpt*) que enviou o evento/mensagem.
- **dest** só é valido para eventos descendentes e que herdem de *Send* e deve conter um array com os índices na vista corrente dos destinatários.

Atenção que estes atributos devem ser encarados na sua maioria, e na maioria das situações, como sendo *apenas de leitura*.

Referências

[1] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing With the ISIS Toolkit*. Number ISBN 0-8186-5342-6. IEEE CS Press, March 1994.

[2] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical report, Department of Computer Science, Cornell University, March 1995.

[3] H. Miranda, A. Pinto, and L. Rodrigues. Aplication program interface specification of appia. Technical report, Universidade de LIsboa, Faculdade de Ciências, 2000.

[4] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cancún, México, September 1999. IEEE.

[5] "evento.getDirection().direction == Direction.UP"