

PHÁT TRIỂN PHẦN MỀM HƯỚNG AGENT

**BÀI GIẢNG DÀNH CHO SINH VIÊN
NGÀNH CÔNG NGHỆ THÔNG TIN**

TRẦN ĐÌNH QUẾ

Hà nội - 2010

LỜI NÓI ĐẦU

Phát triển phần mềm hướng agent được xem là cách tiếp cận tiến hóa của phát triển phần mềm dựa trên thành phần cho các hệ phân tán. Sự khác biệt của agent với các thành phần thông thường là các tính tự chủ, tính linh hoạt, tính xã hội... được tăng cường thêm vào trong các thành phần phần mềm. Cho đến nay, các đặc trưng này đã được nghiên cứu rộng rãi trong lĩnh vực trí tuệ nhân tạo. Các kiểu kiến trúc bên trong của agent cũng như hệ thống đa agent cùng các thuật toán cho các dạng tương tác như thương lượng, đấu giá...cũng đang được nhiều cộng đồng nghiên cứu. Agent và hệ đa agent đã đạt được nhiều tiến bộ đáng kể và đã được ứng dụng trong các lĩnh vực khác nhau đặc biệt trên môi trường mạng Internet như thương mại, dịch vụ y tế, giáo dục...và đã được đưa vào giảng dạy ở nhiều trường Đại học trên thế giới.

Tài liệu này nhằm phục vụ cho sinh viên năm cuối ngành Công nghệ thông tin. Tài liệu được soạn thảo phần lớn dựa trên cuốn sách *Developing Multi-Agent Systems with JADE* của Fabio Bellifemine et al. và các tài liệu tham khảo được liệt kê ở cuối tài liệu nhằm hỗ trợ cho sinh viên gặp khó khăn khi đọc tài liệu nguyên bản tiếng Anh*. Nội dung tài liệu bao gồm:

- Các kiến thức cơ bản về agent, hệ đa agent và các vấn đề liên quan đến phát triển hệ phần mềm phức tạp hướng agent.
- Môi trường phát triển hệ đa agent Jade và phương pháp luận cho phân tích và thiết kế hệ đa agent.

Tài liệu được cấu trúc như sau:

Chương 1 Giới thiệu agent và hệ đa agent

Chương này trước hết giới thiệu các khái niệm agent và hệ đa agent cùng các đặc trưng cơ bản của agent. Tiếp theo sẽ trình bày giao tiếp và phối hợp trong agent cùng các công cụ và ngôn ngữ lập trình agent cũng như ứng dụng hệ đa agent.

Chương 2 Giới thiệu Jade

Chương này tập trung trình bày lịch sử phát triển của nền tảng Jade, kiến trúc cơ bản của Jade cùng những đặc trưng của Jade. Chương này còn hướng dẫn cách chạy JADE bằng dòng lệnh và bằng giao diện đồ họa.

Chương 3 Những đặc điểm cơ bản của Jade

Chương này sẽ trình bày cách phát triển hệ đa agent với JADE dựa vào những tính năng cơ bản mà JADE cung cấp như tạo các agent, thực thi nhiệm vụ của agent, tạo điều kiện cho agent giao tiếp với nhau, và đưa ra các dịch vụ cũng như là tìm kiếm các dịch vụ trong mục trang vàng

* Tài liệu này được hình thành với sự đóng góp trực tiếp hoặc gián tiếp của nhiều thế hệ sinh viên Khoa CNTT Học viện CNBCVT, đặc biệt sinh viên chuyên ngành Công nghệ phần mềm lớp D06CNPM đã cập nhật bản thảo, bổ sung các chương trình ví dụ.

(yellow page). Bằng những tính năng này ta đã có thể cài đặt ứng dụng phân tán với một độ phức tạp nhất định.

Chương 4 Những đặc điểm nâng cao của Jade

Mặc dù Chương 3 đã mô tả những đặc điểm cơ bản của JADE khi phát triển những hệ đa agent phân tán nhưng trong thực tế việc cài đặt các ứng dụng với các đặc tính đó tương đối phức tạp và người phát triển sẽ phải giải quyết rất nhiều vấn đề. Chương này trình bày nâng cao những vấn đề liên quan tới việc kiểm soát các biểu thức nội dung phức tạp bằng ontology và ngôn ngữ nội dung codes, khả năng xây dựng các hành vi phức tạp từ những hành vi đơn giản, hỗ trợ cho việc tạo ra các cuộc hội thoại được tuân theo các giao thức tương tác được định nghĩa bởi FIPA.

Chương 5 Khả năng di động của agent trong Jade

Khả năng di động của Agent là một cách tiếp cận xuất phát từ 2 ngành khác nhau là Trí tuệ nhân tạo – tạo ra khái niệm về agent và Các hệ thống phân tán – định nghĩa khái niệm về mã di động. Đặc trưng này càng ngày càng được nhìn nhận là quan trọng trong hệ phân tán, đặc biệt cho các thiết bị di động. Chương này nhằm trình bày một số khái niệm liên quan agent di động và nền tảng Jade trong thiết kế các agent di động.

Chương 6 Kiến trúc bên trong của Jade

Cho tới thời điểm này chỉ có những tính năng được hỗ trợ bởi Jade run-time và API để truy cập các tính năng đó được trình bày. Trong phần này, chúng ta sẽ hướng vào kiến trúc cốt lõi bên trong của Jade bao gồm cả việc làm thế nào để xác định và mở rộng các hành vi của agent.

Chương 7 Phát triển hệ đa agent với phương pháp luận MaSE và Jade

Chương này bàn về vấn đề sử dụng phương pháp luận MaSE kết hợp với Jade khi phát triển các hệ đa agent phân tán. Cho đến nay có rất nhiều phương pháp luận cũng như rất nhiều khung (framework) được phát triển để xây dựng các hệ đa agent phức tạp. Một số phương pháp luận như Gaia, Tropos, Ingenias, MaSE (Multi-agent Systems Engineering)...và một số khung như TuCSoN (TUPLE Centre Spread Over the Network), JADE (Java Agent DEvelopment Framework), Jadex, DESIRE (Design and Specification of Interacting Reasoning components)...đã được giới thiệu. Chương này tập trung trình bày phương pháp luận MaSE kết hợp với nền tảng JADE và một case study Quản lý Hội nghị.

MỤC LỤC

Chương 1 GIỚI THIỆU AGENT VÀ HỆ ĐA AGENT.....	1
1.1 AGENT	1
1.1.1 Khái niệm Agent	1
1.1.2 Các kiểu kiến trúc	2
1.1.3 Giao tiếp và phối hợp	4
1.1.4 Ngôn ngữ lập trình và công cụ	7
1.1.5 Ứng dụng của các hệ thống đa agent	9
1.2 CƠ SỞ HÌNH THÀNH AGENT VẬT LÝ VÀ AGENT THÔNG MINH (FIPA)	10
1.2.1 Lịch sử và mục đích của FIPA	11
1.2.2 Các khái niệm cốt lõi của FIPA	13
1.2.3 Một số đặc tả FIPA chính	18
1.2.4 Liên quan giữa FIPA và JADE	30
Chương 2 GIỚI THIỆU JADE	34
2.1 TÓM TẮT LỊCH SỬ	34
2.2 JADE VÀ MÔ HÌNH AGENT	35
2.3 KIẾN TRÚC JADE	38
2.4 BIÊN DỊCH VÀ CHẠY CHƯƠNG TRÌNH	40
2.5 CÁC GÓI CỦA JADE	43
2.6 DỊCH VỤ VẬN CHUYỂN THÔNG ĐIỆP	46
2.6.1 Các giao thức vận chuyển thông điệp	46
2.6.2 Giao thức truyền thông điệp nội bộ (IMTP)	48
2.7 CÁC CÔNG CỤ QUẢN TRỊ VÀ GỖ LÕI	49
2.7.1 Cửa sổ quản trị JADE	51
2.7.2 Dummy agent	52
2.7.3 Sniffer Agent	54
2.7.4 Introspector agent	55
2.7.5 Log Manager Agent	56
2.7.6 Dịch vụ thông báo sự kiện (event notification service) và mô hình công cụ JADE	58
Chương 3 NHỮNG ĐẶC ĐIỂM CƠ BẢN CỦA JADE	65
3.1 TẠO AGENT	65
3.1.1 Định danh agent	66
3.1.2 Khởi tạo Agent	67

3.1.3 Kết thúc agent.....	67
3.1.4 Truyền tham số cho agent.....	68
3.1.5 Cài đặt dự án Book-Trading.....	69
3.2 CÀI ĐẶT NHIỆM VỤ CHO AGENT	72
3.2.1 Lập lịch và thực thi Behaviour	72
3.2.2 One-shot behaviour, cyclic behavior và generic behaviour	73
3.2.3 Bổ sung thêm về hành vi của agent	75
3.2.4 Lập lịch cho các hành vi của agent.....	75
3.2.5 Các hành vi trong ví dụ bookTrading	76
3.3 TRUYỀN THÔNG GIỮA CÁC AGENT	78
3.3.1 Gửi thông điệp	79
3.3.2 Nhận thông điệp.....	80
3.3.3 Khóa hành vi đợi thông điệp.....	80
3.3.4 Lựa chọn thông điệp từ hàng đợi.....	82
3.3.5 Các cuộc hội thoại phức tạp	82
3.3.6 Nhận thông điệp tại node đang khóa	85
3.4 KHÁM PHÁ AGENT – DỊCH VỤ TRANG VÀNG	85
3.4.1 DF agent.....	85
3.4.2 Tương tác với DF agent.....	86
3.4.3 Tìm kiếm dịch vụ	87
3.5 AGENT VỚI GIAO DIỆN ĐỒ HỌA	88
3.5.1 Thực hành lập trình tốt với bộ lắng nghe sự kiện AWT	88
3.5.2 Thực hành lập trình tốt bằng cách sửa đổi giao diện đồ họa trong luồng thực thi của Agent....	89
Chương 4 NHỮNG ĐẶC ĐIỂM NÂNG CAO CỦA JADE.....	96
4.1 ONTOLOGY VÀ NGÔN NGỮ NỘI DUNG	96
4.1.1 Các thành phần chính	97
4.1.2 Mô hình tham chiếu nội dung	97
4.1.3 Sử dụng Ontology và ngôn ngữ nội dung	98
4.1.4 Sử dụng Protégé và BeanGenerator add-on để tạo Ontology cho JADE	107
4.2 HỢP CÁC HÀNH VI ĐỂ XÂY DỰNG CÁC TÁC VỤ PHỨC TẠP	108
4.2.1 Lớp SequentialBehaviour	109
4.2.2 Lớp FsmBehaviour.....	110
4.2.3 Lớp ParallelBehaviour.....	112
4.2.4 Chia sẻ dữ liệu giữa các hành vi con: DATASTORE	112
4.2.5 Bổ sung về hành vi gộp	114

4.3 HÀNH VI LUÔNG.....	115
4.4 CÁC GIAO THỨC TƯƠNG TÁC	116
4.4.1 Gói jade.proto	117
4.4.2 Sử dụng các lớp giao thức	118
4.4.3 Lồng giao thức	120
4.5 KHỞI ĐỘNG JADE TỪ MỘT ỨNG DỤNG JAVA BÊN NGOÀI.....	121
4.5.1 Giao tiếp giữa Object và Agent.....	123
Chương 5 KHẢ NĂNG DI ĐỘNG CỦA AGENT TRONG JADE.....	129
5.1 THẾ NÀO LÀ TÍNH DI ĐỘNG CỦA AGENT	129
5.1.1 Một số ưu điểm và nhược điểm của agent di động	130
5.1.2 Di chuyển mạnh và di chuyển yếu.....	131
5.1.3 Kế hoạch di chuyển	131
5.2 DI CHUYỂN TRONG CÙNG PLATFORM.....	131
5.2.1 Các phương thức truy cập tới khả năng di động của Agent.....	131
5.2.2 Agent serialization	132
5.2.3 Lớp tải (classloader) của agent di động	132
5.2.4 Nhân bản Agent	133
5.2.5 Tuyên bố khả năng di động gián tiếp.....	133
5.3 DỊCH VỤ DI ĐỘNG LIÊN PLATFORM.....	133
5.3.1 Quá trình di cư	134
5.3.2 Tích hợp các dịch vụ di động	135
5.3.3 Nhóm code của các Agent với nhau	135
5.3.4 Ontology di động của JADE	136
5.4. SỬ DỤNG CÁC DỊCH VỤ DI CHUYỂN CỦA JADE	136
5.4.1. Dịch vụ di chuyển nội platform.....	136
5.4.2 Dịch vụ di chuyển liên platform.....	138
5.4.3 Xem xét vấn đề bảo mật của IPMS	138
5.4.4 Lập trình agent di động.....	138
5.4.5 Truy cập vào AMS bằng khả năng di động của agent.....	141
5.4.6 Ví dụ về tính di động của agent.....	144
Chương 6 KIẾN TRÚC BÊN TRONG CỦA JADE.....	147
6.1 GIỚI THIỆU CÁC BỘ LỌC CỘNG TÁC PHÂN TÁN	147
6.1.1 Ý tưởng và động cơ thúc đẩy	147
6.1.2 Các thành phần chính	148
6.1.3 Các thành phần dịch vụ	148

6.1.4 Lựa chọn các dịch vụ được kích hoạt	151
6.2 TẠO MỘT DỊCH VỤ LÕI TRONG JADE.....	152
6.2.1 Cài đặt lớp của dịch vụ	152
6.2.2 Khởi động dịch vụ.....	153
6.2.3 Sử dụng bộ lọc để chặn các lệnh đọc	154
6.2.4 Cài đặt một dịch vụ phân tán trong JADE.....	156
6.2.5 Tương tác giữa Agent và dịch vụ	159
Chương 7 PHÁT TRIỂN HỆ ĐA AGENT VỚI PHƯƠNG PHÁP LUẬN MaSE VÀ JADE	163
7.1 GIỚI THIỆU	163
7.2 GIỚI THIỆU NỀN TẢNG JADE	164
7.2.1 Một số nền tảng hỗ trợ phát triển hệ đa agent	164
7.2.2 Một số đặc điểm nổi bật của JADE	165
7.3 PHƯƠNG PHÁP LUẬN MaSE	167
7.3.1 Tổng quan về các pha trong MaSE.....	167
7.3.2 Phân tích và thiết kế với MaSE	168
7.4 PHÁT TRIỂN HỆ THỐNG PAPERMANAGEMENT	171
7.4.1 Miêu tả hệ thống	171
7.4.2 Phân tích.....	172
7.4.3 Thiết kế	173
7.5 CÀI ĐẶT HỆ THỐNG	175

TÀI LIỆU THAM KHẢO

CHƯƠNG 1

GIỚI THIỆU AGENT VÀ HỆ ĐA AGENT

Chương này trước hết giới thiệu các khái niệm về agent, tổng quan các công nghệ agent, kiến trúc agent, các ngôn ngữ lập trình và các công cụ phát triển. Tiếp theo sẽ mô tả các đặc tả của FIPA - tập các tiêu chuẩn phổ biến nhất và được chấp nhận rộng rãi cho phát triển các nền tảng và ứng dụng đa agent. JADE là một nền tảng tuân theo các đặc tả FIPA và hơn nữa nó còn mở rộng mô hình FIPA trong một số lĩnh vực như agent cho thiết bị di động, agent cho dịch vụ web.

1.1 AGENT VÀ HỆ ĐA AGENT

1.1.1 Khái niệm Agent

Thuật ngữ “agent”, hay agent phần mềm, đã được sử dụng rộng rãi và xuất hiện trong nhiều lĩnh vực nghiên cứu như trí tuệ nhân tạo, cơ sở dữ liệu, các tài liệu về hệ điều hành và mạng máy tính. Mặc dù cho đến nay vẫn chưa có một định nghĩa thống nhất về agent (Genesereth và Ketchpel (1994), Wooldridge và Jennings (1995), Russel và Norvig (2003)) nhưng tất cả các định nghĩa đều có chung một điểm rằng một agent, về bản chất, là một phần mềm máy tính đặc biệt có thể tự chủ và cung cấp một interface có khả năng tương thích với một hệ thống bất kì và/hoặc cư xử như là một agent con người hay đại diện cho một số client để thực thi các đích cho riêng mình. Mặc dù một hệ agent có thể chỉ cần dựa trên một agent đơn lẻ để làm việc trong một môi trường và tương tác với người dùng của nó khi cần thiết, tuy nhiên các hệ agent thường bao gồm nhiều agent. Những hệ thống đa agent (MAS: Multiagent System) có thể sử dụng để mô hình hóa các hệ thống phức tạp bao gồm các agent với các mục tiêu chung hoặc riêng. Những agent có thể tương tác với nhau một cách gián tiếp (qua tác động lên môi trường) hoặc trực tiếp (thông qua giao tiếp và thương lượng). Các agent có thể quyết định hợp tác cùng có lợi hoặc có thể cạnh tranh để phục vụ cho mục tiêu của mình.

Như vậy, agent có **tính tự chủ**, vì nó hoạt động mà không có sự can thiệp trực tiếp của con người hoặc các hệ thống khác và có khả năng kiểm soát được hành động và trạng thái bên trong của mình. Agent có **tính xã hội**, vì nó tương tác với con người hoặc các agent khác để hoàn thành nhiệm vụ của mình. Agent có **tính phản ứng**, bởi vì nó nhận thức được môi trường và đáp ứng một cách kịp thời với những thay đổi xảy ra trong môi trường. Agent có **tính hướng đích**, vì nó không chỉ đơn giản là hoạt động để phản ứng với môi trường của nó mà còn có khả năng thể hiện hoạt động hướng đích một cách chủ động. Agent có thể có **tính di động**, với khả năng di chuyển giữa các node trong một mạng máy tính. Nó có thể có **tính trung thực** nghĩa là luôn cung cấp sự thật. Nó có thể **tốt bụng**, luôn cố gắng thực hiện những gì được yêu cầu. Nó có thể **sáng**

suốt, luôn hoạt động nhằm đến để đạt được mục tiêu và không bao giờ ngăn cản việc đạt được mục tiêu của mình.

1.1.2 Các kiểu kiến trúc

Kiến trúc agent là cơ chế nằm bên dưới các thành phần tự chủ nhằm hỗ trợ hành vi của agent trong thế giới thực, môi trường động và môi trường mở. Trong thực tế, những nỗ lực ban đầu trong lĩnh vực tính toán dựa trên agent tập trung vào sự phát triển của các kiến trúc agent thông minh và đã đưa ra khá nhiều kiểu kiến trúc.

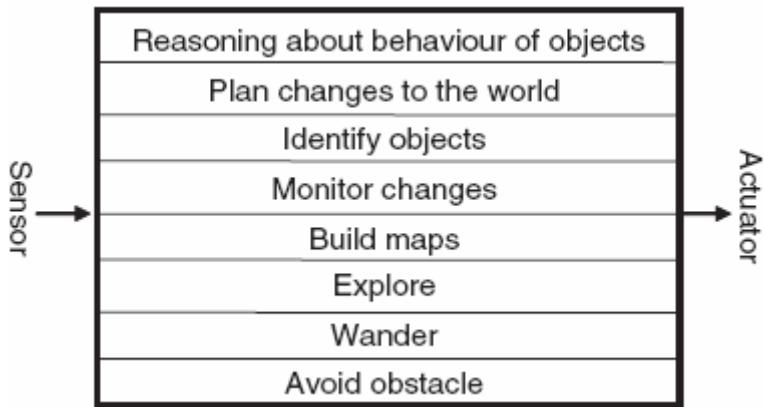
- Kiến trúc kiểu phản ứng (hoặc hành vi) hoạt động theo một kiểu “kích thích-đáp ứng” đơn giản, ví dụ những kiến trúc dựa trên kiến trúc gộp của Brooks (1991)
- Kiến trúc có thể lý giải các hành động của chúng, ví dụ như những loại dựa trên mô hình BDI (beliefs, desires, intentions) (Rao và Georgeff, 1995).
- Kiến trúc dựa trên sự kết hợp của cả hai kiến trúc trên, hay còn được gọi là các kiến trúc phân lớp, tận dụng những ưu điểm tốt nhất của mỗi loại.

Các kiến trúc agent có thể được chia thành 4 nhóm chính: dựa trên logic, có tính phản ứng, BDI và phân lớp. Những kiến trúc dựa trên logic (logic-based) lấy nền tảng từ kỹ thuật dựa trên tri thức truyền thống trong đó một môi trường được thể hiện và hoạt động bằng cách sử dụng các cơ chế lập luận. Ưu điểm của cách tiếp cận này là tri thức của con người được biểu diễn bởi các ký hiệu và vì thế mà việc mã hóa trở nên dễ dàng hơn và cũng làm cho con người hiểu logic hoạt động của nó dễ dàng hơn. Nhược điểm là rất khó để biến đổi thế giới thực thành những mô tả hình tượng một cách chính xác và đầy đủ. Hơn nữa việc biểu diễn và xử lý dưới dạng các ký hiệu có thể mất nhiều thời gian để có được kết quả và thường là được đưa ra quá muộn, không còn có ích nữa.

Những kiến trúc có tính phản ứng (reactive) thực thi quá trình đưa ra quyết định khi ánh xạ trực tiếp tình huống sang hành động và được dựa trên một cơ chế kích thích - phản ứng được tạo ra bởi dữ liệu của thiết bị cảm biến. Không giống như những kiến trúc dựa trên logic, chúng không có bất kỳ mô hình biểu diễn tri thức và vì thế, không tận dụng được các kiểu lập luận phức tạp nào. Kiến trúc có tính phản ứng nổi tiếng nhất là kiến trúc gộp của Brooks (Brooks, 1991). Những ý tưởng chính mà dựa trên đó Brooks đã tìm ra kiến trúc này là:

- Một cách ứng xử thông minh có thể được tạo ra mà không cần biểu diễn rõ ràng và lập luận được cung cấp bởi các kỹ thuật của trí tuệ nhân tạo.
- Thông minh là một tính chất riêng biệt của những hệ thống phức tạp.

Kiến trúc gộp xác định các tầng của các máy hữu hạn trạng thái – các máy được kết nối với thiết bị cảm biến – các thiết bị truyền thông tin theo thời gian thực (một ví dụ của kiến trúc gộp được thể hiện trong hình 1.1). Các tầng này tạo thành sự phân cấp các hành vi của agent



Hình 1.1: Kiến trúc gộp

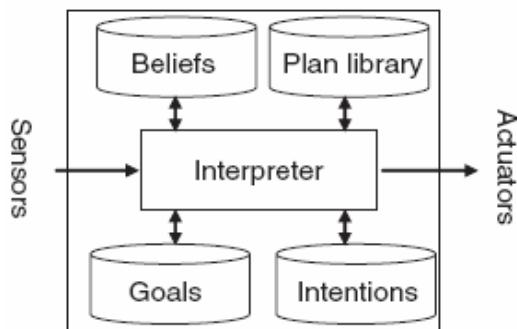
trong đó, mức độ thấp nhất được điều khiển ít hơn so với mức độ cao hơn trong ngăn xếp, vì thế, việc ra quyết định được đưa ra thông qua những hành vi hướng đích. Những agent được thiết kế gộp hiểu được điều kiện và hành động, nhưng không đưa ra được kế hoạch.

Điểm mạnh của phương pháp tiếp cận này là nó có thể thực thi tốt hơn trong những môi trường động, cũng như chúng thường được thiết kế đơn giản hơn so với những agent dựa trên logic. Tuy nhiên, nhược điểm là những agent có khả năng phản ứng không áp dụng được khi những mô hình là kết quả tác động của môi trường của chúng. Do đó, các dữ liệu của sensor có thể không đủ để xác định một hành động thích hợp và thiếu các trạng thái của agent khiến cho hầu như không thể thiết kế các agent có thể học hỏi từ kinh nghiệm. Hơn nữa, những hành động của agent dựa trên cơ sở của sự tương tác giữa các hành vi khác nhau khiến cho các kỹ sư rất khó có thể thiết kế các agent có khả năng phản ứng để thi hành những tác vụ đặc biệt khi những agent phải được nhận ra thông qua rất nhiều hành vi.

Các kiến trúc BDI (Belief, desire, intention) là những kiến trúc agent phổ biến nhất (Rao và Georgeff, 1995). Chúng có nguồn gốc triết học và dựa trên lý thuyết logic. Lý thuyết này dựa trên những quan điểm về tinh thần của niềm tin, mong muốn và dự định bằng cách sử dụng logic hình thức. Một trong những kiến trúc BDI nổi tiếng nhất là hệ thống lập luận theo thủ tục (PRS – Procedural Reasoning System) (Georgeff và Lansky, 1987). Kiến trúc này dựa trên 4 kiểu dữ liệu chính: Lòng tin (beliefs), tác vụ (desires), ý định (intentions) và kế hoạch (plans) và một bộ phận phiên dịch (xem hình 2.2). Trong hệ thống PRS, lòng tin biểu diễn những thông tin mà agent có về môi trường của nó, có thể không đầy đủ hoặc không chính xác. Tác vụ biểu diễn những tác vụ được phân công cho agent và tương ứng là những mục tiêu, hoặc là mục đích mà nó sẽ hoàn thành. Ý định thể hiện những mong muốn mà agent cần phải đạt được. Cuối cùng, kế hoạch chỉ rõ một vài quá trình của hành động mà agent sẽ phải làm để đạt được mục đích. Bốn cấu trúc dữ liệu này được quản lý bởi bộ phận phiên dịch agent chịu trách nhiệm cập nhật lòng tin từ những quan sát từ môi trường, sinh ra những tác vụ mới dựa trên cơ sở của các lòng tin mới, và lựa chọn trong tập những tác vụ hiện tại một vài tập con để hoạt động, chúng được gọi là ý định. Cuối

cùng, bộ phận phiên dịch phải lựa chọn một hành động để thực thi dựa trên cơ sở của những ý định hiện tại của agent và tri thức về mặt thủ tục.

Kiến trúc phân tầng (layered architecture) cho phép hành vi của agent vừa mang tính phản xạ vừa có tính kế hoạch. Để có được sự linh hoạt này, các hệ thống con được sắp xếp thành các tầng của một hệ thống phân cấp nhằm thích ứng với cả hai loại hành vi của agent. Có hai loại luồng điều khiển trong một kiến trúc phân lớp: phân lớp ngang (Ferguson, 1991) và phân lớp dọc (Muller et al, 1995). Trong phân lớp nằm ngang, các lớp kết nối một cách trực tiếp với đầu vào của sensor và đầu ra của hành động (xem hình 2.3); về cơ bản là có mỗi tầng hoạt động giống như một agent. Điểm mạnh chính của cách phân lớp này là sự dễ dàng trong thiết kế bởi vì nếu agent cần n loại hành vi khác nhau, thì kiến trúc chỉ yêu cầu n tầng. Tuy nhiên, bởi vì mỗi tầng đều bị ảnh hưởng bởi agent, nên không cần có một chức năng trung gian hòa giải để kiểm soát các hành động. Sự phức tạp khác là một lượng lớn các tương tác có thể xảy ra giữa những tầng ngang - m^n (với m là số lượng hành động tại mỗi tầng).



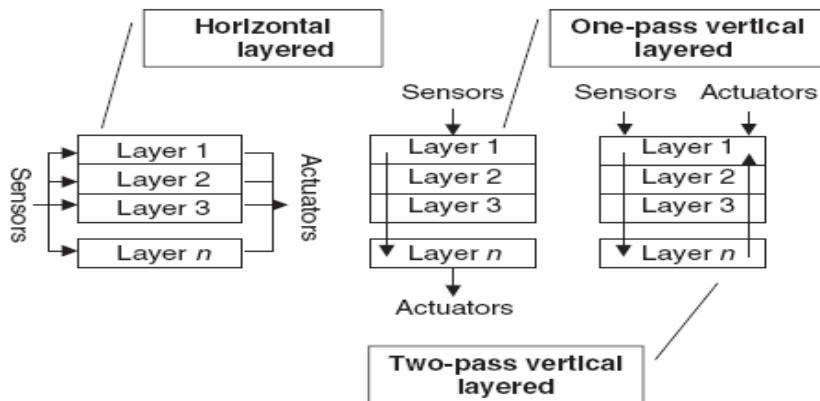
Hình 1.2: Kiến trúc PRS

Một kiến trúc phân lớp loại trừ một số vấn đề trên vì đầu vào của sensor và đầu ra của hành động được giải quyết phần lớn tại mỗi tầng. Kiến trúc phân lớp dọc có thể được chia nhỏ thành những kiến trúc điều khiển một chiều và hai chiều. Trong kiến trúc một chiều, luồng điều khiển đi từ tầng đầu, tầng nhận dữ liệu từ các sensor, xuống đến tầng cuối, tầng sinh ra đầu ra của hành động (xem Hình 1.2). Trong kiến trúc hai chiều, luồng dữ liệu đi lên xuyên qua các tầng và điều khiển, tiếp đó lại có luồng dữ liệu trở về theo thứ tự ngược lại (xem hình 1.3). Điểm mạnh chủ yếu của kiến trúc phân lớp dọc là sự tương tác giữa các tầng được làm giảm đáng kể còn $m^2(n-1)$. Nhược điểm là kiến trúc này phụ thuộc vào tất cả các tầng và không chấp nhận lỗi, vì thế nếu một tầng lỗi, toàn bộ hệ thống sẽ lỗi.

1.1.3 Giao tiếp và phối hợp

Một trong những thành phần chính của những hệ thống đa agent là giao tiếp. Trong thực tế, các agent cần có khả năng giao tiếp với người dùng, với tài nguyên hệ thống, và với agent khác nếu chúng cần hợp tác, cộng tác, đàm phán... Cụ thể, các agent tương tác với agent khác bằng cách sử dụng một vài ngôn ngữ giao tiếp đặc biệt, được gọi là những *ngôn ngữ giao tiếp agent*, dựa trên

lý thuyết lời nói hành động (Searle, 1969) và đem lại sự phân biệt giữa hành động giao tiếp và ngôn ngữ nội dung.



Hình 1.3: Luồng dữ liệu và luồng điều khiển trong kiến trúc phân lớp

Ngôn ngữ giao tiếp agent đầu tiên là KQML (Mayfield et al, 1996). KQML được phát triển vào đầu những năm 1990 là một phần của dự án ARPA của chính phủ Mỹ. Nó là một ngôn ngữ và giao thức để trao đổi thông tin và tri thức, xác định nhiều động từ biểu hiện và cho phép nội dung thông điệp được thể hiện trong một ngôn ngữ giống logic đầu tiên được gọi là KIF (Genesereth và Ketchpel, 1994). Hiện nay, ngôn ngữ giao tiếp agent được nghiên cứu và sử dụng nhiều nhất là FIPA ACL (xem phần 1.2.2), nó kết hợp nhiều khía cạnh của KQML. Đặc điểm chính của FIPA ACL là khả năng sử dụng những ngôn ngữ nội dung khác nhau và sự quản lý các cuộc hội thoại thông qua các giao thức tương tác được xác định trước.

Phối hợp là một tiến trình mà trong đó, các agent tham gia nhằm đảm bảo rằng một cộng đồng các agent đơn lẻ hành động một cách chật chẽ (Nwana et al, 1996). Có khá nhiều lý do lý giải tại sao nhiều agent cần phối hợp với nhau:

- (1) Các mục đích của các agent có thể gây ra sự xung đột giữa các hành động của agent
- (2) Các mục đích của các agent có thể phụ thuộc lẫn nhau
- (3) Các agent có thể có những khả năng và tri thức khác nhau
- (4) Các mục đích của các agent có thể nhanh chóng đạt được nếu có sự cộng tác giữa các agent khác nhau.

Sự phối hợp giữa các agent có thể được điều khiển với nhiều phương pháp tiếp cận khác nhau bao gồm cơ cấu tổ chức (Organizational structuring), lập hợp đồng (contracting), lập kế hoạch và đàm phán.

Cơ cấu tổ chức cung cấp một nền tảng để hoạt động và tương tác thông qua việc định nghĩa các vai trò, đường truyền thông và các mối quan hệ về quyền hạn (Dufee, 1999). Cách đơn giản nhất để đảm bảo hành vi rõ ràng và giải quyết xung đột là kết hợp một nhóm với một agent có một quan điểm rộng về hệ thống, qua đó tạo thành một cơ cấu tổ chức hoặc cấu trúc phân cấp. Đây là kỹ thuật phối hợp đơn giản nhất và tạo ra một kiến trúc chủ/tớ (master/slave) hoặc

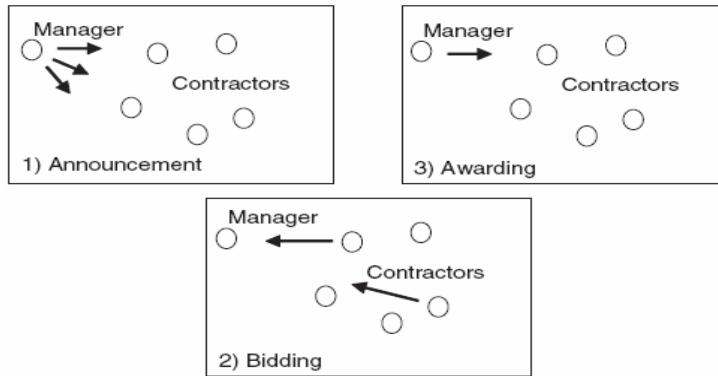
client/server cổ điển dùng cho việc phân bổ nhiệm vụ và phân bổ tài nguyên giữa các agent tách biệt. Bộ điều khiển của agent chủ có thể thu thập thông tin từ các agent trong nhóm, lập kế hoạch và giao các công việc cho từng agent để đảm bảo tính chặt chẽ trong toàn bộ hệ thống. Tuy nhiên, đây là một cách tiếp cận không thực tế trong các ứng dụng bởi vì rất khó để tạo ra một bộ điều khiển trung tâm, và trong bất kỳ trường hợp nào, bộ điều khiển trung tâm, như trong kỹ thuật chủ/tớ, là ngược lại với bản chất phân tán của những hệ thống đa agent.

Một kỹ thuật phối hợp quan trọng dùng cho việc phân bổ nhiệm vụ và phân bổ tài nguyên giữa các agent và xác định cơ cấu tổ chức là *giao thức mạng hợp đồng* (contract net protocol) (Smith and Davis, 1980). Phương pháp tiếp cận này dựa trên một cơ cấu thị trường phân quyền, mà trong đó, các agent có thể đảm nhiệm 2 vai trò, quản lý và đấu thầu. Những tiền đề cơ bản của thê thức phối hợp này là nếu một agent không thể giải quyết một vấn đề được giao khi chỉ sử dụng nguồn lực hoặc chuyên môn của mình, nó sẽ phân rã vấn đề thành các vấn đề con và cố gắng tìm các agent sẵn sàng khác với nguồn lực/chuyên môn cần thiết để giải quyết những vấn đề con này. Nài toán giao vấn đề con được giải quyết bằng cơ chế lập hợp đồng bao gồm:

- (1) Agent quản lý thông báo hợp đồng
- (2) Giao hồ sơ dự thầu cho agent lập hợp đồng để đáp ứng lại thông báo
- (3) Nhà thầu đánh giá các chào giá được gửi đến, trao một hợp đồng phụ cho nhà thầu (một hoặc nhiều) với giá thích hợp nhất (xem Hình 2.4).

Một phương pháp tiếp cận khác là coi vấn đề phối hợp các agent là vấn đề lập kế hoạch. Để ngăn chặn những hành động hoặc tương tác xung đột hay không phù hợp, các agent có thể xây dựng một kế hoạch chi tiết hóa toàn bộ những hành động và tương tác trong tương lai để đạt được mục đích và bổ sung thêm các kế hoạch hoặc lập lại kế hoạch. Lập kế hoạch đa agent có thể tập trung, hoặc là phân tán. Trong lập kế hoạch đa agent tập trung, thường có một *agent tổ chức*, nhận được một phần hoặc tất cả các kế hoạch từ agent đơn lẻ, phân tích chúng để xác định những tương tác không phù hợp hoặc xung đột có thể xảy ra (như xung đột giữa các agent vượt quá giới hạn tài nguyên). Tiếp theo agent tổ chức có gắng chỉnh sửa các kế hoạch thành phần và kết hợp chúng thành một kế hoạch đa agent đã loại bỏ những tương tác xung đột (Georgeff, 1983). Trong lập kế hoạch đa agent phân tán, ý tưởng là cung cấp cho mỗi agent một mô hình của kế hoạch của các agent khác. Các agent giao tiếp để xây dựng và cập nhật vào kế hoạch cá nhân và cập nhật mô hình của mình trong các agent khác cho đến khi tất cả những xung đột được loại bỏ (Georgeff, 1984).

Lập kế hoạch bộ phận - toàn cục tích hợp những thế mạnh của các cách tiếp cận về mặt tổ chức, quy hoạch, và lập hợp đồng bằng cách thống nhất chúng thành một phương pháp duy nhất (Durfee and Victor, 1987). Mục đích của phương pháp tiếp cận này là để đạt được những lợi ích của việc lập kế hoạch đa agent về sự phối hợp được chi tiết hóa, trong tình hình cụ thể trong khi tránh tính toán quá nhiều và chi phí giao tiếp lớn.



Hình 1.4: Các pha của giao thức mạng hợp đồng

Điều này có thể xảy ra bởi vì những cơ cấu tổ chức được coi là có hiệu quả là những cơ cấu giám bớt được không gian mà kế hoạch khả thi chiếm giữ. Ngoài ra, kế hoạch bộ phận - toàn cục coi hợp đồng là kế hoạch phối hợp tổ chức với chỉ rõ những trao đổi về nhiệm vụ và kết quả trong tương tác giữa các agent. Vì thế, trong kế hoạch bộ phận - toàn cục, sự phối hợp bao gồm cả việc chia sẻ công việc và chia sẻ kết quả; cả hai đều tuân theo những vai trò dài hạn về mặt tổ chức và lập kế hoạch phản ứng lại để đạt được các mục tiêu ngắn hạn.

Đàm phán có thể là một kỹ thuật đáng tin cậy nhất để phối hợp các agent. Cụ thể, đàm phán là quá trình giao tiếp của một nhóm các agent để đạt được một thỏa thuận chấp nhận lẫn nhau về một vấn đề nào đó (Bussmann and Muller, 1992). Đàm phán có thể mang tính cạnh tranh hoặc hợp tác tuỳ thuộc vào hành vi của các agent có liên quan. Đàm phán cạnh tranh được dùng trong trường hợp khi có sự tương tác giữa các agent có những mục đích độc lập với nhau; chúng không phải là một sự hợp tác có ưu tiên, chia sẻ thông tin hay sẵn sàng giảm yêu cầu để có lợi ích lớn hơn. Đàm phán hợp tác được dùng trong trường hợp các agent cần đạt được một mục đích chung hoặc thực thi một tác vụ đơn. Trong trường hợp này, hệ thống đa agent được thiết kế tập trung để theo đuổi một mục đích thống nhất cho toàn hệ thống.

1.1.4 Ngôn ngữ lập trình và công cụ

Ngôn ngữ lập trình, nền tảng và các công cụ phát triển của hệ thống đa agent là thành phần quan trọng mà có ảnh hưởng đến việc ứng dụng rộng rãi các công nghệ agent. Trong thực tế, sự thành công của hệ thống đa agent phần lớn là phụ thuộc vào sự sẵn có của công nghệ (tức là ngôn ngữ lập trình, thư viện phần mềm và các công cụ phát triển) để cho phép thực thi các khái niệm và các kỹ thuật đã hình thành cơ sở cho hệ thống đa agent.

Hệ thống agent có thể được cài đặt bằng cách sử dụng một loại ngôn ngữ lập trình nào đó. Cụ thể, ngôn ngữ hướng đối tượng được coi là một phương tiện phù hợp, vì khái niệm về agent không khác nhiều so với từ khái niệm đối tượng. Trong thực tế, các agent chia sẻ nhiều tính chất với các đối tượng như đóng gói (encapsulation), và đôi khi có cả kế thừa (inheritance) và truyền thông điệp (message passing). Tuy nhiên, các agent cũng khác với các đối tượng ở một số điểm chính: tính tự chủ (autonomous) (nghĩa là chúng có thể tự quyết thực hiện hay không thực hiện

một hành động theo yêu cầu từ các agent khác); chúng có thể có hành vi linh hoạt; và mỗi agent của một hệ thống có thể điều khiển luồng của riêng mình.

Ngôn ngữ lập trình hướng agent là một loại ngôn ngữ lập trình mới. Nó tập trung vào những đặc điểm chính của hệ thống đa agent. Tối thiểu, một ngôn ngữ lập trình hướng agent phải bao gồm một vài cấu trúc tương ứng với một agent, nhưng nhiều ngôn ngữ lập trình cũng cung cấp các cơ chế để hỗ trợ các thuộc tính bổ sung của agent như niềm tin (beliefs), mục đích (goals), kế hoạch (plans), vai trò (roles) và quy tắc (norms).

Ngày nay, một số ngôn ngữ hướng agent đã xuất hiện (Bordini et al, 2006). Một số được thiết kế từ đầu, trực tiếp mã hóa một số lý thuyết về agent, trong khi một số khác mở rộng ngôn ngữ đã có để phù hợp với tính chất riêng biệt của agent. Ngoài ra, một số ngôn ngữ mang quan điểm lập trình hoàn toàn có tính chất khai báo hoặc có tính chất bắt buộc. Ví dụ điển hình là FLUX (Thielscher, 2005) và ngôn ngữ Jack Agent (Winikoff, 2005).

Nền tảng là phương tiện chính cho phép phát triển các hệ thống đa agent. Hầu hết chúng cung cấp một phương tiện để triển khai nhiều hệ thống agent trên các phần cứng và hệ điều hành khác nhau, thường là cung cấp một chương trình trung gian (middleware) để hỗ trợ thực thi và các hoạt động cần thiết của chúng như giao tiếp (communication) và phối hợp (coordination). Một số nền tảng có mục đích chung là cung cấp các chức năng theo các chuẩn FIPA để hỗ trợ cộng tác giữa nhiều hệ thống agent khác nhau. Ngoài ra, một số nền tảng cũng có mục tiêu hỗ trợ các loại phần cứng, mạng truyền thông và kiến trúc agent, ví dụ như JADE (Bellifemine et al, 2001) và một số hỗ trợ các loại agent đặc biệt, ví dụ như các agent điện thoại di động (Lange và Oshima, 1998).

Một đặc điểm quan trọng mà các hệ thống đa agent nên cung cấp là khả năng hỗ trợ sự tương tác giữa các hệ thống phần mềm kế thừa từ các hệ thống trước. Do đó, sự sẵn sàng tích hợp các công cụ phần mềm với các công nghệ khác có thể là chìa khóa dẫn đến thành công của chúng. Internet là một trong các lĩnh vực ứng dụng quan trọng nhất và là phương tiện truyền thông quan trọng nhất mà nhiều hệ thống đa agent có thể sử dụng để cung cấp khả năng tương tác giữa các hệ thống phần mềm kế thừa. Do vậy, rất nhiều công trình nghiên cứu và phát triển hiện nay hướng đến việc cung cấp các kỹ thuật và công cụ phần mềm thích hợp cho việc tích hợp các hệ thống đa agent với các công nghệ web như Web Service và Semantic Web.

Dịch vụ web (Web Service) là một công nghệ làm thay đổi Internet một cách nhanh chóng thông qua việc cung cấp một mô hình lập trình trung tính với ngôn ngữ và môi trường (language-neutral, environment-neutral) nhằm sử dụng Web để tích hợp các ứng dụng cả trong và ngoài doanh nghiệp (Tsalgatidou và Pilioura, 2002; Weikum, 2001). Nhiều công trình quan trọng khác đã đề xuất việc tích hợp giữa các công nghệ agent và các công nghệ Web Service thành một phương tiện lý tưởng hỗ trợ khả năng tương tác của cả hai và để cung cấp nhiều dịch vụ phức tạp hơn. Đặc biệt, các agent đã được chứng minh rất hữu ích khi hành động trực tiếp như

là một Web Service, cung cấp các dịch vụ dựa trên agent để cho các Web Service sử dụng, và phối hợp một cách chủ động việc thực thi của một tập các Web Service bằng cách cung cấp các dịch vụ mới là gộp của các dịch vụ khác (Buhler và Vidal, 2005; Greenwood et al, 2005.; Negri et al, 2006).

Web ngữ nghĩa (Semantic Web) là một phần mở rộng của Web hiện tại, trong đó thông tin bao gồm ngữ nghĩa rõ ràng để cho phép phối hợp giữa các máy tính cũng như với con người tốt hơn. Đặc biệt, Semantic Web cung cấp một cơ sở hạ tầng và một tập các công nghệ cho phép không chỉ các trang Web, mà còn cơ sở dữ liệu, dịch vụ, chương trình, cảm biến, thiết bị cá nhân...vừa lấy dữ liệu vừa tạo ra dữ liệu trên Web (Hendler et al, 2002).

1.1.5 Ứng dụng của các hệ thống đa agent

Các hệ thống đa agent đang ngày càng được sử dụng rộng rãi trong rất nhiều ứng dụng, từ các hệ thống tương đối nhỏ để hỗ trợ cá nhân đến các hệ thống mở, phức tạp, và đặc biệt quan trọng dành cho các ứng dụng công nghiệp.

Ứng dụng công nghiệp là những ứng dụng rất quan trọng cho các hệ thống đa agent bởi vì chúng là nơi mà công nghệ đa agent đầu tiên được thử nghiệm và chứng minh tiềm năng ban đầu của nó. Các ví dụ về việc áp dụng các hệ thống đa agent trong lĩnh vực công nghiệp bao gồm kiểm soát tiến trình (Jenning, 1994), chẩn đoán hệ thống (Albert, 2003), sản xuất (Parunak, 1987), dịch vụ vận tải (Neagu, 2006), và quản lý mạng (Greenwood, 2006).

Một trong những lĩnh vực ứng dụng quan trọng nữa của hệ thống đa agent là quản lý thông tin. Thật ra, Internet đã chứng tỏ là một miền lý tưởng cho các hệ thống đa agent nhờ bản chất phân tán tự nhiên của nó và khối lượng thông tin sẵn có đang ngày càng tăng lên mạnh mẽ. Ví dụ, các agent có thể được sử dụng để tìm kiếm và lọc thông tin. Internet đã thúc đẩy việc sử dụng công nghệ agent trong lĩnh vực quản lý tiến trình nghiệp vụ và thương mại điện tử. Trong thực tế, trước sự gia tăng của thương mại điện tử, việc quản lý tiến trình nghiệp vụ đã gần như được điều khiển bởi các tương tác của con người: con người quyết định khi nào mua hàng hóa, họ có khả năng chi trả bao nhiêu...Giờ đây thương mại điện tử và tiến trình nghiệp vụ được tự động hóa đang ngày càng đóng vai trò then chốt trong nhiều tổ chức bởi vì chúng cung cấp những cơ hội để cải thiện đáng kể cách thức mà các thực thể khác nhau tham gia vào tương tác của tiến trình nghiệp vụ. Trong kịch bản này các hệ thống đa agent đã chứng tỏ nó vừa phù hợp cho việc mô hình hóa và thiết kế các hệ thống quản lý tiến trình nghiệp vụ, vừa là các thành phần quan trọng cho tự động hóa của một số hoặc tất cả các bước của các tiến trình này.

Giao thông và vận tải cũng là một lĩnh vực quan trọng, nơi mà bản chất phân tán của các tiến trình giao thông và vận tải và sự độc lập mạnh mẽ giữa các thực thể có liên quan trong các tiến trình đó làm cho các hệ thống đa agent trở thành một công cụ có giá trị cho việc thực hiện các giải pháp thương mại thực sự có hiệu quả (Neagu et al, 2006). Một số lĩnh vực đã được giải quyết như OASIS cung cấp bằng chứng mạnh mẽ rằng nhiều hệ thống đa agent là phương tiện lý

tưởng cho các hệ thống mở, phức tạp và đặc biệt quan trọng. OASIS là một hệ thống điều khiển không lưu phức tạp dựa trên mô hình agent BDI, được triển khai và được sử dụng thành công tại sân bay Sydney ở Australia.

Các hệ thống viễn thông là một lĩnh vực ứng dụng đã sử dụng thành công các hệ đa agent. Trong thực tế, các hệ thống viễn thông là các mạng lưới lớn và phân tán gồm các thành phần được kết nối với nhau. Những thành phần đó cần phải được theo dõi và quản lý trong thời gian thực. Các hệ thống viễn thông cũng hình thành nên cơ sở của một thị trường cạnh tranh, nơi các công ty viễn thông và các nhà cung cấp dịch vụ nhắm đến để phân biệt chính họ với đối thủ cạnh tranh của họ bằng cách cung cấp các dịch vụ tốt hơn, nhanh hơn hoặc đáng tin cậy hơn. Vì vậy, các hệ đa agent được sử dụng cả trong việc quản lý các mạng lưới phân tán lẫn cho việc cài đặt các dịch vụ viễn thông tiên tiến (Fricke et al., 2001; Hayzelden and Bourne, 2001; Greenwood et al., 2006).

Nhiều hệ thống đa robot cũng sử dụng các kỹ thuật lập kế hoạch phân tán và đa agent để phối hợp các robot khác nhau. FIRE phối hợp các hành động của nhiều robot ở nhiều mức trừu tượng hóa. Tầng lập kế hoạch trên cùng sử dụng một chiến lược dựa trên thị trường để phân phối các công việc giữa các robot, trong đó thời gian robot di chuyển là thước đo chính để tính chi phí. MISUS phối hợp các kỹ thuật từ việc lập kế hoạch và lập lịch với học máy để thực hiện việc thăm dò có tính khoa học một cách tự chủ (Estlin et al., 2005). Kỹ thuật lập kế hoạch và lập lịch phân tán được sử dụng để tạo ra các kế hoạch cộng tác nhiều lần, việc thực thi kế hoạch giám sát có hiệu quả và thực hiện làm lại kế hoạch khi cần thiết. Các thành phần thuộc về học máy được sử dụng để suy diễn các quan hệ giữa những dữ liệu thu thập được và lựa chọn các hoạt động khoa học mới. Hơn nữa, những hệ thống này có thể suy luận về những mục đích phụ thuộc lẫn nhau để thực hiện tối ưu hóa kế hoạch và để tăng giá trị của dữ liệu thu thập được.

Một số ứng dụng đa agent đáng quan tâm khác có thể được tìm thấy trong hệ thống chăm sóc sức khỏe (Moreno và Nealon, 2003). Trong thực tế, các hệ thống đa agent đã được đề xuất để giải quyết nhiều loại vấn đề khác nhau trong lĩnh vực chăm sóc sức khỏe, bao gồm lập kế hoạch và quản lý bệnh nhân, chăm sóc sức khỏe người cao tuổi và cộng đồng, truy cập và quản lý thông tin y tế và hỗ trợ quyết định. Một vài ứng dụng đã cài đặt cho thấy rằng hệ thống đa agent có thể là giải pháp đúng đắn cho việc xây dựng các hệ thống hỗ trợ quyết định y học (Hudson và Cohen, 2002) và cải thiện sự phối hợp giữa các chuyên gia khác nhau tham gia vào quá trình chăm sóc sức khỏe (Lanzola và Boley, 2002).

1.2 CƠ SỞ HÌNH THÀNH AGENT VẬT LÝ VÀ AGENT THÔNG MINH (FIPA)

Tập các đặc tả hoàn chỉnh của FIPA có sẵn trên website của FIPA (FIPA). Phần này sẽ cung cấp lịch sử và phạm vi của FIPA và nêu ra một vài đặc tả cụ thể có liên quan đến JADE. Vì JADE là một nền tảng cài đặt phần lớn các đặc tả FIPA nên nó phụ thuộc rất nhiều vào các ý tưởng phát sinh trong quá trình đặc tả và được trình bày trong các tài liệu FIPA. JADE mở rộng mô hình

FIPA trong một số lĩnh vực, nên các đặc tả sẽ không thể bao phủ được toàn bộ. Tuy nhiên, các khía cạnh có liên quan đến khả năng tương tác, mục tiêu cốt lõi của FIPA, JADE đều tuân thủ.

1.2.1 Lịch sử và mục đích của FIPA

FIPA được thành lập năm 1996 với tư cách là một tổ chức quốc tế phi lợi nhuận nhằm phát triển một tập các chuẩn liên quan đến công nghệ agent. Các thành viên ban đầu gồm một nhóm các tổ chức nghiên cứu và công nghiệp đã xây dựng một bộ luật hướng dẫn xây dựng các đặc tả chuẩn, hợp pháp cho các công nghệ agent. Vào lúc đó, các phần mềm agent đã được biết đến rộng rãi trong cộng đồng nghiên cứu nhưng chỉ thu hút được sự quan tâm hạn chế từ các tổ chức doanh nghiệp. FIPA đã quyết định tạo ra các chuẩn nhằm làm nền tảng cho một ngành công nghiệp mới bằng cách xem xét rất nhiều ứng dụng. Cốt lõi của FIPA tuân theo các nguyên lý sau:

- (1) Các công nghệ agent cung cấp một cách tiếp cận mới để giải quyết các vấn đề cũ và mới.
- (2) Một số công nghệ agent đã đạt tới trình độ tăng trưởng đáng kể.
- (3) Để sử dụng được một số công nghệ agent đòi hỏi phải chuẩn hóa.
- (4) Việc chuẩn hóa các công nghệ chung phải được công nhận là khả thi và đem lại hiệu quả sử dụng bởi các hội đồng chuẩn hóa khác.
- (5) Việc chuẩn hóa các cơ chế bên trong của agent không phải là vấn đề trọng tâm, mà đúng hơn là cơ sở hạ tầng và ngôn ngữ cần thiết cho sự tương tác mở.

FIPA ban đầu được thành lập với kì hạn 5 năm để đặc tả các khía cạnh chọn lọc của các hệ đa agent, kì hạn này được mở rộng vào năm 2001. Đến cuối năm 2005, FIPA được quản lý bởi một Hội đồng lãnh đạo gồm các thành viên được bầu, hội đồng này chịu trách nhiệm đưa ra các chỉ dẫn có tính chiến lược và quản lý các công việc mang tính hình thức. Các quyết định liên quan đến việc thành lập các nhóm kỹ thuật để xây dựng các đặc tả và dự đoán vòng đời của các đặc tả đang phát triển được quản lý bởi Hội đồng Kiến trúc FIPA (FAB). Các thành viên của FAB được bổ nhiệm bởi Hội đồng lãnh đạo. Các công việc kỹ thuật hướng đến việc xây dựng các đặc tả được xây dựng trong Cộng đồng Kỹ thuật (TCs), được thành lập khi có một đề xuất công việc mới được chấp nhận và giải tán khi công việc được hoàn thành hoặc bị hủy bỏ. Ngoài ra, các Nhóm làm việc (WGs) được thành lập như một diễn đàn để thảo luận các vấn đề kỹ thuật và hình thành cơ sở cần thiết trước khi thành lập. Cuối cùng, các Nhóm quan tâm đặc biệt (SIGs) được thành lập từng thời kỳ để thảo luận các công việc liên quan đến FIPA mà không được dự định để tạo ra các đặc tả kỹ thuật.

Vào thời kì đỉnh cao, FIPA có tới hơn 60 thành viên từ hơn 20 quốc gia khác nhau trên khắp thế giới, và qua nhiều lần lặp lại trong nhiều năm đã cho ra đời một tập các đặc tả thông qua ba vòng rà soát: FIPA'97, FIPA'98 và FIPA2000. Kèm theo lần lặp cuối cùng là Kiến trúc trùu tượng FIPA (FIPA Abstract Architecture). Kiến trúc này trùu tượng hóa việc cài đặt các nguyên lý của “thuyết không thể biết” (agnostic) được trình bày trong đặc tả FIPA2000 nhằm tạo ra một

đặc tả định nghĩa mọi thành phần kiến trúc lõi và quan hệ giữa chúng. Điều này được thảo luận trong phần 2.2.3. Lịch sử vắn tắt của FIPA qua các thời kỳ:

1996: FIPA đưa ra đề xuất đầu tiên nhằm kiểm các lĩnh vực ứng dụng khác nhau được hội đồng quan tâm và từ đó hình thành nền tảng của tập đặc tả FIPA'97. Trong số 12 phản hồi nhận được, có 4 phản hồi được hoàn toàn nhất trí lựa chọn: trợ lý cá nhân, trợ lý việc di chuyển của cá nhân, quảng bá giải trí nghe-nhin, cuối cùng là quản lý và dự trữ mạng. Tập các công nghệ agent cần thiết để xây dựng ứng dụng trên cũng đã được xác định.

1997: FIPA'97, tập đặc tả đầu tiên, được xác định với 7 phần. 3 phần đầu là các công nghệ lõi, chuẩn và cụ thể về quản lý agent, giao tiếp agent, và tương tác agent-phần mềm. 4 phần còn lại là các ứng dụng thông tin bao gồm trợ lý cá nhân, trợ lý cá nhân di chuyển, quảng bá giải trí nghe-nhin, cuối cùng là quản lý và dự trữ mạng.

Từ quan điểm truyền thông, FIPA đã quyết định chấp nhận ARCOL (Sadek, 1991) từ France Telecom làm nền tảng cho ngôn ngữ truyền thông agent, ngay sau đó ARCOL được biết đến với cái tên FIPA-ACL, hay gọi tắt là ACL. Quyết định chấp nhận ARCOL xuất phát từ một cuộc tranh luận gay gắt về chất lượng giữa ARCOL và KQML (Labrou et al., 1999). Thật không may, ARCOL đã giành chiến thắng vì nó được cung cấp bởi những ngữ nghĩa mang tính hình thức (formal semantic). Tiếp đó, FIPA cũng quyết định chấp nhận ngôn ngữ SL làm tiêu chuẩn để biểu diễn nội dung các thông điệp và một vài giao thức cộng tác, cũng được cung cấp bởi France Telecom.

Về khía cạnh quản lý agent, FIPA'97 đã định nghĩa FIPA Agent Management Ontology đầu tiên bao gồm khái niệm nền tảng agent (agent platform) bao gồm Kênh truyền thông agent (ACC: Agent Communication Channel), Hệ thống quản lý agent (AMS: Agent Management System) và Tiện ích thư mục (DF: Directory Facilitator). IIOP (1999) được chọn là giao thức cơ sở.

9/1998: Các đặc tả cốt lõi được duyệt lại và bổ sung thêm vào cơ chế quản lý agent cơ sở khả năng quản lý di động, và bổ sung vào cơ chế truyền thông các giao thức tương tác và tương tác giữa con người với agent. Các cơ chế mới này được kiểm soát trên hệ quản lý bảo mật agent (agent security management) và dịch vụ từ vựng (ontology service) nhằm tiếp nhận và phục vụ từ vựng trong miền xác định.

Các kế hoạch cuối năm 1998 đã được tạo ra nhằm kiểm thử khả năng kết nối các phần mềm agent tuân theo FIPA cho giai đoạn đầu năm 1999. Ngoài JADE còn có một số platform đã được giới thiệu như FIPA-OS (Buckle et al., 2002) và Contec Agent Platform (Suguri, 1998).

Đầu năm 1999, bộ đặc tả FIPA'98 bị trì hoãn đã được đưa ra với nhiều cải tiến, và phân loại nhiều hơn bộ FIPA'97. Hai TC mới được thiết lập, một cho phát triển kiến trúc trừu tượng FIPA (FIPA Abstract Architecture) và một để phát triển đặc tả cho các trợ giúp ứng dụng thường xuyên di chuyển.

2/2002: FIPA đưa ra một tuyên bố mới: “Đây mạnh các công nghệ và các đặc tả về khả năng liên kết tạo điều kiện thuận tiện cho sự liên kết các hệ thống agent thông minh trong lĩnh vực thương mại và công nghiệp”. Điều này được thực hiện bằng cách tập trung đổi mới các truyền thông, liên kết dữ liệu trên ngữ nghĩa mức cao giữa các agent, các thỏa thuận và tương tác giữa các agent.

Quá trình phát triển ban đầu của kiến trúc trừu tượng FIPA đã bị dừng lại khi mô hình kiến trúc tổng thể FIPA mới xuất hiện. Kiến trúc đó có các mức trừu tượng được định nghĩa tốt mà không bị phá vỡ khi công nghệ thay đổi. Nó có sự ảnh xạ tới các công nghệ thường được sử dụng (như CORBA, JINI) và hỗ trợ các cơ chế khác như: vận chuyển thông điệp, mã hóa nội dung và định nghĩa rõ ràng về các thuật ngữ agent.

Mô hình vòng đời mới dành cho các chuẩn cũng đã được chấp nhận bao gồm 3 pha chính: khởi tạo, thử nghiệm và chuẩn hóa. Giai đoạn thử nghiệm nghĩa là bản đặc tả phải trải qua quá trình kiểm chứng dựa trên cài đặt.

Hai TC mới được thiết lập. TC đầu tiên dùng để giải quyết vấn đề gateway, hỗ trợ cho thiết bị di động và vấn đề tập hợp các giao thức tương tác tạo thành một định dạng thư viện mới. TC thứ 2 dùng phát triển 1 framework ngữ nghĩa để giải quyết mối liên hệ giữa các tín hiệu bên ngoài và trạng thái bên trong và liệt kê các hành động giao tiếp (communicative act), giao thức tương tác, hợp đồng, chính sách, mô hình phục vụ và từ vựng.

Trong suốt năm 2000 và đầu năm 2001, một số đặc tả được xúc tiến chuyển sang trạng thái thực nghiệm và được xuất bản trong tập các đặc tả FIPA2000. Việc cài đặt platform dựa trên những tài liệu này trải qua các cuộc thử nghiệm về khả năng tương tác (hay còn gọi là “bake-off”) như: JADE, FIPA-OS, Zeus.

Năm 2002, một TC đặc biệt, được gọi là “X2S”, được ủy nhiệm để phối hợp tất cả các đặc tả thực nghiệm đã có để đẩy mạnh thành một chuẩn. Cuối 2002, 25 đặc tả của FIPA cuối cùng đã được thúc đẩy sang trạng thái chuẩn, chiếm 56% của toàn bộ bản đặc tả.

4/2004: Sự hỗ trợ về công nghiệp cho việc chuẩn hóa các đặc tả cốt lõi bị giảm dần, nên FIPA đã tập trung vào mảng giao tiếp ad hoc, ngữ nghĩa, bảo mật, dịch vụ, mô hình hóa và các phương pháp luận. Kết quả của những nỗ lực này là xây dựng được AUML (Agent Unified Modelling Language) và phát triển được một thư viện phân đoạn và tương ứng là cơ sở phương pháp (method base).

Cuối 2004, tổ chức FIPA bị gián đoạn vì thiếu kinh phí hỗ trợ.

Từ 2005 tới nay: Giữa 2005, FIPA được sát nhập với chuẩn IEEE, và gọi là FIPA-IEEE. Nhiều nhóm làm việc được thiết lập để tập trung nghiên cứu các lĩnh vực về khả năng liên kết agent và Web service, giao tiếp agent với con người, agent di động và agent thường xuyên di chuyển trong mạng ngang hàng.

Một số thành tựu của FIPA:

- Tập chuẩn đặc tả hỗ trợ quá trình giao tiếp giữa các agent và các dịch vụ chính ở tầng trung gian
- Một kiến trúc trừu tượng cung cấp cái nhìn hoàn thiện thông qua các chuẩn FIPA 2000.
- Một ngôn ngữ giao tiếp agent rõ ràng và được sử dụng nhiều (FIPA-ACL), kèm theo một tập các ngôn ngữ nội dung (ví dụ như FIPA-SL) và một tập các giao thức chính áp dụng từ quá trình trao đổi thông điệp đơn giản cho đến các quá trình giao dịch phức tạp
- Một số công cụ agent thương mại và nguồn mở, như JADE, ngày nay đã được coi là công nghệ mã nguồn mở tuân theo FIPA được sử dụng rộng rãi.
- Ngoài FIPA, có nhiều dự án đã hoàn thành như dự án Agentcities đã tạo một mạng lưới các platform tuân theo FIPA và các dịch vụ ứng dụng agent
- Một mở rộng của UML chuyên về agent là AUML đã được đề xuất

1.2.2 Các khái niệm cốt lõi của FIPA

Trong quá trình phát triển của FIPA, nhiều ý tưởng liên quan tới agent được đề xuất. Một số ý tưởng trở thành chuẩn, một số khác được phát triển nhưng chưa hoàn thành, số còn lại bị thất bại bởi nhiều nguyên nhân nào đó. Các ý tưởng này đều xoay quanh một số khía cạnh chính là giao tiếp giữa agent, quản lý agent, và kiến trúc agent. Mục này sẽ thảo luận các khái niệm chính liên quan tới các khía cạnh đó.

1.2.2.1 Giao tiếp giữa các agent

Các agent về cơ bản là một dạng của các tiến trình lập trình phân tán và vì vậy tuân theo khái niệm cổ điển của mô hình tính toán phân tán bao gồm 2 phần: thành phần (component) và kết nối (connector). Thành phần là người tiêu dùng, người sản xuất và người trung gian truyền các thông điệp được trao đổi thông qua kết nối. Các chuẩn như ISO và IETF mang cách tiếp cận hướng mạng (network-oriented) trong việc phát triển các ngăn xếp giao thức được phân lớp (layered

protocol stack) chiếm đa số trong các giao tiếp máy tính (computer communication) chúng ta biết ngày nay như Mô hình tham chiếu OSI và mô hình TCP/IP. Cả 2 đều được sử dụng thông qua các interface của các dịch vụ phần mềm cài đặt các giao thức.

Trong suốt những năm 1990, các mô hình hướng mạng được bổ sung vào các tổ chức sử dụng mô hình hướng dịch vụ như OMG, DCE, WC3, GGF và FIPA. Một mô hình hướng dịch vụ cơ bản là một ngăn xếp giao thức giao tiếp (communication protocol stack) với nhiều giao thức ứng dụng lớp con (sub-layer) thay vì giao thức ứng dụng một lớp. Mô hình FIPA sẽ được mô tả một cách ngắn gọn nhưng đầu tiên chúng ta đưa ra một số ngữ cảnh bổ sung.

FIPA – ACL dựa trên lý thuyết lời nói hành động (speech act theory). Lý thuyết này chỉ ra rằng các thông điệp biểu diễn các hành động, hoặc các hành động giao tiếp – cũng được biết đến như là các hành động lời nói (speech act) hoặc biểu hiện (performative). Một ví dụ đơn giản là câu “Tên tôi là Ngọc” khi được nói ra, nó sẽ cho bên nhận biết được một thông tin. Tập 22 biểu hiện giao tiếp hành động của FIPA-ACL dựa trên đề xuất ARCOL của France Telecom trong đó mỗi hành động được mô tả bằng cách sử dụng cả dạng tường thuật lẫn ngữ nghĩa hình thức dựa trên logic hình thức gọi là modal logic (Garson, 1984). Logic hình thức chỉ rõ tác động của việc gửi thông điệp lên thái độ về mặt tinh thần của agent gửi và agent nhận. Dạng logic này phù hợp với mô hình BDI hay còn gọi là mô hình suy diễn Belief, Desire, Intention (Rao và Georgeff 1995).

Một số hoạt động được sử dụng chung nhất là cho biết (inform), yêu cầu (request), đồng ý (agree), không hiểu (not understood) và từ chối (refuse). Chúng ghi lại những dạng thường gặp nhất trong giao tiếp cơ bản và sẽ được miêu tả trong phần 2.2.3. Nó được xác định trong các chuẩn FIPA để được tuân theo một cách đầy đủ khi agent nhận bắt kì thông điệp giao tiếp hành động FIPA – ACL nào và ít nhất là đáp ứng bằng một thông điệp not-understood nếu thông điệp nhận được không thể xử lý được.

Dựa vào các kiểu giao tiếp hành động này, FIPA định nghĩa một tập các giao thức tương tác, mỗi cái bao gồm một chuỗi các giao tiếp hành động để kết hợp các hành động đa thông điệp, như mạng hợp đồng (contract net) dành cho việc thiết lập các thỏa thuận và các kiểu đấu giá. Một lựa chọn của các giao thức này được chỉ rõ ở phần 1.2.3

Bên trong cấu trúc của mỗi thông điệp, FIPA – ACL không uỷ thác việc sử dụng một ngôn ngữ cụ thể nào để biểu diễn nội dung, mặc dù nhiều đặc tả dành cho một số kiểu biểu diễn nội dung như FIPA – SL, FIPA – KIF và FIPA – RDF đã được đề xuất.

1.2.2.2 Các lớp con của FIPA

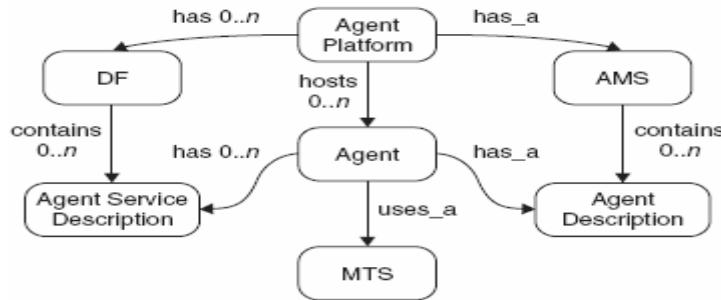
Như đã đề cập ở trước, ngăn xếp giao tiếp FIPA có thể được phân chia thành một số lớp con trong lớp ứng dụng ngăn xếp OSI hoặc TCP/IP. Chúng được trình bày chi tiết như dưới đây:

- Sub-layer 1 (*Transport*): trong mô hình giao thức phân lớp FIPA – ACL, giao thức lớp con thấp nhất là giao thức vận chuyển. FIPA định nghĩa các giao thức vận chuyển thông điệp (message transport protocol) như IIOP (IIOP, 1999), WAP (WAP) và HTTP (HTTP)
- Sub-layer 2 (*Encoding*): Ngoài việc gửi các thông điệp được mã hoá theo từng byte, FIPA còn định nghĩa một vài cách biểu diễn thông điệp sử dụng cho các cấu trúc dữ liệu ở mức cao bao gồm XML, String và Bit – Efficient. Bit – Efficient dự kiến sẽ sử dụng khi giao tiếp qua các kết nối băng thông thấp.
- Sub-layer 3 (*Messaging*): Trong FIPA, cấu trúc thông điệp được xác định độc lập với việc mã hoá để khuyến khích sự linh hoạt. Khía cạnh quan trọng ở mức này là các tham số chính cần thêm vào payload hoặc nội dung được trao đổi, ví dụ người gửi và người nhận, kiểu thông điệp, thời gian đáp ứng. Một ví dụ của cấu trúc thông điệp FIPA – ACL được đưa ra trong phần 2.2.3
- Sub-layer 4 (*Ontology*): thuật ngữ này chứa trong payload hoặc nội dung của một thông điệp FIPA có thể được tham chiếu một cách rõ ràng sang mô hình khái niệm chuyên về một ứng dụng cụ thể (application-specific) hoặc bản chất (ontology). Mặc dù về bản chất FIPA cho phép sử dụng các ontology khi biểu diễn nội dung thông điệp, nhưng nó không chỉ ra bất kì cách biểu diễn nào cho các ontology hoặc cung cấp các ontology cho một lĩnh vực cụ thể nào. Nó có thể tham chiếu đến các ontology dựa trên web nếu được yêu cầu.
- Sub-layer 5 (*Content expression*): Dữ liệu thật của các thông điệp FIPA có thể có một dạng nào đó, nhưng FIPA đã định nghĩa các hướng dẫn sử dụng các công thức và vị từ logic chung, và các phép tính đại số để kết hợp và lựa chọn các khái niệm. Ngôn ngữ thường được sử dụng nhất cho việc biểu diễn nội dung là FIPA – SL, ví dụ của các công thức logic bao gồm: *not*, *or*, *implies* (kéo theo), *equiv...* và ví dụ của các toán tử đại số như *any* và *all*.
- Sub-layer 6 (*Communicative act*): việc phân loại thông điệp đơn giản là chia chúng thành các loại: action hay performative. Ví dụ: *inform*, *request* và *agree*.
- Sub-layer 7 (*Interaction protocol or IP*): các thông điệp hiếm khi được trao đổi một cách riêng biệt mà thường được hình thành một số chuỗi tương tác. FIPA định nghĩa một số giao thức tương tác chỉ ra các chuỗi trao đổi thông điệp điển hình như *request* (được miêu tả trong phần 2.2.3), nó miêu tả một nhóm thông điệp tham gia vào việc tạo một yêu cầu tới các agent khác và phản hồi là *agree* hoặc *refuse*.

1.2.2.3 Quản lý agent

Ngoài giao tiếp, khía cạnh cơ bản thứ 2 của các hệ thống agent được đề cập trong các đặc tả FIPA ban đầu là quản lý agent: một nền tảng chuẩn trong đó các agent tuân theo FIPA có thể tồn tại, hoạt động và được quản lý. Nó thiết lập mô hình tham chiếu logic cho việc tạo, đăng ký, định vị,

giao tiếp, di trú và hoạt động của các agent. Mô hình tham chiếu quản lý agent bao gồm các thành phần được mô tả trong Hình 1.5



Hình 1.5: Minh họa mô hình tham chiếu quản lý agent

Agent Platform (AP): cung cấp cơ sở hạ tầng vật lý trong đó agent được triển khai. AP bao gồm các cơ chế, các hệ điều hành, các thành phần FIPA quản lý agent, các agent và phần mềm hỗ trợ. Thiết kế cụ thể bên trong của AP không được miêu tả ở đây. Một AP đơn có thể trải rộng trên nhiều máy tính, các agent cư trú trên đó cũng không phải đặt trên cùng một host.

Agent: một agent là một tiến trình có sử dụng máy tính. Nó nằm trong AP và thường cung cấp một hoặc nhiều dịch vụ có sử dụng máy tính. Những dịch vụ này có thể được xuất bản dưới một bản miêu tả dịch vụ. Bản thiết kế cụ thể của những dịch vụ này, ngoài việc được biết đến như là khả năng của agent và không phải là vấn đề quan tâm của FIPA, chỉ đưa ra cấu trúc và cách mã hóa các thông điệp đưa sử dụng để trao đổi thông tin giữa các agent. Một agent phải có ít nhất một đối tượng sở hữu nó và phải hỗ trợ ít nhất một khái niệm để xác định cái nào có thể được miêu tả bởi FIPA Agent Identifier (AID). AID là cái dùng để gán nhãn cho một agent để nó có thể được phân biệt một cách rõ ràng. Một agent có thể được đăng ký một số địa chỉ vận chuyển để có thể liên hệ.

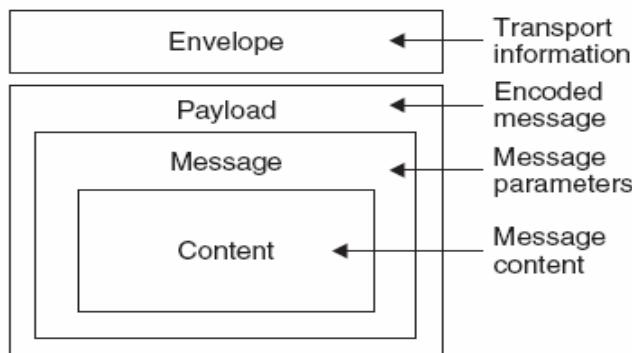
Directory Facilitator (DF): DF là một thành phần tùy chọn của AP. Nó cung cấp các dịch vụ của các trang vàng cho các agent khác. Nó duy trì một danh sách các agent đúng đắn, hoàn chỉnh và hợp thời và phải cung cấp các thông tin phổ biến nhất về agent trong thư mục của nó trên cơ sở không có sự phân biệt đối xử giữa tất cả các agent đã được chứng thực. AP có thể hỗ trợ nhiều DF mà những DF này có thể đăng ký cùng với nhau để hình thành một liên đoàn.

Mọi agent muốn công khai các dịch vụ của nó với các agent khác nên tìm một DF thích hợp và yêu cầu đăng ký bằng bản miêu tả của agent đó. Việc đăng ký đó không bao gồm cam kết hoặc trách nhiệm của agent trong tương lai. Các agent có thể yêu cầu hủy đăng ký bản miêu tả. Khi đó, không còn sự cam kết trong hành vi của DF đối với những thông tin liên quan đến agent đó nữa. Dù bất kỳ thời điểm nào với bất kỳ lý do nào, một agent có thể yêu cầu DF sửa đổi bản miêu tả của agent đó. Ngoài ra, agent có thể phát đi yêu cầu *tìm kiếm* tới DF để tìm bản miêu tả thỏa mãn điều kiện tìm kiếm đó. DF không đảm bảo giá trị của thông tin trong thông điệp phản hồi yêu cầu tìm kiếm của agent. Tuy nhiên, DF có thể hạn chế các truy cập tới thông tin trong thư mục của nó và sẽ xác minh mọi quyền truy cập của những agent đang lỗ lực truyền cho nó những thay đổi trạng thái của agent đó.

Agent Management System (AMS): AMS là một thành phần bắt buộc của AP và chịu trách nhiệm quản lý các thao tác của AP, như tạo mới và xóa agent, và dự đoán việc đến và đi của agent. Mỗi agent phải đăng ký với một AMS để có được AID, cái mà sau đó được giữ lại làm thư mục cho mọi agent và trạng thái hiện tại của chúng (như active, suspended hay waiting) trong AP. Các miêu tả của

agent sau đó có thể được sửa đổi bởi AMS trong giới hạn cho phép. Sau khi hủy đăng ký, AID có thể bị xóa và có thể dùng để phục vụ các agent khác đang yêu cầu nó. Các miêu tả agent cũng có thể được tìm kiếm trong phạm vi của AMS, và AMS cũng là người canh giữ bản miêu tả của AP – cái mà có thể lấy được bởi yêu cầu *get-description*. AMS có thể yêu cầu một agent thực hiện một chức năng quản lý cụ thể như kết thúc quá trình thực thi của nó, và có quyền làm cho thao tác đó có hiệu lực nếu yêu cầu bị bỏ qua. Chỉ một AMS đơn mới có thể tồn tại trong mỗi AP và nếu AP trải rộng trên nhiều máy, AMS cũng có quyền trên tất cả các máy đó.

Message Transport Service (MTS): là một dịch vụ được cung cấp bởi AP để vận chuyển các thông điệp FIPA-ACL giữa các agent trong một AP và giữa các agent trong các AP khác nhau. Các thông điệp cung cấp một tem vận chuyển chứa tập các tham số chi tiết như người nhận... Cấu trúc chung của thông điệp như sau:



Hình 1.6: Cấu trúc thông điệp FIPA

1.2.2.4 Kiến trúc trùu tượng

Ngoài việc giao tiếp và quản lý agent, giữa năm 2000 và 2002 một kiến trúc agent trùu tượng được tạo ra và chuẩn hóa làm phương tiện để tránh sự ảnh hưởng của nền tảng cài đặt lên đặc tả cốt lõi. Điều này có được bằng cách trùu tượng hóa các khía cạnh chính của các cơ chế quan trọng nhất như vận chuyển thông điệp và các dịch vụ trong thư mục thành một đặc tả thống nhất. Mục tiêu chính của cách này là để cho phép việc tạo ra các hệ thống được tích hợp một cách nhuần nhuyễn trong môi trường tính toán của chúng trong khi vẫn tương tác với các hệ thống agent đang cư trú trong các môi trường riêng biệt.

Các hệ agent được xây dựng theo các đặc tả ban đầu của FIPA 97 và FIPA 98 có thể tương tác với các hệ agent khác được xây dựng theo kiến trúc trùu tượng thông qua các cổng (gateway) vận chuyển với một số hạn chế. Kiến trúc FIPA2000 là một ánh xạ gần gũi hơn và cho phép tương tác một cách đầy đủ thông qua các gateway. Vì kiến trúc trùu tượng cho phép tạo nhiều thể hiện rõ ràng, nên nó cũng cung cấp các cơ chế cho phép các thể hiện đó có thể tương tác với nhau như chuyển đổi gateway cho cả việc vận chuyển và mã hóa. Kiến trúc này đặc tả cấu trúc thông điệp ACL, vận chuyển thông điệp, các dịch vụ trong thư mục của agent và các dịch vụ trong thư mục của dịch vụ như bắt buộc (mandatory).

1.2.3 Một số đặc tả FIPA chính

FIPA có 25 đặc tả chuẩn, với hơn 14 đặc tả thuộc giai đoạn thử nghiệm, 3 đặc tả thuộc giai đoạn sơ khai. Trong phần này chúng ta sẽ làm sáng tỏ một số đặc tả chính chuẩn theo thứ tự logic chứ không theo thứ tự đánh số.

1.2.3.1 Đặc tả kiến trúc trừu tượng (SC00001)

Như đã thảo luận từ trước, kiến trúc trừu tượng FIPA cung cấp những điểm tham chiếu chung và không thay đổi dành cho việc cài đặt theo FIPA sao cho giữ được những đặc tính cốt yếu và nổi bật của hệ agent trong khi vẫn cung cấp một số sự cô lập từ những thay đổi nhỏ lặp đi lặp lại mà có thể tác động lên tập đặc tả cơ bản của FIPA2000.

Kiến trúc được định nghĩa ở một mức độ trừu tượng sao cho các agent có thể tồn tại và giao tiếp với nhau bằng cách tự đăng ký và trao đổi thông điệp. Các Agent giao tiếp bằng cách trao đổi các thông điệp biểu diễn các lời nói hành động (speed act) và được mã hóa bằng ngôn ngữ giao tiếp của các agent. Các dịch vụ cung cấp dịch vụ hỗ trợ (support service) cho agent và bao gồm các dịch vụ chuẩn (standard service) của dịch vụ chỉ dẫn (directory service), dịch vụ vận chuyển thông điệp (message transport service) và dịch vụ chỉ dẫn dịch vụ (service directory service). Kiến trúc nêu rõ các dịch vụ được biểu diễn như thế nào bằng cách chỉ ra rằng chúng có thể được cài đặt như các agent hoặc như phần mềm (được truy nhập thông qua lời gọi hàm bằng cách lập trình). Agent cung cấp dịch vụ bị ràng buộc trong các hành vi của nó nhiều hơn so với các agent mang nhiệm vụ chung. Đặc biệt, những agent này được yêu cầu duy trì ngữ nghĩa của các dịch vụ. Điều này nghĩa là những agent này không ở mức độ tự chủ thông thường của agent. Chúng không thể tùy ý từ chối cung cấp dịch vụ. Một vài điều quan trọng nhất được chỉ ra trong kiến trúc trừu tượng FIPA là:

- Thông điệp agent là khuôn dạng truyền thông nền tảng giữa các agent. Cấu trúc của một thông điệp là một tập các giá trị khóa được viết trong FIPA-ACL. Nội dung của thông điệp được diễn tả trong ngôn ngữ nội dung, như FIPA-SL hoặc FIPA-KIF, và diễn đạt nội dung có thể được nhóm bởi ontologies tham chiếu đến. Thông điệp có thể chứa các thông điệp khác một cách đệ quy bên trong nội dung của chúng và phải được mã hóa thành một payload và một phong bì thông điệp vận chuyển để sử dụng một giao thức cụ thể nào đó.
- Một dịch vụ vận chuyển thông điệp được định nghĩa là các phương tiện gửi và nhận thông điệp vận chuyển giữa các agent. Nó được coi là không thể thiếu của các hệ agent tuân theo FIPA
- Một dịch vụ chỉ dẫn agent được định nghĩa là một kho thông tin chia sẻ mà ở đó các agent có thể đưa ra các bản ghi (entry) trong thư mục của nó và cũng là nơi mà chúng có thể tìm kiếm các entry trong thư mục nếu muốn. Đây được là thành phần quan trọng của các hệ agent tuân theo FIPA.

- Một dịch vụ thư mục các dịch vụ được định nghĩa là một kho chia sẻ mà ở đó các agent và các dịch vụ có thể tìm kiếm các dịch vụ. Các dịch vụ bao gồm: các dịch vụ vận chuyển thông điệp, dịch vụ thư mục agent, dịch vụ gateway và bộ đệm lưu tạm thông điệp. Một dịch vụ thư mục dịch vụ có thể được sử dụng để chứa miêu tả dịch vụ của cá dịch vụ hướng ứng dụng, như các dịch vụ thương mại và hướng nghiệp vụ. Đây được coi là thành phần không thể thiếu của các hệ agent tuân theo FIPA.

1.2.3.2 FIPA-ACL - Đặc tả kiến trúc thông điệp (SC00061)

Một thông điệp FIPA-ACL chứa một tập một hoặc nhiều tham số của thông điệp. Những tham số cần thiết cho truyền thông agent hiệu quả sẽ khác nhau tùy theo từng tình huống; tham số duy nhất không thể thiếu trong tất cả thông điệp ALC là performative, mặc dù người ta mong đợi rằng hầu hết thông điệp ACL cũng sẽ chứa các tham số: sender, receiver và content. Các tham số của thông điệp FIPA-ACL được đưa ra ở bảng 2.1 mà không đề cập tới các kiểu mã hóa cụ thể. FIPA định nghĩa ba kiểu mã hóa cụ thể: String (ký hiệu EBNF), XML và Bit-Eficient. Các tham số thông điệp mà người dùng định nghĩa cũng có thể được đưa vào bằng cách thêm vào đầu trước tham số chuỗi “X-”

Bảng 1.1: Các tham số của thông điệp ACL

Tham số	Giới thiệu
Performative	Kiểu hoạt động truyền thông của thông điệp
Sender	Chỉ ra người gửi thông điệp
Receiver	Chỉ ra người nhận mong muốn của thông điệp
Reply-to	Agent nào sẽ được chuyển thông điệp tới ngay sau đó trong luồng hội thoại
Content	Nội dung của thông điệp
Language	Ngôn ngữ mà ở đó tham số nội dung được biểu diễn
Encoding	Mã hóa của nội dung thông điệp
Ontology	Tham chiếu đến một ontology để đưa ra ý nghĩa các ký tự trong nội dung thông điệp
Protocol	Giao thức tương tác được sử dụng để xây dựng nền cuộc hội thoại
Conversation-id	Định danh duy nhất của luồng hội thoại

Reply-with	Một cách diễn đạt được sử dụng bởi agent phản ứng để chỉ ra thông điệp
In-reply-to	Tham chiếu đến hành động trước đó mà thông điệp đáp lại nó
Reply-by	Thời gian/ngày xác định thời điểm mà thông điệp phản hồi sẽ nhận được

Đây là ví dụ đơn giản của một thông điệp FIPA-ACL biểu diễn yêu cầu:

```
(request
  :sender (agent-identifier :name alice@mydomain.com)
  :receiver (agent-identifier :name bob@yourdomain.com)
  :ontology travel-assistant
  :language FIPA-SL
  :protocol fifa-request
  :content
    """((action
      (agent-identifier :name bob@yourdomain.com)
      (book-hotel :arrival 15/10/2006
        :departure 05/07/2002 ... )
    ))"""
)
```

1.2.3.3 Đặc tả thư viện communicative act của thông điệp FIPA-ACL (SC00037)

FIPA-ACL định nghĩa truyền thông ở mức chức năng hoặc hành động, được gọi là hành vi truyền thông (communicative act: CA), được thực hiện bởi hành động giao tiếp. Chuẩn này cung cấp thư viện các CA đã được xác định bởi FIPA. Tập CA được chuẩn hóa bởi FIPA được liệt kê ở bảng 1.2. Nhìn chung CA được dựa trên lý thuyết hành động lời nói (Searle, 1969). Lý thuyết này định nghĩa chức năng của các hành động đơn giản. Những chức năng này được chi tiết hóa trong đặc tả thư viện CA của FIPA (FIPA37) ví dụ như: *interrogatives* nghĩa là truy vấn thông tin, *exercitives* nghĩa là yêu cầu hành động được thực hiện, *referentials* nghĩa là chia sẻ những nhận định về môi trường, *phatics* nghĩa là thiết lập, gia hạn hoặc ngắt quá trình truyền thông, *paralinguistics* nghĩa là liên kết một thông điệp với nhiều thông điệp khác, và *expressives* nghĩa là diễn đạt thái độ, dự định và niềm tin.

Một thông điệp có thể được thực hiện một vài chức năng ở cùng một thời điểm. Ví dụ Agree được miêu tả là hành động chấp nhận thực hiện một số hành động trong tương lai. Việc này có tính chất *phatics* theo nghĩa đồng ý tiến hành và có tính chất *paralinguistics* theo nghĩa đề cập đến một CA FIPA khác; Agree chủ yếu được coi là một chức năng có tính chất *phatics*. Tất cả các CAs của FIPA vốn hỗ trợ các chức năng diễn đạt như là một chức năng phụ vì chúng được định nghĩa dưới một dạng phương thức logic hình thức biểu diễn thái độ, dự định và niềm tin. Ngoài ra, mọi CA của FIPA có thể tham chiếu đến các khái niệm được định nghĩa bởi một ontology.

Bảng 1.2: Các kiểu hành động giao tiếp của FIPA

Communicative act	Mô tả
Accept proposal	Hành động của sự chấp nhận đề xuất đã được đệ trình trước đó để thực hiện một hành động
Agree	Hành động chấp nhận thực hiện hành động nào đó, có thể trong tương lai.
Cancel	Hành động của một agent thông báo với agent khác rằng agent đầu tiên không có ý định để agent thứ hai thực hiện hành động đó nữa.
Call for proposal	Hành động yêu cầu các đề xuất để thực hiện một hành động cho trước.
Confirm	Bên gửi thông báo cho bên nhận rằng một mệnh đề cho trước là đúng, cái mà bên nhận biết được là không chắc chắn về nó.
Disconfirm	Bên gửi thông báo cho bên nhận rằng một mệnh đề cho trước là sai, cái mà bên nhận không tin, hoặc tin rằng nó đúng.
Failure	Hành động bảo agent khác rằng một hành động đã được cố gắng thực hiện nhưng bị thất bại
Inform	Bên gửi thông báo cho bên nhận rằng một đề xuất cho trước là đúng.
Inform If	Một hành động lớn cho hành động của agent thông báo cho người nhận là đề xuất cho trước có đúng hay không.
Inform Ref	Một hành động lớn cho phép bên gửi thông báo cho bên nhận một đối tượng nào đó tin tưởng vào bên gửi tương ứng với một mô tả cụ thể, ví dụ như một tên
Not Understood	Bên gửi của hành động i thông báo cho bên nhận j là nó đã cảm nhận thấy rằng j đã thực hiện một vài hành động, nhưng rằng i không hiểu j đã làm những gì. Một trường hợp phổ biến là i nói với j rằng i đã không hiểu thông điệp j đã gửi cho i.
Propagate	Bên gửi dự định rằng bên nhận đối xử với thông điệp được nhúng vào khi được gửi trực tiếp tới bên nhận, và muốn bên nhận nhận ra các agent biểu hiện bằng bởi một miêu tả cho trước và gửi thông điệp propagate

	nhận được cho họ.
Propose	Hành động đệ trình một đề xuất thực hiện một hành động nào đó với một số tiền điều kiện nào đó.
Proxy	Bên gửi muốn bên nhận chọn các agent đích được biểu thị bằng một sự mô tả có sẵn và gửi một thông điệp nhúng vào chúng.
Query If	Hành động hỏi agent khác xem đề xuất được đưa ra có đúng hay không
Query Ref	Hành động hỏi agent khác về đối tượng được tham chiếu đến bởi một biểu thức tham chiếu.
Refuse	Hành động từ chối việc thực hiện một hành động cho trước, và giải thích lý do từ chối.
Reject Proposal	Hành động bác bỏ một đề xuất để thực hiện một số hành động trong quá trình thương lượng
Request	Bên gửi yêu cầu bên nhận thực hiện một số hành động
Request When	Bên gửi muốn bên nhận thực hiện một số hành động khi một đề xuất nào đó được đưa ra trước đó trở nên đúng
Request Whenever	Bên gửi muốn bên nhận thực hiện một số hành động ngay sau khi một số đề xuất trở nên đúng và sau mỗi khi đề xuất lại trở nên đúng
Subscribe	Hành động yêu cầu một dự định lâu dài để thông báo cho bên gửi về giá trị của việc tham chiếu, và thông báo lại bất cứ khi nào đối tượng được xác định bởi những thay đổi liên quan.

1.2.3.4 Đặc tả ngôn ngữ nội dung FIPA -SL (SC00008)

Ngôn ngữ nghĩa của FIPA (SL: FIPA Semantic Language) được sử dụng để định nghĩa ngôn ngữ cho FIPA CAs như logic về thái độ và hành động, được hình thức hóa thành ngôn ngữ logic vị từ. SL được định nghĩa là một ngữ pháp diễn đạt bằng chuỗi (string expression grammar), được xác định là ngữ pháp con của cú pháp s-expression (s-expression syntax) chung. Các cách biểu diễn nội dung là *action expressions* hoặc *propositions*. Chúng được biểu diễn thành các công thức *well-formed formulas* (wff) gồm các thuật ngữ (constant, set, sequence, functional term, action expression) và các hằng số (numerical constants, string, datetime). Một *well-formed formulas* được xây dựng từ một công thức nguyên tử bằng cách áp dụng các toán tử modal hoặc các toán tử liên kết logic. Ví dụ: phép phủ định, phép hội, phép tuyễn, phép kéo theo,

phép tương đương, phép lượng tử hóa tương tự, lòng tin, dự định, đã hoàn thành, lượng rất bé (iota), any và all.

FIPA xác định 3 tập hợp con của SL (SL0, SL1 và SL2), mỗi tập con cung cấp một tập các toán tử khác nhau. FIPA SL1 mở rộng hơn FIPA SL0 bằng việc thêm vào đại số Boolean để biểu diễn các mệnh đề, như *not*, *and*, *or*. FIPA SL2 mở rộng SL1 bằng việc thêm vào toán tử logic hình thức và toán tử hành động. SL2 cho phép logic vị từ và logic hình thức nhưng bị hạn chế bởi việc chắc rằng nó phải có khả năng quyết định. Các CA khác nhau yêu cầu sử dụng các tập con SL khác nhau, ví dụ *queries* yêu cầu sử dụng của một toán tử hành động “done” được xác định trong SL0. Một cách biểu diễn nội dung FIPA -SL có thể được sử dụng như nội dung của 1 thông điệp ACL. Có 3 trường hợp sau:

- (1) Một mệnh đề, trong đó có thể được gán một giá trị chân lý trong ngữ cảnh cho trước. Một mệnh được sử dụng trong thông điệp CA *inform* và CAs khác được dẫn xuất từ nó.
- (2) Một hành động có thể được thực hiện. Một hành động có thể là một hành động riêng lẻ hoặc một hành động gộp được xây dựng bởi các toán tử chuỗi và toán tử thay thế. Một hành động được sử dụng như là một biểu thức nội dung khi hành động đó là *request* và các CAs khác dẫn xuất từ nó.
- (3) Một biểu thức tham chiếu (reference expression) được xác định là một đối tượng trong miền. Đây là toán tử tham chiếu và được sử dụng trong hành động *inform-ref* và các CAs dẫn xuất từ nó.

Một ví dụ đơn giản sau sẽ miêu tả một sự tương tác giữa agent A và B sử dụng toán tử iota, trong đó agent A có cơ sở tri thức sau $KB = \{P(A), Q(1,A), Q(1,B)\}$. Toán tử iota giới thiệu một phạm vi cho biểu thức cho trước, trong đó xác định một định danh (identifier). Biểu thức (*iota x (Px)*) có thể đọc là “các x sao cho P x đúng”. Toán tử iota xây dựng cho các thuật ngữ biểu diễn các đối tượng trong một miền.

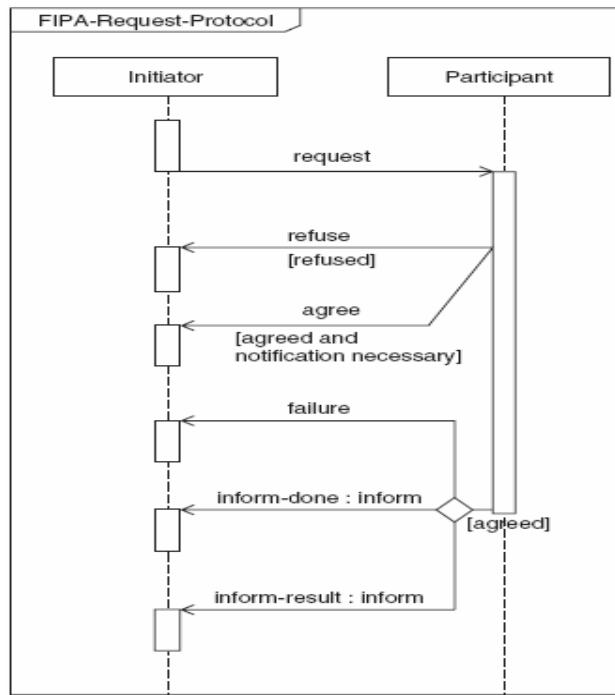
```
(query-ref
  :sender (agent-identifier :name B)
  :receiver (set (agent-identifier :name A))
  :content
    "((iota ?x (p ?x)))"
  :language fipa-sl
  :reply-with query1)

(inform
  :sender (agent-identifier :name A)
  :receiver (set (agent-identifier :name B))
  :content
    " ((= (iota ?x (p ?x)) alpha)) "
  :language fipa-sl
  :in-reply-to query1)
```

Đối tượng duy nhất thỏa mãn mệnh đề $P(x)$ là alpha, do vậy, thông điệp query-ref được phản hồi bằng thông điệp inform như trên.

1.2.3.5 Đặc tả giao thức yêu cầu tương tác (SC00026)

Trong biểu đồ trình tự trong Hình 1.7, giao thức yêu cầu tương tác (IP) cho phép một agent Initiator (agent khởi tạo) yêu cầu agent khác (Participant) thực hiện một hành động. Participant xử lý các yêu cầu và đưa ra một quyết định chấp nhận (*accept*) hay là từ chối (*refuse*) yêu cầu.



Hình 1.7: Giao thức yêu cầu tương tác

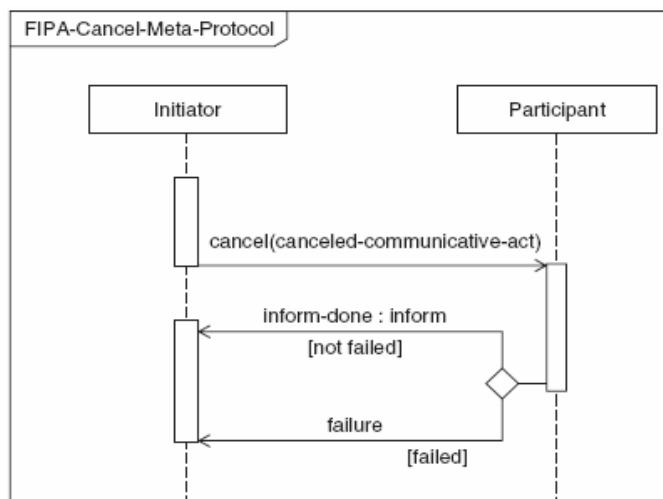
Nếu điều kiện chỉ ra rằng một thỏa thuận rõ ràng là cần thiết (nghĩa là, “notification necessary” có giá trị true), khi đó Participant đưa ra một thông điệp *agree*. Thỏa thuận này là tùy chọn, nó phụ thuộc vào từng trường hợp, ví dụ, nếu hành động được yêu cầu rất nhanh và có thể xảy ra trước thời gian xác định trong tham số *reply-by*. Một khi yêu cầu đã được đồng ý, Participant phải phản hồi bằng một trong các trường hợp sau:

- *failure* nếu nỗ lực thực hiện yêu cầu bị thất bại
- *inform-done* nếu nó thực hiện thành công yêu cầu và muốn chỉ ra rằng nó đã hoàn thành
- *inform-result* nếu nó muốn chỉ ra rằng nó đã hoàn thành và thông báo kết quả cho Initiator.

Bất kỳ tương tác nào sử dụng giao thức tương tác này đều được xác định là duy nhất, tham số *conversation-id* không null, được gán bởi Initiator và được thiết lập trong cấu trúc thông điệp ACL. Các agent liên quan đến sự tương tác phải gắn thẻ *định danh cuộc hội thoại* vào tất cả các gói tin. Điều này cho phép mỗi agent quản lý các chiến lược và các hoạt động truyền thông của nó; ví dụ, nó cho phép một agent xác định các hội thoại cá nhân và suy luận thông qua các ghi chép lịch sử về các hội thoại.

Tại bất kỳ điểm nào trong IP (interaction protocol), bên nhận có thể cho bên gửi biết rằng nó không hiểu cái gì đã được truyền thông. Điều này được thực hiện bằng cách gửi trả về gói tin *not-understood*. Hình 1.7 không miêu tả giao tiếp *not-understood* vì nó có thể xảy ra tại bất kỳ điểm nào trong IP. Truyền thông điệp *not-understood* trong một giao thức tương tác có thể chấm dứt toàn bộ IP và việc chấm dứt tương tác ám chỉ rằng bất kỳ cam kết nào đã tạo ra trong suốt quá trình tương tác đều là *null* và *void*.

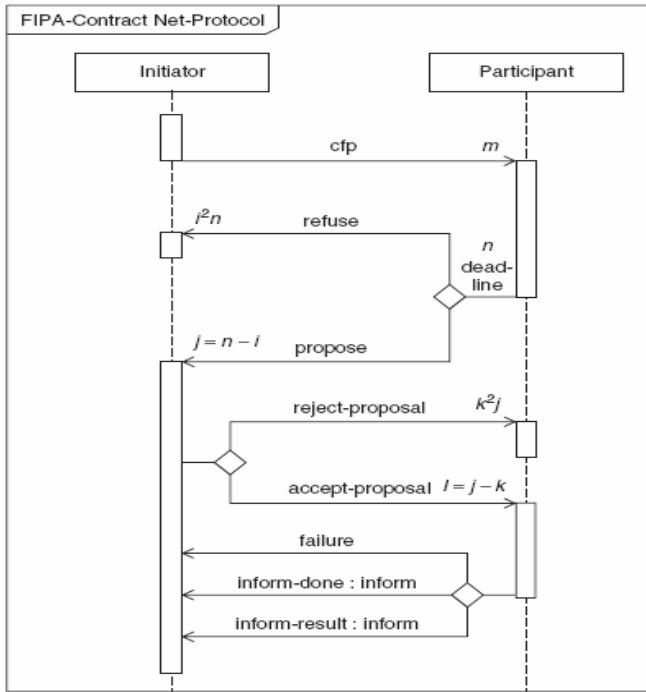
Ngoài ra, tại bất kỳ điểm nào trong IP, Initiator có thể hủy bỏ tương tác bằng cách khởi tạo giao thức Cancel Meta-Protocol như trong Hình 1.8. Tham số *conversation-id* của tương tác *cancel* chính là tham số *conversation-id* của tương tác mà Initiator dự định hủy bỏ. Ngữ nghĩa của việc hủy bỏ được hiểu một cách đại khái là Initiator không còn quan tâm đến việc tiếp tục tương tác và nó nên được kết thúc theo một cách thức chấp nhận được cho cả Initiator và Participant. Participant thông báo cho Initiator rằng tương tác được thực hiện bằng cách sử dụng thông điệp *inform-done* hoặc chỉ ra việc hủy bỏ bị thất bại bằng cách sử dụng thông điệp *failure*.



Hình 1.8: giao thức Cancel Meta

1.2.3.6 Giao thức tương tác mạng hợp đồng (FIFA Contract Net Interaction Protocol) (SC00026)

Một ví dụ về một giao thức tương tác phức tạp hơn, FIPA Contract Net Interaction Protocol (IP) mô tả trường hợp một agent (Initiator) muốn có một số tác vụ được thực hiện bởi một hay nhiều agent khác (Participants) và hơn nữa muốn tối ưu hóa một chức năng mô tả nhiệm vụ đó. Đối với một công việc nhất định, bất kỳ một Participant nào cũng có thể đáp ứng lại lời đề nghị; các Participant còn lại phải từ chối. Sau đó các cuộc đàm phán lại tiếp tục với các Participant được đề xuất. Giao thức IP được mô tả trong Hình 1.9.



Hình 1.9: Giao thức mạng tương tác hợp đồng

Initiator mời chào m lời đề nghị đến các agent khác bằng cách phát hành một lời gọi đề xuất (call for proposals – CFP) đặc tả nhiệm vụ và các điều kiện mà Initiator đặt vào khi thực hiện nhiệm vụ. Các Participant tiếp nhận cfp được coi là các nhà thầu tiềm năng và có khả năng tạo ra n đáp ứng. Trong số này, j là các đề xuất thực hiện nhiệm vụ, được xác định là *proposal* CAs.

Đề xuất của Participant bao gồm các tiền điều kiện mà Participant đang thiết lập cho tác vụ, có thể là chi phí, thời gian khi công việc sẽ được thực hiện... Ngoài ra, có $i = n - j$ Participant có thể từ chối đề xuất. Một khi deadline trôi qua, Initiator đánh giá j lời đề xuất nhận được và chọn các agent để thực hiện nhiệm vụ; có thể có một agent, một số agent hoặc không có agent được chọn. l agent của các đề xuất được chọn sẽ được gửi một *accept-proposal* CA và k agent còn lại sẽ nhận được một *reject-proposal* CA. Các đề xuất phải ràng buộc vào Participant, để khi Initiator chấp nhận đề nghị, Participant phải cam kết thực hiện nhiệm vụ. Một khi đã hoàn thành nhiệm vụ, Participant sẽ gửi một thông báo hoàn thành dưới dạng *inform-done* hoặc *inform-result* cho Initiator. Tuy nhiên, nếu Participant không hoàn thành nhiệm vụ, một thông điệp *failure* được gửi đi.

Giao thức IP này yêu cầu Initiator biết khi nào nó đã nhận được mọi phản hồi. Trong trường hợp Participant thất bại khi phản hồi thông điệp propose hoặc refuse, Initiator có khả năng chờ đợi mãi mãi. Để chống lại điều này, CFP CA bao gồm một deadline xác định trong bao lâu Initiator sẽ nhận được phản hồi. Các đề xuất nhận được sau thời hạn deadline sẽ đưa ra lý do đề xuất đến. Deadline được đặc tả bởi tham số reply-by trong thông điệp ACL.

1.2.3.7 Đặc tả dịch vụ vận chuyển thông điệp (FIFA Message Transport Service) (SC00067)

Dịch vụ FIFA Message Transport Service (MTS) cung cấp cơ chế vận chuyển các thông điệp FIPA-ACL giữa các agent sử dụng giao thức Message Transport Protocol (MTP). Các agent có liên quan có thể nằm trong một AP đơn hoặc trên các AP khác nhau. Trên một AP cho trước bất kì, dịch vụ MTS được cung cấp bởi một kênh truyền thông agent (Agent Communication Channel – ACC).

Một thông điệp được tạo thành bởi hai phần: một phong bì thư (envelope) để thể hiện thông tin vận chuyển và trọng tải (payload) bao gồm thông điệp ACL đã được mã hóa của phiên truyền thông agent. Bất kì MTP nào cũng có thể sử dụng một cách biểu diễn bên trong khác để mô tả một phong bì thông điệp, nhưng phải thể hiện các thuật ngữ giống nhau, biểu diễn cùng một ngữ nghĩa và thực hiện các hành động tương ứng. Một phong bì thông điệp bao gồm một tập các tham số, mỗi tham số trong số đó là một cặp tên/giá trị với các trường bắt buộc *to*, *from*, *date* và *acl-representation*.

ACC cung cấp một MTS và chịu trách nhiệm gửi và nhận thông điệp trên một AP. ACC phải truyền các thông điệp mà nó nhận được theo các hướng dẫn vận chuyển chứa trong phong bì thông điệp và chỉ được yêu cầu đọc phong bì thông điệp; nó không được yêu cầu chuyển đổi payload. Trong việc thực hiện nhiệm vụ chuyển thông điệp, ACC có thể được yêu cầu để có được thông tin từ AMS hoặc DF trên AP của riêng mình. Một số cài đặt ACC có thể cung cấp một số dạng khả năng bộ đệm (buffering capability) để giúp các agent quản lý thông điệp của chúng. Mỗi ACC đang xử lý một thông điệp có thể thêm các thông tin mới vào phong bì thông điệp, nhưng nó không bao giờ có thể ghi đè lên các thông tin đã tồn tại. Các ACC cũng có thể thêm các thông số mới vào một phong bì thông điệp để có thể ghi đè lên các tham số có cùng tên tham số; cơ chế làm cho rõ ràng các entry trong phong bì thông điệp được đặc tả bởi mỗi cú pháp phong bì thông điệp cụ thể. Thông điệp chuyển tiếp hành vi của một ACC được xác định bởi những hướng dẫn phân phối thông điệp được thể hiện trong phong bì thông điệp, các thông số đó được mô tả trong Bảng 1.3.

Người nhận thông điệp được xác định trong tham số “*to*” của phong bì thư và theo mẫu của các AID. Tuỳ thuộc vào sự hiện diện của tham số *intended-receiver*, ACC chuyển tiếp thông điệp đi theo một trong những cách sau:

- Nếu ACC nhận được một phong bì thư mà không có một tham số *intended-receiver* nào, thì nó tự tạo ra một tham số *intended-receiver* mới từ tham số “*to*” (có thể chứa nhiều AID). Nó cũng có thể tạo ra nhiều bản sao của thông điệp với các tham số *intended-receiver* khác nhau nếu có nhiều bên nhận được chỉ định. Trong mọi trường hợp, ACC được yêu cầu để xử lý tất cả các entry trong trường tham số “*to*” và không thêm và không bớt bất cứ AID nào được chứa trong thông điệp gốc. Các tham số *intended-receiver* tạo thành một đường chuyền giao chỉ ra tuyến đường mà một thông điệp sẽ đi qua.

- Nếu ACC nhận được một phong bì thư với một biến *intended-receiver*, nó được sử dụng cho việc chuyển giao của thể hiện của thông điệp và tham số “to” được bỏ qua.

Bảng 1.3: Các tham số trong phần envelope của thông điệp FIPA

STT	Tham số	Miêu tả
1	To	Tên agent nhận đầu tiên của thông điệp
2	From	Tên agent gửi thông điệp
3	Comments	Các ghi chú văn bản
4	Acl-representation	Biểu diễn cú pháp của phần payload của thông điệp
5	Payload-length	Chiều dài (số byte) của phần payload
6	Payload-encoding	Ngôn ngữ mã hóa phần payload của thông điệp
7	Date	Ngày tháng và thời gian tạo envelope
8	Intended-receiver	Tên của các agent mà thông điệp được chuyển tới
9	Received	Tem xác định agent nhận thông điệp theo ACC
10	Transport-behaviour	Các yêu cầu vận chuyển thông điệp

- Nếu ACC nhận được thông điệp có nhiều tham số intended-receiver thì tham số gần nhất được sử dụng.

Trước khi chuyển tiếp thông điệp, ACC thêm tham số received vào phần envelope của thông điệp. Một khi ACC đã chuyển tiếp thông điệp, nó không cần phải lưu giữ bất kỳ ghi chép nào về sự tồn tại của thông điệp nữa. AID có trong tham số to hoặc intended-receiver có thể chứa nhiều địa chỉ vận chuyển của một agent nhận. ACC sử dụng các cách sau để phân phối thông điệp:

- Thủ phân phối thông điệp đến địa chỉ vận chuyển đầu tiên trong tham số addresses; việc địa chỉ đầu tiên được chọn thể hiện rằng danh sách các địa chỉ vận chuyển trong AID được sắp xếp theo ý thích.
- Nếu thất bại, do agent hoặc AP không sẵn sàng hoặc do ACC không hỗ trợ giao thức vận chuyển thông điệp thích hợp, thì ACC tạo một tham số intended-receiver mới chứa AID và xóa đi địa chỉ vận chuyển gây thất bại ở trên. Sau đó, ACC cố gắng gửi thông điệp đến địa chỉ vận chuyển tiếp theo trong AID trong danh sách bên nhận dự định.
- Nếu việc phân phối vẫn không thành công khi tất cả các địa chỉ vận chuyển đã được thử (hoặc AID không còn địa chỉ vận chuyển nào), ACC có thể thử quyết định AID bằng cách sử dụng các dịch vụ quyết định tên (name resolution services) được liệt kê trong tham số resolvers của AID. Tóm lại, các dịch vụ quyết định tên nên được thử theo thứ tự xuất hiện của chúng.

Cuối cùng, nếu tất cả các thông điệp được phân phối thất bại thì các thông điệp báo lỗi tương ứng cho lần thất bại cuối cùng sẽ được gửi trả lại agent đã gửi thông điệp đi. ACC sử dụng các quy tắc sau trong việc phân phối các thông điệp tới nhiều agent nhận dự kiến:

- Nếu ACC nhận được thông điệp không có tham số intended-receiver và một tham số chứa nhiều AID, nó có thể hoặc không tách chúng ra để tạo thành các thông điệp riêng. Mỗi thông điệp sẽ chứa một tập con các agent có tên trong tham số to và intended-receiver.
- Nếu ACC nhận được thông điệp có một tham số intended-receiver chứa nhiều AID, nó có thể hoặc không tách chúng ra để tạo thành các thông điệp riêng.
- Nếu ACC tách một thông điệp như miêu tả ở trên thì nó không được thêm hay bớt bất kỳ AID nào trong thông điệp ban đầu.

Các thông điệp kết quả được xử lý như trong trường hợp có một agent nhận. Một agent có ba lựa chọn khi gửi một thông điệp tới agent khác trên AP từ xa:

- Agent A gửi thông điệp tới ACC cục bộ của nó sử dụng một giao diện chuẩn hoặc độc quyền. Sau đó ACC tìm cách gửi thông điệp tới đúng ACC ở xa sử dụng MTP thích hợp.
- Agent A gửi thông điệp trực tiếp tới ACC trên AP ở xa mà agent B cư trú trên đó. Sau đó ACC ở xa này sẽ phân phối thông điệp tới B. Sử dụng cách này, agent A phải hỗ trợ việc truy cập tới một trong các giao diện MTP của ACC.
- Agent A gửi thông điệp trực tiếp tới agent B bằng cách sử dụng cơ chế truyền thông trực tiếp. Việc truyền, đánh địa chỉ, lưu tạm thông điệp và các thông điệp lỗi phải được xử lý bởi agent gửi và agent nhận. Chế độ truyền thông này không được miêu tả bởi FIPA.

Cuối cùng, việc miêu tả quá trình vận chuyển là một phần của AP và được biểu diễn trong fipa-s10. Miêu tả quá trình vận chuyển sau đây là dành cho AP, cái mà hỗ trợ vận chuyển dựa trên IIOP và HTTP.

```
(ap-description
  :name myAPDescription
  :ap-services
  (set
    (ap-service
      :name myIIOPMTP
      :type fipa.mts.mtp.iiop.std
      :addresses
      (sequence
        corbaloc:iiop:agents.fipa.org:10100/acc
        IOR:000000000233
        corbaname::agents.fipa.org:10000/nameserver#acc)
    )
    (ap-service
      :name myHTTPMTP
      :type fipa.mts.mtp.http.std
      :addresses
      (sequence
```

)
)
)

1.2.4 Liên quan giữa FIPA và JADE

Nhờ rằng FIPA dựa trên nguyên lý là chỉ đặc tả các hành vi bên ngoài của các thành phần trong hệ thống, bỏ qua kiến trúc và chi tiết cài đặt bên trong. Điều này đảm bảo sự kết nối liền mạch giữa các nền tảng biên dịch đầy đủ. JADE tuân theo quan điểm này ở chỗ nó đảm bảo tính tương thích trọn vẹn với đặc tả FIPA2000 (truyền thông, quản lý và kiến trúc) – đặc tả này cung cấp một framework chuẩn trong đó các agent có thể tồn tại, vận hành và giao tiếp trong khi vẫn chấp nhận một kiến trúc bên trong thống nhất và độc quyền và chấp nhận cài đặt các dịch vụ và các agent chính.

JADE tất nhiên chỉ là một trong các nền tảng ứng dụng và dự án có tính cộng tác về agent tuân theo các chuẩn của FIPA. Việc tuân theo này đã được kiểm tra thông qua một số sự kiện: cuộc kiểm tra tính tương kết của FIPA năm 1999 và 2001, dự án Agentcities. Về mặt độ bao phủ của các chuẩn của FIPA, JADE cài đặt hoàn chỉnh đặc tả quản lý agent bao gồm các dịch vụ: AMS, DF, MTS và ACC. Thông qua việc sử dụng và thử nghiệm, các dịch vụ này đã được mở rộng với việc bổ sung các tính năng, nhưng cốt lõi vẫn tuân thủ theo FIPA. JADE cũng cài đặt hoàn chỉnh ngăn xếp giao tiếp agent như FIPA-ACL dành cho cấu trúc thông điệp, FIPA-SL dành cho diễn đạt nội dung thông điệp, ngoài ra còn hỗ trợ nhiều giao thức vận chuyển và tương tác của FIPA.

Một ví dụ chứng tỏ JADE vẫn tiếp tục các chuẩn của FIPA là cơ chế vận chuyển của JADE. Cơ chế này hỗ trợ tất cả các thao tác cụ thể, có khả năng thích nghi với một kiểu kết nối bằng các chọn giao thức sẵn có tốt nhất tương ứng với tình huống sử dụng cụ thể. Một số khía cạnh bổ sung vào các chuẩn của FIPA như một số giao thức tương tác đặc biệt hơn và các công việc không được chuẩn hóa như dịch vụ ontology đã được phát triển cho JADE thậm chí các lập trình viên có thể tìm thấy mọi công cụ và trừu tượng hóa cần thiết để cài đặt chúng. Tuy nhiên, có nhiều thành phần của JADE vượt quá các đặc tả của FIPA. Ví dụ, JADE cung cấp kiến trúc bộ chúa phân tán, kiến trúc dịch vụ nội bộ, phân phối thông điệp lâu dài, framework ngữ nghĩa, các cơ chế bảo mật, hỗ trợ tính di động của agent, tương tác web-service, giao diện đồ họa... Rất nhiều trong số chúng được miêu tả trong cuốn sách này vì chúng là những khía cạnh rất quan trọng trong các hệ hướng agent mà FIPA chưa đề cập đến. Do việc hỗ trợ mã nguồn mở từ cộng đồng người dùng và từ nền công nghiệp mà ngày nay JADE được coi là framework về agent mã nguồn mở hàng đầu tuân theo FIPA.

CHƯƠNG 2

GIỚI THIỆU JADE

Chương này cung cấp một cái nhìn tổng quan về nền tảng JADE và các thành phần chính tạo thành kiến trúc của nó. Ngoài ra, chương này còn hướng dẫn cách chạy JADE bằng dòng lệnh và bằng giao diện đồ họa.

2.1 TÓM TẮT LỊCH SỬ CỦA JADE

Những phần mềm phát triển đầu tiên, cuối cùng trở thành nền tảng JADE, đã được bắt đầu xây dựng bởi Telecom Italia (viết tắt là CSELT) cuối năm 1998, do cần sớm có sự xác nhận các đặc tả kỹ thuật FIPA. Ban đầu nhóm phát triển không hoàn toàn mong đợi đạt được mục tiêu phát triển một nền tảng song nhò có sự hỗ trợ tài chính của Ủy ban Châu Âu và sự nhiệt tình cũng như năng lực của đội (vào thời gian đó có Fabio Bellifemine, Agostino Poggi và Giovanni Rimassa), nên đã quyết định đơn giản hóa việc xác minh các đặc tả của FIPA để phát triển một nền tảng chung đầy đủ. Với quan điểm là để cung cấp các dịch vụ cho người phát triển ứng dụng và để dễ dàng sử dụng được và truy cập được cho cả những người phát triển lâu năm và người mới có ít hoặc không có chút kiến thức nào về những đặc tả của FIPA. JADE đặc biệt nhấn mạnh vào sự đơn giản và tiện dụng của các phần mềm API.

JADE đã trở thành mã nguồn mở từ năm 2000 và được phân phối bởi Telecom Italia theo giấy phép LGPL. Giấy phép này đảm bảo tất cả các quyền cơ bản để tạo thuận lợi cho việc sử dụng phần mềm có trong các sản phẩm thương mại: quyền làm bản sao của phần mềm và phân phối các bản sao, quyền được truy cập mã nguồn, và quyền được thay đổi mã và thực hiện các cải tiến của nó. Không giống như GPL, giấy phép LGPL không đặt bất kỳ hạn chế sử dụng phần mềm JADE, nó cho phép có quyền sở hữu phần mềm để kết hợp với phần mềm bản quyền bởi GPL. Mặt khác, giấy phép cũng đòi hỏi rằng bất kỳ công việc dẫn suất từ JADE, hoặc bất kỳ công việc nào dựa vào nó, sẽ được trả lại công đồng với cùng một loại giấy phép.

JADE có một website, <http://jade.tilab.com>, từ đó các phần mềm, tài liệu, mã nguồn ví dụ, và rất nhiều thông tin về cách sử dụng của JADE đều có sẵn. Dự án hoan nghênh sự tham gia của cộng đồng mã nguồn mở với nhiều cách thức để tham gia và đóng góp cho dự án, chúng đều được chi tiết hóa trên trang web, ví dụ:

- Gửi email jade-contrib@avalon.tilab.com với một mô tả công khai các trường hợp sử dụng của bạn cho JADE, các dự án nghiên cứu hoặc các khóa học sử dụng JADE, hoặc bất kỳ hội thảo hoặc sự kiện công khai có thể có ích cho cộng đồng JADE.
- Tham gia vào các cuộc thảo luận trong JADE mailing lists bằng cách trả lời và đưa ra hỗ trợ cho người dùng ít kinh nghiệm hơn. Hòm thư: jade-develop@avalon.tilab.com để thảo luận về các vấn đề và ý tưởng liên quan đến việc sử dụng và phát triển của JADE, và jade-news@avalon.tilab.com được sử dụng bởi các quản trị viên dự án để thông báo cho cộng đồng về những bản phần mềm mới phát hành và các sự kiện liên quan đến JADE.
- Cung cấp các báo cáo lỗi và khi nào có thể sửa lỗi.
- Đóng góp những add-on mới và các module phần mềm để sử dụng bởi cộng đồng.

Để tạo điều kiện tham gia tốt hơn, tháng 5 năm 2003 Telecom Italia Lab và Motorola Inc, đã đưa ra một thỏa thuận hợp tác và thành lập Ủy Ban Quản Lý JADE, một tổ chức phi lợi nhuận của các công ty quan tâm đóng góp cho sự phát triển của JADE. Ủy ban được thành lập như là một tập hợp của các thỏa thuận với các quy định quản lý tốt các quyền và nghĩa vụ nhằm mục đích tạo ra IPR. Tổ chức là mở với các thành viên có thể tham gia và rời khỏi tùy theo nhu cầu của họ. Hiện thời Telecom Italia, Motorola, France Telecom R&D, Whitespace Technologies AG và Profactor GmbH đã trở thành thành viên của Ủy ban.

Khi JADE lần đầu tiên được công bố bởi Telecom Italia, nó đã được sử dụng hầu như chỉ bởi cộng đồng FIPA nhưng khi tích hợp các chức năng lại vượt xa các chi tiết kỹ thuật FIPA. Do đó nó đã được sử dụng bởi một cộng đồng các nhà phát triển được phân phối trên toàn cầu. JADE góp phần phổ biến rộng rãi các chi tiết kỹ thuật bởi việc cung cấp một tập trùu tượng hóa phần mềm và các công cụ để ẩn các đặc trưng của chúng; những người lập trình có thể thực thi tùy theo các chi tiết kỹ thuật mà không cần phải nghiên cứu chúng. Chúng ta xem đây là một trong những điểm mạnh của chính JADE đối với FIPA.

Một trong những phần mở rộng của lõi JADE được cung cấp bởi LEAP, một dự án tài trợ một phần bởi Ủy ban châu Âu đã góp phần đáng kể từ năm 2000 và 2002 nhằm hướng JADE tới Java Micro Edition và môi trường mạng không dây. Công việc này được dẫn dắt bởi Giovanni Caire. Ngày nay, nó được dùng như một JADE run-time cho các nền tảng J2ME-CLDC và J2ME-CDC, và nó được sử dụng để giải quyết các vấn đề và thách thức đặt ra trong viễn thông di động, đây được coi là một trong những tính năng hàng đầu của JADE.

2.2 JADE VÀ MÔ HÌNH AGENT

JADE là một nền tảng phần mềm cung cấp chức năng cơ bản cho tầng giữa, độc lập với các ứng dụng cụ thể và đơn giản hóa việc thực hiện của các ứng dụng phân tán – những ứng dụng khai thác sự trùu tượng của các agent phần mềm. Một đặc điểm đầy ý nghĩa của JADE là nó thực thi

sự trùu tượng này trên ngôn ngữ hướng đối tượng, Java, cung cấp một API đơn giản và thân thiện. Những lựa chọn thiết kế đơn giản đều bị ảnh hưởng bởi sự trùu tượng của agent.

Một agent có tính tự chủ và hướng đích: một agent không thể cung cấp các call-back hoặc tham chiếu đối tượng của chính nó tới các agent khác để làm giảm đi cơ hội điều khiển của các thực thể lên các dịch vụ của nó. Một agent phải có luồng thực thi của chính nó, sử dụng nó để điều khiển vòng đời của nó và tự chủ quyết định khi nào thực thi các hành động.

Các agent có thể nói không, và chúng được gắn kết lỏng lẻo: Việc giao tiếp không đồng bộ dựa trên thông điệp là hình thức giao tiếp cơ bản giữa các agent trong JADE; một agent muốn giao tiếp phải gửi thông điệp đến một điểm được xác định (hoặc thiết lập các điểm đến). Việc này không phụ thuộc vào thời gian giữa người gửi và người nhận: một người nhận có thể không có mặt khi người gửi gửi thông điệp đến. Cũng không cần phải lấy tham chiếu đối tượng của agent nhận mà cần có các định danh tên để hệ vận chuyển thông điệp có thể dựa vào đó để chuyển thông điệp đến đúng địa chỉ. Thậm chí bên gửi có thể không cần biết về định danh của bên gửi, nó có thể định nghĩa một danh sách bên nhận sử dụng intentional grouping (nhóm người nhận dự kiến) hoặc sử dụng một proxy agent trung gian.

Hơn nữa, dạng thức giao tiếp này cho phép bên nhận có quyền lựa chọn thông điệp sẽ xử lý hay loại bỏ, cũng như có quyền xác định các mức ưu tiên xử lý của chính nó (VD: đọc tất cả thông điệp đến từ miền ‘book.it’ đầu tiên). Cách truyền thông này còn cho phép bên gửi có thể điều khiển luồng thực thi của nó và như vậy không bị khóa cho đến khi bên nhận xử lý thông điệp. Cuối cùng, cách truyền thông này còn có một ưu điểm đáng chú ý khi cài đặt truyền thông quảng bá (multi-cast) như một hành động nguyên tử chứ không gọi N phương thức liên tiếp nhau (VD: một hành động gửi với một danh sách gồm nhiều bên nhận thông điệp thay vì một lời gọi phương thức tới mỗi đối tượng ở xa mà bạn muốn giao tiếp với nó).

Hệ thống có kiểu Peer-to-Peer: mỗi agent được xác định bởi một tên toàn cục duy nhất. Nó có thể tham gia vào và rời khỏi một nền tảng máy chủ ở bất kỳ thời điểm nào và có thể nhận ra các agent khác thông qua cả 2 dịch vụ white-page và yellow-page cung cấp trong JADE bởi AMS và DF mà đã được định nghĩa bởi FIPA. Một agent có thể là chủ thể khởi tạo sự giao tiếp tới bất kỳ agent khác trong bất kỳ thời gian nào nó mong muốn và tương tự nó có thể là đối tượng để các agent khác khởi tạo giao tiếp đến ở bất kỳ thời điểm nào.

Trên cơ sở những lựa chọn thiết kế này, JADE đã được cài đặt để cung cấp cho các nhà lập trình các chức năng cốt lõi sẵn sàng để sử dụng và dễ dàng để tùy biến sau đây:

- Một hệ thống hoàn toàn phân tán mà các agent cư trú trên đó, mỗi agent hoạt động như là một luồng riêng biệt, và có khả năng giao tiếp một cách trong suốt với agent khác. Ví dụ, nền tảng cung cấp một API độc lập về vị trí duy nhất mà có thể trùu tượng hóa cơ sở hạ tầng giao tiếp bên dưới.

- Tuân thủ đầy đủ các đặc tả của FIPA. Nền tảng tham gia thành công vào tất cả các sự kiện phối hợp hoạt động của FIPA và được sử dụng như là tầng giữa của nhiều nền tảng trong mạng lưới Agentcities. Điều này đã tạo nên sự đóng góp lớn lao của đội JADE vào quá trình chuẩn hóa của FIPA.
- Phương tiện vận chuyển hiệu quả của các thông điệp không đồng bộ thông qua một API trong suốt về vị trí. Nền tảng lựa chọn các phương tiện sẵn có tốt nhất của truyền thông và khi có thể, tránh sự sắp xếp theo thứ tự hoặc không theo thứ tự các đối tượng Java. Khi đi qua ranh giới nền tảng, các thông điệp tự động được biến đổi từ cách biểu diễn bằng Java bên trong của JADE sang các cú pháp tuân theo FIPA, cách giải mã và các giao thức vận chuyển.
- Thực thi cả 2 dịch vụ white-page và yellow-page. Hệ thống có thể được cài đặt để biểu diễn các miền và các miền con như một đồ thị các thư mục.
- Quản lý vòng đời agent đơn giản nhưng hiệu quả. Khi các agent đã được tự động gán một định danh toàn cục duy nhất và một địa chỉ vận chuyển được sử dụng để đăng ký với dịch vụ white-page của nền tảng. Các API đơn giản và các công cụ đồ họa cũng được cung cấp để quản lý vòng đời agent vừa từ xa và vừa cục bộ, như tạo, định chỉ, phục hồi, đóng băng, tan băng, di chuyển, lặp lại và xóa.
- Cung cấp tính di động của agent. Cả mã và trạng thái của agent đều có thể di chuyển giữa các tiến trình và các máy. Sự di chuyển của được tạo ra để các agent giao tiếp một cách trong suốt mà có thể tiếp tục tương tác thậm chí là trong suốt quá trình di chuyển.
- Một cơ chế đặt trước (subscription) cho mỗi agent, và thậm chí là cả các ứng dụng bên ngoài, mà muốn đăng ký với một nền tảng để được thông báo về tất cả các sự kiện của platform, bao gồm các sự kiện có liên quan đến vòng đời và các sự kiện trao đổi thông điệp.
- Một tập các công cụ đồ họa để hỗ trợ người lập trình khi debug và monitor. Chúng đặc biệt quan trọng và phức tạp trong các hệ thống đa luồng, nhiều tiến trình, nhiều máy ví dụ như một ứng dụng JADE điển hình.
- Hỗ trợ các Ontology và các ngôn ngữ nội dung. Việc kiểm tra ontology và việc mã hóa nội dung được thực hiện tự động bởi nền tảng, các nhà lập trình có thể lựa chọn các ngôn ngữ nội dung và ontologies yêu thích. Những người lập trình còn có thể cài đặt những ngôn ngữ mới để thực hiện các yêu cầu ứng dụng cụ thể.
- Một thư viện của các giao thức tương tác: mô hình các mẫu đặc trưng của truyền thông nhằm đạt được một hoặc nhiều mục đích. Các skeleton độc lập với ứng dụng là một tập các lớp Java có sẵn và có thể tùy chọn. Các giao thức tương tác cũng có thể được thể hiện và được cài đặt như một tập các máy trạng thái đồng thời.

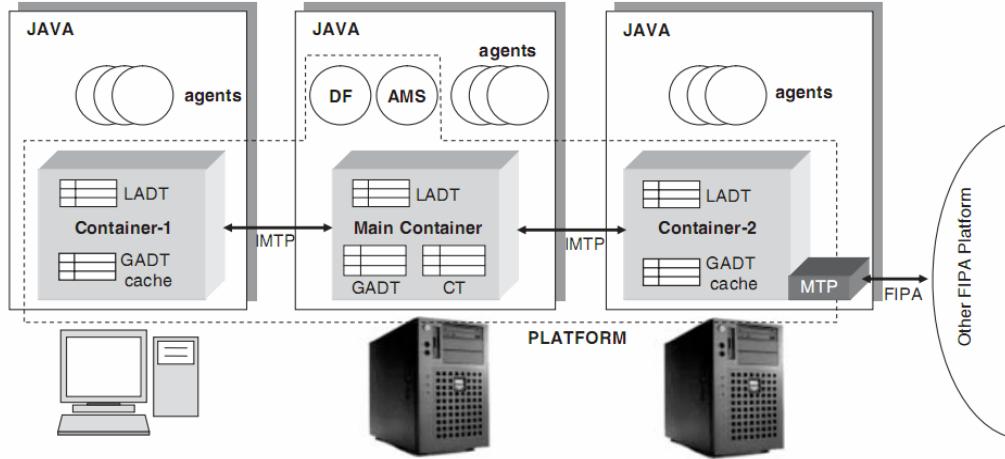
- Sự tích hợp với các công nghệ khác nhau dựa trên Web bao gồm các công nghệ JSP, Servlet, applet và Web Service. Nền tảng cũng có thể được cấu hình một cách dễ dàng để xuyên qua tường lửa.
- Hỗ trợ nền tảng J2ME và môi trường không dây. JADE run-time có thể dùng cho các nền tảng J2ME – CDC và J2ME-LCDC thông qua một tập không đổi của các API che phủ cả 2 môi trường J2ME và J2SE.
- Một giao diện tiến trình bên trong (in-process) cho việc khởi chạy và việc điều khiển một platform và các thành phần phân tán của nó từ một ứng dụng bên ngoài.
- Một nhân có thể mở rộng được thiết kế để cho phép những người lập trình mở rộng các chức năng của nền tảng thông qua việc bổ sung các dịch vụ phân tán mức nhân. Cơ chế này xuất phát từ phương pháp lập trình hướng giao diện trong đó các giao diện thành phần khác nhau có thể kết hợp vào trong mã ứng dụng và được sắp xếp ở mức nhân. Để duy trì khả năng tương thích với môi trường J2ME nơi mà không hỗ trợ lập trình hướng khía cạnh (aspect-oriented), JADE sử dụng một cách tiếp cận lọc gộp đặc biệt (composition filter) được miêu tả trong chương sau.

2.3 KIẾN TRÚC JADE

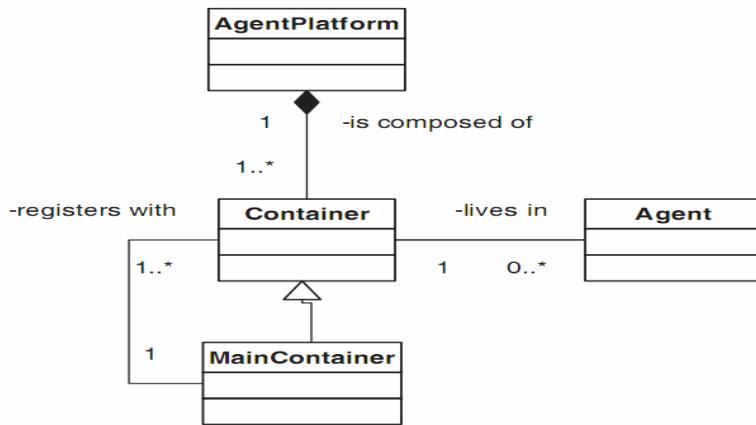
Hình 2.1 chỉ ra các thành phần kiến trúc chính của JADE platform. Một JADE platform bao gồm các khung chứa agent (agent containers), có thể được phân tán trên mạng. Các agent sống trong khung chứa là các tiến trình Java, cung cấp JADE run-time và tất cả các dịch vụ cần cho việc lưu trú và thực thi các agent. Có một khung chứa đặc biệt, được gọi là khung chứa chính (*main container*), thể hiện nét nổi bật của platform: Nó là khung chứa đầu tiên được khởi chạy và tất cả các khung chứa khác phải đăng ký để gia nhập vào khung chứa chính. Biểu đồ UML trong hình 2.2 miêu tả quan hệ giữa các thành phần kiến trúc chính trong JADE.

Người lập trình sử dụng tên logic để xác định các khung chứa; mặc định, khung chứa chính được gọi là “Main Container” trong khi các khung chứa khác có tên lần lượt là “Container-1”, “Container-2”,... Các dòng lệnh khác nhau đã sẵn có để thay đổi các tên mặc định đó. Main container có những nhiệm vụ đặc biệt sau:

- Quản lý bảng khung chứa (container table - CT), nơi đăng ký các tham chiếu của đối tượng và các địa chỉ giao dịch của tất cả các khung chứa có trong platform;
- Quản lý bảng miêu tả agent cục bộ (Global agent descriptor table -GADT), là nơi đăng ký của tất cả các agent trong platform, bao gồm cả trạng thái hiện tại và vị trí của chúng
- Hosting AMS và DF, hai agent đặc biệt cung cấp việc quản lý agent (agent management) và dịch vụ trang trắng (white page service), và dịch vụ trang vàng mặc định của platform (default yellow page service).



Hình 2.1: Các thành phần kiến trúc chính



Hình 2.2: quan hệ giữa các thành phần kiến trúc chính

Một câu hỏi thường gặp là liệu khung chứa chính có phải là nút thắt cổ chai (bottleneck) của hệ thống không. Câu trả lời là không, vì JADE cung cấp một bộ nhớ cho GADT để mỗi container quản lý cục bộ. Thông thường, các hoạt động của platform, không liên quan tới khung chứa chính, mà chỉ liên quan tới bộ nhớ cục bộ và hai khung chứa lưu trữ các agent - là chủ thể và đối tượng của hoạt động (ví dụ, người gửi và người nhận thông điệp). Khi một khung chứa muốn tìm ra nơi mà agent nhận thông điệp cư ngụ, đầu tiên nó tìm kiếm trong LADT của nó (Local agent descriptor table), nếu việc tìm kiếm không đưa lại kết quả, thì khung chứa chính được liên hệ để lấy tham chiếu từ xa phù hợp, sau đó tham chiếu này được lưu vào LADT để sử dụng sau này. Vì hệ thống là động (các agent có thể cư ngụ, chấm dứt, hay agent mới có thể xuất hiện), nên đôi khi chúng có thể sử dụng một giá trị ánh xạ có được từ một địa chỉ vô giá trị. Trong trường hợp này, khung chứa nhận một ngoại lệ và buộc phải làm mới lại bộ nhớ để chống lại khung chứa chính. Chính sách thay thế của bộ nhớ là LRU (least recently use), được thiết kế để tối ưu các cuộc đàm thoại dài, khách quan hơn là cho các cuộc đàm thoại trao đổi thông điệp đơn, rời rạc trong các ứng dụng đa agent.

Tuy nhiên, mặc dù khung chúa chính không phải là nút thắt cổ chai, nhưng nó có **một điểm gây ra lỗi** trong platform. Để quản lý điều này, Jade cung cấp dịch vụ Main Replication Service để đảm bảo JADE platform **vẫn hoạt động một cách đầy đủ ngay cả trong trường hợp main container thất bại**. Với dịch vụ này, bộ phận quản trị có thể điều khiển mức độ chịu đựng lỗi của platform, mức độ co giãn của platform và mức phân tán của platform. Một tầng điều khiển bao gồm một số thể hiện phân tán của khung chúa chính, có thể được cấu hình để cài đặt một hệ thống phân tán và một hệ thống điều khiển phân tán. Trong trường hợp cực đoan, mỗi khung chúa có thể được tạo ra để gia nhập Main Replication Server và hoạt động như một phần của tầng điều khiển.

Định danh của agent được chứa trong Agent Identifier (AID), gồm một tập các khe tuân thủ cấu trúc và ngữ nghĩa được đưa ra bởi FIPA. Các thành phần cơ bản nhất của AID là tên agent và địa chỉ của nó. Tên của agent là định danh toàn cục duy nhất mà JADE xây dựng bằng cách kết hợp nickname được định nghĩa bởi người dùng (được biết như tên cục bộ sử dụng trong giao tiếp intra-platform) với tên của platform. Địa chỉ của agent là địa chỉ giao dịch được kê thừa từ platform, mỗi địa chỉ platform tương ứng với một điểm cuối MTP (Message Transport Protocol), nơi các thông điệp theo chuẩn FIPA có thể được gửi và nhận. Người lập trình agent cũng được phép thêm các địa chỉ giao vận riêng vào AID, khi họ muốn tự cài đặt MTP.

Khi khung chúa chính được khởi chạy, hai agent đặc biệt được tự động khởi tạo và được bắt đầu bởi JADE, vai trò của chúng được định nghĩa bởi chuẩn quản lý agent của FIPA (FIPA Agent Management standard):

- Hệ thống quản lý agent (Agent Management System -AMS) là agent quản lý toàn bộ platform. Nó là điểm kết nối cho tất cả các agent muốn tương tác để truy cập trang trzáng của platform cũng như để quản lý chu trình sống của chúng. Mọi agent phải đăng ký với AMS (được thực hiện một cách tự động bởi JADE lúc agent khởi tạo) để có một AID hợp lệ. Thông tin chi tiết về AMS được trình bày sau này.
- Directory Facilitator (DF) là agent triển khai dịch vụ trang vàng, được sử dụng bởi các agent khi chúng muốn đăng ký các dịch vụ của chúng hoặc tìm kiếm các dịch vụ có sẵn khác. JADE DF cũng chấp nhận các đặc tả từ các agent với mong muốn nhận thông báo bất cứ khi nào có một dịch vụ được đăng ký hay sửa đổi. Nhiều DF có thể được bắt đầu đồng thời để phân tán dịch vụ trang vàng tới nhiều miền khác nhau. Các DF này có thể được hợp nhất thành liên đoàn nếu cần thiết, bằng cách thiết lập các đăng ký (cross-registration) với một DF khác (là DF cho phép truyền bá các yêu cầu của agent tới toàn bộ liên đoàn đó).

2.4 BIÊN DỊCH VÀ CHẠY CHƯƠNG TRÌNH

Tất cả các phần mềm liên quan tới JADE có thể được download từ trang web của JADE: <http://jade.tilab.com>. Phần mềm liên quan tới JADE được chia thành hai mục: phân tán chính

(main distribution) và tích hợp (add ons). Cụ thể, phần tích hợp (add ons) bao gồm các modun tự chúa, cài đặt các đặc trưng mở rộng đặc tả. Trong nhiều trường hợp, chúng không được phát triển bởi đội phát triển của JADE mà được phát triển bởi các thành viên trong cộng đồng mã nguồn mở. Sự phân tán chính (main distribution) bao gồm 5 file archive chính với các nội dung sau:

- *jadeBin.zip* chứa các file archive tiền biên dịch của JADE java đã sẵn sàng trong trạng thái sử dụng được
- *jadeDoc.zip* chứa các tài liệu bao gồm các tài liệu hướng dẫn cho lập trình viên hoặc người quản trị. Tài liệu này cũng có trực tuyến trên trang web.
- *jadeExamples.zip* chứa các mã nguồn của các ví dụ khác nhau
- *jadeSrc.zip* chứa tất cả các mã nguồn của JADE
- *jadeAll.zip* chứa tất cả 4 file trên.

Nếu các file zip trên được download và giải nén, cấu trúc thư mục sẽ giống như Hình 2.3 (chỉ những file và thư mục quan trọng nhất được chỉ ra). Các file/thư mục quan trọng như:

- Giấy đăng kí (license), giấy đăng kí mã nguồn mở, quy định cách sử dụng của phần mềm.
- File *jade/doc/index.html* là điểm bắt đầu tốt nhất cho những người bắt đầu làm quen với jade, bao gồm các liên kết (link) tới các chuyên đề (thematic tutorial), tài liệu hướng dẫn cho lập trình viên và người quản trị, tài liệu javadoc của tất cả các mã nguồn mở, cùng với nhiều tài liệu hỗ trợ khác.
- Thư mục *jade/lib* chứa tất cả các file .jar chứa trong CLASSPATH của java để có thể thực thi jade. Nó bao gồm thư mục con lib/commons- codec, chứa các mã 64bit có trong CLASSPATH của java



Hình 2.3: Cấu trúc thư mục JADE

- Thư mục *jade/src* chứa 4 thư mục con. Thư mục Demo chứa mã nguồn của các ví dụ đơn giản. Thư mục Examples chứa mã nguồn của nhiều ví dụ hữu ích theo các phân đoạn khác

nhau về agent. Thư mục FIPA chứa mã nguồn của một mô đun được định nghĩa bởi FIPA. Thư mục Jade chứa tất cả các mã nguồn của chính nó.

Mã nguồn của jade có thể được biên dịch bằng cách sử dụng công cụ ANT. Các mục tiêu ANT quan trọng nhất là:

- Jade – biên dịch các mã nguồn và tạo các file .class trong thư mục con classes.
- Lib – tạo các file archive của java trong thư mục con lib
- Doc – tạo các file tài liệu javadoc trong thư mục con doc.
- Example – biên dịch tất cả các ví dụ.

Những người có kinh nghiệm có thể tìm thấy nhiều hữu ích khi truy cập trực tiếp vào kho mã nguồn của cộng đồng jade để tìm hiểu khi cần thiết. Nó được duy trì và cải thiện bởi người quản trị; các hướng dẫn về cách truy nhập vào kho mã nguồn đó cũng có trên website của JADE. Thư mục *lib* chứa 5 file archive, trong các file đó có chứa lớp mà JADE cần:

- *jade.jar* chứa tất cả các gói jade ngoại trừ add ons, MTP và các công cụ đồ họa.
- *jadeTool.jar* chứa tất cả các công cụ đồ họa.
- *http.jar* chứa MTP dựa trên HTTP, nó là MTP mặc định khi platform được khởi tạo.
- *iiop.jar* chứa MTP dựa trên IIOP. MTP này không được sử dụng thường xuyên, nhưng là đối tượng nghiên cứu của các ví dụ sau này và nó cài đặt các đặc tả MTP IIOP FIPA.
- *commons-codec/commons-codec-1.3.jar* chứa các mã 64 bit được sử dụng bởi JADE

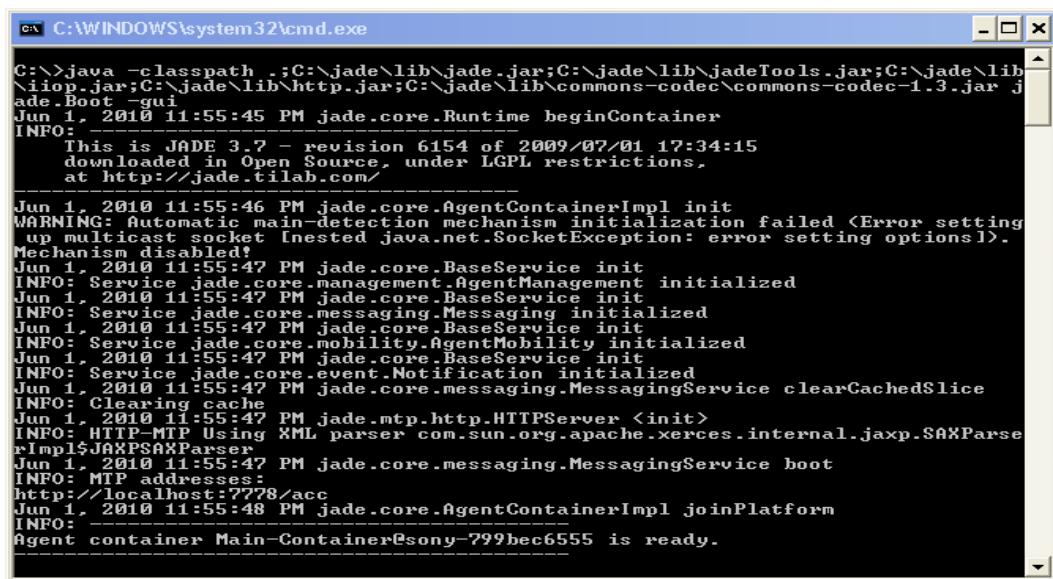
Thư mục *classes* chứa các file class của các ví dụ. Lưu ý, để giảm kích thước của các file phân tán, các ví dụ được phân tán như mã nguồn và do đó, phải được biên dịch trước khi sử dụng các câu lệnh *ant examples*.

Sau khi download JADE và giải nén vào ổ C, ta tạo file runjade.bat ở ổ C với nội dung sau:

```
java -classpath  
. ;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop  
.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-  
codec-1.3.jar jade.Boot -gui
```

Sau khi chạy file runjade.bat ta được kết quả như Hình 2.4

Phần đầu tiên hiển thị thông tin về JADE mỗi lần nó chạy. Sau đó, các dịch vụ chuẩn của JADE được khởi tạo. Thông tin chi tiết về nó được nghiên cứu tiếp ở chương sau. Khi đó, khung chứa chính, và một HTTP MTP được khởi tạo mặc định và địa chỉ cục bộ của nó được hiển thị. Cuối cùng là dòng thông báo khung chứa chính đã sẵn sàng hoạt động, JADE đã sẵn sàng để sử dụng.



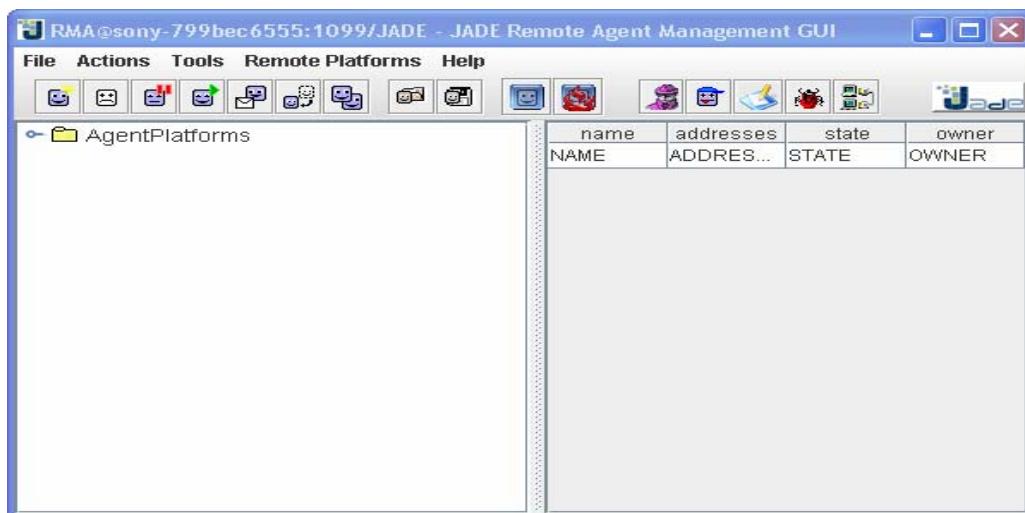
```

C:\>java -classpath .;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar J
ade.Boot -gui
Jun 1, 2010 11:55:45 PM jade.core.Runtime beginContainer
INFO: _____
This is JADE 3.7 - revision 6154 of 2009/07/01 17:34:15
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
Jun 1, 2010 11:55:46 PM jade.core.AgentContainerImpl init
WARNING: Automatic main-detection mechanism initialization failed <Error setting
up multicast socket [nested java.net.SocketException: error setting options]>.
Mechanism disabled!
Jun 1, 2010 11:55:47 PM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
Jun 1, 2010 11:55:47 PM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
Jun 1, 2010 11:55:47 PM jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
Jun 1, 2010 11:55:47 PM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
Jun 1, 2010 11:55:47 PM jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
Jun 1, 2010 11:55:47 PM jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParse
rImpl$JAXPSAXParser
Jun 1, 2010 11:55:47 PM jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://localhost:7778/acc
Jun 1, 2010 11:55:48 PM jade.core.AgentContainerImpl joinPlatform
INFO: _____
Agent container Main-Container@sony-799bec6555 is ready.

```

Hình 2.4: Kết quả của việc khởi chạy JADE

Tham số `-gui` có tác động đến việc khởi chạy giao diện đồ họa của JADE như hình 2.5. GUI này được cung cấp bởi một agent hệ thống của JADE gọi là Remote Monitoring Agent (RMA) và cho phép người quản trị platform thao tác và quản lý platform lúc đang chạy.



Hình 2.5: Giao diện của JADE RMA

Lưu ý, việc sử dụng RMA GUI và tất cả các công cụ đồ họa khác, có thể gây ảnh hưởng tiêu cực tới hiệu năng hoạt động của hệ thống. Đây là nguyên nhân tại sao tùy chọn `-gui` được đưa ra. Nếu hiệu năng hoạt động là yếu, thì không nên sử dụng RMA GUI lúc triển khai.

2.5 CÁC GÓI CỦA JADE

Các gói chính là:

- jade.core cài đặt lõi của JADE, môi trường thời gian chạy phân tán hỗ trợ toàn bộ platform và các công cụ của nó. Nó chứa lớp gốc `jade.core.Agent` cũng như các lớp run-

time cơ bản cần để thực thi các container agent. Nó còn chứa 1 tập các gói con, mỗi gói thực thi 1 sức năng đặc biệt ở mức lõi. Đó là:

- jade.core.event cài đặt dịch vụ thông báo các sự kiện phân tán. Nó giúp cho người dùng thấy được các sự kiện phát sinh bởi các thành phần phân tán khác nhau trong 1 platform;
- jade.core.management cài đặt dịch vụ quản lý vòng đời agent phân tán;
- jade.core.messaging cài đặt dịch vụ phân tán thông điệp;
- jade.core.mobility cài đặt dịch vụ nhân bản và di chuyển agent, bao gồm việc truyền cả trạng thái và mã nguồn của 1 agent;
- jade.core.nodeMonitoring cho phép các container quản lý lẫn nhau và phát hiện các container không quản lý được hoặc đã chết;
- jade.core.replication cho phép tái tạo 1 main container mới nếu có lỗi nghiêm trọng trong main container ban đầu.
- jade.core.behaviors là 1 gói con của jade.core chứa 1 hệ thống các hành vi lỗi độc lập với ứng dụng. Một hành vi JADE biểu diễn 1 tác vụ mà 1 agent thực hiện, được mô tả trong Phần 4.2.
- jade.content và các gói con của nó chứa 1 tập các lớp hỗ trợ lập trình viên tạo và thao tác với các biểu thức nội dung phức tạp theo 1 ngôn ngữ nội dung cho trước và ontology. Nó chứa tất cả các cấu trúc được mã hóa để tự động chuyển đổi giữa cách biểu diễn nội dung JADE và định dạng truyền nội dung thông tin điệp theo FIPA. Phần 5.1 mô tả chi tiết các chức năng được cài đặt trong gói này.
- jade.domain chứa phần cài đặt của các agent AMS và DF, theo như chuẩn FIPA, cùng với các mở rộng đặc tả JADE của chúng sẽ được nói đến sau. Mỗi gói con chứa các lớp biểu diễn các thực thể khác nhau của 1 ontology JADE. Các ontology này được liệt kê trong Bảng 2.1.

Bảng 2.1: các ontology đã định nghĩa sẵn trong JADE

Ontology	Gói	Miêu tả
FIPA-Agent-Management	Jade.domain.FIPAAgentManagement	Các thực thể các ngoại lệ và các hành động cần thiết để tương tác với AMS và DF theo các đặc tả của FIPA
JADE-Agent-Management	Jade.domain.JADEAgentManagement	Mở rộng của JADE đối với ontology FIPA-Agent-Management

JADE-Introspection	Jade.domain.introspection	Mở rộng của JADE liên quan đến việc giám sát các sự kiện của platform
JADE-Mobility	Jade.domain.mobility	Mở rộng của JADE liên quan đến tính di động của agent
JADE-Persistence	Jade.domain.persistence	Mở rộng của JADE liên quan đến việc lưu trữ agent lâu dài
DEApplet-Management	Jade.domain.DFGUIManagement	Ontology sử dụng bởi DF GUI để tương tác với DF. Nó cho phép nhiều giao diện đồ họa của cùng một DF, kể cả các giao diện cài đặt applet

- jade.gui chứa 1 vài thành phần Java chung và các icon dùng để xây dựng các giao diện GUI dựa trên Swing dùng cho các agent JADE. Gói này cung cấp các thành phần đồ họa sẵn dùng để biểu diễn các khái niệm trừu tượng JADE điển hình, như AID, ACLMessage, và AgentDescription.
- jade.imtp chứa các cài đặt JADE IMTP (Giao thức truyền thông didepj bên trong). Về nguyên tắc, gói con jade.imtp.rmi là IMTP mặc định của JADE dựa trên Java RMI.
- jade.lang.acl chứa hỗ trợ cho FIPA ACL (Ngôn ngữ giao tiếp agent) chứa lớp ACLMessage, chương trình phân tích, mã hóa, và 1 lớp hỗ trợ các mẫu biểu diễn của các thông điệp ACL.
- jade.mtp chứa 1 tập các giao diện Java nên được cài đặt bởi JADE MTP. Nó còn chứa 2 gói con với 1 gói cài đặt dựa trên giao thức HTTP (là cài đặt mặc định) và 1 gói dựa trên giao thức IIOP.
- jade.proto chứa các cài đặt của 1 vài giao thức tương tác chung, trong đó 1 số được đặc tả bởi FIPA. Phần 5.4 cung cấp mô tả về 1 số giao thức tương tác.
- jade.tools chứa cài đặt của tất cả các công cụ đồ họa JADE, được mô tả trong Phần 2.7.
- jade.util chứa 1 số lớp hữu dụng khác.
- jade.wrapper cùng với các lớp jade.core.Profile và jade.core.Runtime cung cấp hỗ trợ giao diện đang chạy JADE cho phép các ứng dụng Java bên ngoài sử dụng JADE như 1 thư viện, được mô tả trong Phần 5.6.
- FIPA là 1 gói chứa module IDL (Ngôn ngữ định nghĩa giao diện) đặc tả bởi FIPA dùng cho MTP dựa trên IIOP.

2.6 DỊCH VỤ VẬN CHUYỂN THÔNG ĐIỆP

Theo các đặc tả FIPA, một Dịch vụ vận chuyển thông điệp (MTS) là 1 trong 3 dịch vụ quan trọng mà mọi nền tảng agent phải cung cấp (2 dịch vụ khác là Dịch vụ quản lý MS và Xúc tiến thư mục DS). Một MTS quản lý tất cả các thông điệp trao đổi bên trong và giữa các platform.

2.6.1 Các giao thức truyền thông điệp

Để hỗ trợ tương tác giữa các platform khác nhau (ví dụ, với platform không phải của JADE), JADE cài đặt tất cả các chuẩn Giao thức vận chuyển thông điệp (MTP) định nghĩa bởi FIPA, nơi mỗi MTP bao gồm 1 định nghĩa giao thức vận chuyển và 1 chuẩn mã hóa của phong bì chứa thông điệp. Như trong Hình 2.4, một trong các thông điệp được đưa ra đầu ra chuẩn khi chạy main container có dạng tương tự như sau:

INFO: MTP addresses:

<http://anduril:7778/acc>

Mặc định, JADE luôn luôn khởi động bằng 1 MTP dựa trên HTTP được khởi tạo trong main container, không MTP nào được hoạt động trong các container thường. Nó tạo ra một server socket trên host main container và lắng nghe các kết nối qua HTTP tại URL đặc tả trong dòng lệnh thứ 2 phía trên. Khi 1 kết nối tới được xác định và thông điệp hợp lệ được nhận từ kết nối, MTP sẽ gửi thông điệp đến đích cuối cùng, thường là 1 agent nằm trong platform phân tán. Phía trong, platform sử dụng 1 giao thức vận chuyển tên là IMTP (Giao thức truyền thông điệp bên trong) sẽ được mô tả trong phần tiếp theo. JADE thực hiện việc truyền thông điệp cho cả các thông điệp đến và đi sử dụng 1 bảng đơn bước yêu cầu IP trực tiếp giữa các container.

Sử dụng các lệnh tùy chọn, vô số MTP có thể hoạt động trong mỗi JADE container, bao gồm cả các MTP cài đặt các giao thức truyền khác nhau. MTP còn thể được 'nhúng' và khởi tạo tại thời gian chạy nhờ sử dụng RMA GUI. Nó cho phép người quản lý platform tự do thao tác với topo bằng cách, chẳng hạn, cách ly 1 host với 1 kết nối mở tới 1 mạng bên ngoài nhằm tăng cường bảo mật. Khi 1 MTP hoạt động trong 1 platform, JADE platform sẽ nhận được 1 địa chỉ truyền mới, 1 đầu mút nơi các thông điệp có thể nhận. Địa chỉ này còn được thêm vào cấu trúc dữ liệu sau:

- Thông tin platform, có thể đọc từ AMS nhờ lệnh get-description.
- Toàn bộ các đối tượng ams-agent-description chứa trong kho chứa AMS, có thể đọc nhờ 1 lệnh tìm kiếm.
- Định danh agent (AID) cục bộ của bất kỳ agent trong bất kỳ container nào có thể đọc nhờ phương thức getAID() của lớp Agent.

Giao diện public Java jade.mtp.MTP cho phép hát triển MTP tùy chọn cho các yêu cầu ứng dụng và môi trường mạng cụ thể. Giao diện MTP mô hình 1 kênh song hướng có thể vừa gửi và nhận các thông điệp ACL bằng cách kế thừa giao diện jade.mtp.OutChannel và jade.mtp.InChannel là

các kênh 1 hướng. Giao diện jade.mtp.TrasnportAddress chỉ đơn giản biểu diễn 1 URL cung cấp truy nhập đến các trường như giao thức, host, cổng và tệp. Khi các MTP được liệt kê trong Bảng 2.2 có trong public domain, mỗi MTP được triển khai dưới dạng các tệp jar riêng lẻ.

Bảng 2.2: Các giao thức truyền trong JADE hiện nay

Giao thức truyền	Mã hóa thông điệp	Nhà phát triển
HTTP và HTTPS	XML	Đại học Autonoma ở Barcelona, Tây Ban Nha (UAB)
IIOP (bản cài đặt của Sun)	CORBA IDL	Nhóm JADE
IIOP (bản cài đặt của CORBA IDL ORBacus)	CORBA IDL	Giovanni Rimassa, Đại học Parma, Italia
JMS	Cấu trúc dữ liệu Java	Edward Curry, đại học Galway
Jabber XMPP	Cấu trúc dữ liệu Java	Đại học Politecnica ở Valencia, Tây Ban Nha

Trong khi HTTP và IIPO MTP được đính kèm trong bản phân phối chính JADE, các giao thức còn lại đều phải download dưới dạng bản add-on tại trang chủ JADE. HTTP là MTP mặc định để chạy với main container. HTTP được chọn làm MTP mặc định vì bản cài đặt cung cấp bởi UAB có những ưu điểm sau đây:

- Số cổng cục bộ của các kết nối đến và đi có thể được chọn cho cấu hình firewall sử dụng các biến jade_mtp_http_port và jade_mtp_http_outPort.
- Proxy có thể được cấu hình thông qua các kết nối bát biến: thay vì thực hiện bắt tay TCP với mỗi thông điệp, các kết nối có thể được lưu lại và sử dụng lại khi các thông điệp được trao đổi thường xuyên giữa 2 platform khác nhau. Điều này được điều khiển với các biến jade_mtp_http_numKeepAll và jade_mtp_http_timeout.
- HTTPS có thể được sử dụng để thiết lập bảo mật và các kênh chứng thực giữa các platform. Để sử dụng HTTPS, 1 địa chỉ truyền đi phải đơn giản bắt đầu bằng https. Tất nhiên phải lưu ý rằng, mặc dù HTTPS tăng cường bảo mật, nó vẫn gặp phải 1 số khuyết điểm khi thực hiện; ước tính sơ lược cho thấy rằng HTTPS MTP chậm hơn 15% so với HTTP MTP chuẩn.

JADE RMA cho phép quản lý MTP linh hoạt bằng việc cho phép kích hoạt hoặc tắt chúng trong khi platform đang chạy. Nhấn chuột phải vào 1 nút cây container agent trong bảng bên trái của RMA GUI sẽ hiện ra 1 menu trong đó có 2 mục là Install a new MTP (Cài đặt MTP mới) và Unistall an MTP (Gỡ bỏ MTP). Lựa chọn đầu sẽ tạo ra 1 cửa sổ để người dùng chọn MTP mới để cài đặt, tên đầy đủ của lớp cài đặt giao thức, và (nếu giao thức được chọn hỗ trợ) địa chỉ truyền lắng nghe được ưu tiên. Nếu chọn Unistall an MTP, 1 cửa sổ sẽ xuất hiện để người dùng có thể chọn MTP trong danh sách đang hoạt động để gỡ nó ra khỏi platform.

Một vài ứng dụng có thể không cần tới các giao tiếp bên ngoài platform cục bộ. Trong trường hợp đó, lệnh tùy chọn –nompt sẽ tạo ra 1 HTTP MTP mặc định trong main container. Tuy nhiên nó sẽ cách ly platform khỏi mọi giao tiếp với các platform từ xa. Lưu ý rằng 1 container từ xa chỉ là 1 container mà không nằm chung host với main container, nhưng vẫn nằm chung platform; nói cách khác, 1 container từ xa không được là 1 phần của 1 platform từ xa. Các container trong cùng platform luôn giao tiếp bằng JADE IMTP.

2.6.2 Giao thức truyền thông điệp nội bộ (IMTP)

JADE IMTP (Giao thức truyền thông điệp nội bộ) chuyên dùng để trao đổi thông điệp giữa các agent sống trong các container khác nhau trong cùng 1 platform. Nó tương đối khác với các MTP ngoài platform, như HTML. Thứ nhất, vì nó chỉ được dùng cho việc giao tiếp bên trong platform, nên không cần phải tương thích với các chuẩn FIPA; nó có thể thuộc và do đó được thiết kế để hỗ trợ việc vận hành platform. Thực tế JADE IMTP không chỉ được dùng để truyền thông điệp mà còn truyền các lệnh bên trong cần thiết để quản lý platform phân tán, cũng như giám sát trạng thái của các container từ xa. Ví dụ, nó được sử dụng để truyền lệnh tắt 1 container, cũng như giám sát việc 1 container bị tắt hoặc nằm ngoài kiểm soát.

JADE được thiết kế để cho phép lựa chọn IMTP trong thời gian platform chạy. Hiện tại, đã có 2 cách cài đặt ITMP chính. Một cách dựa trên Java RMI và là tùy chọn mặc định. Cách thứ hai dựa trên 1 giao thức sử dụng TCP socket giúp loại bỏ đi sự thiếu sót hỗ trợ Java RMI trong môi trường J2ME; nó được khởi động mặc định khi chạy platform JADE LEAP và sẽ được mô tả trong Chương 8. Cả 2 cách cài đặt này đều cung cấp các lựa chọn cấu hình cho phép điều chỉnh IMTP theo mạng và các thiết bị nhất định.

2.6.2.1 Giao thức truyền thông điệp nội bộ theo chuẩn RMI (RMI-IMTP)

RMI-IMTP được cài đặt bởi gói jade.imtp.rmi. Khi main container khởi động, nó sẽ tìm 1 đăng ký RMI trong host cục bộ và gọi đến các đối tượng tham chiếu; nếu không tìm thấy, nó sẽ tạo ra 1 đăng ký mới. Khi 1 container thường khởi động, nó sẽ xác định đăng ký RMI trên host đặc tả main container và tìm đối tượng tham chiếu của main container. Sau đó nó sẽ gọi phương thức từ xa addNode() của main container để tham gia platform và đăng ký tham chiếu của nó với main container.

Các thông điệp agent và thông tin điều khiển hệ thống được trao đổi giữa các container được cài đặt thông qua 1 mẫu lệnh khi nút yêu cầu (ví dụ, 1 container) tạo ra 1 đối tượng Command và truyền đi đối tượng này, với 1 yêu cầu thực thi, đến nút thực thi.

Hai biến dòng lệnh sau có sẵn trong RMI-IMTP:

-host <hostName>

sẽ đặc tả host đang cung cấp main container để đăng ký với nó; giá trị mặc định là localhost. Lựa chọn này cũng được sử dụng khi chạy main container để override giá trị của localhost, ví dụ để đọc toàn bộ tên miền của host với –host anduril.cselt.it khi localhost chỉ trả về là 'anduril'.

- port <portNumber>

sẽ đặc tả số cổng mà đăng ký RMI được tạo ra bởi main container để nhận các yêu cầu tìm kiếm. Giá trị mặc định là 1099.

2.7 CÁC CÔNG CỤ QUẢN TRỊ VÀ GỠ LỖI

Các ứng dụng đa agent thường khá phức tạp. Chúng thường được phân tán qua một số host. Chúng được tạo thành từ hàng trăm tiến trình đa luồng (ví dụ các container với một số agent, mỗi agent có một luồng của nó); chúng là động nghĩa là các agent có thể xuất hiện, biến mất và di trú. Các khía cạnh này dẫn đến những khó khăn trong việc quản lý và đặc biệt trong việc gỡ lỗi. Để giảm đi những khó khăn này, JADE có một dịch vụ thông báo sự kiện được tạo thành từ cơ sở của JADE RMA management console và một tập các công cụ đồ họa nó được cung cấp để giúp đỡ trong pha quản lý và gỡ lỗi. Tất cả các công cụ này được đóng gói trong jadeTools.jar.

Trong phần này miêu tả một số công cụ được cung cấp với JADE distribution, Event Notification Service (ENS) và mô hình các công cụ JADE. Các thông tin bổ sung được cung cấp để giúp đỡ người sử dụng với tạo ra các công cụ của chính họ. Để giải thích các công cụ nền tảng, đoạn code sau minh họa một HelloWorldAgent đơn giản thực thi cyclic behaviour: mỗi lần một thông điệp được nhận, nó in thông điệp ra output chuẩn và hồi âm cho người gửi với thông điệp "Hello!".

```
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
public class HelloWorldAgent extends Agent {
    public void setup() {
        System.out.println("Hello. My name is "+getLocalName());
        addBehaviour(new CyclicBehaviour() {
            public void action() {
                ACLMessage msgRx = receive();
                if (msgRx != null) {
                    System.out.println(msgRx);
                    ACLMessage msgTx = msgRx.createReply();
                    msgTx.setContent("Hello!");
                    send(msgTx);
                } else {
                    block();
                }
            }
        });
    }
}
```

Biên dịch agent

Tạo file compilejade.bat lưu ở ô C với nội dung sau:

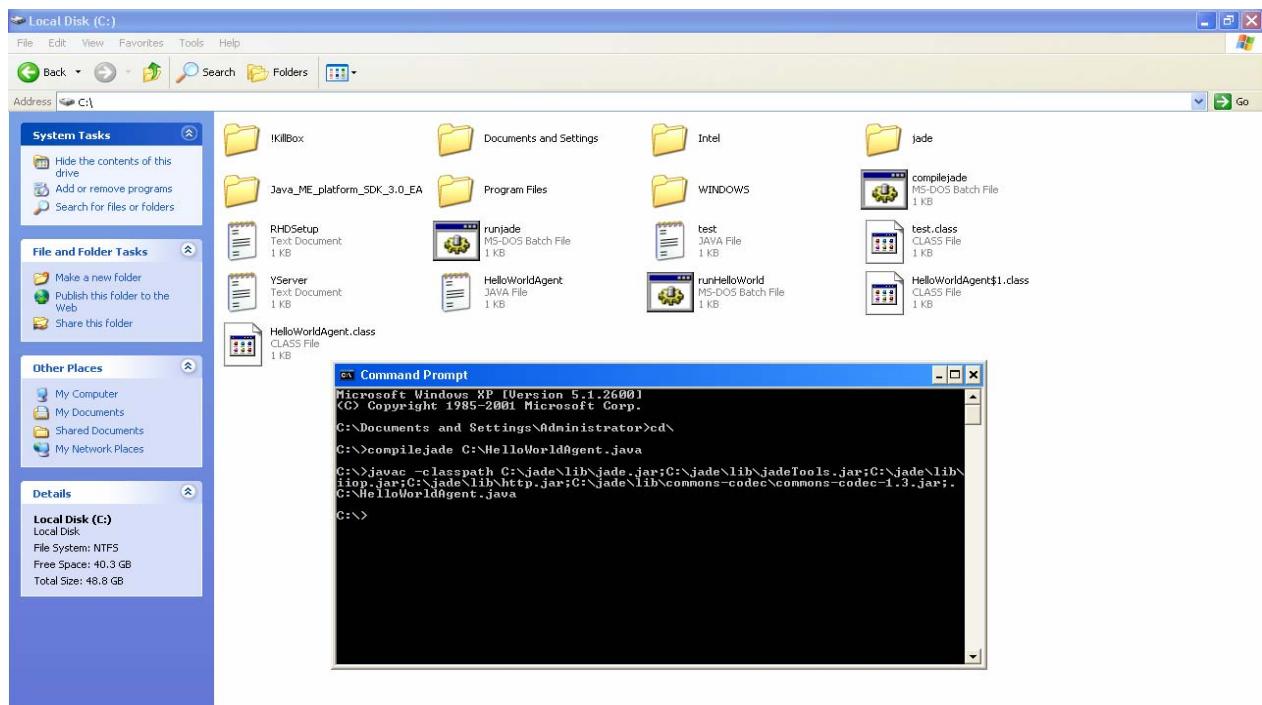
```
javac -classpath
```

```
C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar;. %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Lưu file HelloWorldAgent.java với nội dung như trên ở ô C

Biên dịch file HelloWorldAgent.java bằng cách gõ lệnh sau:

```
compilejade C:\HelloWorldAgent.java
```

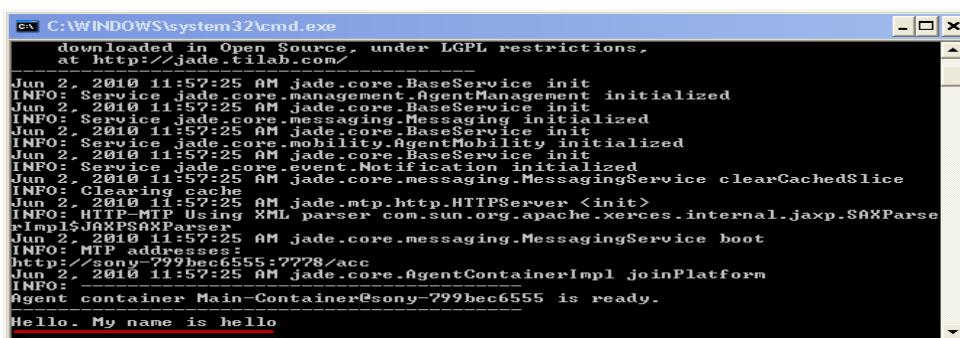


Tạo file runHelloWorld.bat với nội dung sau:

```
java -classpath
```

```
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar jade.Boot hello>HelloWorldAgent
```

Chạy file runHelloWorld.bat ta có kết quả sau:



2.7.1 Cửa sổ quản trị JADE

JADE RMA (Remote Monitoring Agent) là một công cụ hệ thống thực thi một giao diện quản lý nền tảng đồ họa. Công cụ được thực thi bởi lớp jade.tools.rma.rma nhưng nó thường được bắt đầu trực tiếp từ dòng lệnh sử dụng tuỳ chọn –gui. Nó cung cấp một giao diện đồ họa để giám sát và quản lý nền tảng JADE phân tán được tạo thành từ một hoặc một số host và các nút container. Nó bao gồm một menu “Tools” qua đó các công cụ khác có thể được khởi hoạt. Một số RMA có thể được khởi hoạt trong cùng một nền tảng nếu một tên agent khác được đăng kí cho mỗi thẻ hiện.

Tại lúc khởi động RMA agent đăng kí với AMS để được thông báo tất cả các sự kiện cấp nền tảng; Hình 2.5 hiện thị giao diện sử dụng đồ họa của nó. Panel trái cung cấp cái nhìn của topo nền tảng được biểu diễn như một cây của các container các lá là các agent. Panel này được thực thi bởi lớp jade.gui.AgentTree và được sử dụng lại bởi tất cả các công cụ khác. Nói cụ thể, có 3 kiểu của nút: agent platform, container và agent. Với mỗi nút, chu kì sống của thực thể được biểu diễn được điều khiển thông qua menu sổ xuống xuất hiện bằng việc click chuột phải vào nút đó.

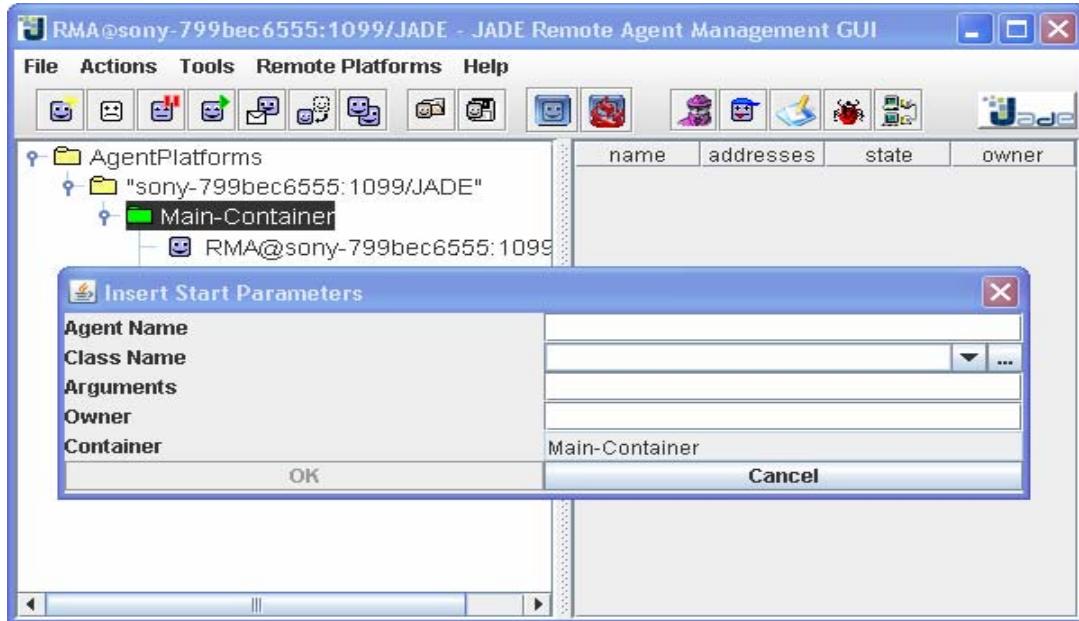
Nếu một agent được chọn, menu sổ xuống cho phép agent được treo (suspend), hồi phục lại (resume), giết (kill), tạo bản sao (clone), lưu (saved), đóng băng (frozen) hoặc di chuyển đến một container khác. Nó cũng cho phép cấu hình và gửi một thông điệp tuỳ chỉnh, đặc biệt

Nếu một container được chọn, menu sổ xuống cho phép tạo một agent mới, tải một agent đang tồn tại, cài đặt hoặc xoá bỏ một MTP, lưu/tải container bao gồm tất cả các agent của nó và kết thúc container. Sử dụng RMA đã khởi hoạt như ví dụ trong phần trước, chúng ta có thể cố gắng tạo một HelloWouldAgent mới, gọi Bill như hiện thị trong Hình 3.6

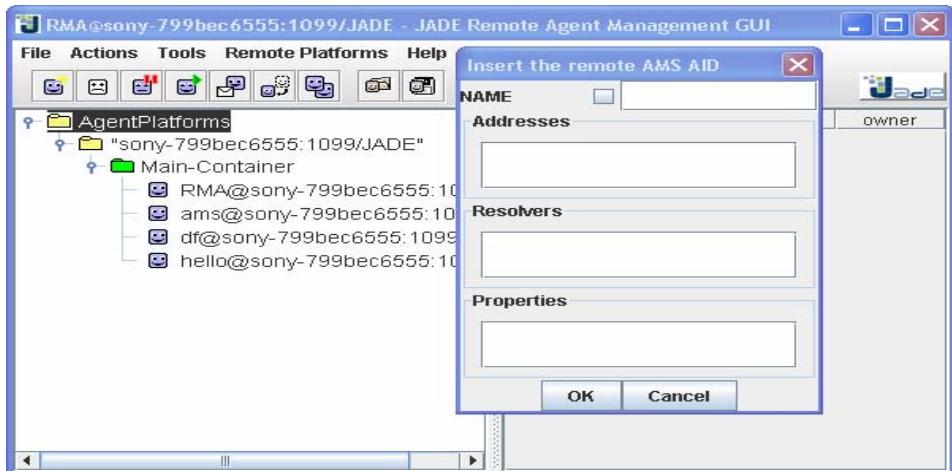
Nếu một platform được chọn, menu pop-up cho phép platform profile được hiện thị, ví dụ như cấu trúc dữ liệu, được gọi là AP (Agent Platform), nó miêu tả một FIPA – compliant platform và liệt kê tên nền tảng và các dịch vụ của nó. Menu cũng cho phép quản lý platform MTP, ví dụ như cài đặt và xoá các MTP trong/tù các container được chỉ ra.

Chú ý rằng gốc của cây được gọi là “Agent Platforms” trong số nhiều. Nó biểu thị sự thật rằng RMA có thể được sử dụng để điều khiển một tập platform được cung cấp chúng là tất cả FIPA – compliant. Tất nhiên, mức độ của điều khiển được giới hạn khi việc tương tác với một platform ở xa khi đó chỉ việc quản lý các thông điệp và action được định nghĩa trong FIPA có thể được sử dụng, thay vì thông qua JADE IMTP trong bất kì JADE platform nào. Ví dụ, có thể xem AP Description của một platform ở xa và danh sách các agent active của nó. Tuy nhiên, khi sự truy cập container không được chỉ ra bởi FIPA, tree view của platform từ xa không thể trình bày một cách trực tiếp. Để giao tiếp với một platform ở xa, nhận dạng của AMS của nó phải được cung cấp (ví dụ: AMS AID), nó phải bao gồm tên và ít nhất 1 địa chỉ truyền (transport address) hợp lệ. Điều này hiện thị trong Hình 2.7, bằng việc chọn RMA agent và việc yêu cầu nó giao tiếp với AMS cục bộ của bạn như nếu nó là một platform ở xa: sau khi pop – up menu, khi được yêu

cầu để chèn AID của AMS, kiểu “ams” và kiểm tra hộp để chỉ ra rằng nó không là một GUID. Một platform thứ 2 nên xuất hiện trong cây platform, một cách ngẫu nhiên, có AP Description và danh sách các agent tương tự như platform của bạn.



Hình 2.6: Giao diện để chạy agent mới



Hình 2.7: Giao tiếp với platform từ xa

2.7.2 Dummy agent

Dummy agent là một công cụ rất đơn giản hữu dụng cho việc gửi các tác nhân kích thích theo dạng các ACL thông điệp tùy chỉnh để kiểm tra hành vi của các agent khác. Nó được thực thi bởi lớp `jade.tools.DummyAgent.DummyAgent`. Khả năng của nó là gửi và nhận các thông điệp tùy chỉnh có thể được tạo ra sử dụng một GUI đơn giản và được tải/lưu từ/vào một file. Khi một ứng dụng agent được khởi hoạt, một Dummy Agent có thể được sử dụng để giả vờ nó bằng việc gửi các thông điệp được người dùng chỉ ra và việc phân tích các phản ứng của nó trong thời hạn các thông điệp được nhận. Nó là một công cụ đơn giản nhưng hiệu quả được sử dụng rộng rãi trong

suốt quá trình phát triển ứng dụng. Hình 2.8 hiển thị Dummy Agent GUI với panel bên phải dành để hiện thị danh sách các thông điệp gửi và nhận. Panel bên trái sử dụng để tạo ra các thông điệp tùy chỉnh. Cả 2 panel và các thành phần khác được sử dụng lại và được cung cấp như là các lớp riêng biệt; cụ thể là jade.gui.AclGui và jade.gui.AIDGui là các lớp hữu ích cho việc tạo/hình dung một ACL thông điệp và một AID.

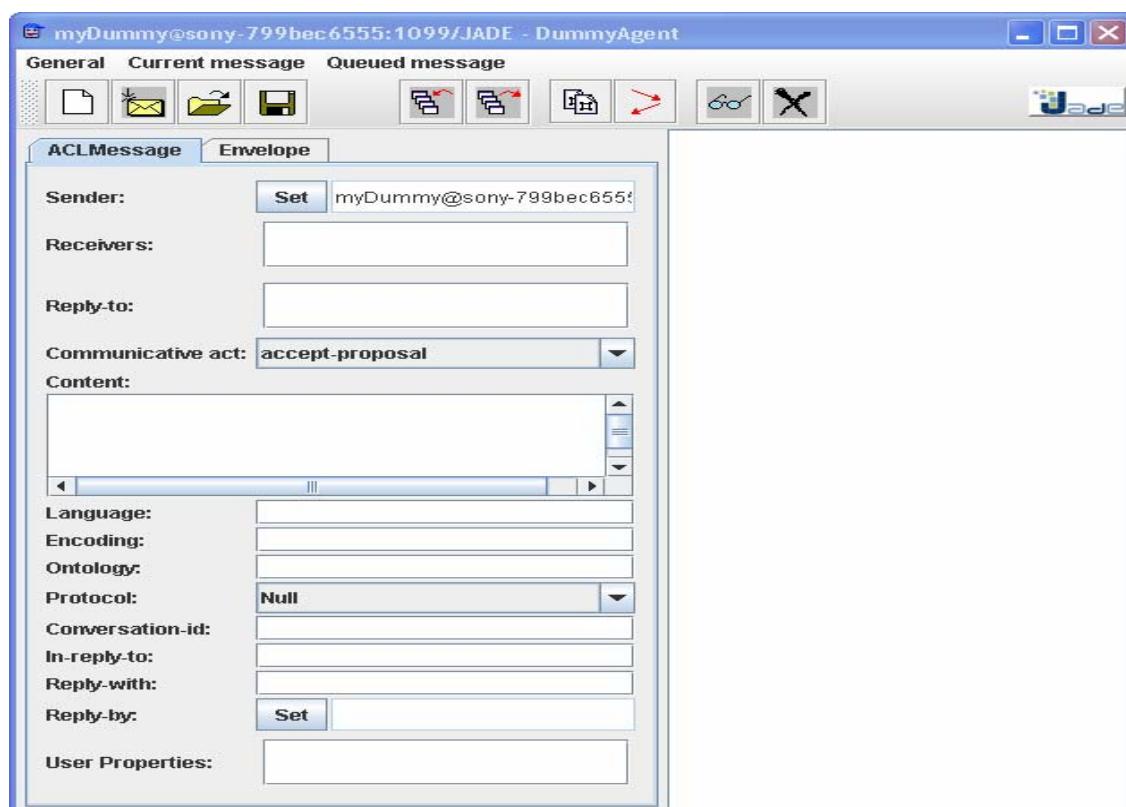
Chạy DummyAgent từ dòng lệnh:

Tạo file runDummyAgent.bat lưu ở ô C với nội dung:

```
java -classpath
```

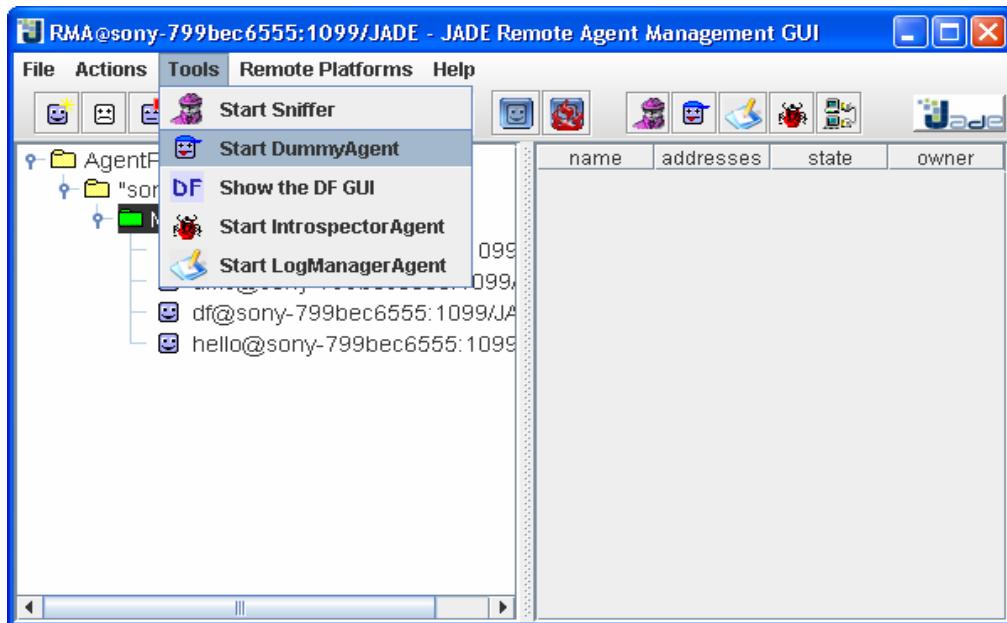
```
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar jade.Boot  
myDummy:jade.tools.DummyAgent.DummyAgent
```

Chạy file runDummyAgent.bat ta được kết quả sau:



Hình 2.8: Kết quả chạy DummyAgent từ dòng lệnh

Chạy DummyAgent từ giao diện quản trị platform:



Hình 2.9: Kết quả chạy DummyAgent từ giao diện quản trị platform

2.7.3 Sniffer Agent

Trong khi tất cả các công cụ khác phần lớn được sử dụng cho việc gõ lỗi một agent đơn, công cụ này được sử dụng rộng rãi cho việc gõ lỗi, hoặc đơn giản là viết các cuộc nói chuyện giữa các agent. Nó được thực thi bởi lớp jade.tools.sniffer.Sniffer. “sniffer” đăng kí với platform AMS để được thông báo tất cả các sự kiện của platform và tất cả các sự trao đổi thông điệp giữa một tập các agent xác định. Hình 2.10 hiện thị GUI của Sniffer Agent. Panel trái là trình duyệt tương tự như RMA, nhưng được sử dụng cho việc duyệt agent platform và việc chọn các agent được sniff. Phần bên phải cung cấp biểu diễn đồ họa của các thông điệp được trao đổi giữa các agent được sniff, nơi mỗi mũi tên biểu diễn một thông điệp và mỗi màu xác định một cuộc nói chuyện.

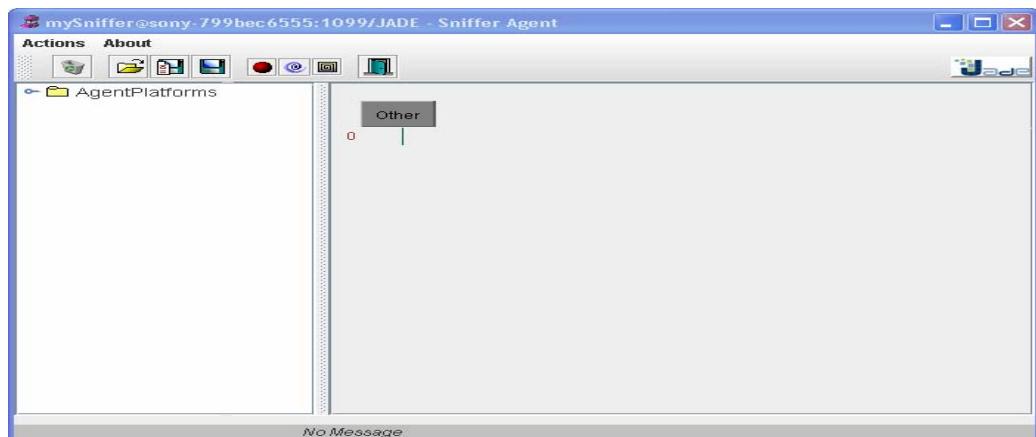
Khi người sử dụng quyết định sniff một agent hoặc một nhóm các agent, mỗi thông điệp gửi đi hoặc đến, agent/nhóm được lưu vết và được hiện thị trong sniffer GUI. Người sử dụng có thể chọn và xem chi tiết của mỗi thông điệp, lưu thông điệp vào đĩa như một file văn bản hoặc serialize một cuộc nói chuyện như một file nhị phân

Chạy SnifferAgent từ dòng lệnh:

Tạo file runSnifferAgent.bat lưu ở ổ C với nội dung:

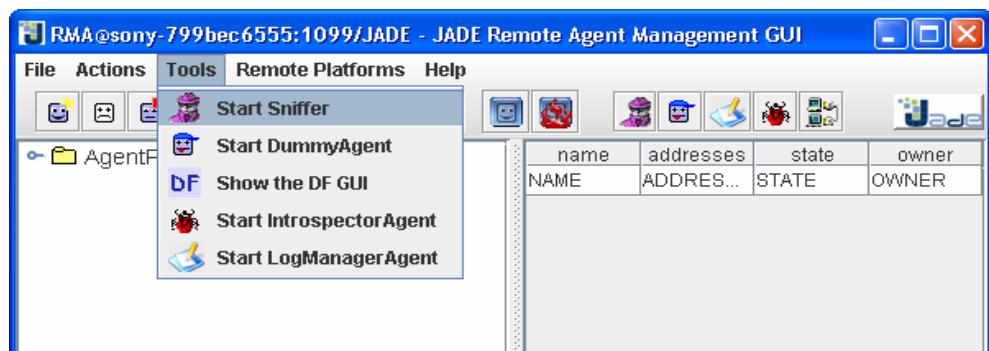
```
java -classpath
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar jade.Boot mySniffer;jade.tools.sniffer.Sniffer
```

Chạy file này ta có kết quả:

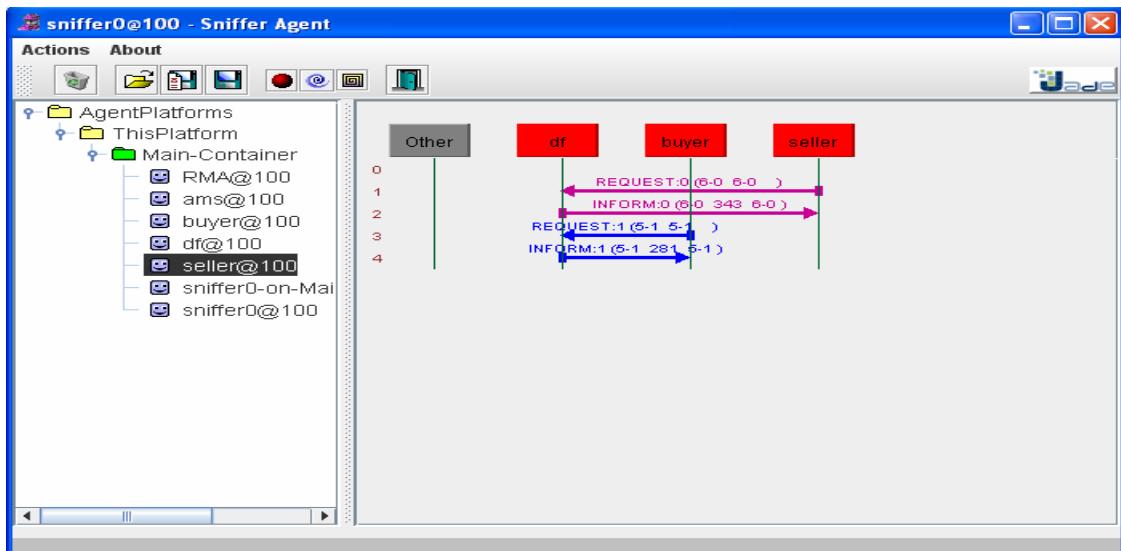


Hình 2.10: Giao diện của SnifferAgent chạy bằng dòng lệnh

Chạy SnifferAgent từ giao diện quản trị platform:



Hình 2.11: Chạy SnifferAgent từ giao diện quản trị platform



Hình 2.12: Giao diện SnifferAgent của ví dụ bookTrading

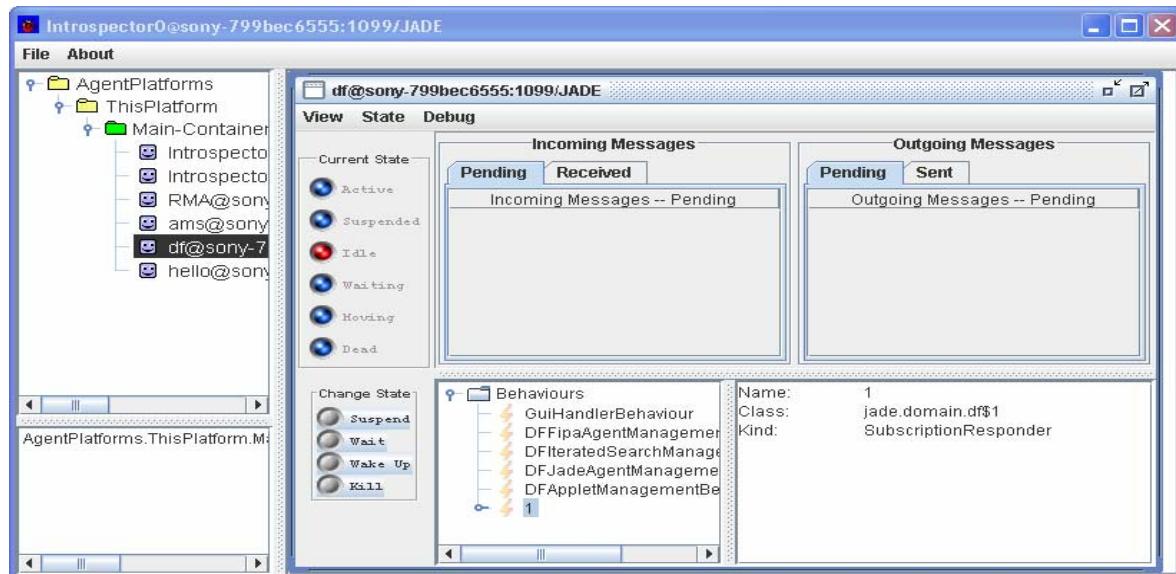
2.7.4 Introspector agent

Trong khi Sniffer Agent có ích trong việc đánh hơi, giám sát và debug các cuộc hội thoại giữa các agent, thì Introspector Agent được dùng để debug hành vi của một agent. Công cụ này cho

phép giám sát và điều khiển vòng đời agent, hàng đợi thông điệp gửi đi và nhận được của agent. Nó cũng cho phép giám sát hàng đợi các hành vi đã được lập lịch sẵn bao gồm khả năng thực thi các hành vi từng bước một. Chú ý một bước của hành vi là một lần thực thi phương thức *action ()* của lớp Behaviour và không được liên kết một cách trực tiếp với việc thực thi mã Java. Tóm lại, công cụ này cho phép giám sát việc thực thi của agent, cụ thể là những hành vi nào được thực thi, những hành vi nào được đưa vào hàng đợi, và cho phép giám sát những phản ứng của chúng đối với kích thích bên trong, cụ thể là các thông điệp sắp nhận được. Hình 2.13 biểu diễn giao diện của Introspector Agent khi đang giám sát agent DF.

2.7.5 Log Manager Agent

Log Manager Agent là công cụ đơn giản hóa việc quản lý động và phân tán khả năng logging (logging facility) bằng cách cung cấp một giao diện người dùng cho phép các mức logging của mỗi thành phần trong JADE platform được thay đổi đúng lúc, nghĩa là bao gồm mọi thành phần đang được xử lý tại các node ở xa, gồm cả các thông điệp logging của một ứng dụng cụ thể. Bộ quản lý log khai thác khả năng ngầm của thư viện *java.util.logging* mà JADE logging dựa trên đó. Mỗi lớp sử dụng một thẻ hiện của lớp *Logger*. Mỗi đối tượng *Logger* có thể được cấu hình ở một mức logging và có một tập các *Handler*. Việc cấu hình này là tinh bǎng cách xác định file cấu hình *java.util.logging* tại thời điểm nạp hoặc động bằng cách sử dụng Log Manager Agent.



Hình 2.13: Giao diện của Introspector Agent khi đang giám sát agent DF.

Ví dụ, lệnh sau đây khởi động một container trong JADE và xác định một file cấu hình để khởi tạo hệ thống logging của JVM:

Đầu tiên ta chạy file runjade.bat với nội dung sau:

```
java -classpath
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\co
mmons-codec\commons-codec-1.3.jar jade.Boot -gui
```

Sau đó, ta tạo file logging.properties với nội dung sau:

```
handlers = java.util.logging.ConsoleHandler
.level = OFF
jade.core.messaging.level = FINEST
```

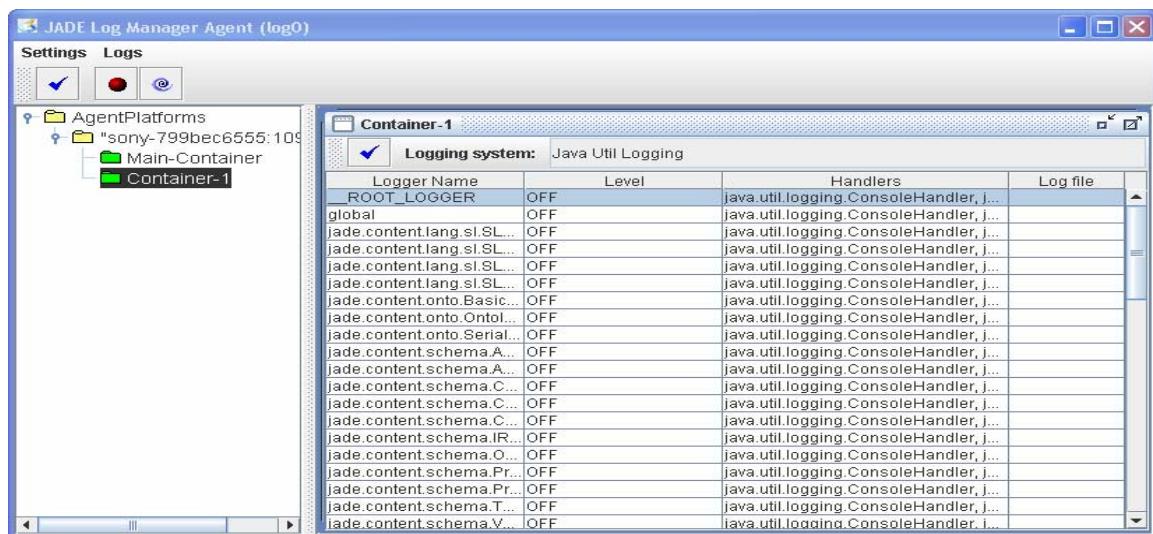
Đây là file cấu hình tuân theo định dạng chuẩn *java.util.Properties*. Xem tài liệu javadoc về lớp *java.util.logging.LogManager* để biết thông tin chi tiết về file cấu hình và định dạng của nó. Container đã được tạo trong JADE sẽ chỉ ghi lại các thông điệp log của hệ thống Messaging, cụ thể là mọi lớp trong gói *jade.core.messaging*.

Tạo file createLogFile.bat với nội dung:

```
java -classpath
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\co
mmons-codec\commons-codec-1.3.jar -Djava.util.logging.config.file=logging.properties
jade.Boot -container
```

Tắt cả các file được ở ô C.

Chạy file runjade.bat, file createLogFile.bat. Sau khi container mới được tạo ra trên platform, khởi động Log Manager Agent trên container đó, ta thấy mức độ logging của container này đều được thiết lập là OFF và các handler đều được thiết lập là *java.util.logging.ConsoleHandler*:



Hình 2.14: Giao diện Log Manager Agent của Container-1

2.7.6 Dịch vụ thông báo sự kiện (event notification service) và mô hình công cụ JADE

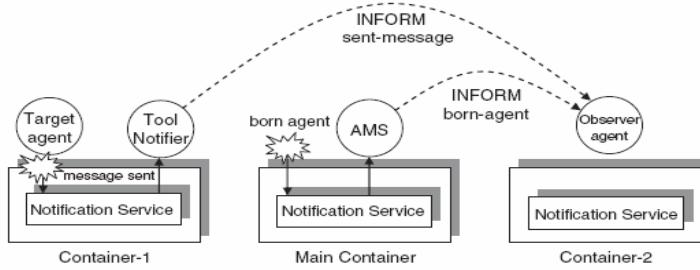
Dịch vụ thông báo sự kiện (ENS) là một dịch vụ ở mức platform quản lý các thông báo phân tán của tất cả các sự kiện được sinh ra bởi mỗi node của platform. Dịch vụ này được gọi là *jade.core.event.Notification*, được cài đặt trong gói *jade.core.event* và được khởi chạy mặc định trong mỗi container. Mỗi khi có một sự kiện được sinh ra bởi một container, nó sẽ bị chặn bởi ENS và được định tuyến tới mọi agent đã đặt trước để được thông báo về các kiểu sự kiện. Nếu không có agent nào đặt trước thì ENS có hiệu năng không đáng kể. Thực tế, những node có hiệu năng thấp (do việc thông báo sự kiện) là container nơi cư trú của các agent đã đặt trước và là container sinh ra sự kiện được thông báo. Vì tất cả các agent công cụ có thể hoạt động khi cần, thậm chí tại thời điểm chạy trong quá trình vận hành platform, việc cải thiện hiệu năng có thể đạt được bằng cách chỉ bắt đầu chúng khi cần thiết. Có 4 loại sự kiện chính:

- (1) Sự kiện liên quan đến vòng đời, còn được gọi là sự kiện kiểu platform (*platform-type*) vì chúng luôn liên quan đến container chính. Những sự kiện này liên quan đến những thay đổi trong vòng đời agent (ví dụ: born, dead, moved, suspended, resumed, frozen, thawed) và liên quan đến những thay đổi trong vòng đời container (ví dụ: added, removed).
- (2) Sự kiện kiểu *MTP-type* được sinh ra bởi platform khi một MTP được kích hoạt (kết thúc) và khi một thông điệp được gửi/nhận bởi/từ một MTP, cụ thể là khi có một số phiên truyền thông liên platform (inter-platform).
- (3) Sự kiện kiểu *message-passing-type* được sinh ra khi một thông điệp ACL được gửi, nhận, định tuyến hoặc được đưa vào hàng đợi thông điệp. Chúng là những sự kiện mà Sniffer thường sử dụng để giám sát.
- (4) Sự kiện kiểu *agent-internal-type* liên quan đến những thay đổi trong trạng thái và hành vi của agent. Chúng là những sự kiện mà Introspector thường sử dụng để giám sát.

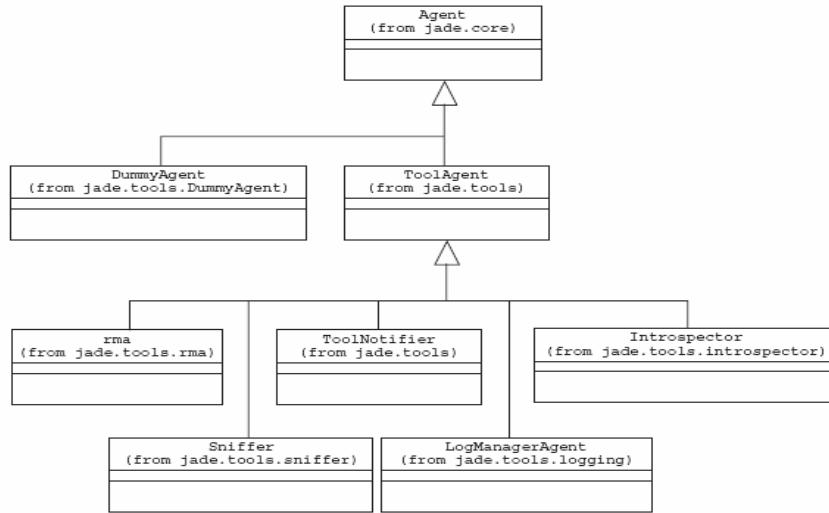
Các agent tương tác với ENS bằng cách trao đổi các thông điệp ACL với AMS. Agent có thể đặt trước với AMS để được thông báo về các kiểu sự kiện bằng cách sử dụng lớp *AMSSubscriber* trong gói *jade.domain.introspection*.

Kiểu sự kiện message-passing-type và agent-internal-type nói cách khác chỉ được nằm trong container chính nơi mà agent tạo ra chúng cư trú. Việc chuyển chúng vào container chính thực tế sẽ làm giảm đáng kể hiệu năng của platform. Hậu quả là để được thông báo về các sự kiện message-passing và agent-internal liên quan đến agent đích cho trước, agent quan sát (observe agent) sẽ phải yêu cầu AMS đánh hơi và debug một cách rõ ràng các agent đích. Điều này được thực hiện bằng các phương tiện của hành động SnifferOn và hành động DebugOn của JADEManagementOntology. Phần 2.5.1 miêu tả chi tiết cách yêu cầu các thao tác quản lý platform của AMS. Kết quả của hành động SniffOn là một agent phụ ToolNotifier được tạo ra trong container mà agent đích cư trú ở đó. Agent này lắng nghe sự kiện message-passing cục bộ

và chuyển tiếp chúng tới agent quan sát. Hệ thống thông báo sự kiện của JADE được minh họa trong Hình 2.15:



Hình 2.15: Hệ thống thông báo sự kiện của JADE



Hình 2.16: Biểu đồ lớp các công cụ của JADE

Mỗi công cụ của JADE, ngoại trừ DummyAgent, đều kế thừa từ lớp *jade.tools.ToolAgent* – lớp cung cấp khả năng nhận các thông báo theo một cách thống nhất. Hình 2.16 minh họa biểu đồ lớp UML về các công cụ của JADE. Mỗi công cụ được cài đặt là một Agent mở rộng lớp *jade.core.Agent* cơ sở. Điều này cho phép một số tính năng và sự đơn giản hóa quan trọng:

- (1) Vòng đời của một công cụ JADE có thể được quản lý như các agent khác của platform.
- (2) Khả năng truyền thông điệp của Agent cơ sở có thể được sử dụng để cho phép sự tương tác giữa công cụ và AMS, cụ thể là việc đặt trước các thông báo sự kiện của platform.
- (3) Một số thể hiện của cùng một công cụ có thể cùng tồn tại trên cùng một platform và thậm chí trong cùng container.

Lớp *jade.tools.ToolNotifier* cài đặt các agent phụ trợ được dùng để chuyển tiếp các sự kiện message-passing và agent-internal tới các agent quan tâm chính là một *ToolAgent*. Cách này cho phép phát hiện xem agent đích hoặc agent quan sát đã kết thúc hay chưa. Lớp *ToolNotifier* được ghép với ENS và không được dự định dành cho các lập trình viên sử dụng.

CHƯƠNG 3

NHỮNG ĐẶC ĐIỂM CƠ BẢN CỦA JADE

Trong chương 2 chúng ta đã có được một cái nhìn tổng quan về JADE, phác họa kiến trúc ở mức cao và nêu ra chức năng của nó. Trong phần này chúng ta sẽ trình bày làm thế nào để phát triển hệ đa agent với JADE dựa vào những tính năng cơ bản mà JADE cung cấp như tạo các agent, thực thi nhiệm vụ của agent, làm cho agent có thể giao tiếp với nhau, và đưa ra các dịch vụ cũng như là tìm kiếm các dịch vụ trong mục trang vàng (yellow page). Bằng những tính năng này, những cái mà chỉ nằm trong dưới 10 lớp trong thư viện của JADE, là đã có thể cài đặt ứng dụng phân tán với một độ phức tạp nhất định. Trong chương sau, chúng ta sẽ mô tả các tính năng nâng cao, những tính năng mà tập trung vào những vấn đề phức tạp hơn. Tuy nhiên JADE phù hợp với câu triết lý “đi tới đâu trả tiền tới đó”, ngũ ý rằng các lập trình viên không cần quan tâm đến các tính năng nâng cao cho đến khi họ cần hoặc mong muốn như vậy.

Như đã giới thiệu trong chương 2, JADE là công cụ thuần java bởi vậy tạo một hệ đa agent trên jade đơn thuần là tạo các lớp java mà không cần lập trình java quá chuyên nghiệp. Để minh họa các bước cần thiết để phát triển ứng dụng với Jade, phần này sẽ giới thiệu các case study đơn giản mà sử dụng xuyên suốt tài liệu này, đó là một hệ thống hướng agent cho phép người sử dụng trao đổi sách cũ. Trong hệ thống trao đổi sách này sẽ có 2 loại agent: *agent bán* và *agent mua*. Mỗi một agent mua lấy đầu vào là các sách mà nó cần phải mua và cố gắng tìm kiếm các agent bán hàng để mua các sách đó với giá chấp nhận được. Tương tự như vậy mỗi một agent bán hàng sẽ lấy đầu vào là các sách dùng để bán và cố gắng sao cho sách bán được với giá cao nhất có thể. Cả agent bán và agent mua đều cài đặt một vài chiến lược đơn giản để thực hiện các đàm phán mua bán sao cho đạt được kết quả tốt nhất cho người sử dụng mà nó đại diện. Cả agent bán và mua có thể xuất hiện và biến mất trong hệ thống một cách động. Các vấn đề liên quan đến mua sách cụ thể, phân phối và thanh toán được coi là ngoài phạm vi và không được quan tâm.

3.1 TẠO AGENT

Tạo một Agent trong jade chỉ đơn giản là định nghĩa một lớp extends lớp jade.core.Agent và cài đặt phương thức setup() như ví dụ dưới đây:

```
import jade.core.Agent;
public class HelloWorldAgent extends Agent {
    protected void setup() {
        // Printout a welcome message
        System.out.println("Hello World. I'm an agent!");
    }
}
```

Lớp HelloWorldAgent ở trên đại diện cho một loại agent, chính xác hơn là một lớp thông thường biến thị cho một đối tượng. Một vài thể hiện của lớp HelloWorldAgent có thể chạy lúc run-time. Không giống như đối tượng java thông thường được xử lý qua tham chiếu của chúng, một agent luôn luôn được thể hiện bởi JADE run-time và tham chiếu của nó không bao giờ được đặt ngoài agent chính nó (tất nhiên là trừ khi các agent đó rõ ràng). *Các agent không bao giờ tương tác qua lời gọi các phương thức mà là tương tác bằng cách trao đổi các thông điệp không đồng bộ*, sẽ được giới thiệu ở mục sau.

Phương thức setup() có mục đích để gộp các khởi tạo của agent. Thông thường công việc chính xác của các agent được thực hiện bên trong các hành vi “behaviours” sẽ được giới thiệu ở mục 3.2. Ví dụ về các hoạt động điển hình mà agent thực hiện trong hàm setup() của nó là : đưa ra một GUI, mở kết nối đến cơ sở dữ liệu, đăng ký các dịch vụ nó cung cấp trong mục các trang vàng (xem phần 3.4) và bắt đầu khởi tạo các behaviours. Tốt nhất là không nên xây dựng hàm khởi tạo trong lớp agent và thực hiện tất cả các khởi tạo trong phương thức setup(). Điều này là vì tại thời điểm xây dựng , agent vẫn chưa được liên kết với JADE run-time phía dưới và vì vậy một vài phương thức kế thừa từ lớp Agent có thể không làm việc một cách chắc chắn.

3.1.1 Định danh agent

Để nhất quán với đặc tả FIPA, mỗi một thể hiện agent được định danh bởi một “bộ định danh agent” (agent identifier). Một bộ định danh agent được biểu diễn dưới dạng là một thể hiện của lớp jade.core.AID. Phương thức getAID() của lớp agent cho phép gọi lại định danh cục bộ của agent. Một đối tượng AID gồm một tên toàn cục đơn nhất (GUID) cộng thêm một số địa chỉ. Tên trong jade có dạng <local-name>@<platform-name>. Ví dụ một agent là *Peter* đang tồn tại trên platform là *foo-platform* sẽ có tên toàn cục đơn nhất là [Peter@foo-platform](#) . Địa chỉ gộp trong AID là địa chỉ của platform mà agent sống. Nhưng địa chỉ này chỉ được sử dụng khi một agent cần giao tiếp với agent khác sống trên một platform khác cũng tuân theo chuẩn FIPA.

Lớp AID cung cấp một số phương thức phục vụ cho việc gọi lại tên cục bộ (getLocalName()), GUID (getName()) và địa chỉ (getAllAddresses()). Vậy nên ta có thể làm phong phú thêm thông điệp chào đón của agent HelloWorldAgent của chúng ta như sau:

```
protected void setup() {
    // Printout a welcome message
    System.out.println("Hello World. I'm an agent!");
    System.out.println("My local-name is
"+getAID().getLocalName());
    System.out.println("My GUID is "+getAID().getName());
    System.out.println("My addresses are:");
    Iterator it = getAID().getAllAddresses();
    while (it.hasNext()) {
        System.out.println("- "+it.next());
    }
}
```

Tên cục bộ của một agent được gán tại thời điểm khởi động bởi người tạo và phải là đơn nhất trong platform. Nếu đã tồn tại một tên cục bộ tương tự trên platform, JADE run-time sẽ ngăn chặn việc tạo agent mới mà trùng tên. Tạo một agent và AID của nó có thể làm như sau:

```
String localname = "Peter";  
AID id = new AID(localname, AID.ISLOCALNAME);
```

tên platform được tự động gắn vào GUID của AID mới được tạo bởi JADE run-time. Tương tự như vậy, tạo GUID của một agent và AID của nó được thực hiện như sau:

```
String guid = "Peter@foo-platform";  
AID id = new AID(guid, AID.ISGUID);
```

3.1.2 Khởi tạo Agent

Tạo lớp HelloWorldAgent mới với nội dung của hàm setup() như trên và biên dịch, chạy như hướng dẫn ở Chương 2, ta có kết quả:

```
C:\WINDOWS\system32\cmd.exe  
INFO: Service jade.core.management.AgentManagement initialized  
Jun 2, 2010 5:30:50 PM jade.core.BaseService init  
INFO: Service jade.core.messaging.Messaging initialized  
Jun 2, 2010 5:30:50 PM jade.core.BaseService init  
INFO: Service jade.core.mobility.AgentMobility initialized  
Jun 2, 2010 5:30:50 PM jade.core.BaseService init  
INFO: Service jade.core.event.Notification initialized  
Jun 2, 2010 5:30:50 PM jade.core.messaging.MessagingService clearCachedSlice  
INFO: Clearing cache  
Jun 2, 2010 5:30:51 PM jade.mtp.http.HTTPServer <init>  
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParse  
rImpl$JAXPSAXParser  
Jun 2, 2010 5:30:51 PM jade.core.messaging.MessagingService boot  
INFO: MTP addresses:  
http://sony-799bec6555:7778/acc  
Hello World. I'm an agent!  
My local-name is hello  
My GUID is hello@sony-799bec6555:1099/JADE  
My addresses are:  
- http://sony-799bec6555:7778/acc  
Jun 2, 2010 5:30:51 PM jade.core.AgentContainerImpl joinPlatform  
INFO:  
Agent container Main-Container@sony-799bec6555 is ready.
```

Hình 3.1: Kết quả chạy file HelloWorldAgent

hello là tên cục bộ được chỉ ra ở dòng lệnh. Vì chúng ta không chỉ ra tên một platform cụ thể nào, nên jade tạo mặc định sử dụng công local và công của main container:

sony-799bec6555:1099/JADE.

Bởi vậy GUID của agent là hello@sony-799bec6555:1099/JADE.

Cần phải chú ý rằng, mặc dù GUID này trông khá giống một địa chỉ nhưng nó không phải là một địa chỉ. Cuối cùng chúng ta nhận thấy rằng AID của agent hello chỉ bao gồm một địa chỉ đó là địa chỉ MTP kích hoạt trong platform. Còn có các cách khác để chạy agent như là dùng bộ công cụ quản lý GUI đề cập ở mục 2.7.1 hoặc phát sinh ra request tới AMS (xem mục 5.5), hoặc sử dụng giao diện tiền trình bên trong (mục 5.6)

3.1.3 Kết thúc agent

Sau khi in ra thông điệp welcome, agent vẫn tiếp tục tồn tại thậm chí nó chẳng có việc gì để làm. Để kết thúc agent thì hàm doDelete() phải được gọi. Tương tự, hàm setup() được gọi để tham gia vào việc khởi tạo agent thì hàm takeDown() chỉ tham gia trước khi agent kết thúc để thực hiện

công việc đơn dẹp.

3.1.4 Truyền tham số cho agent

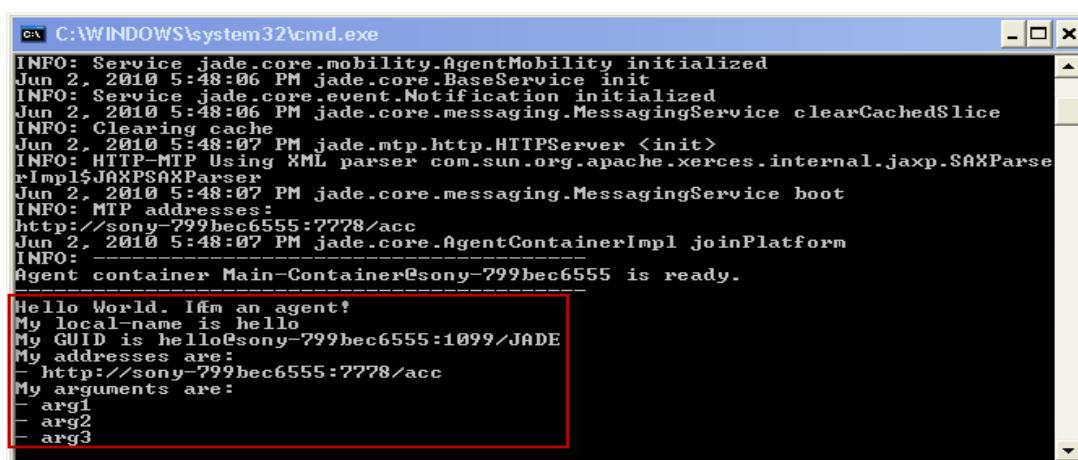
Các agent có thể lấy các tham số khởi động, những tham số này là một mảng Object có thể lấy được bằng cách sử dụng phương thức getArguments() của lớp Agent. Khi khởi chạy một agent trong dòng lệnh như mô tả trong phần 2.7, các tham số khởi động có thể được xác định bằng cách thêm vào sau tên agent trong dấu ngoặc đơn như sau:

```
hello:HelloWorldAgent (arg1 arg2 arg3)
```

Tất nhiên trong trường hợp dòng lệnh thì tham số truyền vào phải là String, nhưng khi chạy một agent trực tiếp từ code, thì tham số Object có thể được truyền vào. Nếu chúng ta chỉnh sửa hàm setup() của HelloWorldAgent như sau:

```
protected void setup() {  
    // Printout a welcome message  
    System.out.println("Hello World. I'm an agent!");  
    System.out.println("My local-name is "+getAID().getLocalName());  
    System.out.println("My GUID is "+getAID().getName());  
    System.out.println("My addresses are:");  
    Iterator it = getAID().getAllAddresses();  
    while (it.hasNext()) {  
        System.out.println("- "+it.next());  
    }  
    System.out.println("My arguments are:");  
    Object[] args = getArguments();  
    if (args != null) {  
        for (int i = 0; i < args.length; ++i) {  
            System.out.println("- "+args[i]);  
        }  
    }  
}
```

thì sau khi thực thi dòng lệnh tương tự thì đầu ra như sau:



```
C:\WINDOWS\system32\cmd.exe  
INFO: Service jade.core.mobility.AgentMobility initialized  
Jun 2, 2010 5:48:06 PM jade.core.BaseService init  
INFO: Service jade.core.event.Notification initialized  
Jun 2, 2010 5:48:06 PM jade.core.messaging.MessagingService clearCachedSlice  
INFO: Clearing cache  
Jun 2, 2010 5:48:07 PM jade.ntp.http.HTTPServer <init>  
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser  
Jun 2, 2010 5:48:07 PM jade.core.messaging.MessagingService boot  
INFO: MTP addresses:  
http://sony-799bec6555:7778/acc  
Jun 2, 2010 5:48:07 PM jade.core.AgentContainerImpl joinPlatform  
INFO: Agent container Main-Container@sony-799bec6555 is ready.  
  
Hello World. I'm an agent!  
My local-name is hello  
My GUID is hello@sony-799bec6555:1099/JADE  
My addresses are:  
- http://sony-799bec6555:7778/acc  
My arguments are:  
- arg1  
- arg2  
- arg3
```

Hình 3.2: Kết quả chạy lớp HelloWorldAgent với tham số truyền vào

```

bookTrading/
|
|---src/
|   |
|   |---bookTrading/
|       |
|       |---buyer/
|       |---seller/
|---classes/
|---lib/
|   |
|   |--- jade.jar
|   |--- jadeTools.jar
|   |--- http.jar
|   |--- iiop.jar
|---commons-codec/
|---dist/
|---build.xml

```

Hình 3.3 Cấu trúc thư mục của dự án bookTrading

3.1.5 Cài đặt dự án Book-Trading

Đến đây chúng ta đã có đầy đủ các yếu tố cần thiết để cài đặt dự án mua bán sách (book-trading). Để dịch và tạo file jar mà user sẽ cần thiết để truy nhập hệ thống, chúng tôi chọn bộ công cụ xây dựng ANT - một sản phẩm của Apache Software Foundation có thể download tại <http://ant.apache.org>. Chúng tôi tổ chức dự án theo như Hình 3.3.

Trong thư mục src chúng ta sẽ lưu các mã nguồn. Chúng ta sẽ bắt đầu tạo gói gốc là book-trading và hai gói phụ cho mã nguồn của agent bán hàng và cái còn lại cho agent mua hàng. Thư mục class dùng để chứa các file .class sau khi biên dịch. Thư mục lib để chứa các thư viện phụ thuộc của jade. Cuối cùng là thư mục dist nơi chứa các file .zip được phân phát cho người sử dụng. Thêm nữa là chúng ta phải tạo ra file ANT build.xml để chỉ cho ANT làm sao để xây dựng dự án bookTrading

Khi đã có môi trường cần thiết cho dự án chúng ta bắt đầu viết bộ khung của lớp BookBuyerAgent bằng cách cài đặt buyer agent. Tất nhiên là bookSellerAgent tất nhiên là cũng tương tự như vậy, nhưng ở phần này chúng ta sẽ giả sử rằng agent bán hàng là cố định, và tên của chúng được biết được truyền vào các agent mua hàng như các toán tử. Khi giới thiệu dịch vụ trang vàng trong phần 3.4 sẽ bỏ các hạn chế này.

```

package bookTrading.buyer;
import jade.core.Agent;
import jade.core.AID;
import java.util.Vector;
import java.util.Date;
public class BookBuyerAgent extends Agent {
    // The list of known seller agents
    private Vector sellerAgents = new Vector();
    // The GUI to interact with the user
    private BookBuyerGui myGui;
    /**
     * Agent initializations
     */
    protected void setup() {
        // Printout a welcome message
        System.out.println("Buyer-agent "+getAID().getName()+" is
            ready.");
        // Get names of seller agents as arguments
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            for (int i = 0; i < args.length; ++i) {
                AID seller = new AID((String) args[i], AID.ISLOCALNAME);
                sellerAgents.addElement(seller);
            }
        }
        // Show the GUI to interact with the user
        myGui = new BookBuyerGuiImpl();
        myGui.setAgent(this);
        myGui.show();
    }
    /**
     * Agent clean-up
     */
    protected void takeDown() {
        // Dispose the GUI if it is there
        if (myGui != null) {
            myGui.dispose();
        }
        // Printout a dismissal message
        System.out.println("Buyer-agent "+getAID().getName()+"
            terminated.");
    }
    /**
     * This method is called by the GUI when the user inserts a new
     * book to buy
     * @param title The title of the book to buy
     * @param maxPrice The maximum acceptable price to buy the book
     * @param deadline The deadline by which to buy the book
     */
}

```

```

public void purchase(String title, int maxPrice, Date deadline) {
    // To be implemented
}
}

```

Các bước biên dịch và chạy ứng dụng bookTrading:

- Tạo file compilejade.bat với nội dung như sau và lưu ở thư mục C:\jade\src\examples\bookTrading:

```
javac -classpath
```

```
C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar; %1 %2 %3 %4 %5 %6 %7 %8 %9
```

- Tạo file runjade.bat với nội dung như sau và lưu ở thư mục C:\jade\src\examples\bookTrading:

```
java -classpath
```

```
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar jade.Boot %1 %2 %3 %4 %5 %6 %7 %8 %9
```

- Biên dịch các file bằng dòng lệnh:

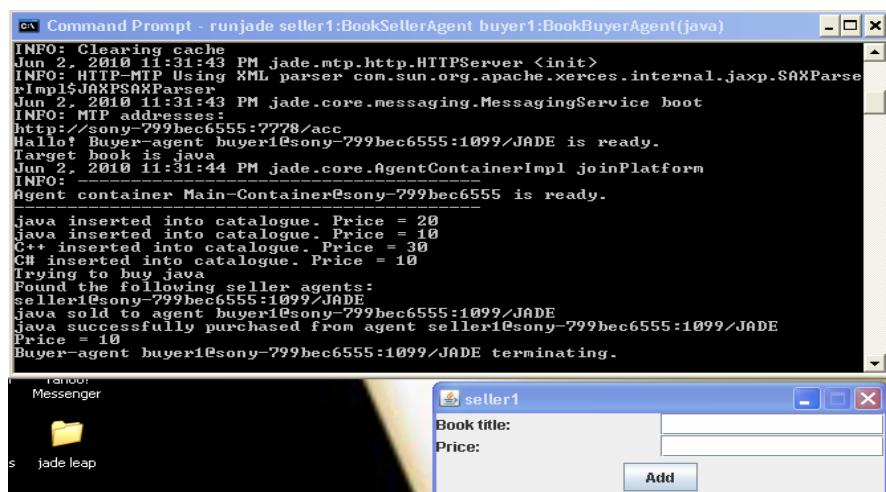
Trong cửa sổ dòng lệnh, chuyển về thư mục C:\jade\src\examples\bookTrading.

Gõ lệnh: compilejade *.java

- Chạy seller agent và buyer agent:

Làm tương tự như trên, nhưng gõ lệnh: runjade seller1:BookSellerAgent buyer1:BookBuyerAgent(<tên sách>)

Kết quả:



Hình 3.4: Kết quả chạy seller agent và buyer agent

Cả agent buyer và seller đều có GUI để tương tác với người sử dụng của chúng. Ví dụ, GUI của agent buyer nên cho phép một người sử dụng để chỉ ra đề mục của các sách mà họ muốn mua và kèm theo các thông tin khác (ví dụ như giá cao nhất và thời hạn mua sách) để thuận lợi cho việc

giao dịch. Hình 3.4 là GUI của agent seller. Vì việc phát triển các GUI này là nằm ngoài phạm vi của tia liệu này nên chúng ta sẽ chỉ tập trung vào giao diện cung cấp bởi mỗi GUI và xem các lớp cài đặt GUI là đã có sẵn. Định nghĩa của lớp BookBuyerGui như sau :

```
package bookTrading.buyer;  
public interface BookBuyerGui {  
    void setAgent(BookBuyerAgent a);  
    void show();  
    void hide();  
    void notifyUser(String message);  
}
```

3.2 CÀI ĐẶT NHIỆM VỤ CHO AGENT

Như được trình bày trong phần 3.1, việc cài đặt agent thể hiện trong các hành vi (behaviour). Một behaviour đại diện cho một nhiệm vụ mà agent có thể thực hiện và được cài đặt như một đối tượng của một lớp kế thừa *jade.core.behaviours.Behaviour*. Để một agent thực thi nhiệm vụ được cài đặt trong đối tượng behaviour, behaviour phải được add vào agent bằng phương thức *addBehaviour()* của lớp Agent. Các Behaviour có thể được add vào bất kì thời gian nào khi agent bắt đầu (trong phương thức *setup()*) hoặc từ trong các behaviour khác.

Mỗi lớp kế thừa Behaviour phải cài đặt 2 phương thức abstract. Phương thức *action()* định nghĩa các hoạt động được thực hiện khi behaviour thực thi. Phương thức *done()* trả một giá trị boolean chỉ ra behaviour được hoàn thành hay chưa và được xóa khỏi luồng hành vi của agent đang thực thi.

3.2.1 Lập lịch và thực thi Behaviour

Một agent có thể thực thi đồng thời vài behaviour. Tuy nhiên, điều quan trọng cần lưu ý là việc lập lịch của các behaviour trong agent không có sự ưu tiên (giống threads của java), nhưng có sự hợp tác với nhau. Điều này có nghĩa khi một behaviour được lập lịch cho việc thực thi phương thức *action()* của nó được gọi và chạy cho đến khi trả về. Đó là điều người lập trình xác định khi một agent chuyển từ việc thực thi một behaviour sang thực hiện behaviour khác.

Cách tiếp cận này thường tạo những khó khăn cho các nhà phát triển JADE thiếu kinh nghiệm và phải thường xuyên chú tâm khi viết các agent JADE. Mặc dù đòi hỏi thêm sự nỗ lực của cộng đồng, nhưng mô hình này hiện có một số lợi thế:

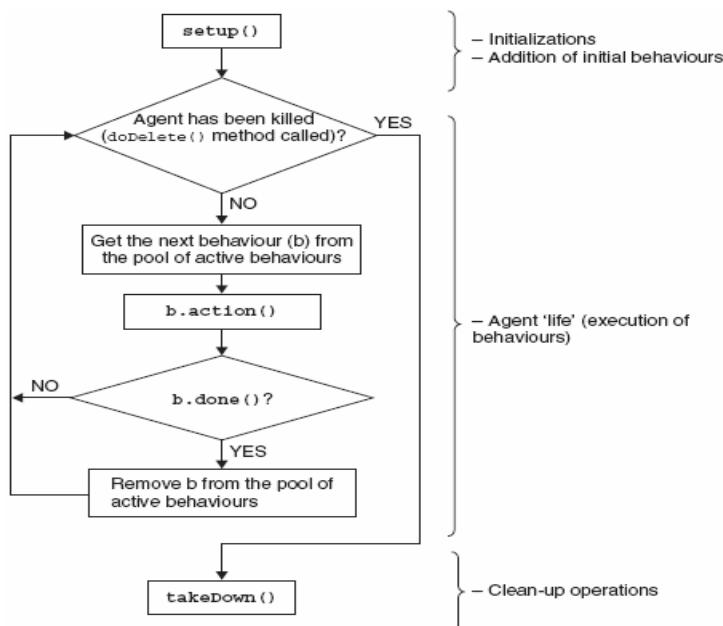
- Nó chấp nhận một luồng Java đơn giản bằng agent. Điều này rất quan trọng trong môi trường giới hạn về nguồn lực như điện thoại di động.
- Nó cung cấp cải thiện hiệu suất trong việc chuyển hành vi nhanh hơn so với chuyển luồng Java.
- Nó loại bỏ tất cả các vấn đề đồng bộ giữa các behaviour đồng thời truy cập vào cùng tài nguyên từ tất cả các behaviours được thực thi bởi cùng Java thread. Điều này cũng làm nâng cao hiệu suất.
- Khi chuyển đổi behaviour xảy ra, tình trạng của agent không bao gồm bất kì thông tin ngăn xếp nào. Điều này cho phép việc thực hiện liên tục một số tính năng nâng cao quan

trọng, chẳng hạn như lưu lại trạng thái của agent trong bộ lưu trữ lâu dài, hoặc chuyển các agent đến container khác để thực thi từ xa (agent di động). Các tính năng nâng cao sẽ được giải quyết chi tiết trong phần sau.

Các bước thực hiện của luồng agent được mô tả trong Hình 3.5. Điều quan trọng cần chú ý là một behaviour như phần dưới đây sẽ giải quyết trước bất kì behaviour khác đang được thực thi bởi phương thức action() của nó và không trả về.

```
public class OverbearingBehaviour extends Behaviour {
    public void action() {
        while (true) {
            // do something
        }
    }
    public boolean done() {
        return true;
    }
}
```

Khi không có các behaviour để thực thi, thread của agent sẽ sleep để đỡ tốn thời gian CPU. Luồng này sẽ được đánh thức trở lại một khi có một behaviour để thực thi.



Hình 3.5: Luồng thực thi của agent

3.2.2 One-shot behaviour, cyclic behavior và generic behaviour

Có 3 kiểu behaviour chính sẵn có trong JADE như sau:

- (1) “One -shot” behaviours được thiết kế để kết thúc một giai đoạn thực thi. Phương thức `action()` chỉ được thực thi một lần. Lớp `jade.core.behaviours.OneShotBehaviour` đã cài đặt phương thức `done()` return “true” và thuận lợi khi mở rộng để cài đặt các one-shot behaviour mới.

```
public class MyOneShotBehaviour extends OneShotBehaviour {
```

```

        public void action() {
            // perform operation X
        }
    }

```

trong ví dụ, operation X được thực hiện một lần.

- (2) “Cyclic” behaviours được thiết kế không bao giờ kết thúc. Phương thức action() thực hiện các operation cùng lúc mỗi khi được gọi. Lớp *Jade.core.behaviours.CyclicBehaviour* đã cài đặt phương thức done() return “false” và thuận lợi khi mở rộng để cài đặt các cyclic behaviour mới.

```

public class MyCyclicBehaviour extends CyclicBehaviour {
    public void action() {
        // perform operation Y
    }
}

```

trong ví dụ, operation Y được thực hiện lặp lại cho tới khi agent thực hiện hết behaviour xong.

- (3) Các behaviours được nhúng vào 3 trạng thái và thực thi các operation khác nhau phục thuộc vào giá trị trạng thái. Chúng kết thúc khi một điều kiện nhất định được đáp ứng.

```

public class ThreeStepBehaviour extends Behaviour {
    private int step = 0;
    public void action() {
        switch (step) {
            case 0:
                // perform operation X
                step++;
                break;
            case 1:
                // perform operation Y
                step++;
                break;
            case 2:
                // perform operation Z
                step++;
                break;
        }
    }
    public boolean done() {
        return step == 3;
    }
}

```

trong ví dụ, biến “step” cài đặt trạng thái của behaviour. Thao tác X , Y, Z được thực hiện tuần tự tới khi behaviour kết thúc.

JADE cũng cung cấp khả năng hợp tác với nhau của các behaviours để tạo ra các behaviour phức tạp. Tính năng này, đặc biệt thuận lợi khi cài đặt các nhiệm vụ phức tạp, được mô tả trong phần 3.3.5.

3.2.3 Bổ sung thêm về hành vi của agent

Tất cả các behaviours đều kế thừa các phương thức `onStart()` và `onEnd()` từ lớp Behaviour. Các phương thức này được thực thi chỉ một lần trước khi gọi phương thức `action()` và sau khi phương thức `done()` trả về true. Chúng nhằm thực hiện các nhiệm vụ đặc biệt để khởi tạo và chấm dứt các operation. Không giống với các phương thức `action()` và `done()` được khai báo abstract, chúng cài đặt mặc định rõ ràng cho phép người phát triển cài đặt chúng theo ý họ muốn.

Một behaviour có thể bị hủy ở bất cứ thời gian nào khi gọi phương thức `removeBehaviour()` trong lớp Agent. Do đó nếu behavior bị hủy sử dụng phương thức `removebehaviour()`, phương thức `onEnd()` của nó không được gọi. Mỗi behaviour có một biến gọi là “myAgent” trỏ đến agent được thực thi behaviour. Cung cấp một cách đơn giản để truy cập tài nguyên của agent từ bên trong behaviour. Cuối cùng điều quan trọng cần ghi nhớ là một đối tượng Behaviour đã được thực thi, nếu nó thực thi lần thứ 2, nó cần gọi phương thức `reset()` trước tiên. Nếu không làm điều này có thể dẫn đến kết quả không mong muốn.

3.2.4 Lập lịch cho các hành vi của agent

JADE cung cấp 2 lớp (trong package `jade.core.behaviours`) mà có thể cài đặt để tạo các behaviour thực thi khi chọn thời gian cho nó.

- (1) WakerBehaviour có các phương thức `action()` và `done()` được cài đặt trước để thực thi phương thức abstract `onWake()` sau 1 thời gian xác định kết thúc (đặc tả trong cấu trúc). Sau khi thực thi phương thức `onWake()` thì behaviour kết thúc.

```
public class MyAgent extends Agent {  
    protected void setup() {  
        System.out.println("Adding waker behaviour");  
        addBehaviour(new WakerBehaviour(this, 10000) {  
            protected void onWake() {  
                // perform operation X  
            }  
        } );  
    }  
}
```

Trong ví dụ này, operation X được thực hiện 10 s sau khi “add behaviour waker”.

- (2) TickerBehaviour có các phương thức `action()` và `done` được cài đặt trước để thực thi lặp đi lặp lại phương thức abstract `onTick()`, chờ đợi một thời gian xác định (đặc tả trong cấu trúc) sau mỗi lần thực thi. Một TickerBehaviour không bao giờ kết thúc trừ phi nó được xóa hoặc phương thức `stop()` của nó được gọi.

```
public class MyAgent extends Agent {  
    protected void setup() {  
        addBehaviour(new TickerBehaviour(this, 10000) {  
            protected void onTick() {  
                // perform operation Y  
            }  
        } );  
    }  
}
```

```

    }
}

```

trong ví dụ này, operation Y được thực hiện chu kì 10s.

3.2.5 Các hành vi trong ví dụ bookTrading

3.2.5.1 Các hành vi của BookBuyerAgent

Khi một agent buyer được yêu cầu mua sách, một phương pháp đơn giản có thể áp dụng để thực hiện nhiệm vụ theo chu kì để hỏi tất cả các agent seller biết nếu chúng có sẵn sách bán, và nếu như vậy, cung cấp 1 giao dịch. Tùy thuộc vào điều này và trên phạm vi giá được xác định bởi người dùng, agent buyer có thể hỏi seller cung cấp giao dịch tốt nhất để bán sách. Chúng ta cài đặt chức năng này bằng việc sử dụng TickerBehaviour, ở mỗi tick, thêm behaviour khác để yêu cầu các agent seller. TickerBehaviour này được add phương thức setup():

```

protected void setup() {
    ...
    // Add a TickerBehaviour that schedules a request to seller
    // agents every minute
    addBehaviour(new TickerBehaviour(this, 60000) {
        protected void onTick() {
            ...
            // Perform the request
            myAgent.addBehaviour(new RequestPerformer());
        }
    });
    ...
}

```

Hành vi RequestPerformer có nhiệm vụ nhận phản hồi từ các seller và gửi thông điệp đáp ứng của buyer tới seller. Cài đặt lớp behaviour bên trong lớp agent sẽ thực thi chúng tốt vì nó cho phép các behaviours truy cập trực tiếp tới tài nguyên của agent giống như biến “myGui” của lớp BookBuyerAgent.

3.2.5.2 Hành vi của BookSellerAgent

Người dùng phải cung cấp tiêu đề sách và giá ban đầu của mỗi quyền sách được bán. Catalogue chứa các sách đang được bán là một hash table. Ngoài ra, seller agent có hai hành vi CyclicBehaviour là OfferRequestsServer và PurchaseOrderServer để phục vụ các yêu cầu đang được gửi tới.

```

public class BookSellerAgent extends Agent {
    // The catalogue of books for sale (maps the title of a book
    // to its price)
    private Hashtable catalogue;
    // The GUI by means of which the user can add books in the
    catalogue
    private BookSellerGui myGui;
    // Put agent initializations here
    protected void setup() {

```

```

// Create the catalogue
catalogue = new Hashtable();

// Create and show the GUI
myGui = new BookSellerGui(this);
myGui.show();

// Register the book-selling service in the yellow pages
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("book-selling");
sd.setName("JADE-book-trading");
dfd.addServices(sd);
try {
    DFService.register(this, dfd);
} catch (FIPAException fe) {
    fe.printStackTrace();
}

// Add the behaviour serving queries from buyer agents
addBehaviour(new OfferRequestsServer());

// Add the behaviour serving purchase orders from buyer
agents
addBehaviour(new PurchaseOrdersServer());
}

// Put agent clean-up operations here
protected void takeDown() {
    // Deregister from the yellow pages
    try {
        DFService.deregister(this);
    } catch (FIPAException fe) {
        fe.printStackTrace();
    }
    // Close the GUI
    myGui.dispose();
    // Printout a dismissal message
    System.out.println("Seller-agent " + getAID().getName() +
" terminating.");
}

/**
 * This is invoked by the GUI when the user adds a new book for
 * sale
 */
public void updateCatalogue(final String title, final int
price) {

```

```

        addBehaviour(new OneShotBehaviour() {
            public void action() {
                catalogue.put(title, new Integer(price));
                System.out.println(title + " inserted into catalogue. Price = " + price);
            }
        });
    }
}

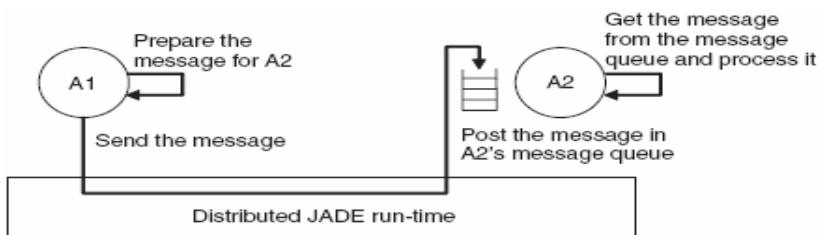
```

3.3 TRUYỀN THÔNG GIỮA CÁC AGENT

Truyền thông giữa các agent có lẽ là tính năng cơ bản nhất của Jade và được thực hiện theo các đặc tả FIPA được mô tả trong Phần 1.2. Các mô hình truyền thông dựa trên truyền thông điệp bất đồng bộ. Như vậy, mỗi agent có một “hộp thư” (hàng đợi thông điệp của các agent), nơi Jade tại thời gian chạy gửi thông điệp được gửi đến bởi các agent khác. Bất cứ khi nào một thông điệp được gửi vào trong hàng đợi hộp thư thì agent được gửi thông điệp đó sẽ nhận được thông báo. Tuy nhiên khi nào, hoặc nếu, agent chọn lấy thông điệp từ hàng đợi để xử lý là do việc lựa chọn thiết kế của lập trình viên. Quá trình này được mô tả trong hình 3.6.

Các định dạng cụ thể của thông điệp trong Jade tương thích với định dạng được định nghĩa trong cấu trúc thông điệp FIPA-ACL được mô tả trong Phần 1.2.3. Mỗi thông điệp bao gồm các trường sau:

- *Người gửi* thông điệp.
- Danh sách *những người nhận*.
- Những *hành động giao tiếp* (còn gọi là 'performative - biểu hiện') chỉ ra những gì người gửi dự định đạt được bằng cách gửi thông điệp. Ví dụ, nếu hành động là một yêu cầu (request), tức là người gửi muốn người nhận thực hiện một hành động, nếu là thông báo (inform) tức là người gửi muốn người nhận biết một thông tin gì đó, nếu là một đề xuất (proposal) hay một yêu cầu được đề xuất (CFP) nghĩa là người gửi muốn tham gia vào một đàm phán.



Hình 3.6: Cơ chế truyền thông điệp không đồng bộ trong JADE

- Nội dung có chứa các thông tin thực sự được trao đổi bằng thông điệp (ví dụ, hành động được thực hiện trong một thông điệp YÊU CẦU, hoặc thực tế là người gửi muốn tiết lộ trong thông điệp THÔNG BÁO...).
- Ngôn ngữ nội dung chỉ ra cú pháp được sử dụng để thể hiện nội dung. Cả người gửi và người nhận phải có khả năng mã hóa và phân tích các biểu thức tuân thủ với cú pháp này

cho việc giao tiếp có hiệu quả.

- Các ontology chỉ ra vốn từ vựng của những biểu tượng được sử dụng trong nội dung. Cả người gửi và người nhận phải quy về cùng ý nghĩa như nhau với các biểu tượng này để việc giao tiếp có hiệu quả.
- Một số trường bổ sung được sử dụng để kiểm soát một số cuộc hội thoại đồng thời và để xác định những thời gian trễ khi nhận một trả lời như conversation – id, reply – with, in – reply – to và reply – by.

Một thông điệp trong Jade được thực hiện như là một đối tượng của lớp *jade.lang.acl.ACMessage* cung cấp các phương thức get và set để truy cập vào tất cả các trường được đặc tả bởi định dạng ACL. Tất cả các performatives được định nghĩa trong đặc tả FIPA được ánh xạ như các hằng số trong lớp ACMessage.

3.3.1 Gửi thông điệp

Gửi thông điệp đến agent khác đơn giản như việc điền vào các trường của một đối tượng ACMessage và sau đó gọi phương thức send() của lớp Agent. Đoạn code dưới đây tạo ra một thông điệp để thông báo một agent có biệt danh là Peter rằng ngày hôm nay trời mưa:

```
ACMessage msg = new ACMessage(ACMessage.INFORM);  
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));  
msg.setLanguage("English");  
msg.setOntology("Weather-forecast-ontology");  
msg.setContent("Today it's raining");  
send(msg);
```

Các ACL performative được định nghĩa bởi FIPA cũng đã định nghĩa các ngữ nghĩa hình thức có thể được khai thác để làm cho một agent tự động đưa ra các quyết định đúng đắn khi nhận được một thông điệp. Tính năng cao cấp này không được sử dụng trong ví dụ về bán sách của chúng ta, nhưng sẽ được miêu tả sau này trong chương sau. Thay vào đó, chúng tôi sẽ chọn các performative để sử dụng trong các thông điệp trao đổi giữa các agent mua và bán trên cơ sở ý nghĩa trực quan của chúng. Đặc biệt chúng ta có thể sử dụng khá tiện lợi performative CFP (Call for proposal – kêu gọi các đề xuất) cho những thông điệp mà các agent mua gửi đến các agent bán để yêu cầu cung cấp một cuốn sách. performative PROPOSE có thể được sử dụng cho những thông điệp mang theo những lời chào hàng của người bán và performative ACCEPT_PROPOSAL cho những thông điệp mang theo những chấp nhận lời chào hàng đó, tức là việc đặt mua hàng. Cuối cùng là performative REFUSE sẽ được sử dụng cho các thông điệp được gửi bởi các agent bán khi cuốn sách được yêu cầu không có trong danh mục của họ.

Để giữ cho mọi thứ đơn giản đến mức có thể, chúng ta sẽ đặt tiêu đề của cuốn sách muốn mua vào trong nội dung của thông điệp CFP được gửi bởi các agent mua. Tương tự, nội dung của các thông điệp PROPOSAL mang theo những lời chào hàng của các agent lý bán sẽ là giá của cuốn sách. Đây là cách một thông điệp CFP có thể được tạo ra và gửi bởi một agent mua:

```
ACMessage cfp = new ACMessage(ACMessage.CFP);  
for (int i = 0; i < sellerAgents.length; ++i) {  
    cfp.addReceiver(sellerAgents[i]);
```

```

    }
    cfp.setContent(targetBookTitle);
    cfp.setConversationId("book-trade");
    cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique
    value
    myAgent.send(cfp);
}

```

3.3.2 Nhận thông điệp

Như đã đề cập ở trên, tại thời gian chạy Jade sẽ tự động đưa các thông điệp vào hàng đợi thông điệp cá nhân của một người nhận ngay sau khi chúng tới. Một agent có thể lấy các thông điệp từ hàng đợi thông điệp của mình bằng phương thức receive(). Phương thức này sẽ trả về thông điệp đầu tiên trong hàng đợi (do đó gây ra việc nó sẽ được bỏ khỏi hàng đợi), hoặc null nếu hàng đợi rỗng, và ngay lập tức trả lại.

```

ACLMensaje msg = receive();
if (msg != null) {
    // Xử lý thông điệp
}

```

3.3.3 Khóa hành vi đợi thông điệp

Lập trình viên thường cần phải thực hiện các hành vi xử lý các thông điệp nhận được từ các agent khác. Đây là trường hợp đối với hành vi OfferRequestsServer và PurchaseOrderServer được giới thiệu trong phần 3.2.5.2, ở đây chúng ta cần phục vụ các thông điệp từ các agent mua. Những hành vi này phải được liên tục thực hiện (cyclic behaviours) và, ở mỗi lần thực hiện phương thức action(), phải kiểm tra xem thông điệp đã được nhận chưa và xử lý nó. Trong trường hợp của chúng ta hai hành vi là rất giống nhau. Ở đây chúng ta đưa ra hành vi OfferRequestsServer. Đây là hành vi sử dụng bởi các agent bán sách để phục vụ cho các yêu cầu đến từ việc đặt mua sách của các agent mua. Nếu cuốn sách chỉ ra là ở trong danh mục có của người bán thì agent bán trả lời với một thông điệp PROPOSAL xác định giá cả. Nếu không thì một thông điệp REFUSE được gửi lại.

```

private class OfferRequestsServer extends CyclicBehaviour {
    public void action() {
        MessageTemplate mt =
MessageTemplate.MatchPerformativ(ACLMensaje.CFP);
        ACLMensaje msg = myAgent.receive(mt);
        if (msg != null) {
            // CFP Message received. Process it
            String title = msg.getContent();
            ACLMensaje reply = msg.createReply();

            Integer price = (Integer) catalogue.get(title);
            if (price != null) {
                // The requested book is available for sale.
//Reply with the price
        }
    }
}

```

```

        reply.setPerformative(ACLMessage.PROPOSE);
        reply.setContent(String.valueOf(price.intValue()));
    } else {
        // The requested book is NOT available for
        // sale.
        reply.setPerformative(ACLMessage.REFUSE);
        reply.setContent("not-available");
    }
    myAgent.send(reply);
} else {
    block();
}
}
}
}

```

Như thường lệ, chúng tôi thực hiện hành vi OfferRequestsServer như là một lớp bên trong của lớp BookSellerAgent. Điều này đơn giản hóa mọi thứ vì chúng ta có thể truy cập trực tiếp vào danh mục sách để bán. Tất nhiên phương pháp này chỉ được đề xuất, không bắt buộc.

Phương thức createReply() của lớp ACLMessage tự động tạo ra một ACLMessage mới, tự động cài đặt những người nhận và bất kỳ một trường cần thiết nào cho việc kiểm soát cuộc hội thoại (ví dụ như conversation-id, reply-with, in-reply-to).

Tuy nhiên, chúng ta có thể nhận thấy rằng ngay khi chúng tôi thêm các hành vi trên, luồng hoạt động của agent bắt đầu một vòng lặp liên tục mà cực kỳ tốn dung lượng CPU. Một khác, chúng tôi muốn phương thức action() của hành vi OfferRequestsServer như được thực thi chỉ khi nhận được một thông điệp mới. Để làm được điều này, chúng ta phải sử dụng phương thức block() của lớp Behaviour, trong đó, mặc dù như những gì tên phương thức gợi ý, nó không phải là một cuộc gọi bị chặn, mà chỉ đánh dấu hành vi như 'bị chặn' để các agent không còn lập lịch để thực hiện nó. Khi một thông điệp mới được đưa vào hàng đợi của các agent, tất cả các hành vi bị cấm trở nên có hiệu lực thực hiện lại để chúng có cơ hội xử lý thông điệp nhận được. Phương thức action() do đó phải được sửa đổi như sau:

```

public void action() {
    ACLMessage msg = myAgent.receive();
    if (msg != null) {
        // Message received. Process it
        ...
    } else {
        block();
    }
}

```

Code trên được xem là tiêu biểu và là khuôn mẫu cho việc nhận các thông điệp bên trong một hành vi.

3.3.4 Lựa chọn thông điệp từ hàng đợi

Ta thấy *OfferRequestsServer* và *PurchaseOrderServer* đều là các hành vi vòng với một phương thức *action()* bắt đầu với một lời gọi đến *myAgent.receive()*. Chúng ta có thể nhận thấy một vấn đề là làm thế nào có thể chắc chắn rằng hành vi *OfferRequestsServer* chỉ đọc từ hàng đợi những thông điệp CFP và hành vi *PurchaseOrderServer* chỉ đọc những thông điệp chứa yêu cầu mua hàng? Để giải quyết vấn đề này chúng ta phải sửa đổi mã hiện tại bằng cách xác định các “mẫu” (template) để được sử dụng khi gọi phương thức *receive()*. Khi một mẫu được xác định, phương thức *receive()* trả về thông điệp đầu tiên thỏa mãn và bỏ qua tất cả các thông điệp không thỏa mãn. Mẫu này được cài đặt là thể hiện của lớp jade.lang.acl.MessageTemplate cung cấp một số phương thức để tạo ra các mẫu một cách rất đơn giản và linh hoạt. Như đã đề cập trong Phần 3.3.1, chúng ta sử dụng CFP performative cho thông điệp mang theo các lời gọi để được mua sách và ACCEPT_PROPOSAL performative cho thông điệp mang theo lời chấp nhận đề xuất. Vì vậy chúng ta có thể sửa đổi các phương thức *action()* của *OfferRequestsServer* sao cho các lời gọi đến *myAgent.receive()* bỏ qua tất cả các thông điệp, ngoại trừ những người có performative là CFP:

```
MessageTemplate mt =  
MessageTemplate.MatchPerformative(ACLMessage.CFP);  
ACLMessage msg = myAgent.receive(mt);
```

3.3.5 Các cuộc hội thoại phức tạp

Hành vi BookNegotiator thảo luận trong Phần 3.2.5.1 đại diện cho một ví dụ về một hành vi thực hiện một cuộc hội thoại phức tạp. Một cuộc hội thoại là một chuỗi các thông điệp trao đổi giữa hai hay nhiều agent với những quan hệ nhân quả và tạm thời được định nghĩa rõ ràng. Hành vi BookNegotiator gửi một thông điệp CFP cho các agent (các agent bán được biết đến) và nhận được tất cả phản hồi. Sau đó, nếu nhận được ít nhất một phản hồi PROPOSE, nó phải gửi thêm một thông điệp ACCEPT_PROPOSAL (cho agent bán mà có lời đề xuất tốt nhất) và chờ xác nhận. Bất cứ khi nào một cuộc hội thoại như này phải được tiến hành thì đó là một sự thực hành tốt để xác định các trường kiểm soát hội thoại trong các thông điệp trao đổi bên trong hội thoại đó. Điều này cho phép dễ dàng tạo ra các mẫu rõ ràng phù hợp với những phản hồi có thể có.

```
private class BookNegotiator extends Behaviour {  
    private String title;  
    private int maxPrice;  
    private PurchaseManager manager;  
    private AID bestSeller; // The seller agent who provides the  
    best offer  
    private int bestPrice; // The best offered price  
    private int repliesCnt = 0; // The counter of replies from  
    seller agents  
    private MessageTemplate mt; // The template to receive  
    replies  
    private int step = 0;  
    public BookNegotiator(String t, int p, PurchaseManager m) {
```

```

        super(null);
        title = t;
        maxPrice = p;
        manager = m;
    }
    public void action() {
        switch (step) {
            case 0:
                // Send the cfp to all sellers
                ACLMessage cfp = new
                ACLMessage(ACLMessage.CFP);
                for (int i = 0; i < sellerAgents.length; ++i)
                {
                    cfp.addReceiver(sellerAgents[i]);
                }
                cfp.setContent(title);
                cfp.setConversationId("book-trade");
                cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
                myAgent.send(cfp);
                // Prepare the template to get proposals
                mt = MessageTemplate.and(
                MessageTemplate.MatchConversationId("book-
                trade"),
                MessageTemplate.MatchInReplyTo(cfp.getReplyWi
                th()));
                step = 1;
                break;
            case 1:
                // Receive all proposals/refusals from seller
                agents
                ACLMessage reply = myAgent.receive(mt);
                if (reply != null) {
                    // Reply received
                    if (reply.getPerformative() ==
                    ACLMessage.PROPOSE) {
                        // This is an offer
                        int price =
                        Integer.parseInt(reply.getContent()
                        );
                        if (bestSeller == null || price <
                        bestPrice) {
                            // This is the best offer at
                            present
                            bestPrice = price;
                            bestSeller =
                            reply.getSender();
                        }
                    }
                }

```

```

        repliesCnt++;
        if (repliesCnt >= sellerAgents.length) {
            // We received all replies
            step = 2;
        }
    }
    else {
        block();
    }
    break;
case 2:
if (bestSeller != null && bestPrice <= maxPrice) {
    // Send the purchase order to the seller that
    provided the
    best offer
    ACLMessage order = new
    ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
    order.addReceiver(bestSeller);
    order.setContent(title);
    order.setConversationId("book-trade");
    order.setReplyWith("order"+System.currentTimeMillis());
    myAgent.send(order);
    // Prepare the template to get the purchase
    order reply
    mt = MessageTemplate.and(
    MessageTemplate.MatchConversationId("book-
    trade"),
    MessageTemplate.MatchInReplyTo
    (order.getReplyWith()));
    step = 3;
}
else {
    // If we received no acceptable proposals,
    terminate
    step = 4;
}
break;
case 3:
// Receive the purchase order reply
reply = myAgent.receive(mt);
if (reply != null) {
    // Purchase order reply received
    if (reply.getPerformative() ==
    ACLMessage.INFORM) {
        // Purchase successful. We can terminate
        myGui.notifyUser("Book           "+title+
        successfully purchased.Price = " +
        bestPrice);
}

```

```

        manager.stop();
    }
    step = 4;
}
else {
    block();
}
break;
}
}

public boolean done() {
    return step == 4;
}
} // End of inner class BookNegotiator

```

Các hội thoại phức tạp thường được thực hiện dựa theo một giao thức được xác định rõ ràng, chẳng hạn như những gì được xác định bởi FIPA. Jade cung cấp hỗ trợ phong phú cho một số các giao thức tương tác được sử dụng phổ biến nhất trong gói jade.proto. Cuộc hội thoại mà chúng ta thực hiện ở trên, ví dụ, theo giao thức 'Contract-net' giao thức mà có thể là rất dễ dàng thực hiện bằng cách khai thác lớp jade.proto.ContractNetInitiator. Điều này sẽ tiếp tục được mô tả trong các phần sau.

3.3.6 Nhận thông điệp tại node đang khóa

Bên cạnh phương thức receive(), lớp Agent cũng cung cấp phương thức *blockingReceive()*, như tên cho thấy, là một lời gọi khóa: nó không trả lại cho đến khi có một thông điệp trong hàng đợi thông điệp của agent. Một phiên bản quá tải mà dùng MessageTemplate như một tham số (nó không trả lại cho đến khi có một thông điệp phù hợp với mẫu quy định) cũng khả dụng.

Điều quan trọng cần nhấn mạnh rằng phương thức *blockingReceive()* thực sự chặn thread của agent. Vì vậy nếu bạn gọi *blockingReceive()* từ trong một hành vi, điều này ngăn cản tất cả các hành vi khác cho đến khi thực hiện lời gọi đến *blockingReceive()* trả về. Hãy cùng xem xét, một việc lập trình tốt là để nhận được các thông điệp sử dụng *blockingReceive()* trong phương thức setup() và takeDown(); sử dụng receive() trong sự kết hợp với Behaviour.block () (như đã được nêu trong Phần 3.3.3) bên trong các hành vi.

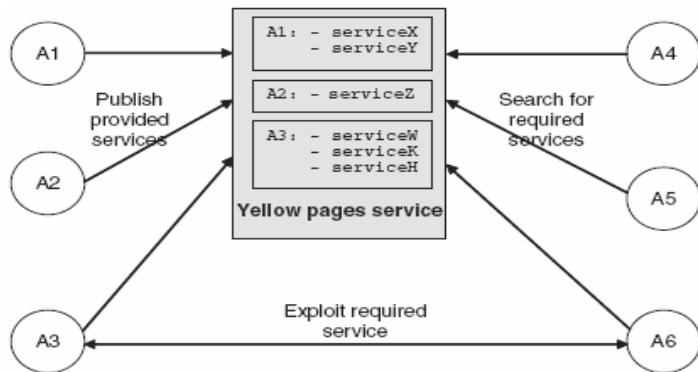
3.4 KHÁM PHÁ AGENT – DỊCH VỤ TRANG VÀNG

Khi viết code chúng ta giả định rằng có một tập cố định các agent người bán (qua mỗi agent người mua như là các đối số ban đầu). Trong chương này chúng ta loại bỏ giả định này bằng cách khai thác dịch vụ các trang vàng được cung cấp bởi JADE platform cho phép các agent người mua tự động phát hiện các agent người bán sẵn có tại một điểm được đưa ra trong thời gian.

3.4.1 DF agent

Một dịch vụ “yellow pages” cho phép các agent đưa ra các mô tả của một hoặc nhiều các dịch vụ mà chúng cung cấp theo thứ tự mà các agent khác có thể dễ dàng phát hiện và khai thác chúng. Điều này được mô tả trong hình 3.5. Bất kỳ agent nào cũng có thể đăng ký (xuất bản) các dịch vụ

và tìm kiếm cho các dịch vụ (khai thác). Đăng ký, xóa, sửa đổi và tìm kiếm có thể được thực hiện bất cứ lúc nào trong thời gian sống của agent.



Hình 3.7: Dịch vụ trang vàng

Dịch vụ các trang vàng trong Jade, phù hợp với các đặc tả quản lý agent của FIPA (FIPA Agent Management), được cung cấp bởi một agent đặc biệt gọi là DF (Directory Facilitator). Mỗi FIPA-compliant platform nên đăng cai một agent DF mặc định (tên địa phương là ‘*df@<platform-name>*’). Các agent DF khác có thể được triển khai nếu được yêu cầu và một vài agent DF (bao gồm cả mặc định) có thể được tổ chức thành liên đoàn để cung cấp một danh mục liệt kê duy nhất các trang vàng phân tán.

3.4.2 Tương tác với DF agent

DF là một agent, nó có thể tương tác với nó như với bất kỳ agent khác bằng cách trao đổi các thông điệp ACL sử dụng một ngôn ngữ có nội dung thích hợp (ví dụ ngôn ngữ SL0) và một ontology thích hợp (ví dụ FIPA-agent-management ontology) như được định nghĩa trong các đặc tả FIPA. Để đơn giản hóa các tương tác này, Jade cung cấp lớp *jade.domain.DFService* mà có thể xuất bản và tìm kiếm các dịch vụ qua nhiều lời gọi phương thức.

3.4.2.1 Công bố dịch vụ

Một agent muốn đưa ra một hoặc nhiều dịch vụ phải cung cấp DF với một mô tả bao gồm AID riêng của mình, một danh sách các dịch vụ cung cấp tùy chọn danh sách các ngôn ngữ và ontology để các agent khác sử dụng để tương tác với nó. Mỗi mô tả dịch vụ được công bố phải bao gồm loại dịch vụ, tên dịch vụ, các ngôn ngữ và các ontology được yêu cầu để sử dụng dịch vụ và tập các thuộc tính đặc trưng của dịch vụ dưới dạng cặp giá trị khóa. Các lớp *DFAgentDescription*, *ServiceDescription* và *Property*, bao gồm trong gói *jade.domain.FIPAAGentManagement*, đại diện cho các khái niệm trừu tượng này.

Để công bố một dịch vụ, một agent phải tạo một mô tả thích hợp (như một thể hiện của lớp *DFAgentDescription*) và gọi phương thức tĩnh *register()* của lớp *DFService*. Với ví dụ tham khảo book-trading, các agent người bán đăng ký khả năng bán của chúng (một dịch vụ kiểu ‘Book-selling’) trong phương thức *setup()* của chúng như sau :

```

protected void setup() {
    ...

```

```

    // Register the book-selling service in the yellow
    pages
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("Book-selling");
    sd.setName(getLocalName()+"-Book-selling");
    dfd.addServices(sd);
    try {
        DFService.register(this, dfd);
    }
    catch (FIPAException fe) {
        fe.printStackTrace();
    }
    ...
}

```

Chú ý rằng trong ví dụ đơn giản này chúng ta không đặc tả bất kỳ ngôn ngữ, ontology hoặc các thuộc tính đặc trưng dịch vụ nào. Khi một agent kết thúc, nó thực hiện xóa các dịch vụ đã công bố:

```

protected void takeDown() {
    // Deregister from the yellow pages
    try {
        DFService.deregister(this);
    }
    catch (FIPAException fe) {
        fe.printStackTrace();
    }
    ...
}

```

3.4.3 Tìm kiếm dịch vụ

Một agent muốn tìm kiếm các dịch vụ phải cung cấp DF với một mô tả mẫu. Kết quả tìm kiếm là một danh sách tất cả các mô tả mà phù hợp với mẫu đã cung cấp. Theo các đặc tả FIPA, một mô tả phù hợp với mẫu nếu tất cả các lĩnh vực được đặc tả trong mẫu được diễn tả trong mô tả có giá trị như nhau. Phương thức tĩnh *search()* của lớp *DFService* có thể được sử dụng như ví dụ code được sử dụng bởi các agent người mua sách để duy trì một danh sách mới nhất các agent người bán:

```

public class BookBuyerAgent extends Agent {
    // The list of known seller agents
    private Vector sellerAgents;
    protected void setup() {
        ...
        // Update the list of seller agents every minute
        addBehaviour(new TickerBehaviour(this, 60000) {
            protected void onTick() {
                // Update the list of seller agents

```

```

        DFAgentDescription template      =      new
        DFAgentDescription();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Book-selling");
        template.addServices(sd);
        try {
            DFAgentDescription[] result      =
            DFService.search(myAgent,
            template);
            sellerAgents.clear();
            for (int i = 0; i < result.length; ++i) {
                sellerAgents.addElement(result
                [i].getName());
            }
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
    }
}
}

```

Lưu ý rằng tìm kiếm được lặp đi lặp lại một lần trong một phút kể từ khi các agent người bán có thể tự động xuất hiện và biến mất trong hệ thống. Jade DF cũng cung cấp một kỹ thuật cho phép các agent được thông báo ngay khi các agent khác đăng ký hoặc xóa khỏi các dịch vụ. Khai thác kỹ thuật này (điều đó có thể sẽ thích hợp hơn trong trường hợp của chúng ta) yêu cầu khởi tạo một giao thức FIPA-Subscribe với DF. Chúng ta sẽ mô tả điều này trong phần 5.4 khi thảo luận về các giao thức tương tác.

3.5 AGENT VỚI GIAO DIỆN ĐỒ HỌA

Một vấn đề điển hình mà những người phát triển phải làm là cách quản lý các agent Jade tương tác với GUI (giao diện đồ họa người dùng) của họ và ngược lại. Vấn đề ở đây là các mô hình lập trình giao tiếp giữa các luồng thích hợp phải được sử dụng giữa các luồng agent. Nó phải được đánh thức bất cứ khi nào nhận được một thông điệp ACL và AWT gửi đi và nó thức dậy bất cứ khi nào các thành phần của AWT (tức là các thành phần GUI) kích hoạt các kiểu sự kiện khác nhau (ví dụ người dùng nhấn vào một nút).

Vấn đề tiêu biểu cần tránh được phản ứng với một sự kiện AWT bằng cách chặn các sự kiện gửi đi cho tới khi một thông điệp ACL được nhận, hoặc cập nhật GUI từ luồng bên trong hoặc sửa đổi các biến không đồng bộ từ cả hai luồng. Bên dưới là một vài gợi ý về cách thực hành lập trình tốt để tránh một vài vấn đề này.

3.5.1 Thực hành lập trình tốt với bộ lắng nghe sự kiện AWT

Khi một agent có một GUI, nó cần phản ứng lại các hành động của người dùng, như là khởi đầu một hội thoại mới khi người dùng nhấn một nút. Khi sự kiện hành động AWT xảy ra, phương

thức *actionPerformed()* được gọi bằng sự kiện luồng gửi đi trên *ActionListener* đã đăng ký nguồn gốc sự kiện. Bên trong phương thức này, một thực hành lập trình tốt là để chuẩn bị một đối tượng lịch trình JADE *Behaviour* để thực hiện bằng luồng sự kiện.

Bên dưới là một đoạn mã từ RMA Agent of JADE (lớp *jade.tools.rma.rma*). Phương thức này được gọi khi người dùng tương tác với GUI và chọn một agent để tiêu diệt. Phương thức chỉ ra cách mà một hành vi được minh họa, các đối số của nó được chuẩn bị và lên lịch để thực hiện.

```
public void actionPerformed(ActionEvent e) {
    /*omissis*/
    AgentTree.Node curNode
        = (AgentTree.Node) panel.treeAgent.tree.getSelectionPath(
            );
    rma.killAgent(new AID(curNode.getName(), AID.ISLOCALNAME));
    /*omissis*/
}
public void killAgent(AID name) {
    KillAgent ka = new KillAgent();
    ka.setAgent(name);
    try {
        Action a = new Action();
        a.setActor(getAMS());
        a.setAction(ka);
        ACLMessage requestMsg = getRequest();
        requestMsg.setOntology(JADEManagementOntology.NAME);
        getContentManager().fillContent(requestMsg, a);
        addBehaviour(new AMSClientBehaviour("KillAgent",
            requestMsg));
    } catch(Exception fe) {
        fe.printStackTrace();
    }
}
```

Nhắc lại rằng các hành vi được lên lịch để thực hiện chỉ sau khi phương thức *setup()* của đối tượng agent đã được kết thúc. Hành vi luôn được thực hiện bởi các luồng agent. Kết quả là không có đồng bộ giữa các hành vi khác nhau được yêu cầu.

Tất nhiên, trong một vài trường hợp, thêm một hành vi là gánh nặng không cần thiết khi phản ứng đến hành động AWT phải được thay đổi một cách đơn giản giá trị của một biến hoặc chuẩn bị và gửi một ACLMessage (nhớ là phân phối thông điệp là hoàn toàn không đồng bộ). Đây là tất cả các hoạt động hợp lệ cho luồng AWT, nói chung, không gây ra vấn đề. Ngược lại, ngăn chặn các cuộc gọi (ví dụ *Agent.blockingReceive()*) không bao giờ được thực thi trong luồng AWT

3.5.2 Thực hành lập trình bằng cách sửa đổi giao diện đồ họa trong luồng thực thi của Agent

Có ý kiến cho rằng agent có các luồng thực thi của riêng nó, các chuyên gia lập trình Java sẽ ngay lập tức suy ra rằng cập nhật GUI từ bên trong các luồng này có thể dẫn đến các vấn đề bất

ngờ do các vấn đề đồng bộ hóa. AWT, Swing, MIDP (và hầu hết các framework giao diện người dùng khác) cung cấp một phương thức đặc biệt thích hợp xếp một đối tượng *Runnable* và khiến nó được thực hiện đồng bộ trên luồng sự kiện gửi đi GUI :

- `java.awt.EventQueue.invokeLater()` cho AWT
- `javax.swing.SwingUtilities.invokeLater()` cho Swing
- `javax.microedition.lcdui.Display.callSerially()` cho MIDP

Vì vậy, việc thực hành lập trình được giới thiệu là để đóng gói trong một đối tượng *Runnable* tất cả các truy cập đến các đối tượng GUI từ một hành vi JADE, nói chung, từ một luồng mà không phải là *EventDispatchThread*. Sau đó, các phương thức thích hợp nên được sử dụng để gửi đối tượng *Runnable* này tới *EventDispatchThread*.

Đoạn code bên dưới từ RMA Agent của lớp JADE (*jade.tools.rma.rma*). Phương thức này được gọi khi RMA Agent nhận một thông điệp thông báo rằng một agent mới đã được tạo ra trên một container đã định. Phương thức chỉ ra cách mà luồng agent tạo ra một thẻ hiện của đối tượng *Runnable* mới và gửi nó đến *EventDispatchThread*. Đối tượng *Runnable* này chịu trách nhiệm cập nhật GUI bằng cách tạo ra đối tượng *javax.swing.tree.TreeNode* mới và thêm nó vào *JTree* của container đã định.

```
public void addAgent(final String containerName, final AID agentID) {  
    Runnable addIt = new Runnable() {  
        public void run() {  
            String agentName = agentID.getName();  
            AgentTree.Node node =  
                tree.treeAgent.createNewNode(agentName, 1);  
            /* [omissis] */  
            tree.treeAgent.addAgentNode((AgentTree.AgentNode) node, c  
                ontainerName, agentName, agentAddresses, "FIPAAGENT");  
        }  
    };  
    SwingUtilities.invokeLater(addIt);  
}
```

CHƯƠNG 4

NHỮNG ĐẶC ĐIỂM NÂNG CAO CỦA JADE

Trong chương 3, chúng ta đã mô tả những đặc điểm cơ bản của JADE. Với những đặc tính này, chúng ta hoàn toàn có khả năng phát triển những hệ đa agent phân tán trên nền Jade. Tuy nhiên trong thực tế việc cài đặt các ứng dụng với các đặc tính đó tương đối phức tạp và người phát triển sẽ phải giải quyết rất nhiều vấn đề mà có thể sẽ trở nên dễ dàng hơn khi sử dụng các đặc tính nâng cao của nền JADE sẽ được trình bày trong chương này. Những vấn đề liên quan tới việc kiểm soát các biểu thức nội dung phức tạp bằng ontology và ngôn ngữ nội dung codes, khả năng xây dựng các hành vi phức tạp từ những hành vi đơn giản, hỗ trợ cho việc tạo ra các cuộc hội thoại được tuân theo các giao thức tương tác được định nghĩa bởi FIPA.

4.1 ONTOLOGY VÀ NGÔN NGỮ NỘI DUNG

Như chúng ta đã biết, cách đơn giản để truyền thông tin giữa các agent là sử dụng thông điệp ACL với nội dung là các chuỗi hoặc có kiểu chuỗi byte. Tuy nhiên, trong thực tế, các agent cần phải giao tiếp những thông tin phức tạp hơn. Ví dụ, để phân biệt các cuốn sách với nhau, ít nhất chúng ta cũng phải đưa ra các đặc điểm như tên sách, tên tác giả và nhà sản xuất. Khi mô tả những chuỗi thông tin như vậy, ta cần phải điều chỉnh sao cho đúng cú pháp để bên nhận có thể hiểu được nội dung thông điệp sau khi tách ra từng gói tin cụ thể (tên sách, tên tác giả, nhà sản xuất) theo các thuật ngữ của FIPA. Những cú pháp đó được gọi là ngôn ngữ nội dung. FIPA đã định nghĩa ngôn ngữ SL dùng để giao tiếp giữa hai agent đặc biệt của JADE là AMS và DF. Ví dụ sau đây mã hóa thông tin liên quan tới sách theo ngôn ngữ SL:

```
(Book :title "Programming Multi Agent System with JADE" :authors  
(sequence "F.Bellifemine" "G.Caire" "D.Greenwood") :editor Wiley)
```

Ý nghĩa của biểu thức này là miêu tả các đặc điểm của một cuốn sách có:

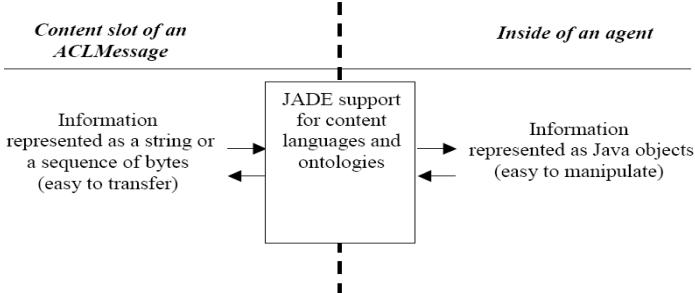
- Tên sách: Programming Multi Agent System with JADE
- Các tác giả: F.Bellifemine, G.Caire và D.Greenwood
- Nhà sản xuất: Wiley

Để hiểu được biểu thức này, agent bên nhận phải có những hiểu biết chung với agent bên gửi về các khái niệm: book, title, authors, editor. Tập các khái niệm đó được định nghĩa trong một tài liệu gọi là Ontology. Ontology có tính chất phụ thuộc vào miền ứng dụng. Mỗi miền ứng dụng khác nhau có tập Ontology khác nhau. Ví dụ, Ontology trong miền mua bán chứng khoán khác với Ontology trong miền mua bán sách.

Tuy nhiên, cách trao đổi thông tin theo kiểu chuỗi chỉ phù hợp cho việc đưa trực tiếp thông tin đó vào trong một thông điệp ACL. Điều này gây bất tiện cho agent khi xử lý thông điệp vì tại mỗi thời điểm có thông điệp được trao đổi, agent phải thực hiện một số bước:

- Bên gửi cần chuyển đổi mô tả bên trong thành biểu thức nội dung ACL và bên nhận phải thực hiện chuyển đổi ngược lại.
- Bên nhận cần phải thực hiện một số thao tác kiểm tra ngữ nghĩa để xác minh rằng các thông tin nhận được tuân theo các quy tắc của Ontology chung.

Với sự hỗ trợ của JADE trong việc cung cấp Ontology và ngôn ngữ nội dung, các bước này được thực hiện một cách tự động.



Hình 4.1: Hỗ trợ của Ontology và ngôn ngữ nội dung trong JADE

Có một số công nghệ khác có thể sử dụng như Serialization trong Java, hay XML...để miêu tả thông tin phức tạp. Tuy nhiên, chúng nằm ngoài phạm vi của bài viết này. Bạn đọc có thể tự tìm hiểu về ưu nhược điểm của từng công nghệ để có những lựa chọn phù hợp.

4.1.1 Các thành phần chính

Các thao tác chuyển đổi và kiểm tra như đã giới thiệu ở trên được thực hiện bởi một đối tượng quản lý nội dung thuộc lớp ContentManager trong gói jade.content. Mỗi agent có một đối tượng quản lý nội dung và có thể sử dụng đối tượng này bằng cách gọi phương thức `getContentManager()`. Lớp `ContentManager` cung cấp tất cả các phương thức dùng cho việc chuyển đổi đối tượng Java sang chuỗi (hoặc chuỗi byte) và ngược lại cũng như các phương thức để chèn chúng vào phần content trong thông điệp ACL.

Đối tượng quản lý nội dung cung cấp một giao diện để có thể làm việc này, nhưng thực chất nó lại giao cho Ontology và ngôn ngữ nội dung Codec. Ontology thực hiện kiểm tra thông tin dựa trên quan điểm về ngữ nghĩa, trong khi ngôn ngữ nội dung lại nhìn về mặt cấu trúc.

4.1.2 Mô hình tham chiếu nội dung

Để có thể thực hiện tốt việc kiểm tra ngữ nghĩa của biểu thức nội dung, JADE cần phân loại tất cả các thành phần liên quan đến miền theo các đặc điểm ngữ nghĩa chung của chúng. Việc phân loại này bắt nguồn từ ngôn ngữ ACL của FIPA. Ngôn ngữ này đòi hỏi nội dung của mỗi thông điệp ACLMessage phải có ngữ nghĩa tuân theo chức năng thực hiện (performative) của nó.

- **Vị từ (Predicate):** là những biểu thức diễn đạt trạng thái của sự vật hiện tượng và thường trả về giá trị true hoặc false. Chúng chủ yếu được sử dụng trong nội dung của các thông điệp dạng INFORM hoặc QUERY-IF, và không có ý nghĩa nếu sử dụng trong thông điệp REQUEST. Ví dụ, vị từ Works-for diễn đạt trạng thái một người có tên là John đang làm việc cho công ty có tên là TILAB

(Works-for (Person :name John) (Company :name TILAB))

- **Thuật ngữ (Term)**: là những biểu thức xác định những thực thể tồn tại trong thế giới thực mà agent nói chuyện và suy luận về chúng. Thuật ngữ được phân thành hai loại:

Khái niệm (Concept): là những biểu thức xác định những thực thể có cấu trúc phức tạp được định nghĩa trong các thuộc tính (slot). Ví dụ, khái niệm Person xác định một thực thể có các đặc điểm là: tên của người đó là John và tuổi của người đó là 33.

Person :name John :age 33)

Chúng thường không có ý nghĩa nếu sử dụng trực tiếp làm nội dung của thông điệp ACL mà thường được kết hợp bên trong vị từ hoặc khái niệm khác.

Ví dụ, khái niệm Person: (Person :name "J.R.R. Tolkjien") được kết hợp bên trong khái niệm Book trong biểu thức sau:

(Book :title "The Lord of the rings" :author (Person :name "J.R.R. Tolkjien"))

Hành động của agent (Agent action): là những khái niệm chỉ hành động có thể được thực hiện bởi một số agent. Chúng là nội dung của những thông điệp có kiểu REQUEST.

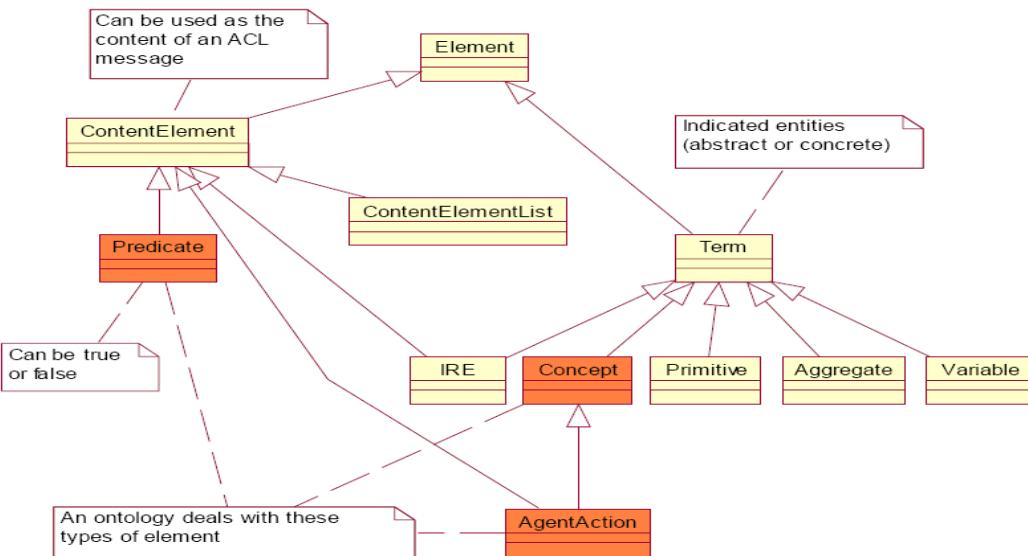
- **Primitive**: là những biểu thức xác định các thực thể nguyên tử như string, integer...
- **Aggregate**: là những biểu thức xác định các thực thể là nhóm các thực thể khác. Ví dụ, đây là một thực thể được tạo thành từ việc nhóm hai thực thể Person khác:
(sequence (Person :name John) (Person :name Bill))
- **Identifying Referential Expressions (IRE)**: là những biểu thức xác định các thực thể làm cho vị từ của nó đúng. Ví dụ, IRE sau có ý nghĩa là tìm tất cả những người đang làm việc cho công ty có tên TILAB:
(all ?x (Works-for ?x (Company :name TILAB)))
- **Variable**: là những biểu thức xác định một phần tử generic mà không biết trước được.

Ngôn ngữ nội dung biểu diễn và phân biệt giữa tất cả các kiểu phần tử trên, còn Ontology là một tập các lược đồ định nghĩa cấu trúc của predicate, agent action và concept.

Trong mô hình tham chiếu nội dung sau, hai kiểu phần tử nữa được giới thiệu. ContentElementList là một danh sách các phần tử thuộc ba kiểu: predicate, agent action và IRE. ContentElement là một kiểu cha được kế thừa bởi các kiểu trên.

4.1.3 Sử dụng Ontology và ngôn ngữ nội dung

Phần này trình bày những đặc trưng cơ bản của Ontology và ngôn ngữ nội dung trong JADE được cung cấp bởi gói jade.content. Những đặc trưng này rất có ý nghĩa trong nhiều tình huống, đặc biệt là cho những người đã làm quen với JADE. Gói jade.content cung cấp một số đặc trưng nâng cao như khả năng tạo các câu truy vấn. Điều này rất quan trọng trong những hệ thống phức tạp nhưng lại không được hỗ trợ trong phiên bản cũ. Những đặc trưng nâng cao này được trình bày trong phần sau.



Hình 4.2: Mô hình tham chiếu nội dung

Việc khám phá Ontology và ngôn ngữ nội dung phải trải qua các bước:

- Định nghĩa Ontology chứa các lược đồ của các kiểu predicate, agent action và concept được sử dụng trong miền ứng dụng.
- Tạo các lớp Java cho các kiểu predicate, agent action và concept.
- Lựa chọn ngôn ngữ nội dung thích hợp trong số những ngôn ngữ được hỗ trợ bởi JADE. JADE cho phép mở rộng bằng cách sử dụng ngôn ngữ nội dung do người dùng định nghĩa, tuy nhiên trong đa số trường hợp, người dùng không cần phải định nghĩa ngôn ngữ nội dung riêng.
- Đăng ký ontology và ngôn ngữ nội dung với agent.
- Tạo và xử lý các biểu thức nội dung như là các đối tượng Java thể hiện của các lớp đã được định nghĩa ở bước 2 và để cho JADE biên dịch chúng sang chuỗi hoặc chuỗi byte và ngược lại sao cho phù hợp với phần content của thông điệp ACL.

4.1.3.1 Định nghĩa Ontology

Ontology trong Jade là một thể hiện của lớp `jade.content.onto.Ontology` mà các giản đồ được thêm vào để xác định các kiểu vị từ; các khái niệm và các hành động của Agent liên quan đến lĩnh vực áp dụng. Các giản đồ là thể hiện của các lớp `PredicateSchema`, `AgentActionSchema` và `ConceptSchema` được chứa trong gói `jade.content.schema`. Mỗi lớp này có vài phương thức, mà nó có thể thể hiện những thuộc tính của các kiểu vị từ, khái niệm và hành động của Agent.

Về cơ bản, một ontology là một tập các giản đồ mà thường không phát triển trong suốt vòng đời của Agent, do đó tốt nhất ta nên khai báo nó là một đối tượng singleton và nên định nghĩa một phương thức static để truy cập đến đối tượng singleton này. Điều này cho phép chia sẻ cùng một đối tượng ontology giữa các agent khác nhau trong cùng JVM.

Trong ví dụ book-trading, chúng ta có thể mô hình lĩnh vực quan tâm bằng một ontology đơn giản mà bao gồm một khái niệm (BOOK), một vị từ (COSTS) và một hành động của Agent

(SELL). Thông thường, mỗi ontology trong Jade kế thừa một ontology cơ bản, được trình bày như một đối tượng duy nhất của lớp *jade.content.onto.BasicOntology*, bao gồm các giản đồ cho:

- Các kiểu nguyên thủy (*STRING, INTEGER, FLOAT, ...*)
- Các kiểu tập hợp
- Một số vị từ chung (tức là không thuộc bất kỳ lĩnh vực đặc biệt nào); các khái niệm và các hành động của Agent, trong đó là khái niệm AID nhận dạng một Agent

Để thể hiện điều đó, ontology O₁ kế thừa ontology O₂ (tức là, tất cả các vị từ, các khái niệm và các hành động của Agent được chứa trong O₂ cũng được chứa trong O₁), khi đó cần truyền O₂ làm tham số khi O₁ được khởi tạo.

Sau đây là khai báo Ontology của ví dụ bookTrading:

```
package bookTrading.ontology;
import jade.content.onto.*;
import jade.content.schema.*;
public class BookTradingOntology extends Ontology {
    // The name identifying this ontology
    public static final String ONTOLOGY_NAME =
        "Book-trading-ontology";
    // VOCABULARY

    public static final String BOOK = "Book";
    public static final String BOOK_TITLE = "title";
    public static final String BOOK_AUTHORS = "authors";
    public static final String BOOK_EDITOR = "editor";
    public static final String COSTS = "Costs";
    public static final String COSTS_ITEM = "item";
    public static final String COSTS_PRICE = "price";
    public static final String SELL = "Sell";
    public static final String SELL_ITEM = "item";
    // The singleton instance of this ontology
    private static Ontology theInstance = new BookTradingOntology();
    // Retrieve the singleton Book-trading ontology instance
    public static Ontology getInstance() {
        return theInstance;
    }
    // Private constructor
    private BookTradingOntology() {
        // The Book-trading ontology extends the basic ontology
        super(ONTOLOGY_NAME, BasicOntology.getInstance());
        try {
            add(new ConceptSchema(BOOK), Book.class);
            add(new PredicateSchema(COSTS), Costs.class);
            add(new AgentActionSchema(SELL), Sell.class);
            // Structure of the schema for the Book concept
            ConceptSchema cs = (ConceptSchema) getSchema(ITEM);
            cs.add(BOOK_TITLE, (PrimitiveSchema)
                getSchema(BasicOntology.STRING));
            cs.add(BOOK_AUTHORS, (PrimitiveSchema)
                getSchema(BasicOntology.STRING), 0,
                ObjectSchema.UNLIMITED);
        }
    }
}
```

```

        cs.add(BOOK_EDITOR, (PrimitiveSchema)
getSchema(BasicOntology.STRING),
ObjectSchema.OPTIONAL);
        // Structure of the schema for the Costs predicate
        . . .
        // Structure of the schema for the Sell agent action
AgentActionSchema as = (AgentActionSchema)
getSchema(SELL);
        as.add(SELL_ITEM, (ConceptSchema) getSchema(BOOK));
    }
    catch (OntologyException oe) {
        oe.printStackTrace();
    }
}

```

Từ đoạn mã trên, chúng ta có thể quan sát thấy rằng:

- Mỗi giản đồ được gửi đến ontology được kết hợp với một lớp Java, ví dụ giản đồ cho khái niệm *BOOK* được kết hợp với lớp *Book.java*. Những lớp Java này phải có 1 cấu trúc thích hợp như miêu tả trong phần 4.1.3.2
 - Mỗi thuộc tính trong giản đồ có một tên và một kiểu, tức là giá trị cho thuộc tính phải tuân theo một giản đồ định sẵn
 - Một thuộc tính có thẻ *OPTIONAL* (tùy ý) nghĩa là giá trị của nó có thể là *null*. Mặt khác, một thuộc tính là *MANDATORY* (bắt buộc). Nếu thuộc tính *MANDATORY* có giá trị là *null* thì một ngoại lệ của lớp *OntologyException* được ném ra.
 - Một thuộc tính có thẻ có *cardinality* (lực lượng) > 1 , tức là một danh sách hay một tập hợp. Ví dụ, thuộc tính *authors* trong giản đồ khái niệm *BOOK* có thể chứa nhiều yếu tố kiểu *String* (nhiều tác giả)

Có thể xác định quan hệ mở rộng/cụ thể hóa giữa các khái niệm. Ví dụ, nếu chúng ta đã mở rộng hệ thống để hỗ trợ việc kinh doanh các loại hàng hóa khác như CDs, chúng ta có thể xác định một giản đồ *ITEM* chung và bổ sung nó như giản đồ cha cho cả giản đồ *BOOK*, giản đồ *CD*... Một giản đồ sẽ tự động được kế thừa tất cả các thuộc tính được chứa trong chính giản đồ cha của nó. Giản đồ cha được bổ sung bằng phương thức *addSuperSchema()* của lớp *ConceptSchema*

4.1.3.2 Phát triển các lớp Java liên quan đến Ontology

Như đã đề cập trong phần 4.1.3.1, mỗi giản đồ được chứa trong một ontology được kết hợp với một lớp Java (hay Interface). Rõ ràng cấu trúc của những lớp này phải liên quan với những giản đồ, tức là chúng phải tuân theo các quy tắc sau:

- (1) Chúng phải thực thi một giao diện thích hợp :

 - Nếu giản đồ là một lớp *ConceptSchema* thì phải thực hiện giao diện *Concept* (hoặc gián tiếp hoặc trực tiếp)
 - Nếu giản đồ là một lớp *PredicateSchema* thì phải thực hiện giao diện *Predicate* (hoặc gián tiếp hoặc trực tiếp)

- Nếu giản đồ là một lớp *AgentActionSchema* thì phải thực hiện giao diện *AgentAction* (hoặc gián tiếp hoặc trực tiếp)
- (2) Chúng phải có quan hệ kế thừa thích hợp, tức là S_1 là giản đồ cha của S_2 , sau đó lớp C_2 liên kết với giản đồ S_2 phải kế thừa lớp C_1 liên kết với giản đồ S_1
- (3) Chúng phải có các phương thức truy cập tới các trường:
- Đối với mỗi thuộc tính trong giản đồ S_1 với tên Nnn và kiểu là S_2 , lớp C_1 liên kết với giản đồ S_1 phải có 2 phương thức truy cập:
- ```
public void setNnn(C2 c);
public C2 getNnn();
```
- $C_2$  là lớp liên kết với giản đồ  $S_2$ . Thực tế, nếu  $S_2$  là giản đồ được xác định trong *BasicOntology*, sau đó :
- o Nếu  $S_2$  là giản đồ cho *STRING*  $\rightarrow C_2$  là *java.lang.String*
  - o Nếu  $S_2$  là giản đồ cho *INTEGER*  $\rightarrow C_2$  là *int, long, java.lang.Integer* hay *java.lang.Long*
  - o Nếu  $S_2$  là giản đồ cho *BOOLEAN*  $\rightarrow C_2$  là *Boolean* hay *java.lang.Boolean*
  - o Nếu  $S_2$  là giản đồ cho *FLOAT*  $\rightarrow C_2$  là *float, double, java.lang.Float* hay *java.lang.Double*
  - o Nếu  $S_2$  là giản đồ cho *DATE*  $\rightarrow C_2$  là *java.util.Date*
  - o Nếu  $S_2$  là giản đồ cho *BYTE SEQUENCE*  $\rightarrow C_2$  là *byte[]*
  - o Nếu  $S_2$  là giản đồ cho *AID*  $\rightarrow C_2$  là *jade.core.AID*
- Đối với mỗi thuộc tính trong  $S_1$  có tên  $Nnn$ , kiểu  $S_2$  và cardinality > 1, lớp  $C_1$  liên kết với giản đồ  $S_1$  phải có 2 phương thức truy cập sau:
- ```
public void setNnn(jade.util.leap.List l);
public jade.util.leap.List getNnn();
```

Sau đây là các lớp liên kết với khái niệm *BOOK* và vị từ *COSTS*:

```
// Class associated to the BOOK schema
package bookTrading.ontology;
import jade.content.Concept;
import jade.util.leap.List;
public class Book implements Concept {
    private String title;
    private List authors;
    private String editor;
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public List getAuthors() {
        return authors;
    }
    public void setAuthors(List authors) {
```

```

        this.authors = authors;
    }
    public String getEditor() {
        return editor;
    }
    public void setEditor(String editor) {
        this.editor = editor;
    }
}

// Class associated to the COSTS schema
package bookTrading.ontology;
import jade.content.Predicate;
import jade.core.AID;
public class Costs implements Predicate{
    private Book item;
    private int price;
    public Book getItem() {
        return item;
    }
    public void setItem(Book item) {
        this.item = item;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}

```

4.1.3.3 Chọn ngôn ngữ nội dung

Gói jade.content gồm các codec cho hai ngôn ngữ nội dung (ngôn ngữ SL và ngôn ngữ LEAP), cả hai đều hỗ trợ mô hình tham chiếu nội dung. Codec của ngôn ngữ nội dung L là đối tượng Java có khả năng quản lý các biểu thức nội dung được viết bằng ngôn ngữ L. Đa số trường hợp, người phát triển phải chấp nhận một trong các ngôn ngữ này và sử dụng codec tương ứng mà không cần phải làm thêm gì nữa. Phần này sẽ cung cấp một số so sánh có thể giúp cho việc lựa chọn ngôn ngữ. Nếu nhà phát triển muốn agent có thể nói được ngôn ngữ khác, anh ta phải định nghĩa một codec cho nó. Xem chi tiết trong phần sau.

Ngôn ngữ SL là ngôn ngữ nội dung được mã hóa dưới dạng các chuỗi ký tự mà con người có thể đọc được và có thể là ngôn ngữ phổ biến nhất trong giới khoa học làm việc về agent thông minh. Tất cả những ví dụ về biểu thức nội dung trong tài liệu này đều là các biểu thức của ngôn ngữ SL. Chúng ta thường sử dụng ngôn ngữ này cho những ứng dụng dựa trên agent có tính mở (nơi mà agent của các nhà phát triển khác nhau và chạy trên các nền tảng khác nhau phải giao tiếp với nhau). SL gồm một số toán tử như:

- Toán tử logic: AND, OR, NOT.
- Toán tử mô hình: BELIEF, INTENTION, UNCERTAINTY.

Ngoài ra, tính chất con người có thể đọc được ngôn ngữ này giúp ích cho việc soát lỗi và kiểm thử ứng dụng.

Ngôn ngữ LEAP là ngôn ngữ được mã hóa dưới dạng các byte mà con người không thể đọc được. Nó xác định cho JADE trong dự án LEAP. Do đó, rõ ràng là chỉ những agent của JADE mới có khả năng nói được ngôn ngữ LEAP. Tuy nhiên, một số trường hợp mà LEAP được ưa chuộng hơn SL:

- Lớp LEAPCodec nhẹ hơn lớp SLCodec. Khi có giới hạn về bộ nhớ nên dùng LEAP.
- Ngôn ngữ SL không hỗ trợ chuỗi byte.

Tuy nhiên, ngôn ngữ SL có thể làm việc với agent action. Do đó, tất cả các agent action trong SL đều phải được insert vào thành phần ACTION để gán hành động cho AID của agent được dự định thực hiện hành động. Do đó, biểu thức

```
(Sell
  (Book :title "The Lord of the rings")
  (agent-identifier :name Peter)
)
```

Không thể sử dụng trực tiếp làm nội dung thông điệp REQUEST kể cả khi nó tương ứng với một agent action trong mô hình tham chiếu nội dung. Cụ thể là ngữ pháp của SL không cho phép nó là biểu thức ở mức đầu tiên. Thay vào đó, chúng ta sử dụng biểu thức:

```
(ACTION
  (agent-identifier :name John)
  (Sell
    (Book :title "The Lord of the rings")
    (agent-identifier :name Peter)
  )
)
```

Trong đó, John là agent được yêu cầu bán quyền sách cho Peter agent.

Ngoài ra còn có ngôn ngữ nội dung XML sử dụng cú pháp XML như sau:

```
<action>
  <agent-identifier>
    <name>seller-X</name>
  </agent-identifier>
  <Sell>
    <Book>
      <title>Developing Multi Agent Systems with
      JADE</title>
    </Book>
  </Sell>
```

```
</action>
```

Ngôn ngữ này được sử dụng khi tập các thực thể của ontology được export/import từ một hệ thống bên ngoài.

4.1.3.4 Đăng ký ngôn ngữ nội dung và Ontology với Agent

Trước khi agent có thể sử dụng ontology và ngôn ngữ nội dung, chúng phải được đăng ký với bộ đối tượng quản lý nội dung của agent. Thao tác này được thực hiện trong quá trình agent thiết lập (nhưng không bắt buộc) (trong phương thức setup() của lớp Agent).

```
public class BookSellerAgent extends Agent {  
    ...  
    private Codec codec = new SLCodec();  
    private Ontology ontology =  
        BookTradingOntology.getInstance();  
    ...  
    protected void setup() {  
        ...  
        getContentManager().registerLanguage(codec);  
        getContentManager().registerOntology(ontology)  
        ...  
    }  
    ...  
}
```

Từ đây, bộ phận quản lý nội dung sẽ liên kết Codec và Ontology đã đăng ký với các chuỗi được trả về bởi các phương thức getName().

4.1.3.5 Tạo và xử lý các biểu thức nội dung như là các đối tượng Java

Sau khi có ontology, ngôn ngữ nội dung và đăng ký chúng với bộ quản lý nội dung, cần tạo và xử lý biểu thức nội dung như là các đối tượng Java. Phương thức fillContent() và extractContent() sẽ lấy đối tượng Ontology và đối tượng Codec và để cho chúng thực hiện các công việc chuyển đổi và kiểm tra cần thiết.

```
private class CallForOfferServer extends CyclicBehaviour {  
    public void action() {  
        ACLMessage msg = myAgent.receive();  
        if (msg != null) {  
            // Message received. Process it  
            ACLMessage reply = msg.createReply();  
            try {  
                ContentManager cm = myAgent.getContentManager();  
                Action act = (Action) cm.extractContent(msg);  
                Sell sellAction = (Sell) act.getAction();  
                Book book = sellAction.getItem();  
                PriceManager pm = (PriceManager)  
                    catalogue.get(book.getTitle());  
                if (pm != null) {  
                    // The requested book is available for sale  
                    reply.setPerformative(ACLMessage.PROPOSE);  
                }  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```

        ContentElementList cel = new
ContentElementList();
        cell.add(act);
        Costs costs = new Costs();
        costs.setItem(book);
        costs.setPrice(pm.getCurrentPrice());
        cel.add(costs);
        cm.fillContent(reply, cel);
    }
    else {
        // The requested book is NOT available for
sale.
        reply.setPerformative(ACLMessage.REFUSE);
    }
}
catch (OntologyException oe) {
    oe.printStackTrace();
    reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
}
catch (CodecException ce) {
    ce.printStackTrace();
    reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
}
myAgent.send(reply);
}
}
} // End of inner class CallForOfferServer

```

Kết quả :content của thông điệp phản hồi PROPOSE chỉ tới cuốn sách “Developing Multi Agent Systems with JADE” đang được bán với giá 30 euro.

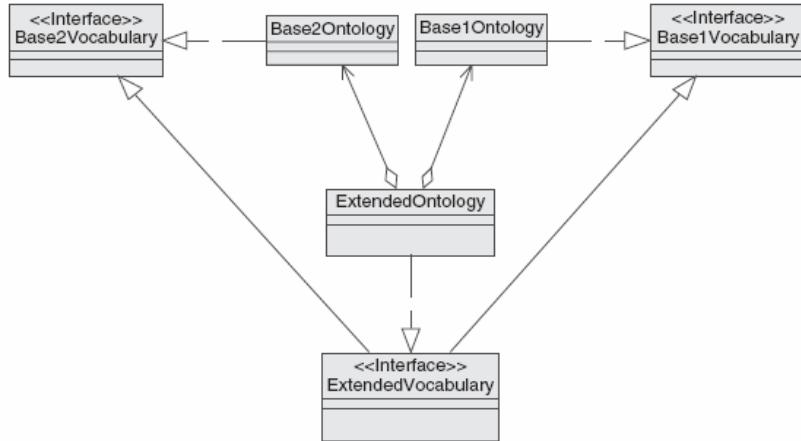
```

((action
  (agent-identifier :name seller@JADE-book-trading)
  (Sell
    (Book :title "Developing Multi Agent Systems with JADE"
          :authors (sequence Bellifemine Caire Greenwood)
          :editor Wiley) )
  (Costs
    (Book :title "Developing Multi Agent Systems with JADE"
          :authors (sequence Bellifemine Caire Greenwood)
          :editor Wiley)
    30))

```

4.1.3.6 Kết hợp các Ontology

Sự hỗ trợ ngôn ngữ nội dung và ontology trong gói jade.content cung cấp một cách dễ dàng để kết hợp các ontology do đó trợ giúp việc tái sử dụng mã nguồn. Cụ thể, có thể xác định một ontology mới mở rộng từ một hoặc nhiều ontology bằng cách đơn giản là xác định các ontology được mở rộng như là các tham số trong hàm khởi tạo được dùng để tạo ra ontology mới.



Hình 4.3: Mẫu xây dựng giao diện từ vựng

Khi thực hiện điều này chúng ta nên sử dụng mẫu xây dựng giao diện từ vựng như trong hình 4.3. Tất cả các biểu tượng được sử dụng cho tên của các khái niệm, thuộc tính, agent actions, và slot của nó được nhóm vào trong một giao diện biểu diễn từ vựng. Ví dụ như trong book-trading, chúng ta có thể có một giao diện BookTradingVocabulary như sau:

```

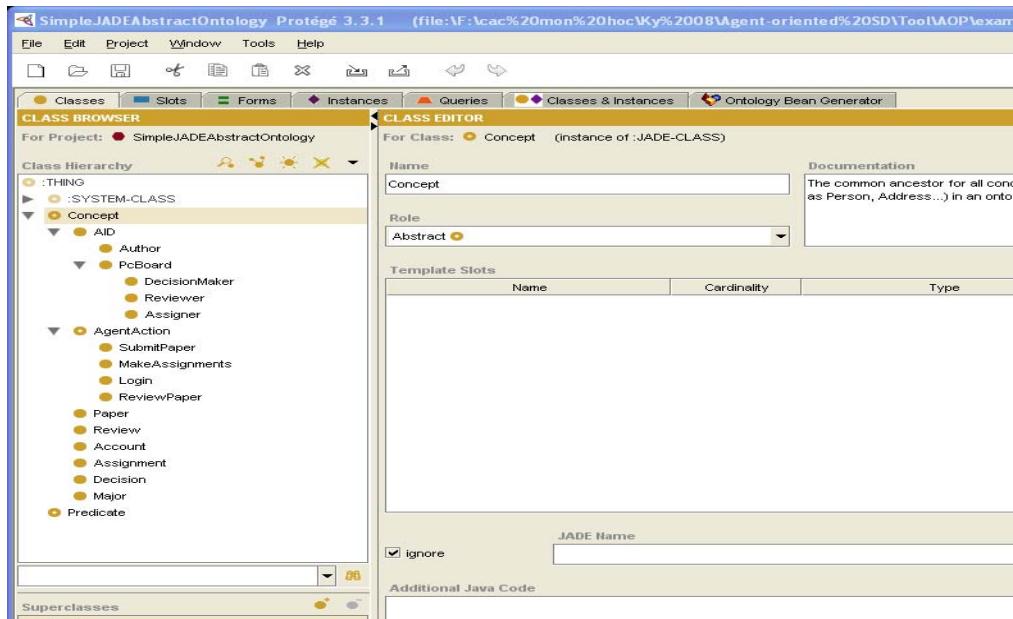
public interface BookTradingVocabulary {
    // VOCABULARY
    public static final String BOOK = "Book";
    public static final String BOOK_TITLE = "title";
    public static final String BOOK_AUTHORS = "authors";
    public static final String BOOK_EDITOR = "editor";
    public static final String COSTS = "Costs";
    public static final String COSTS_ITEM = "item";
    public static final String COSTS_PRICE = "price";
    public static final String SELL = "Sell";
    public static final String SELL_ITEM = "item";
}
  
```

Lớp ontology cài đặt giao diện từ vựng này. Khi mở rộng một hoặc nhiều ontology, lớp ontology kết quả bao gồm các ontology được mở rộng, trong khi giao diện từ vựng kết quả kề thura các giao diện từ vựng được mở rộng.

4.1.4 Sử dụng Protégé và BeanGenerator add-on để tạo Ontology cho JADE

Protégé là một công cụ đồ họa hỗ trợ xây dựng ontology một cách nhanh chóng và chính xác. Protégé hỗ trợ hai cách chính để mô hình hóa ontology là Protégé-Frames và Protégé-OWL. Ontology sau khi được tạo ra có thể được đưa vào các định dạng khác như RDF(S), OWL và XML Schema. Protégé được xây dựng dựa trên Java, có khả năng mở rộng và cung cấp môi trường plug-and-play giúp cho việc xây dựng khuôn mẫu và phát triển ứng dụng một cách nhanh chóng.

BeanGenerator là một plugin cho Protégé cho phép sinh các lớp Java biểu diễn trong ontology mà có thể được sử dụng với môi trường JADE. Với BeanGenerator, chúng ta có thể sinh các ontology tuân theo chuẩn FIPA/JADE từ RDF(S), XML và các dự án Protégé. Giao diện của Protégé:



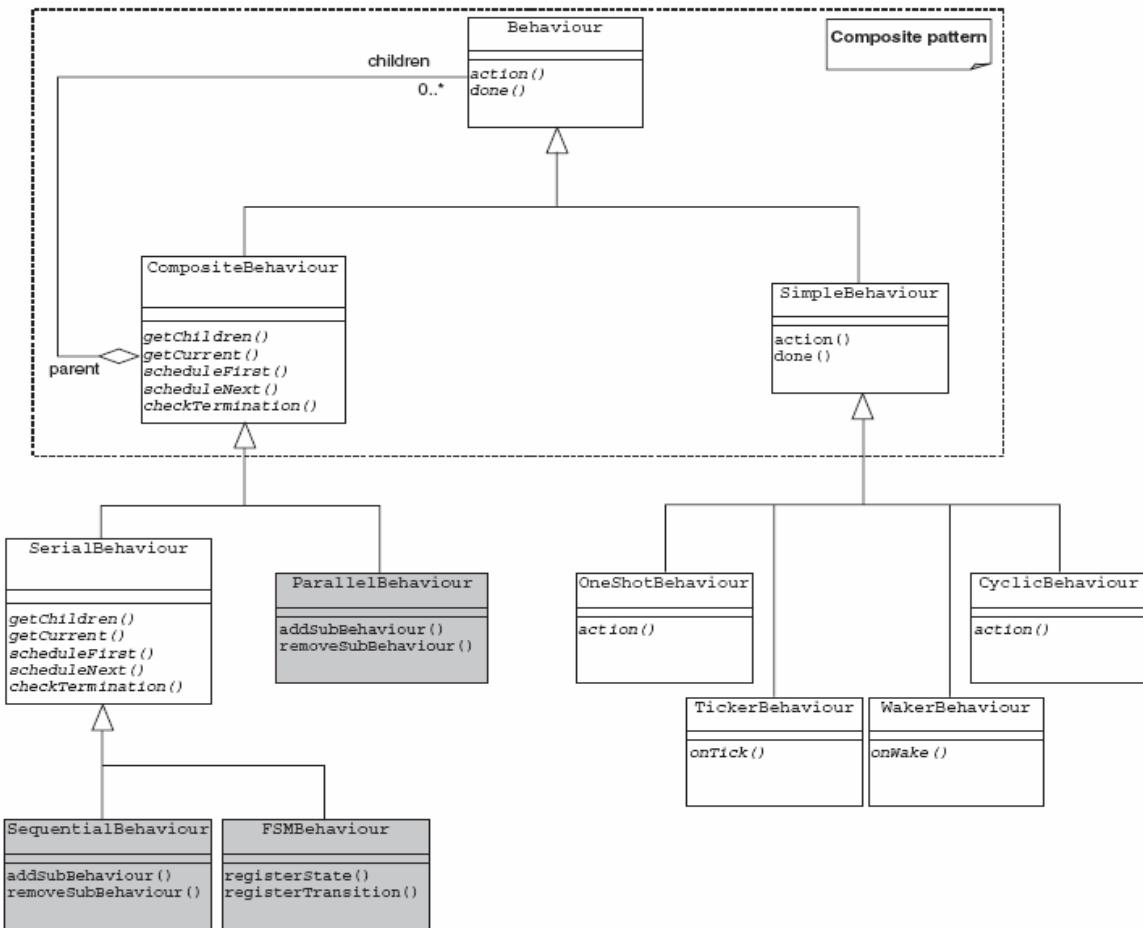
Hình 4.4: Giao diện Protege

4.2 HỢP CÁC HÀNH VI ĐỂ XÂY DỰNG CÁC TÁC VỤ PHÚC TẠP

Như đã mô tả trong phần 3.2, các công việc trong JADE được thực hiện bằng cách xây dựng các lớp mở rộng của lớp *jade.core.behaviours.Behaviour* và cài đặt các phương thức *action()* và *done()*. Tuy nhiên, khi liên kết với các công việc phức tạp liên quan đến các bước tính toán, có thể pha trộn với các agent khác ...thì điều này xem ra không thuận lợi. Chúng ta hãy xem xét ví dụ: hành vi BookNegotiator trình bày trong mục 3.3.5. Mặc dù nó chỉ đơn giản là trao đổi một vài thông điệp và lấy một quyết định, phương thức *action()* của nó khá là phức tạp.

Một phương pháp đơn giản và rõ ràng hơn để thực hiện các nhiệm vụ phức tạp trong Jade là kết hợp các hành vi – tạo nhiệm vụ phức tạp từ các hành vi đơn giản. Cơ sở cho các đặc trưng này được cung cấp bởi lớp *CompositeBehaviour* trong gói *jade.core.behaviours*. Lớp này có các phương thức *scheduleFirst()* và *scheduleNext()* dùng để lập lịch các hành vi con. Những phương pháp này được khai báo trừu tượng và phải được định nghĩa trong các lớp con của *CompositeBehaviour*.

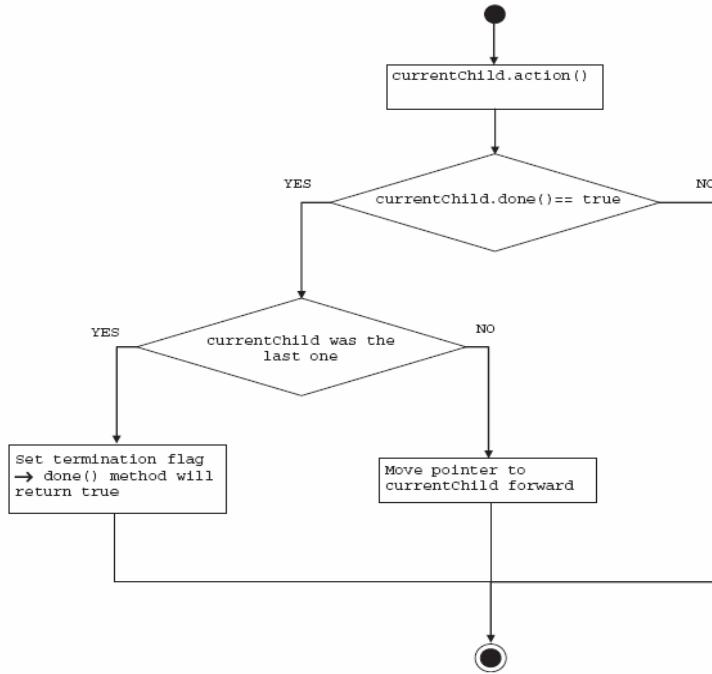
Ba kiểu hành vi kép được cung cấp trong JADE là *SequentialBehaviour*, *FSMBehaviour* và *ParallelBehaviour* sẽ được trình bày chi tiết trong các phần sau. Các nhà phát triển không cần phải trực tiếp mở rộng lớp *CompositeBehaviour* và chỉ sử dụng một thể hiện của các lớp con *SequentialBehaviour*, *FSMBehaviour* và *ParallelBehaviour* của nó.



Hình 4.5: Cấu trúc phân cấp các hành vi trong JADE

4.2.1 Lớp SequentialBehaviour

Lớp SequentialBehaviour cài đặt một hành vi gộp để lập lịch các hành vi con theo một chính sách tuần tự đơn giản. Nó bắt đầu hành vi con đầu tiên, sau khi hoàn thành nó chuyển sang hành vi con tiếp theo và cứ như thế cho đến khi thực hiện hết các hành vi con.



Hình 4.6: Luồng thực hiện phương thức action() của lớp SequentialBehaviour

Các hành vi con được add vào bằng phương thức addSubBehaviour(). Thứ tự được đưa vào chính là thứ tự chúng được lập lịch. Ví dụ, lớp ThreeStepBehaviour:

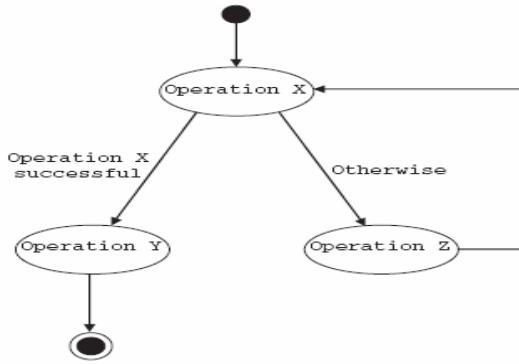
```

SequentialBehaviour threeStepBehaviour = new
    SequentialBehaviour(anAgent);
threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
    public void action() {
        // perform operation X
    }
});
threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
    public void action() {
        // perform operation Y
    }
});
threeStepBehaviour.addSubBehaviour(new OneShotBehaviour(anAgent) {
    public void action() {
        // perform operation Z
    }
});

```

4.2.2 Lớp FsmBehaviour

Lớp FsmBehaviour cài đặt một hành vi gộp trong đó các hành vi con được lập lịch theo một máy hữu hạn trạng thái (FSM). Lớp FsmBehaviour cung cấp 2 phương thức để đăng ký các hành vi con làm các trạng thái của máy hữu hạn trạng thái và để đăng ký sự di chuyển giữa các trạng thái. Tương tự như hành vi chuỗi, một hành vi FSM cũng chứa con trỏ chỉ tới hành vi con hiện thời. Sau khi hoàn thành (phương thức done() trả về true), hành vi FSM kiểm tra bảng chuyển dịch, trên cơ sở đó chọn ra hành vi con mới để thực hiện tiếp.



Hình 4.7: Một máy hữu hạn trạng thái đơn giản

Các chuyển dịch trong hành vi FSM được đánh dấu bằng một nhãn là một số nguyên. Khi hành vi con hiện thời hoàn thành nhiệm vụ, giá trị trả về của phương thức `onEnd()` được lấy làm giá trị thoát và được so sánh với nhãn của các chuyển dịch đang thoát ra từ trạng thái con hiện thời. Chuyển dịch đầu tiên có nhãn trùng với giá trị thoát sẽ được thực hiện và trạng thái đích của nó sẽ là trạng thái con hiện thời mới. Phương thức `registerState()` dùng để thêm trạng thái và `FSMBehaviour` sẽ được thực thi trong trạng thái đó. Phương thức `registerTransition()`, dùng để thêm chuyển dịch vào `FSMBehaviour`, chấp nhận 3 tham số: hai tham số có kiểu String định nghĩa trạng thái nguồn và trạng thái đích của chuyển dịch và một giá trị kiểu int định nghĩa nhãn đánh dấu chuyển dịch đó. Các phương thức `registerFirstState()` và `registerLastState()` cho phép đăng ký trạng thái bắt đầu và trạng thái kết thúc. Tuy nhiên, cần chú ý rằng, chỉ có một trạng thái bắt đầu nhưng chỉ có một trạng thái kết thúc. Toàn bộ hành vi FSM sẽ kết thúc khi đạt được trạng thái kết thúc và đã thực thi đầy đủ. Ví dụ, minh họa code của một hành vi FSM trong hình 4.7:

```

FSMBehaviour sampleFSM = new FSMBehaviour(anAgent);
sampleFSM.registerFirstState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation X
}
public int onEnd() {
return (operation X successful ? 1 : 0);
}
}, "X");
sampleFSM.registerLastState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation Y
}
}, "Y");
sampleFSM.registerState(new OneShotBehaviour(anAgent) {
public void action() {
// Perform operation Z
}
}, "Z");
sampleFSM.registerTransition("X", "Y", 1);
sampleFSM.registerTransition("X", "Z", 0);
sampleFSM.registerDefaultTransition("Z", "X", new String[]{"X",
"Z"});
  
```

Phương thức registerDefaultTransition() của lớp FSMBehaviour cho phép định nghĩa một chuyển dịch mặc định giữa hai trạng thái. Một chuyển dịch mặc định không được đánh dấu bằng nhãn và xảy ra khi và chỉ khi các chuyển dịch khác (nếu có) đang thoát khỏi cùng trạng thái đó không xảy ra. Cả hai phương thức registerTransition() và registerDefaultTransition() đều lấy tham số là String[] với các phiên bản nạp chồng nhau. Tham số này xác định một tập các trạng thái FSM mà phải được thiết lập lại khi xảy ra trạng thái đã đăng ký. Điều này rất hữu ích khi đăng ký các trạng thái backward, cụ thể là các chuyển dịch mà dẫn đến các trạng thái mà đã từng xảy ra. Thực tế, như đã giải thích trong phần 3.2.3, một đối tượng Behaviour một khi đã được thực thi thì phải được thiết lập lại bằng cách gọi phương thức reset trước khi nó lại được thực thi. Ví dụ, trong hình 4.7, nếu chuyển dịch từ Z đến X xảy ra, trạng thái X và có thể là cả trạng thái Z sẽ được thực thi lần nữa. Trước khi điều này xảy ra, chúng phải được reset để tránh những tác động không mong muốn.

4.2.3 Lớp ParallelBehaviour

Lớp này cài đặt một hành vi gộp để lập lịch các hành vi con một cách song song. Thông thường, khi gặp phải các hành vi FSM, việc lập lịch có sự phối hợp và không ngừng. Nghĩa là mỗi khi phương thức action() của hành vi song song được thực thi, nó gọi phương thức action() của hành vi con hiện thời và sau đó chuyển con trỏ tới hành vi con tiếp theo mà không cần quan tâm đến việc nó đã hoàn thành hay chưa. Các hành vi con trong hành vi song song được thêm vào bằng cách gọi phương thức addBehaviour(). Một hành vi song song có thể kết thúc khi tất cả các hành vi con của nó hoàn thành, hoặc khi hành vi con đầu tiên hoàn thành. Chính sách kết thúc được chọn trong thời gian khởi tạo bằng các xác định trong hàm khởi tạo là WHEN-ALL hay WHEN-ANY. Chính sách WHEN-ANY thường được sử dụng để kết thúc một nhiệm vụ trong trường hợp nó không hoàn thành trong khoảng thời gian timeout, ví dụ:

```
Behaviour task = new MyTask();
ParallelBehaviour pb = new ParallelBehaviour(anAgent,
ParallelBehaviour.WHEN_ANY);
pb.addSubBehaviour(task);
pb.addSubBehaviour(new WakerBehaviour(anAgent, 60000) {
public void onWake() {
System.out.println("timeout expired");
}
});
}
```

4.2.4 Chia sẻ dữ liệu giữa các hành vi con: DATASTORE

Khi gộp các hành vi thành hành vi chuỗi, FSM hay song song thì thông thường hành vi con sẽ cần truy cập một số dữ liệu được tạo ra bởi các hành vi con khác. Tuy nhiên, những dữ liệu này không thể được truyền vào như là tham số trong hàm khởi tạo của hành vi con vì tất cả các hành vi con đều thường được khởi tạo trước khi toàn bộ hành vi gộp được thực thi. Thông thường, khi các hành vi cần chia sẻ dữ liệu, cần sử dụng các biến thành viên của agent hoặc của hành vi gộp. Ví dụ, giả sử chúng ta cần một hành vi chuỗi trong đó ở bước n phải nhận một thông điệp và ở bước n+1 phải làm một số công việc xử lý hành vi nhận được. Chúng ta có đoạn code sau:

```
public class MySequentialBehaviour extends SequentialBehaviour {
```

```

private ACLMessage receivedMsg;
public MySequentialBehaviour(Agent a) {
    super(a);
    // . .
    addSubBehaviour(new SimpleBehaviour(a) {
        private boolean finished = false;
        public void action() {
            receivedMsg = myAgent.receive();
            if (receivedMsg != null) {
                finished = true;
            }
            else {
                block();
            }
        }
        public boolean done() {
            return finished;
        }
    });
    addSubBehaviour(new OneShotBehaviour(a) {
        public void action() {
            // Process receivedMsg
        }
    });
}
}

```

Tuy nhiên, trong nhiều trường hợp, việc tạo các hành vi có thể dùng lại trong nhiều ngữ cảnh khác nhau là rất có ích và do đó không phải gắn bó với một agent hoặc với một hành vi gộp cha. Trong những trường hợp này, dữ liệu chia sẻ giữa các hành vi không thể được lưu trong các biến thành viên của agent hoặc của hành vi cha. Lớp DataStore có trong gói jade.core.behaviours cung cấp một giải pháp đơn giản và toàn diện cho vấn đề này. Mỗi hành vi có một kho dữ liệu riêng (cụ thể là một thê hiện của DataStore) có thể truy cập được bằng các phương thức getDataStore() và setDataStore() của lớp Behaviour. Một kho dữ liệu đơn giản là một map (thực tế DataStore extend HashMap) và cung cấp cơ chế chuẩn (các hành vi có thể dùng lại) để chia sẻ dữ liệu. Nghĩa là, bằng cách thiết lập cùng một thê hiện DataStore cho một hoặc nhiều hành vi, các hành vi này sẽ có một không gian chung nơi mà chúng có thể lưu trữ dữ liệu được chia sẻ. Ví dụ, giả sử ta có hành vi MessageReceiver nhận một thông điệp và ta muốn sử dụng nó trong bước n của ví dụ trên. Sau đây là đoạn code:

```

public class MessageReceiver extends SimpleBehaviour {
    public static final String RECV_MSG = "received-message";
    private boolean finished = false;
    public void action() {
        ACLMessage msg = myAgent.receive();
        if (msg!= null) {
            getDataStore().put(RECV_MSG, msg);
            finished = true;
        }
        else {

```

```

        block();
    }
}

public boolean done() {
    return finished;
}
}

```

Ở đây, chúng ta có thể sửa hành vi chuỗi để tận dụng ưu điểm của lớp MessageReceiver:

```

SequentialBehaviour sb = new SequentialBehaviour(anAgent);
Behaviour b = new MessageReceiver(anAgent);
b.setDataStore(sb.getDataStore());
sb.addSubBehaviour(b);
b = new OneShotBehaviour(anAgent) {
    public void action() {
        ACLMessage receivedMsg = getDataStore()
            .get(MessageReceiver.RECV_MSG);
        // Process receivedMsg
    }
};
b.setDataStore(sb.getDataStore());
sb.addSubBehaviour(b);

```

4.2.5 Bổ sung về hành vi gộp

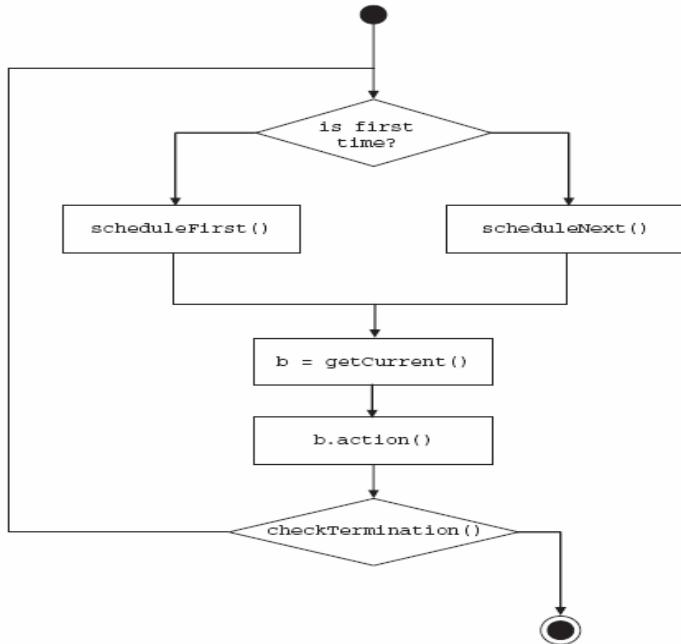
Phần này sẽ trình bày chi tiết hơn về các hành vi gộp mà JADE cung cấp. Ngoài chính sách lập lịch hành vi con, lớp con của *CompositeBehaviour* cũng phải định nghĩa một cơ chế kết thúc và một cơ chế khóa/khởi động lại. Cơ chế kết thúc sẽ xác định thời điểm hành vi gộp sẽ kết thúc. Ví dụ, một SequentialBehaviour lập lịch cho các hành vi con của nó sau những hành vi khác, kết thúc khi hành vi con cuối cùng hoàn tất. Phần sau chỉ ra làm sao block và restart các sự kiện (tức là gọi phương thức block() và restart()) trong các hành vi gộp được truyền tới các hành vi con của nó và ngược lại.

Một chính sách lập lịch con được cài đặt (như đã giới thiệu ở phần đầu của mục này) bằng cách định nghĩa lại theo các phương thức abstract của lớp CompositeBehaviour:

- *getCurrent()* - phương thức này nhằm return hành vi con hiện tại để chạy và được gọi mỗi lần phương thức *action()* của CompositeBehaviour được thực thi.
- *ScheduleFirst()* - phương thức này được gọi một lần ngay khi CompositeBehaviour start và được dùng để thiết lập hành vi con đầu tiên để thực thi.
- *ScheduleNext()* - phương thức này có cùng ý nghĩa với *sheduleFirst()*, nhưng được gọi mỗi lần thành công.

Tiêu chuẩn kết thúc được cài đặt bằng việc định nghĩa lại phương thức trừu tượng *checkTermination()* của lớp CompositeBehaviour. Phương thức này được gọi sau khi thực thi phương thức *action()* của hành vi con hiện tại.

Cần chú ý về hành vi gộp liên quan tới phương thức `getParent()` của lớp `Behaviour`. Phương thức này cho phép một hành vi con có con trỏ trả về hành vi gộp cha. Nếu một hành vi không là một phần của bất kì hệ thống hành vi gộp nào thì phương thức `getParent()` trả về null.



Hình 4.8: Luồng thực thi của phương thức `action()` của lớp `CompositeBehaviour`

4.3 HÀNH VI LUỒNG

Như đã giới thiệu ở mục 3.2.1, việc lập lịch hành vi được thực hiện theo một cách không ưu tiên. Đó là, phương thức `action()` của một hành vi không bao giờ bị ngắt để cho phép một hành vi khác nhảy vào. Chỉ khi phương thức `action()` của hành vi đang chạy trả về, việc điều khiển được truyền cho hành vi tiếp theo. Như đã thảo luận, cách tiếp cận này có một vài ưu điểm về hiệu năng và khả năng thay đổi. Tuy nhiên, khi một hành vi cần thực hiện một vài hoạt động blocking, thì nó block toàn bộ agent chứ không chỉ bản thân nó. Một giải pháp có thể là sử dụng các luồng Java thông thường. Tuy nhiên Jade cung cấp một giải pháp rõ ràng hơn bằng việc đặt luồng hành vi, tức là các hành vi được thực thi trong các luồng riêng biệt.

Bất cứ hành vi Jade (đơn giản hay gộp) có thể được thực thi như một hành vi luồng bằng lớp `jade.core.behaviours.ThreadedBehaviourFactory`. Lớp này cung cấp phương thức `wrap()` để bao bọc hành vi Jade thông thường vào một hành vi luồng wrapper. Hành vi luồng wrapper này bản thân nó là một hành vi. Việc thêm nó vào agent bằng phương thức `addBehaviour()` sẽ làm cho việc thực thi object Behaviour gốc trong một luồng chuyên môn. Nên chú ý rằng người phát triển chỉ đối xử với lớp `ThreadedBehaviourFactory`, trong khi lớp thực của hành vi luồng wrapper là private và không thể truy xuất. Đoạn code mẫu dưới đây chỉ ra cách thực thi một hành vi Jade trong một luồng java chuyên môn.

```

import jade.core.*;
import jade.core.behaviours.*;
public class ThreadedAgent extends Agent {
    private ThreadedBehaviourFactory tbf = new
  
```

```

ThreadedBehaviourFactory();
protected void setup() {
    // Create a normal JADE behaviour
    Behaviour b = new OneShotBehaviour(this) {
        public void action() {
            // Perform some blocking operation that can take a long time
        }
    };
    // Execute the behaviour in a dedicated Thread
    addBehaviour(tbf.wrap(b));
}
}

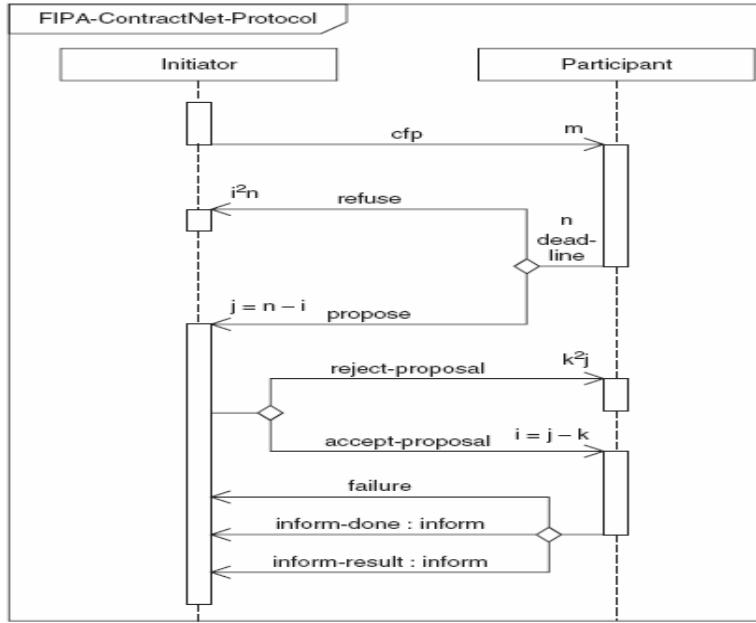
```

Hành vi luồng có thể được kết hợp với hành vi thường trong các hành vi gộp. Ví dụ, một hành vi SequentialBehaviour có thể có 2 hành vi con được thực thi như là các hành vi thông thường và một hành vi con thứ 3 được thực thi trong một luồng chuyên môn. Trong thực tế lớp ParallelBehaviour có thể được sử dụng để gán một nhóm hành vi vào một luồng chuyên môn. Có vài điểm quan trọng cần phải chú ý khi làm việc với hành vi luồng:

- Phương thức removeBehaviour() của lớp Agent không ảnh hưởng tới hành vi luồng. Một hành vi luồng bị xóa bằng việc lấy đối tượng Thread có nó bằng việc gọi phương thức getThread() của lớp ThreadedBehaviourFactory và gọi phương thức interrupt().
- Khi một agent chết, di chuyển hoặc tạm dừng, các hành vi luồng đang hoạt động của nó phần bị kill một cách rõ ràng bằng việc sử dụng kỹ thuật đã mô tả ở trên.
- Nếu một hành vi con của hành vi song song (parallel) được cấu hình với chính sách kết thúc WHEN_ANY là hành vi luồng, việc kết thúc các hành vi con khác không stop nó. Hành vi luồng con phải được kill một cách rõ ràng như mô tả ở trên.
- Khi một hành vi luồng truy xuất một vài tài nguyên agent, cái mà có thể được truy xuất bởi các hành vi luồng hoặc không luồng khác, việc quan tâm tính đúng đắn phải trả giá bằng việc đồng bộ.

4.4 CÁC GIAO THỨC TƯƠNG TÁC

Ở giai đoạn này người đọc khá quen thuộc với ngôn ngữ FIPA-ACL được sử dụng bởi agent JADE để giao tiếp. Ngôn ngữ này cung cấp một tập các biểu diễn chuẩn, mỗi một biểu diễn như vậy được định một cách rõ ràng. Một trong những ưu điểm chính của đặc điểm này là khả năng chỉ ra chuỗi các thông điệp được định nghĩa trước có thể được áp dụng trong một vài hoàn cảnh chia sẻ cùng một kiểu giao tiếp mà không quan tâm miền ứng dụng. Một chuỗi các thông điệp này được biết đến như là các giao thức tương tác.



Hình 4.9: Giao thức mạng hợp đồng (ContractNet)

4.4.1 Gói jade.proto

Tất cả các lớp cung cấp hỗ trợ việc cài đặt các giao thức chuẩn trong JADE nằm trong gói jade.proto. Khi việc tham gia một phiên trao đổi được điều khiển bởi giao thức tương tác một agent có thể đóng vai trò là initiator (bên khởi tạo) hoặc responder (bên đáp ứng). Kết quả là các lớp trong gói jade.proto được chia thành initiator và responder. Ví dụ, chúng ta có lớp ContractNetInitiator và ContractNetResponder, SubscriptionInitiator và SubscriptionResponder...

Tất cả các hàm khởi tạo của lớp initiator chứa một tham số ACLMessage thể hiện một thông điệp được sử dụng để khởi tạo giao thức. Cho ví dụ, lớp ContractNetInitiator có thông điệp CFP được gửi tới responder để khởi tạo tác vụ yêu cầu được đề xuất (call for propose). Tất cả các lớp initiator hỗ trợ cả tương tác one-to-one lẫn one-to-many phụ thuộc vào số lượng người nhận được chỉ ra trong thông điệp khởi tạo.

Các lớp Responder có sẵn hai phiên bản. Phiên bản vòng (cyclic) có một tham số MessageTemplate trong hàm khởi tạo, được dùng để chọn các thông điệp khởi tạo giao thức từ các Initiator. Hành vi của Responder thường được đưa vào phương thức setup() và duy trì hoạt động trong toàn bộ vòng đời của agent. Mỗi khi nhận được một thông điệp khởi tạo giao thức đúng với template, hành vi này sẽ xử lý nó, thực hiện phiên hội thoại và quay lại chờ thông điệp khởi tạo mới.

Phiên bản đơn phiên (single session) có một thông điệp khởi tạo trong hàm khởi tạo của nó, thực hiện phiên hội thoại được khởi tạo bởi thông điệp đó và sau đó kết thúc. Ở đây, hành vi của responder không có trách nhiệm nhận thông điệp khởi tạo. Do đó, cần có một hành vi ngoại để nhận chúng. Đoạn code sau minh họa hành vi SSSContractNetResponder (phiên bản đơn phiên):

```

MessageTemplate template = MessageTemplate.and(
    MessageTemplate.MatchProtocol("fipa-contract-net"),
    MessageTemplate.MatchPerformativ(ACLMessage.CFP) );
addBehaviour(new CyclicBehaviour(this) {
    public void action() {
        ACLMessage cfp = myAgent.receive(template);
        if (cfp != null) {
            myAgent.addBehaviour(new SSContractNetResponder(myAgent, cfp)
            {
                // Redefine callback methods to implement domain-dependent
                // logic
            });
        } else {
            block();
        }
    }
});
}
);

```

Bảng 4.1: Các giao thức tương tác được hỗ trợ bởi JADE

Giao thức	Lớp Initiator	Lớp Responder
FIPA-Request FIPA-Query	AchieveREInitiator	AchieveREResponder
FIPA-Propose	ProposeInitiator	ProposeResponder
Phiên bản được lặp lại của FIPA-Request FIPA-Query	IteratedAchieveREInitiator	SSIteratedAchieveREResponder
Contract-Net	ContractNetInitiator	ContractNetResponder SSContractNetResponder
FIPA-Subscribe	SubscriptionInitiator	SubscriptionResponder

4.4.2 Sử dụng các lớp giao thức

Như đã đề cập ở trên, các lớp giao thức cung cấp một số giao thức gọi lại. Những phương thức này có thể được lập trình viên sử dụng để định nghĩa lại bằng cách tùy biến chúng theo logic của miền ứng dụng. Chúng được khai báo để được bảo vệ và có một cài đặt mặc định. Theo cách này, lập trình viên có thể chọn (tùy thuộc vào yêu cầu cụ thể) phương thức nào sẽ cài đặt và phương thức nào sẽ bỏ qua. Với cả bên khởi tạo và bên đáp ứng, phần lớn các phương thức gọi lại đều được gọi theo việc nhận thông điệp và có dạng

```
protected handle<message-performative>(ACLMessage receivedMessage)
```

Ví dụ, trong ContractNetResponder, nếu thông điệp ACCEPT_PROPOSAL được nhận, thì phương thức handAcceptProposal (ACLMessage accept) được gọi. Khi việc nhận thông điệp kết

thúc một tương tác với bên gửi thông điệp (ví dụ khi thông điệp REFUSE được gửi làm thông điệp phản hồi cho thông điệp CFP trong Contract-Net protocol xác định rằng không còn thông điệp nào nữa cần được gửi trả lại bên đáp ứng), phương thức handleXXX() tương ứng sẽ trả về void. Nói cách khác, nếu thông phản hồi phải được gửi trả lại, chúng ta phân biệt 2 trường hợp. Với những bên đáp ứng mà luôn bị lôi cuốn vào các tương tác one-to-one, phương thức handleXXX() trả về một ACLMessage. Giá trị được trả về sẽ được sử dụng là thông điệp phản hồi. Ví dụ, phương thức handleCfp() của ContractNetResponder thường được định nghĩa như sau:

```
Protected ACLMessage handleCfp (ACLMensaje cfp) {
    ACLMessage reply = cfp.createReply();
    //Evaluate the call
    If (call OK) {
        //prepare a proposal
        reply.setPerformative (ACLMensaje.PROPOSE);
    }
    else {
        reply.setPerformative (ACLMensaje.REFUSE);
    }
    return reply;
}
```

Để bắt đầu việc thiết kế hỗ trợ những tương tác một-nhiều, phương thức handleXXX () nhận thêm một đối số kiểu Vector. Ví dụ, phương thức handlePropose () của ContractNetInitiator thường sẽ được định nghĩa lại như sau:

```
protected void handlePropose (ACLMensaje propose, Vector acceptances)
{
    ACLMessage reply = propose.createReply();
    // Evaluate the proposal
    if (proposal OK) {
        reply.setPerformative (ACLMensaje.ACCEPT_PROPOSAL);
    }
    else {
        reply.setPerformative (ACLMensaje.REJECT_PROPOSAL);
    }
    acceptances.add (reply);
}
```

Để thấy được sức mạnh của các lớp giao thức tương tác, ta có thể đơn giản hóa hành vi BookNegotiator được trình bày trong Phần 3.3.5 dựa trên lớp ContractNetInitiator:

```
public class BookNegotiator extends ContractNetInitiator {
    private String title;
    private int maxPrice;
    private PurchaseManager manager;
    public BookNegotiator (String t, int p, PurchaseManager m) {
        super (null, null);
        title = t;
        maxPrice = p;
        manager = m;
    }
}
```

```

protected Vector prepareCFPs(ACLMensaje cfp) {
    cfp = new ACLMensaje(ACLMensaje.CFP);
    cfp.setContent(title);
    for (int i = 0; i < sellerAgents.size(); ++i) {
        cfp.addReceiver((AID) sellerAgents.get(i));
    }
    Vector v = new Vector();
    v.add(cfp);
    return v;
}
protected void handleAllResponses(Vector responses
Vector acceptances) {
    ACLMensaje bestOffer = null;
    int bestPrice = -1;
    for (int i = 0; i < responses.size(); ++i) {
        ACLMensaje rsp = (ACLMensaje) responses.get(i);
        if (rsp.getPerformative() == ACLMensaje.PROPOSE) {
            int price = Integer.parseInt(rsp.getContent());
            if (bestOffer == null || price < bestPrice) {
                bestOffer = rsp;
                bestPrice = price;
            }
        }
    }
    if (bestOffer != null) {
        ACLMensaje accept = bestOffer.createReply();
        accept.setContent(title);
        acceptances.add(accept);
    }
}
protected void handleInform(ACLMensaje inform) {
    // Book successfully purchased
    int price = Integer.parseInt(inform.getContent());
    myGui.notifyUser("Book "+title+" successfully purchased.
Price = "+price);
    manager.stop();
}
} // End of inner class BookNegotiator

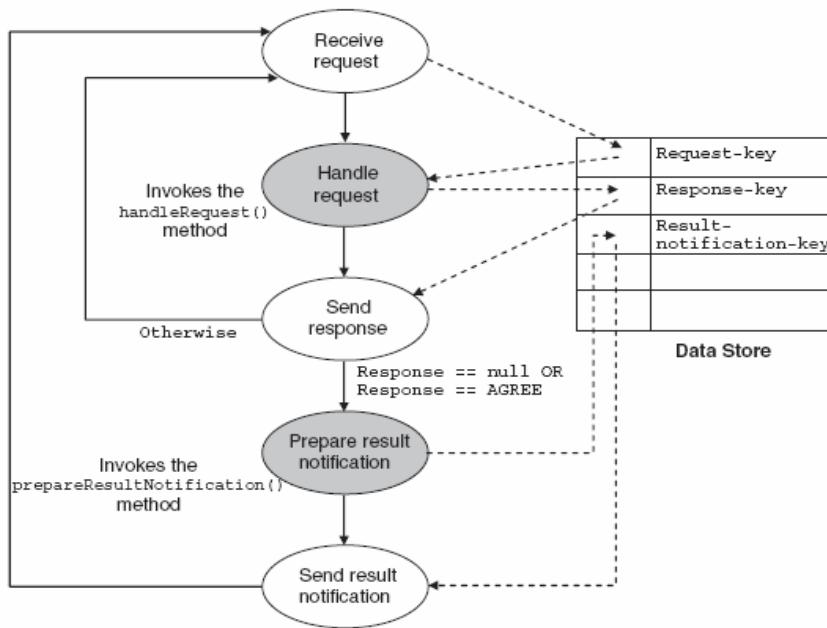
```

Phương thức `prepareCFPs()` được gọi ngay sau khi các hành vi `ContractNetInitiator` bắt đầu. Nó được dự định để điều chỉnh các thông điệp CFP được gửi đến các responder. Nó đặc biệt hữu dụng khi thông điệp CFP không biết đến vào thời gian xây dựng hoặc khi chúng ta cần gửi thông điệp tùy chỉnh tới mỗi responder. Tất cả các lớp khởi tạo giao thức có một phương pháp tương tự.

4.4.3 Lòng giao thức

Như đã trình bày trong các phần trước, cả lớp initiator và responder đều gọi các phương thức callback khi nhận được các thông điệp. Nếu phải gửi lại một thông điệp phản hồi thì phương thức callback có trách nhiệm tạo thông điệp đó. Tuy nhiên, có những trường hợp để tạo được thông điệp phản hồi, cần phải thực thi một hành vi. Rõ ràng là việc này ngăn cản chúng ta sử dụng lớp `ContractNetInitiator` vì nó không thể thực thi một hành vi trong một phương thức.

Để vượt qua giới hạn này, tất cả các lớp giao thức của JADE đều được cài đặt là các lớp con của lớp FSMBehaviour và mỗi phương thức callback được gọi trong một trạng thái của máy hữu hạn trạng thái. Hình 4.10 chỉ ra máy hữu hạn trạng thái của lớp AchieveREResponder. Nhìn chung, với mỗi phương thức callback mmm() có một phương thức registerMmm() cho phép ghi đè trạng thái gọi phươn gthwes mmm() bằng một hành vi cụ thể của ứng dụng. Tất cả các phương thức registerMmm() đều có một tham số là đối tượng Behaviour.



Hình 4.10: Máy hữu hạn trạng thái của lớp AchieveREResponder

Bên cạnh các trạng thái dùng để gọi các phương thức callback, còn có các trạng thái khác có trách nhiệm gửi, nhận thông điệp và thực hiện các kiểm tra liên quan đến luồng giao thức. Tuy nhiên, chúng được ẩn đi và người lập trình không cần quan tâm đến. Đường nét đứt trong hình 4.10 chỉ ra cách dữ liệu được chia sẻ giữa các trạng thái của giao thức sử dụng DataStore. Ví dụ, hành vi cài đặt trạng thái Handler-Request của lớp AchieveREResponder như sau:

```

private class HandleRequest extends OneShotBehaviour {
    public void action() {
        ACLMessage request = getDataStore().get(REQUEST_KEY);
        ACLMessage response = handleRequest(request);
        getDataStore().put(RESPONSE_KEY, response);
    }
}
  
```

Do đó, khi đăng ký một hành vi cụ thể của ứng dụng trong trạng thái của một lớp giao thức, hành vi đó có trách nhiệm lấy các thông điệp đã nhận ở trước và lưu trữ các phản hồi được tạo ra vào DataStore sử dụng các khóa.

4.5 KHỞI ĐỘNG JADE TỪ MỘT ỨNG DỤNG JAVA BÊN NGOÀI

Cho đến nay chúng ta luôn luôn giả định rằng Jade run-time được khởi động từ dòng lệnh. Tuy nhiên trong nhiều trường hợp, chúng ta cần khởi động một hoặc nhiều agent từ ứng dụng bên ngoài, do đó cần tạo JADE run-time để lưu trú chúng. Để hỗ trợ yêu cầu này, từ phiên bản JADE

2.3, một giao diện in-process đã được cài đặt cho phép JADE được sử dụng như một loại thư viện cho phép run-time được khởi chạy từ các chương trình bên ngoài.

JADE run-time được cài đặt bởi lớp jade.core.Runtime. Theo mẫu singleton, một thể hiện của lớp này tồn tại trong một máy ảo JVM và có thể lấy bằng cách gọi phương thức tĩnh instance(). Thể hiện Runtime cung cấp hai phương thức: createMainContainer() để tạo container chính và createAgentContainer() để tạo một container ngoại vi. Cả hai phương thức đều lấy một tham số là đối tượng Profile để lưu các lựa chọn cấu hình cần thiết để khởi động JADE run-time. Tất cả các lựa chọn có thể được đặc tả khi khởi động JADE từ dòng lệnh đều được sử dụng như là các hằng trong lớp Profile và có thể được thiết lập trong profile sử dụng phương thức setParameter(String key, String value).

Cả createMainContainer() và createAgentContainer() đều trả về đối tượng jade.wrapper.ContainerControl. Bộ điều khiển này sẽ bao bọc các chức năng mức cao của container và tạo các agent mới. Phương thức createNewAgent() của lớp này trả về đối tượng AgentController cũng được bao bọc bởi một số chức năng của agent. Cụ thể, lớp AgentController cung cấp các phương thức để ứng dụng bên ngoài có thể điều khiển vòng đời của agent được bao bọc nhưng vẫn đi việc tham chiếu tới đối tượng Agent để nó không thể thực hiện các lời gọi trực tiếp tới đó. Chú ý, phương thức createNewAgent() tạo một thể hiện của lớp Agent nhưng không khởi động nó, mà chỉ có thể gọi phương thức start() của đối tượng AgentController được trả về. Quay lại ví dụ bookTrading, ta muốn tích hợp buyer agent và một hệ thống lớn hơn có thể mua sách thông qua nhiều kênh khác nhau. Khi đó, môi trường book-trading của JADE trở thành một trong cách kênh của hệ thống lớn có thể sử dụng để mua sách. Sau đây là đoạn code của book-buyer agent mà hệ thống bên ngoài sử dụng để mua sách trong môi trường book-trading của JADE:

```
public AgentController startBuyerAgent(
    String host, // JADE Book Trading environment Main Container
    host
    String port, // JADE Book Trading environment Main Container
    port
    String name, // Book Buyer agent name
) {
    // Retrieve the singleton instance of the JADE Runtime
    Runtime rt = Runtime.instance();
    // Create a container to host the Book Buyer agent
    Profile p = new ProfileImpl();
    p.setParameter(Profile.MAIN_HOST, host);
    p.setParameter(Profile.MAIN_PORT, port);
    ContainerController cc = runtime.createAgentContainer(p);
    if (cc != null) {
        // Create the Book Buyer agent and start it
        try {
            AgentController ac = cc.createNewAgent(name,
                "bookTrading.buyer.BookBuyerAgent", null);
            ac.start();
            return ac;
        }
    }
}
```

```

        catch (Exception e) {
            e.printStackTrace();
        }
    }
    return null;
}

```

Trong trường hợp này một container ngoại vi được tạo ra để kết nối với một nền tảng Jade đang tồn tại (môi trường book-trading JADE). Do đó chúng ta cần phải xác định host và port của container chính của flatform. Cũng lưu ý việc sử dụng các lớp ProfileImpl. Điều này là cần thiết vì Profile là một lớp trừu tượng và do đó không trực tiếp khởi tạo. Cả 2 lớp Profile và ProfileImpl đều thuộc gói jade.core.

4.5.1 Giao tiếp giữa Object và Agent

Trong nhiều trường hợp, bên cạnh việc bắt đầu từ một hay nhiều agent, một ứng dụng bên ngoài cần phải tương tác với các agent để chỉ dẫn chúng thực hiện một số tác vụ. Trong kịch bản ở trên, ví dụ hệ thống bên ngoài phải thông báo cho agent book-buyer mỗi lần có một cuốn sách mới để mua. Tương tự, agent book-buyer phải thông báo cho hệ thống bên ngoài khi việc mua sách thành công. Tuy nhiên, như chúng ta đã thấy trong phần trước, các lớp AgentController không phơi bày các tham chiếu của một thể hiện Agent và do đó các ứng dụng bên ngoài không thể gọi trực tiếp bất kỳ một phương thức của agent. Sự tương tác giữa một ứng dụng bên ngoài và một agent bắt đầu bằng giao diện tiến trình được tạo ra sẵn bởi một cơ chế giao tiếp object-to-agent (O2A). Đây chính là một hàng đợi FIFO đồng bộ nơi ứng dụng bên ngoài có thể đặt đối tượng Java mà sau này có thể được lấy bởi một agent, mỗi agent tự có riêng một hàng đợi O2A. Các đối tượng AgentController bọc lấy một agent cung cấp phương thức putO2AObject() có thể được sử dụng bởi các ứng dụng bên ngoài để chèn đối tượng trong hàng đợi O2A của một agent. Tương tự như các lớp Agent cung cấp phương thức getO2AObject() có thể được sử dụng bởi các agent để đọc các đối tượng thông qua các ứng dụng bên ngoài. Các cơ chế truyền thông O2A mặc định bị vô hiệu và do đó một agent mong muốn tương tác với các ứng dụng bên ngoài phải khởi động nó một cách rõ ràng bằng phương thức setEnabledO2ACommunication(). Tương tự với mô hình hàng đợi thông điệp chuẩn, chèn một đối tượng vào hàng đợi O2A của một agent có ảnh hưởng tới việc khởi động lại tất cả các hành vi của agent để tạo ra cho họ một cơ hội để đọc và thực thi đối tượng được chèn vào.

Trở lại ví dụ ở trên, mỗi khi hệ thống buying bên ngoài được yêu cầu để mua một cuốn sách, nó phải thông báo cho agent book-buyer cục bộ. Các đoạn code sau đây được sử dụng bởi các bên ngoài hệ thống sẽ thực hiện điều này:

```

BookInfo info = new BookInfo(title, maxPrice, deadline);
buyerAgentController.putO2AObject(info);

```

BookInfo là một bean ứng dụng được sử dụng để nhóm tiêu đề sách để mua, giá tối đa mà người sử dụng sẵn sàng trả và các hạn giao hàng. BuyerAgentController là đối tượng AgentController bao bọc agent book-buyer giống như khi thảo luận startBuyerAgent() trong phần trước. Các đoạn code sau được thêm vào các phương thức BookBuyerAgent() để xử lý

các thông báo quá trình từ hệ thống bên ngoài (các sửa đổi khác cũng cần phải được giới thiệu để xử lý một giao dịch mua hàng thành công hoặc hạn giao hàng).

```
// Enable O2A communication
setEnabledO2ACommunication(true, 0);
// Add the behaviour serving notifications from the external system
addBehaviour(new CyclicBehaviour(this) {
    public void action() {
        BookInfo info = (BookInfo) myAgent.getO2AObject();
        if (info != null) {
            purchase(info.getTitle(),
                    info.getMaxPrice(),
                    info.getDeadline());
        }
        else {
            block();
        }
    }
});
```

CHƯƠNG 5

KHẢ NĂNG DI ĐỘNG CỦA AGENT TRONG JADE

Khả năng di động của Agent là một cách tiếp cận xuất phát từ 2 ngành khác nhau là Trí tuệ nhân tạo – tạo ra khái niệm về agent và Các hệ thống phân tán – định nghĩa khái niệm về mã di động. Chương này trình bày một số khái niệm liên quan agent di động và nền tảng Jade trong thiết kế các agent di động.

5.1 THẾ NÀO LÀ TÍNH DI ĐỘNG CỦA AGENT

Theo các định nghĩa chuẩn, các agent di động có tất cả đặc tính của một agent thông thường (như tính tự chủ, phản ứng, hướng đích và tính xã hội), nhưng thêm vào đó chúng có khả năng di chuyển – chúng có thể di chuyển giữa các platform để thực hiện các nhiệm vụ được giao.

Từ quan điểm của hệ thống phân tán, một agent di động là một chương trình với một thực thể duy nhất, có thể di chuyển code, dữ liệu và trạng thái của nó giữa các máy đã được nối mạng. Để đạt được điều này, các agent di động có thể tạm dừng quá trình thực thi của chúng tại bất kỳ thời điểm nào và tiếp tục tại một vị trí khác. Chúng ta có thể đặt các agent di động trong mối quan hệ với các cách tiếp cận cổ điển khác:

- Client – server: phương pháp tiếp cận được sử dụng rộng rãi nhất, các dịch vụ được cung cấp bởi một máy chủ và được phục vụ cho một hoặc nhiều máy khách – thường là từ xa
- Thực thi từ xa: một thành phần gửi mã đến thành phần khác để thành phần đó thực thi từ xa do quyết định của chính nó, một yêu cầu từ thành phần từ xa hoặc thậm chí có thể là một phần của một giao ước đã tồn tại trước đó. Sau mỗi lần thực thi, thành phần thực hiện thường trả lại bất kỳ kết quả nào tới thành phần ban đầu (thành phần đã gửi mã tới)
- Các agent di động: một thành phần tự gửi chính bản thân nó (hoặc cả thành phần khác, nếu được phép) tới một máy chủ từ xa để thực thi. Thành phần này chuyển cả code, dữ liệu và có thể toàn bộ trạng thái của nó. Động lực có thể tương tự như trường hợp trên (thực thi từ xa), nhưng thông thường nhất là bởi quyết định của chính thành phần (tức là agent di động), nó muốn di chuyển tới một vị trí thay thế.

Một agent di động, như mô tả trong Hình 5.1, bao gồm 3 thành phần: *code*, *trạng thái* và *dữ liệu*. Code là phần của agent sẽ được thực thi khi nó di chuyển tới một platform khác. Trong trường hợp đơn giản nhất, chỉ có một code đơn nhất. Trạng thái là môi trường thực thi dữ liệu của agent, bao gồm bộ đếm chương trình và ngăn xếp tác vụ. Thành phần này chỉ được tìm thấy ở các agent “di chuyển mạnh” (xem phần 5.1.2). Dữ liệu bao gồm các biến mà các agent sử dụng, như tri thức, định danh tập tin... Trong “di chuyển yếu” (xem phần 5.1.2) thành phần này thực sự cần

thiết vì code agent được cấu tạo như một máy trạng thái và để duy trì thông tin trạng thái đòi hỏi phải có các biến này.



Hình 5.1: Cấu trúc cơ bản của agent di động

5.1.1 Một số ưu điểm và nhược điểm của agent di động

Đã có nhiều tranh cãi về những ưu nhược điểm của các agent di động, thông thường chúng được so sánh với những họ agent “không di động”. Một vài ưu điểm điển hình là :

- *Tiến trình độc lập và không đồng bộ*: mỗi khi agent di chuyển tới một platform mới, các agent không phải liên hệ với chủ đề thực hiện nhiệm vụ của mình. Chúng có thể chỉ cần gửi các kết quả trả về. Điều này đặc biệt hữu ích đối với các thiết bị di động với các nguồn lực hạn chế khi một agent có thể được di chuyển tới một máy khác để thực hiện các nhiệm vụ phức tạp và gửi trả lại kết quả định kỳ.
- *Sự chịu lỗi*: các agent di động có thể giải quyết và trợ giúp trạng thái lỗi bằng cách di chuyển tới một platform thay thế khi các vấn đề được phát hiện. Nếu đích đến hỏng, có thể chọn máy chủ tạm thời. Điều này làm chúng khá thích hợp cho những môi trường thù địch, không thân thiện.
- *Các ứng dụng dữ liệu lớn*: các agent di động rất thích hợp cho các ứng dụng cần phải xử lý số lượng lớn các dữ liệu từ xa. Các agent di động có thể di chuyển tới nơi có chứa dữ liệu, chứ không phải ngược lại (chuyển dữ liệu tới các agent để xử lý). Trong nhiều trường hợp, lựa chọn này hiệu quả hơn rất nhiều.

Tuy nhiên, các agent di động cũng có một số nhược điểm. Như được mô tả trong Mir (2004), những nhược điểm rõ nhất trong số đó là:

- *Khả năng mở rộng và hiệu suất*: mặc dù các agent di động giảm tải mạng, nhưng chúng cũng có xu hướng làm tăng tiến trình. Đó là bởi chúng thường được lập trình với các ngôn ngữ trình diễn và cũng phải tuân theo những tiêu chuẩn về khả năng tương tác nghiêm ngặt, có thể chịu được việc xử lý dữ liệu bên trên.
- *Tính linh động và tiêu chuẩn hóa*: các agent không thể tương tác nếu chúng không theo cùng tiêu chuẩn chung. Sự tuân theo các tiêu chuẩn này, như OMG MASIF hay FIPA, là cần thiết, đặc biệt cho di động giữa các platform.
- *Bảo mật*: việc sử dụng các agent di động có thể mang tới những vấn đề về an ninh. Bất kỳ mã di động nào cũng đem lại một mối đe dọa tiềm năng và nên được xác nhận cẩn thận trước khi gọi tới.

5.1.2 Di chuyển mạnh và di chuyển yếu

Trong các hệ thống agent di động có thể chia thành 2 loại di chuyển cơ bản: di chuyển mạnh và di chuyển yếu.

Di chuyển mạnh thì thường phức tạp hơn. Đó là trường hợp sự thực thi của một agent ổn định, sự di chuyển diễn ra, sau đó sự thực thi được khởi động lại ngay từ chỉ dẫn tiếp theo. Kỹ thuật này đòi hỏi phải bảo vệ và lưu lại trạng thái của agent trong suốt quá trình di chuyển. Việc cài đặt kỹ thuật này có thể phức tạp bởi nó đòi hỏi truy cập các tham số nội bộ của agent - thường chỉ dành cho hệ điều hành; và rất phụ thuộc cấu trúc.

Mặt khác, di chuyển yếu không gửi trạng thái của agent, do đó đơn giản hơn nhiều. Sự thực thi của agent luôn khởi động lại từ đầu mã. Loại di chuyển này đòi hỏi agent được cài đặt như một máy trạng thái hữu hạn để trạng thái được duy trì.

5.1.3 Kế hoạch di chuyển

Một hành trình di chuyển xác định các địa điểm mà một agent di động phải tới để hoàn thành tập các nhiệm vụ. Hai loại hành trình cơ bản :

- Các hành trình tĩnh: được xác định tại thời điểm khởi tạo agent và không có khả năng thay đổi trong suốt quá trình thực thi của agent.
- Các hành trình động: được xác định trong cả quá trình thực thi của agent, theo các nhu cầu và mong muốn.

Ngoài ra có phương pháp lai là sự kết hợp của cả 2 loại trên.

5.2 DI CHUYỂN TRONG CÙNG PLATFORM

JADE cung cấp một dịch vụ nền tảng gọi là dịch vụ agent di động, thực hiện di động nội bộ platform. Dịch vụ này tạo cho các agent phần mềm khả năng di chuyển giữa các container trong cùng platform. Tuy nhiên, cơ chế này không cho phép các agent di chuyển đến các container thuộc các platform khác.

5.2.1 Các phương thức truy cập tới khả năng di động của Agent

Trong JADE, agent di động được điều khiển đơn giản thông qua phương thức *doMove()* trong lớp Agent :

```
void doMove( Location destination )
```

Tham số đích phải là một đối tượng của một lớp thực thi giao diện Location. Trong platform của JADE có 2 lớp thực thi giao diện này, chúng đều chứa trong gói *jade.core*. Đầu tiên là *ContainetID*, được dùng để đặc tả đích đến của agent sẽ là một container của platform. Container đó hiện đang chạy. Thứ 2 là *PlatformID*, được sử dụng để chỉ ra rằng đích đến của agent là container chính của một platform khác. Khi một platform từ xa được chỉ định là đích đến của một agent di động thì các cơ chế di động giữa các platform (mô tả trong phần 5.3) được thực hiện .

Mỗi khi được gọi tới, phương thức này khởi tạo tiến trình di chuyển agent đến container đích được chỉ định. Đa số code để đạt được điều này nằm trong gói *jade.core.mobility*. Lời gọi phương thức *doMove()* được chuyển tiếp tới “Dịch vụ agent di động” thông qua phương thức

move() của *helper* của nó. Hành động đầu tiên *helper* thực hiện là thay đổi trạng thái của agent từ ACTIVE thành TRANSIT buộc agent chấm dứt các hoạt động hiện thời của nó và bị trì hoãn trong khi platform định vị lại vị trí của nó. Người dùng có thể chỉ định các hoạt động được kích hoạt trước khi tiến trình di động được bắt đầu, ví dụ lưu trạng thái agent. Các hoạt động đó được xác định theo phương thức của lớp Agent :

```
void beforeMove()
```

Một phương thức ngược lại là *afterMove()* cũng được chỉ định để kích hoạt các hoạt động được thực thi ngay sau khi agent di chuyển, trước khi nó phục hồi lại trạng thái ACTIVE tại vị trí đích.

5.2.2 Agent serialization

Sau lời gọi phương thức *beforeMove()*, *helper* gọi một câu lệnh đọc (sẽ được trình bày trong Chương 6) yêu cầu agent được chuyển tới đích chứa trong câu lệnh. Nếu câu lệnh chứa một đối tượng đích là một thể hiện của lớp *PlatformID*, “Dịch vụ di động liên platform” (xem phần 5.3) sẽ bắt đầu di chuyển agent tới platform ở xa được chỉ định. Khi đối tượng đích là một thể hiện của lớp *ContainerID*, “Dịch vụ agent di động” sẽ bắt đầu di chuyển agent tới container đích trong platform hiện thời. *ContainerID* của container bất kỳ có thể được yêu cầu thay thế từ platform agent AMS, hơn là xây dựng nó từ lớp *ContainerID*. Sự di chuyển của agent phải bao gồm việc truyền tải ít nhất là code và dữ liệu của nó; và cũng có thể truyền trạng thái. Trong JADE, các agent mô hình trạng thái thực thi của chúng như những dữ liệu nội bộ agent, vì vậy có thể hiểu rằng chỉ cần truyền code và dữ liệu. Dữ liệu agent được chứa trong đối tượng Java đại diện cho agent, do đó, việc truyền đối tượng này cùng với code của nó là đủ để khôi phục lại agent tại đích đến.

Java serialization thường truyền một thể hiện agent qua một kết nối mạng bằng cách ghi lại các giá trị thành phần nội bộ của đối tượng agent trong một luồng byte. Người dùng phải chỉ định các thành phần dữ liệu được truyền cùng với agent bằng cách sử dụng bộ đệm tạm thời. Ví dụ, nếu một agent có một thành phần *FileInputStream* (không phải *Serializable*) và không được khai báo là tạm thời, tiến trình di động sẽ lỗi và đưa ra một ngoại lệ *NotSerializableException*.

5.2.3 Lớp tải (classloader) của agent di động

Khi một thể hiện của một agent đã được xuất bản, nó được chuyển đến các container đích bằng bằng của một lệnh ngang (xem chương 6). Sau đó thực hiện deserialized để phục hồi các đối tượng agent ban đầu. Tuy nhiên, vật chứa Jade thường được đặt tại máy ảo Java khác nhau trên máy khác nhau.

Nếu một agent di chuyển từ một máy chủ tới các máy khác, những tập tin lớp ban đầu gốc cũng phải được làm sẵn có như là quá trình deserialization của đối tượng agent ở một container đích yêu cầu lớp cấu trúc ban đầu. Để quản lý vấn đề này, các AMS (Agent Mobility Service) sử dụng lớp được xây dựng bên trong có khả năng nạp của các lớp yêu cầu từ một container từ xa thông qua các lệnh dịch vụ ngang (horizontal service commands). Bất kỳ container tiếp nhận yêu cầu như vậy đặt tên lớp yêu cầu và gửi nó vào container yêu cầu.

Nếu agent được deserialized thành công, một luồng được tạo ra cho đối tượng tái sinh, và quá trình bật được thực thi để khởi động lại các agent. Tất cả các thông điệp được gửi trong quá trình di cư sau đó được gỡ bỏ từ một bộ đệm tạm thời tại các container gốc và chuyển hướng về phía vị trí agent mới.

Cuối cùng, tại container nguồn, agent chuyển vào vào tình trạng GONE, chỉ ra rằng một di chuyển đã thành công và ở đó các bản sao cục bộ bị chấm dứt. Phương thức `afterMove()` được gọi và agent được loại bỏ ra từ các container.

5.2.4 Nhân bản Agent

Cho đến nay chương này đã được mô tả nội nền tảng di động về các sự kiện xảy ra trong Jade khi một agent di chuyển. Tuy nhiên, Jade cũng cung cấp một cơ chế nhân bản là chụp ảnh bản sao của agent tồn tại bằng cách sử dụng phương thức:

```
public void doClone(Location destination, String newName)
```

Tham số `destination` được sử dụng để chỉ ra các container mà ở đó clone của agent hiện thời sẽ được tạo ra. Tham số `newName` là tên được sử dụng để tạo ra clone AID. Quá trình nhân bản chính nó là giống với di động, ngoại trừ các agent khởi tạo thì không bị chấm dứt. Do đó kết quả là tương đương, thực hiện nhân bản tương tự ở mỗi cách ngoại trừ định danh của chúng.

5.2.5 Tuyên bố khả năng di động gián tiếp

Quá trình di chuyển một agent có thể thực sự được xác nhận một cách trực tiếp bởi các chính các agent hoặc gián tiếp bởi một nền tảng AMS. Nó được khởi tạo khi một agent, hoặc người dùng thông qua giao diện RMA, gửi một thông điệp ACL đến nền tảng của AMS yêu cầu cái mà một agent cụ thể được di cư. Nếu một AMS nhận như một yêu cầu, và quyền hạn của người yêu cầu được xác minh, các AMS tự động gọi phương thức `doMove()` hoặc `doClone()` của các agent cụ thể. Vì lý do này, một số chú ý cần được thực hiện khi tạo phương thức `afterMove()` và `beforeMove()` của một agent mà dễ tránh khỏi khống định gián tiếp việc cư trú. Bởi vì lời gọi của `doMove()` không nhất thiết phải thuộc vào bản thân agent, rất thận trọng để đảm bảo chặn các hành động đó, chẳng hạn như giải phóng tài nguyên và tiết kiệm thông tin, được đưa ra nhằm ngăn chặn vấn đề xảy ra nếu một lời gọi di cư gián tiếp được khởi tạo.

5.3 DỊCH VỤ DI ĐỘNG LIÊN PLATFORM

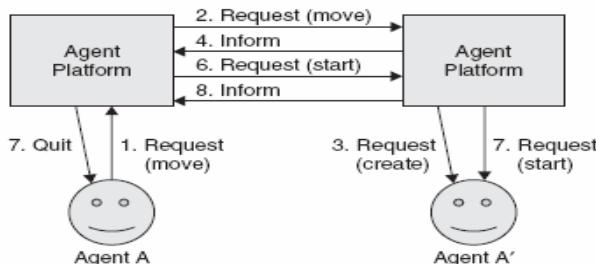
Các dịch vụ di động liên platform (IPMS) là một tiện ích của Jade đã được tạo ra để cung cấp khả năng di động giữa các platform (platform to platform mobility) cho các agent trong Jade, một tính năng mà không có sẵn gắn liền với AMS (Agent Mobile Service). Các IPMS được thiết kế một cách đặc biệt để có thể trong suốt với các lập trình viên agent bởi việc chắt chẽ việc di cư liên nền tảng đơn giản như di cư nội nền tảng. Càng nhiều tuân thủ nhất có thể được duy trì với phản đối (vì thiếu xác nhận cài đặt ban đầu). FIPA Hỗ trợ quản lý Agent cho đặc tả di động (đặc tả phản đối hiện có tại <http://www.fipa.org/specs/fipa00087/index.html>). Phiên bản hiện tại là đơn giản, nhưng thiết kế để mở rộng với các tính năng bổ sung như an ninh và hỗ trợ cho kháng lỗi di cư.

5.3.1 Quá trình di cư

Cơ chế cốt lõi của IPMS là sự di chuyển của các agent giữa các nền tảng sử dụng FIPA-ACL thông điệp như là phương tiện trung chuyển. Các thông điệp này được gửi đi giữa các AMS của các nền tảng thiết bị đầu cuối. Như đã đề cập, một vài thay đổi đã từng được thực hiện cho những đặc tả kỹ thuật FIPA gốc; các ontology bây giờ định nghĩa hai hành động, di chuyển (move) và bật nguồn (power up). Hành động đầu tiên đại diện cho sự di chuyển của code và thể hiện agent; hành động thứ hai đại diện cho sự kích hoạt của agent một lần di cư là hoàn tất. Ngoài ra, một số khái niệm được định nghĩa như mobile-agent-description chứa tất cả các thông tin agent, bao gồm code, dữ liệu của nó và mobile-agent-profile. Khái niệm này cuối cùng xác định đặc tính agent cơ bản để giúp chắc chắn rằng khả năng tương thích với các nền tảng nhận, như tên gọi của hệ thống agent gốc, ngôn ngữ mà nó đã được viết, và giao thức di động được sử dụng.

Giả sử rằng các thông điệp có chứa thông tin này, phù hợp với các ontology di động, sẽ là đủ cho một nền tảng từ xa để quyết định xem nó có thể thực thi một agent đến. Cả hai hành động, move và power up, được thực hiện bằng cách sử dụng các tiêu chuẩn giao thức tương tác theo yêu cầu FIPA. Trong từng trường hợp một thông điệp yêu cầu được gửi đến các nền tảng đích với một thông điệp Thông báo hoặc Thất bại mong đợi như một phản hồi. Như minh họa trong hình 5.2, giao thức yêu cầu di chuyển của FIPA, agent khởi tạo (bên trái) sẽ gửi một yêu cầu để chuyển thông điệp tới nền tảng của nó cái mà sẽ lần lượt gửi một yêu cầu để chuyển các thông điệp chứa mã agent và các dữ liệu vào nền tảng đích cụ thể (bên phải). Nếu nền tảng đích chính chọn thành công một agent từ thông điệp yêu cầu, một thông điệp Thông báo được trả lại cho nền tảng nguồn. Khi nhận, thông điệp Thông báo này gây lên cho nền tảng nguồn việc chấm dứt các agent yêu cầu và gửi Yêu cầu power-up đến nền tảng đích, nơi sẽ bắt đầu các agent. Nếu một cái gì đó không thành công, các trạng thái của agent trong nền tảng ban đầu được khôi phục và bất kỳ những thể hiện còn lại của các agent trong nền tảng đích là được.

Ưu điểm của việc sử dụng thông điệp ACL để vận chuyển các agent từ một nền tảng này tới nền tảng khác khác là không cần thêm vào kênh giao tiếp liên nền tảng. Điều bất lợi ở đây là hiệu suất không phải là đặc biệt cao do bản chất là quy trình mã hóa và giải mã tới ACL thông điệp. Tiêu chuẩn của các MTP thiết kế để cải thiện hiệu suất thông điệp ACL, có lẽ bằng cách sử dụng mã hóa nội dung nhẹ như là nó biểu hiện ở (FIPA23), có thể nâng cao hiệu suất đáng kể trong khi giữ lại khả năng tương thích.

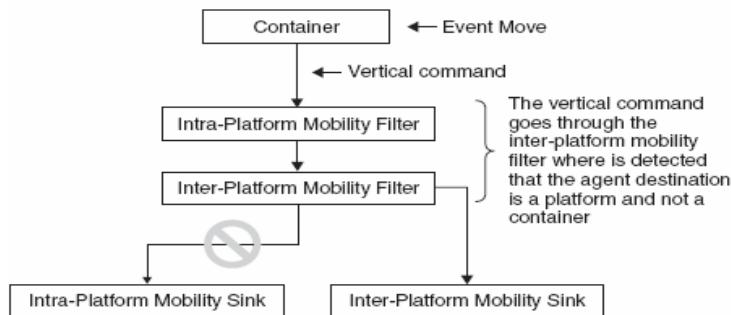


Hình 5.2: Giao thức yêu cầu di chuyển của FIPA

5.3.2 Tích hợp các dịch vụ di động

IPMS cần đến Dịch vụ di động nội platform xây dựng trong Jade. Hai lý do cho việc này là:

- (1) Phương pháp để di cư giữa các nền tảng và phương pháp để di cư giữa các container trên cùng một nền tảng cơ bản là như nhau theo quan điểm của người sử dụng. Phương thức move() tạo tham số Location mới trong liên nền tảng hợp được gọi là PlatformID (một biến của containerID) là tượng trưng cho nền tảng mục tiêu mà agent muốn di chuyển. Như các IPMS được thực hiện bằng cách sử dụng kiến trúc dịch vụ Jade (xem chương 6), nó cần chỉ chờ đợi để nhận được lệnh đọc từ dịch vụ nội platform vận động với một Location của kiểu PlatformID. Nó sau đó chỉ phải cần hủy bỏ lệnh và bắt đầu công việc của mình. Sơ đồ này được mô tả trong hình 5.3
- (2) Bởi vì trong di động nội platform, mã agent thực thi có thể không được cư trú trong cùng một container mà agent muốn di chuyển; như vậy, một cơ chế thu hồi mã là cần thiết. Vì các dịch vụ di động nội platform không di chuyển mã agent, IPMS chứa một cơ chế bổ sung phân tầng trên nền tảng di động để bảo đảm việc di chuyển mã cũng diễn ra khi một agent di chuyển giữa các container....



Hình 5.3: Luồng thực thi lệnh đọc Move

5.3.3 Nhóm code của các Agent với nhau

Để di chuyển một agent từ một địa điểm này tới một địa điểm khác, tất cả mã của nó phải được nhóm lại vào thông điệp ACL. Sự bố trí mã này không nhất thiết là một nhiệm vụ tầm thường, như trong Jade, Classpath dùng để pha trộn các tham chiếu vào mã và thư viện của platform, và mã của agent. Điều này gây khó khăn khi quyết định một thư viện hoặc một lớp trong Classpath có thể được sử dụng bởi các agent trong tương lai.

Hai giải pháp khác nhau được cung cấp cho vấn đề này. Phương pháp lập trình đơn giản (mặc dù không đơn giản đối với người sử dụng) là dành cho các lập trình viên tự đóng gói các agent bên trong một file Jar với tất cả các mã và thư viện mà nó sẽ yêu cầu. Tập tin jar này sau đó có thể được chuyển đi cùng với các agent.

Các dịch vụ di động nội platform cung cấp một cơ chế khởi tạo trong hệ thống di chuyển cái mà tự động tạo một tập tin JAR cho một agent. Tập JAR này có chứa mã tất cả các agent và các lập trình viên sẽ không bao giờ nhận thấy sự tồn tại của nó. Cơ chế này được xây dựng bởi việc kiểm tra để quy các lớp chính của agent để tìm kiếm cho tất cả các lớp phụ thuộc và xây dựng một JAR từ tập cuối cùng thu được. Trong phiên bản hiện tại, có thể là trong một số trường

hợp đặc biệt không phải tất cả các lớp agent được phát hiện đúng. Ví dụ, Điều này có thể xảy ra tại các agent ở đâu đó ánh xạ được sử dụng để tải các lớp (ví dụ: Class.forName ("ClassName"). newInstance ()). Nếu vấn đề xảy ra với các agent di trú nó rất có khả năng rằng một số lớp không được bao gồm trong JAR của agent.

Hiện nay, để tránh những vấn đề này, các tập tin JAR phải được tạo ra bằng tay. Các JAR nên được đặt tên sao cho nó tương ứng với tên lớp chính của agent. Ví dụ, với agent với main class là org.example.MyAgent phải được đóng gói trong một tập tin tên org_example_MyAgent.jar JAR. Các JAR sau đó phải được đặt trong một thư mục được chỉ định trên dòng lệnh khi chạy Jade.

Phương pháp tiếp cận thứ hai, lập trình phức tạp hơn nhưng minh bạch để người sử dụng, bao gồm của định danh tự động tất cả các lớp cần thiết và xây dựng một tập tin JAR chứa chúng. Lựa chọn lớp được điều khiển bởi các dịch vụ mà có thể ngũ ý rằng không phải lựa chọn tối ưu có thể được thực hiện.

5.3.4 Ontology di động của JADE

Jade-mobility-ontology có tất cả các khái niệm và hành động cần thiết để hỗ trợ agent di động. Jade cung cấp lớp jade.domain.mobility. MobilityOntology có chức năng như một singleton cho phép truy cập đến một trường hợp duy nhất được chia sẻ của ontology di động của JADE thông qua các phương thức getInstance (). Các ontology, giúp kéo dài JADEManagementOntology, chứa năm khái niệm và hai hoạt động, mỗi cái liên kết với một lớp của gói jade.domain.mobility. Các lược đồ khái niệm này được mô tả trong các bảng từ 5.1 đến 5.5. Các hành động đó là:

- move-agent. Đây là hành động của một agent di chuyển từ một địa điểm tới một địa điểm khác. Nó đại diện bởi các lớp MoveAction. Hành động này có slot đơn không tên của loại mobileagent-description. Đôi số là bắt buộc.
- clone-agent. Đây là hành động tạo ra một bản sao của một agent, có thể chạy trên một vị trí khác. Nó được đại diện bởi lớp CloneAction. Hành động có hai khe không tên: đầu tiên của loại mobile-agent-description và thứ hai của kiểu String. Cả hai đối số là bắt buộc.

Lưu ý rằng ontology này hiện chưa có bản sao trong các đặc tả FIPA.

5.4. SỬ DỤNG CÁC DỊCH VỤ DI CHUYỂN CỦA JADE

5.4.1. Dịch vụ di chuyển nội platform

Dịch vụ di chuyển nội platform là một dịch vụ được xây dựng trong Jade và do đó không yêu cầu các bước cài đặt đặc biệt để chuẩn bị sử dụng nó. Dịch vụ này chạy một cách mặc định khi Jade được bắt đầu chạy nhưng nếu một số dịch vụ khác đang chạy thì nó phải được thêm vào trong danh sách một cách rõ ràng. Đó là chạy với lệnh sau:

```
java jade.Boot -services jade.core.mobility.AgentMobilityService
```

Bảng 5.1. *Mobile-agent-description*: miêu tả một mobile agent. Nó được biểu diễn bởi lớp *MobileAgentDescription*

Tên slot	Kiểu slot	Bắt buộc/tùy chọn
Name	AID	Bắt buộc
Destination	Location	Bắt buộc
Agent-profile	mobile-agent-profile	Tùy chọn
Agent-version	String	Tùy chọn
Signature	String	Tùy chọn

Bảng 5.2 *Mobile-agent-profile*: miêu tả môi trường tính toán cần cho mobile agent. Nó được biểu diễn bởi lớp *MobileAgentProfile*

Tên slot	Kiểu slot	Bắt buộc/tùy chọn
System	mobile-agent-system	Tùy chọn
language	mobile-agent-language	Tùy chọn
Os	Mobile-agent-os	Bắt buộc

Bảng 5.3 *Mobile-agent-system*: Miêu tả thời gian chạy hệ thống được sử dụng bởi mobile agent. Nó được biểu diễn bởi lớp *MobileAgentSystem*

Tên slot	Kiểu slot	Bắt buộc/tùy chọn
System	String	Bắt buộc
major-version	Integer	Bắt buộc
minor-version	Integer	Tùy chọn
dependencies	String	Tùy chọn

Bảng 5.4 *Mobile-agent-language*: Miêu tả ngôn ngữ lập trình được sử dụng bởi mobile agent. Nó được biểu diễn bởi lớp *MobileAgentLanguage*

Tên slot	Kiểu slot	Bắt buộc/tùy chọn
System	String	Bắt buộc
major-version	Integer	Bắt buộc
minor-version	Integer	Tùy chọn
dependencies	String	Tùy chọn

Bảng 5.5 Mobile-agent-os: Miêu tả hệ điều hành cần bởi mobile agent. Nó được biểu diễn bởi lớp *MobileAgentOS*

Tên slot	Kiểu slot	Bắt buộc/tùy chọn
System	String	Bắt buộc
major-version	Integer	Bắt buộc
minor-version	Integer	Tùy chọn
dependencies	String	Tùy chọn

5.4.2 Dịch vụ di chuyển liên platform

Các IPMS không được xây dựng trong platform và phải được cài đặt như một add-on. Để cài đặt các add-on, các tập tin đóng gói phân phối phải được Unzipped bên trong thư mục Jade nơi lệnh *lib ant* được sử dụng để tạo một tập tin JAR chứa tất cả các file của lớp được biên dịch. Để sử dụng dịch vụ, nó phải được quy định rõ ràng trên dòng lệnh, không quên bao gồm cả dịch vụ di chuyển nội platform bên trên

```
java jade.Boot -services
jade.core.mobility.AgentMobilityService;jade.core.migration.
InterPlatformMobilityService
```

5.4.3 Xem xét vấn đề bảo mật của IPMS

Điều rất quan trọng để nhấn mạnh rằng những tác động bảo mật của việc sử dụng dịch vụ di chuyển liên platform là rất quan trọng, thường nghiêm trọng hơn di chuyển nội platform. Việc kích hoạt các IPMS ngũ ý rằng các máy đang chứa một Jade platform có thể thực thi mã được chuyển tới từ bất kỳ hệ thống tiềm năng nào có được thông qua một kết nối mạng. Vào thời điểm hiện tại, IPMS không có cơ chế kiểm soát truy cập để quyết định agent đang đến nào sẽ được thực thi và agent nào không được thực thi. Hiện nay, kiểm soát truy cập chỉ có thể được thực hiện bằng các phương tiện của một MTP an toàn (ví dụ HTTPS MTP cung cấp một số tính năng xác thực đơn giản mà có thể được sử dụng cho mục đích này), hoặc bằng cách sử dụng tường lửa để ngăn chặn các kết nối mạng từ các platform cụ thể. Cho đến khi một hệ thống điều khiển truy cập được hoàn thiện hơn được cài đặt, người ta khuyến nghị rằng các platform sử dụng IPMS có một số cách cơ bản sau để bảo mật:

- (1) Không sử dụng IPMS trong một môi trường mở hoặc trên các máy có chứa thông tin nhạy cảm.
- (2) Hãy thử hạn chế truy cập mạng cho platform bằng cách sử dụng tường lửa.
- (3) Hãy thử sử dụng các MTP với một số hình thức xác thực nội tại.

Dự kiến các phiên bản sắp tới của IPMS sẽ cung cấp toàn diện và cấu hình cơ chế bảo mật.

5.4.4 Lập trình agent di động

Mục tiêu chính của các dịch vụ di động của Jade là được sử dụng một cách đơn giản nhất có thể. Trong trường hợp này, cần thiết phải nạp chồng một số phương thức và gọi các phương thức

doMove() hoặc *doClone()* của lớp *Agent* để di chuyển hoặc sao chép các *agent* (lưu ý rằng *doClone()* chỉ có sẵn với dịch vụ di chuyển nội platform). Để chỉ ra đích của một agent di động hoặc một agent được nhân bản, Jade định nghĩa giao diện *jade.core.Location*. Hai cài đặt của giao diện này được cung cấp: *jade.core.ContainerID* cho di chuyển nội platform và *jade.core.PlatformID* cho di chuyển liên platform. Các *ContainerID* phải được khởi tạo với tên của container đích và địa chỉ vận chuyển của nó. Các *PlatformID* phải được khởi tạo với agent AID của điều khiển của agent AMS của platform bao gồm địa chỉ vận chuyển của nó.

Để di chuyển một agent khác container hay platform, một hành vi của agent phải bao gồm một lời gọi đến các phương thức *doMove()*. Phương thức này thay đổi agent của trạng thái TRANSIT chỉ ra rằng đó là vận chuyển. Các mã sau minh họa một di chuyển nội platform:

```
// Create some variables
String containerName = "Container-1";
ContainerID destination = new ContainerID();
// Initialize the destination object
destination.setName(containerName);
// Change of the agent state to move
myAgent.doMove(destination);
In the case of inter-platform migration, a similar example is as follows:
// Build the AID of the corresponding remote platform's AMS
AID remoteAMS = new AID("ams@remotePlatform:1099/JADE", AID.ISGUID);
// Specify the MTP by setting the transport address of the remote
// AMS
remoteAMS.addAddresses("http://remotePlatformaddr:7778/acc");
// Create the Location object
PlatformID destination = new PlatformID(remoteAMS);
// Change of the agent state to move
myAgent.doMove(destination);
```

Để sao chép một agent phương thức *doClone()* được sử dụng để thay đổi trạng thái agent để COPY. Để sử dụng nó, hai đối số phải được thông qua: đích đến container và tên agent mới. Đây là một ví dụ:

```
// Create some variables
String containerName = "Container-1";
String newAgentName = "myClone";
ContainerID destination = new ContainerID();
// Initialize the destination object
destination.setName(containerName);
// Change of the agent state to clone
myAgent.doClone(destination, newAgentName);
```

Như các dịch vụ di động cài đặt việc di chuyển yêu, một cấu trúc mã dựa trên một máy hữu hạn trạng thái được yêu cầu. Điều này là do truy cập chương trình của agent thực thi không được truyền, làm cho nó không thể tiếp tục agent thực hiện từ dòng tiếp theo chỉ dẫn của *doMove()* hoặc *doClone()*. Thay vào đó, agent thực hiện có thể tiếp tục chỉ từ đầu đoạn mã hành vi của agent. Sử dụng máy hữu hạn trạng thái để biểu diễn, một cấu trúc có thể được tạo ra trong đó đoạn mã agent thành phần với các biến được gán chỉ ra trạng thái mã agent thực thi.

Một tuyên bố chuyển đổi được sử dụng, ví dụ, một agent với cả người bán và người mua có thể có vai trò mã với cả 2 trạng thái, một cho việc bán và một cho việc mua. Các agent phải thiết lập một biến trạng thái trước khi rời một container hoặc một platform để cho biết có vai trò sẽ được khởi tạo cho thông qua trong các vị trí tiếp theo. Để tạo ra có thể sử dụng cách biểu diễn máy hữu hạn trạng thái. Ví dụ, trong một hành trình hai container trong đó container đích là một mảng các địa điểm, các mã hành vi được thực hiện trong mỗi container được phân cách như trong ví dụ sau đây:

```

addBehaviour(new CyclicBehaviour(this) {
    public void action() {
        switch(_state) {
            case 0:
                // Agent starts to migrate
                _state++;
                myAgent.doMove(_dests[0]);
                break;
            case 1:
                // Agent migrates to the second container
                _state++;
                myAgent.doMove(_dests[1]);
                break;
            case 2:
                // Agent dies
                myAgent.doDelete();
                break;
            default:
                myAgent.doDelete();
        }
    }
    private ContainerID[] _dests = ...;
    private int _state = 0;
});

```

Ví dụ này cho thấy cách mà mã agent phải được cấu trúc trong Jade để bảo đảm các trạng thái của nó bằng cách sử dụng một biến. Trong suốt các tiến trình di chuyển của mã agent của *serialization* và *deserialization*, một số tài nguyên sử dụng bởi agent cũng sẽ được chuyển giao, trong khi những agent khác sẽ bị ngắt kết nối trước khi sự di chuyển của agent và kết nối lại tại điểm đích. Jade cung cấp hai phương pháp kết hợp trong các lớp Agent cho việc quản lý nguồn tài nguyên mà chỉ cần được nạp chòng bởi các lập trình viên:

- *BeforeMove()*: được gọi tại vị trí nguồn khi hoạt động di chuyển hoàn thành công như thể hiện của agent được di chuyển là để được kích hoạt trên container đích và các thể hiện của agent ban đầu là để được chấm dứt. Phương thức này là vị trí chính xác để phát hành bất kỳ nguồn lực địa phương được sử dụng bởi các thể hiện của agent ban đầu (ví dụ như đóng mở các file và ảnh minh họa). Nếu các nguồn lực này đã được sử dụng trước khi biết nếu một di chuyển thành công, chúng vừa mới mở lại một lần nữa. Tuy nhiên, do hậu quả của việc này là bất kỳ thông tin được vận chuyển bởi agent đến vị trí mới phải được thiết lập trước khi các phương thức *doMove()* được gọi. Thiết lập một thuộc tính của

agent trong các phương thức *beforeMove()* sẽ không có tác động đến thể hiện của sự di chuyển.

- *AfterMove()*: được gọi tại vị trí đích đến ngay sau khi các agent đến và việc nhận dạng nó được đặt ra, nhưng trước khi việc lập lịch hành vi khởi động lại.

Đối với agent nhân bản, Jade cung cấp một cặp phương thức tương ứng là phương thức *beforeClone()* và *afterClone()*. Chúng được gọi trong cùng một dạng như phương thức *beforeMove()* và phương thức *afterMove()*. Tất cả bốn phương thức này đều là *protect* và là các thành viên của lớp Agent, được định nghĩa như là placeholders rỗng. Người dùng được định nghĩa các agent di động có thể ghi đè lên 4 phương thức khi cần thiết.

5.4.5 Truy cập vào AMS bằng khả năng di động của agent

JADE AMS có một vài mở rộng hỗ trợ tính di động của agent. Mỗi hành động liên quan đến tính di động, như được đề cập trong phần 5.3.4, có thể được yêu cầu bởi AMS bằng cách sử dụng thông điệp FIPA Request, cùng với thuộc tính ontology được thiết lập và jade-mobility-ontology và ngôn ngữ được thiết lập là FIPA-SL0.

Hành động move-agent sẽ xem mobile-agent-description như là một biến của nó. Hành động này di chuyển agent được định danh bởi thuộc tính tên và địa chỉ của mobile-agent-description đến vị trí được định danh bởi thuộc tính đích. Ví dụ, nếu một agent muốn di chuyển agent Peter đến vị trí tên là Front-End, nó phải gửi ACL Request thông điệp sau đến AMS của nó:

```
(REQUEST
  :sender (agent-identifier:nameRMA@Zadig:1099/JADE)
  :receiver (set(agent-identifier:nameams@Zadig:1099/JADE))
  :content
  (
    action(agent-identifier:nameams@Zadig:1099/JADE)
    (move-agent(mobile-agent-description
      :name (agent-identifier:nameJohnny@Zadig:1099/JADE)
      :destination(location
        :nameMain-Container
        :protocolJADE-IPMT
        :addressZadig:1099/JADE.Main-Container)
    )
  )
)
:reply-withReq976983289310
:languageFIPA-SL0
:ontologyjade-mobility-ontology
:protocolfipa-request
:conversation-idReq976983289310
)
```

Sử dụng hỗ trợ JADE ontology, một agent có thể dễ dàng thêm tính di động vào khả năng của nó mà không cần soạn thông điệp ACL bằng tay. Đầu tiên, agent phải tạo một đối tượng MoveAction mới và điền đối số của nó với một đối tượng MobileAgentDescription phù hợp, nó

lần lượt điền đầy đủ tên và địa chỉ của agent cần di chuyển (hoặc là chính nó, hoặc là agent khác) và vị trí của đích. Một lời gọi đến phương thức Agent.getContentManager().fillContent(.., ..) tiếp đó chuyển đổi đối tượng Java MoveAction sang dạng String và điền nó vào thuộc tính content của thông điệp ACL Request. Hành động nhân bản agent làm việc cũng tương tự như vậy, nhưng có thêm một đối số dạng String để chứa tên của agent mới là kết quả của quá trình nhân bản. AMS còn hỗ trợ bốn hành động liên quan đến tính lưu động được định nghĩa trong JADEManagement Ontology. Hành động where-is-agent chỉ có một đối số, chứa định danh của agent được xác định. Hành động này có kết quả là vị trí để agent đưa và thuộc tính content của thông điệp Inform ACL giúp hoàn thành giao thức. Ví dụ, thông điệp Request để hỏi vị trí agent Peter lưu trữ sẽ là:

```
(REQUEST
  :sender (agent-identifier:namedal@Zadig:1099/JADE)
  :receiver (set(agent-identifier:nameams@Zadig:1099/JADE))
  :content ((action
    (agent-identifier:nameams@Zadig:1099/JADE)
    (where-is-agent (agent-identifier:namePeter@Zadig:1099/JADE)))
  )
  :language FIPA-SL0
  :ontology JADE-Agent-Management:protocolfipa-request
)
```

Vị trí kết quả sẽ được chứa trong một thông điệp Inform như sau:

```
(INFORM
  :sender (agent-identifier:nameams@Zadig:1099/JADE)
  :receiver (set(agent-identifier:namedal@Zadig:1099/JADE))
  :content ((result
    (action
      (agent-identifier:nameams@Zadig:1099/JADE)
      (where-is-agent (agent-identifier:name
        Peter@Zadig:1099/JADE)))
  )
  (set(location
    :nameContainer-1
    :protocol JADE-IPMT
    :address Zadig:1099/JADE.Container-1
  )))
  :reply-with da1@Zadig:1099/JADE976984777740
  :language FIPA-SL0
  :ontology JADE-Agent-Management
  :protocol fipa-request
)
```

Hành động query-platform-locations (truy vấn platform về vị trí) không có đối số những kết quả trong một tập tất cả những đối tượng phù hợp với Location chứa trong JADE platform hiện tại. Thông điệp cho hành động này rất đơn giản:

```
(REQUEST
  :sender (agent-identifier:nameJohnny)
```

```

:receiver(set(Agent-Identifier:nameAMS))
:content((action(agent-identifier:nameAMS)
          (query-platform-locations)))
:languageFIPA-SL0
:ontologyJADE-Agent-Management
:protocolfipa-request
)

```

Nếu platform hiện tại có 3 container, AMS sẽ gửi lại thông điệp Inform sau:

```

(INFORM
  :sender(Agent-Identifier:nameAMS)
  :receiver(set(Agent-Identifier:nameJohnny))
  :content((Result(action(agent-identifier:nameAMS)
                    (query-platform-locations))
            (set(Location
                  :nameContainer-1128 AgentMobility
                  :transport-protocolJADE-IPMT
                  :transport-addressIOR:000....Container-1)
                  (Location
                    :nameContainer-2
                    :protocolJADE-IPMT
                    :addressIOR:000....Container-2)
                  (Location
                    :nameContainer-3
                    :protocolJADE-IPMT
                    :addressIOR:000....Container-3)
                ))))
  :languageFIPA-SL0
  :ontologyJADE-Agent-Management
  :protocolfipa-request
)

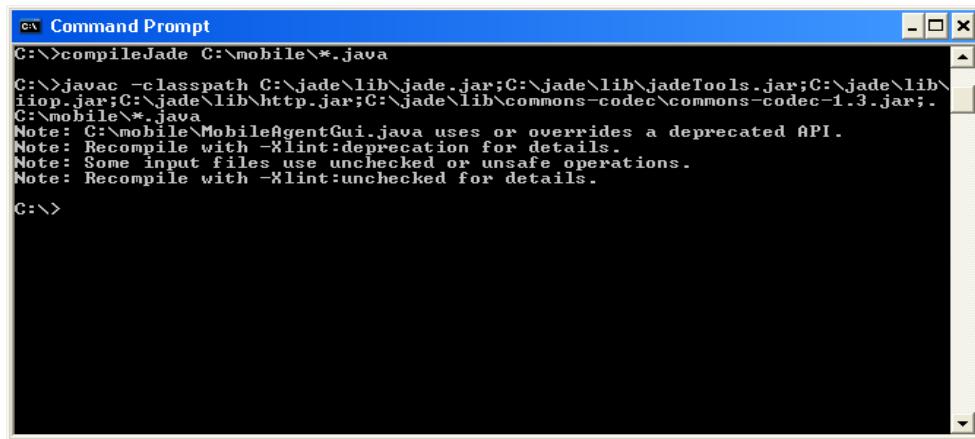
```

Lớp Location thực thi interface jade.core.Location và được truyền đến phương thức Agent.doMove() và Agent.doClone(). Một mẫu hành vi điển hình cho agent di động JADE sẽ hỏi AMS về vị trí (hoặc là danh sách đầy đủ, hoặc là thông qua nhiều hành động where-is-agent) và quyết định, khi nào và ở đâu nên di chuyển.

5.4.6 Ví dụ về tính di động của agent

Đối với tính di động nội platform, các gói ví dụ JADE chứa một ứng dụng chứng minh bao gồm một GUI dành riêng cho việc hỗ trợ sử dụng. Để chạy ứng dụng chứng minh cần có một platform với hai container. Cách chạy ví dụ minh họa:

Copy thư mục mobile trong thư mục C:\jade\src\examples vào ổ C. (Xoá dòng package trong tất cả các lớp của thư mục mobile). Biên dịch các lớp bằng dòng lệnh compileJade (file compileJade.bat đã tạo như ở chương 3):



```
C:\>compileJade C:\mobile\*.*.java
C:\>javac -classpath C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar;C:\mobile\*.java
Note: C:\mobile\MobileAgentGui.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\>
```

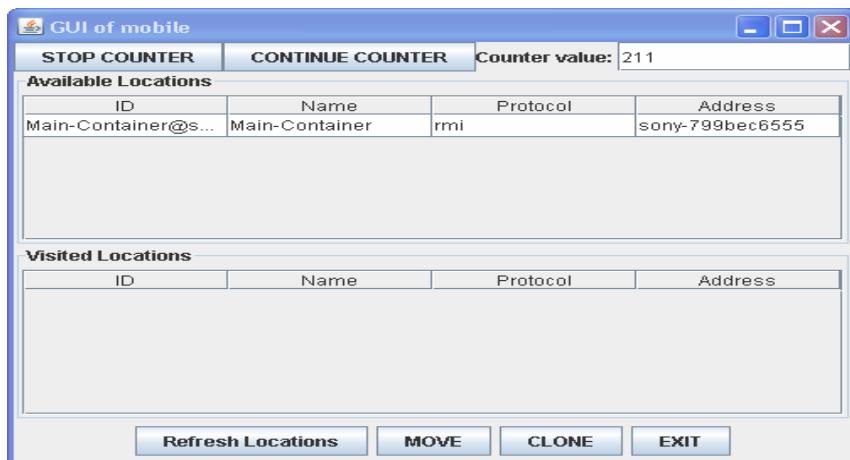
Hình 5.4: Biên dịch các lớp trong ví dụ minh họa

Container đầu tiên có thể được chạy bằng cách:

Tạo file runMobileAgent.bat với nội dung

```
java -classpath
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C
:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar
jade.Boot -gui mobile:MobileAgent
```

và lưu vào thư mục C:\mobile. Chạy file runMobileAgent.bat ta được:



Hình 5.5: giao diện của agent di động

Container thứ hai được chạy bằng cách tương tự nhưng thay đổi nội dung file runMobileAgent.bat như sau:

```
java -classpath  
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C  
:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar  
jade.Boot -container
```

và lưu với tên runNewContainer.bat ở cùng thư mục C:\mobile và chạy file này. Ta thấy một container mới xuất hiện trên platform. Để chạy ví dụ, chỉ cần refresh danh sách vị trí, lựa chọn một contaitner và nhấp vào nút Move hoặc Clone. Tiếp đó, toàn bộ ứng dụng sẽ di chuyển hoặc nhân bản sang container được lựa chọn.

CHƯƠNG 6

KIẾN TRÚC BÊN TRONG CỦA JADE

Cho tới thời điểm này chỉ có những tính năng được hỗ trợ bởi Jade run-time và API để truy cập các tính năng đó được trình bày. Trong phần này, chúng ta sẽ hướng vào kiến trúc cốt lõi bên trong của Jade bao gồm cả việc làm thế nào để xác định và mở rộng các hành vi.

6.1 GIỚI THIỆU CÁC BỘ LỌC CỘNG TÁC PHÂN TÁN

Trước phiên bản Jade 3.2, run-time được cài đặt như một hạt nhân cung cấp khối lượng lớn các tính năng được yêu cầu bởi các Agent để sống và giao tiếp với nhau. Cách tiếp cận này đúng hơn là rất khó sửa đổi, như nó yêu cầu sửa đổi hạt nhân ở bất cứ lúc nào khi có một chức năng mới được giới thiệu. Như vậy, trong bản phát hành phiên bản 3.2, phân phối trong tháng 7 năm 2003, Jade run-time hoàn toàn được cấu trúc lại theo một thiết kế mới gọi là ‘kiến trúc bộ lọc cộng tác phân tán’ (distributed coordinated filters architecture).

6.1.1 Ý tưởng và động cơ thúc đẩy

Kiến trúc mới của Jade được tạo ra bằng cách xem xét các yêu cầu sau:

- Thực thi các tính năng nền tảng như những module tách biệt.
- Khả năng mở rộng linh hoạt để hỗ trợ cho việc tích hợp những tính năng mới và sửa đổi những tính năng đang tồn tại.
- Dễ dàng triển khai các tính năng dựa trên một nền tảng phân tán.
- Chiến lược triển khai: “Triển khai những gì bạn cần” để chỉ các tính năng được bắt đầu và sử dụng mục tiêu thực thi cụ thể. Ví dụ như cho môi trường di động.

Khi nói tới các yêu cầu này, kiến trúc đã thu hút được một số cảm hứng từ lập trình hướng khía cạnh (AOP), trong đó nguyên lý chính là khuyến khích chia nhỏ mối quan tâm một cách rõ ràng. Điều này đạt được bằng cách viết code như một tập hợp các khía cạnh độc lập, mỗi một khía cạnh biểu hiện về những mối quan tâm khác nhau. Các khía cạnh sau đó được kết hợp một cách cẩn thận theo một số quy tắc trong một tiến trình được gọi là quá trình thêu dệt khía cạnh (aspect weaving). Hiện đã có một số phương pháp để cài đặt aspect weaving, cách phổ biến nhất là aspect weaving tại thời điểm tải các lớp thư viện (classload-time aspect weaving). Trong cách tiếp cận này, một ClassLoader được sử dụng để kết hợp các khía cạnh khác nhau khi tải các lớp java. Tuy nhiên, trong một lỗ lực để làm cho công việc này trở nên đơn giản nhất có thể, và xem xét các yêu cầu để chạy Jade trên thiết bị di động, nơi mà các kỹ thuật tải lớp tinh vi thường không sẵn sàng, thì một cách tiếp cận (được đề xuất bởi Aksit et al., 1993) đã được chọn. Trong cách tiếp cận này, được biết đến như các bộ lọc kết hợp, mỗi đối tượng được cung cấp bởi 2 kênh lọc (filter chain) một kênh đến – các bộ lọc của nó được gọi bắt cứ khi nào mà đối tượng nhận một lời gọi phương thức và một kênh đi – các bộ lọc của nó sẽ được gọi khi mà đối tượng sẽ gọi

tới một vài phương thức của đối tượng khác. Kiến trúc bộ lọc cộng tác phân tán bắt nguồn từ cách tiếp cận lọc kết hợp này và hướng đến phân tán qua nhiều container.

6.1.2 Các thành phần chính

Các thành phần chính trong kiến trúc Distributed coordinated filters được mô tả trong Hình 5.1. Mỗi một jade container sẽ đặt ngay phần đầu của mỗi node. Trong khi container được thiết kế để lưu trữ các Agent thì node sẽ lưu trữ các dịch vụ. Một dịch vụ được cài đặt bởi các đặc tính mức nền mà được nhóm lại theo tính gắn kết về mặt khái niệm giữa chúng. Đây là một sự trừu tượng nguyên thủy của kiến trúc lọc cộng tác phân tán. Các dịch vụ như Messaging Service, Agent Management Service và Mobility Service đều cài đặt các cơ chế cho phép agent di cư giữa các container.

Thành phần quản lý hoạt động của các dịch vụ bên trong một node được gọi là dịch vụ quản lý (Service Manager). Trên thực tế, Service Manager sẽ tồn tại với node lưu trữ main container. Service Manager này lưu trữ các bảng của tất cả các node ở trong một platform và danh sách tất cả các dịch vụ đang hoạt động tại mỗi node. Service Manager trên các container ngoại vi chỉ là các proxy.

6.1.3 Các thành phần dịch vụ

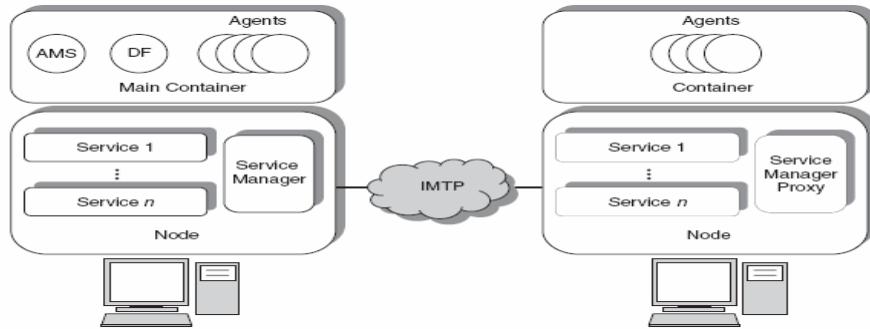
Như đã đề cập, trừu tượng chính ở trong kiến trúc Distributed coordinated filter là dịch vụ. Tất cả các hoạt động mức agent như việc gửi thông điệp, di chuyển, thậm chí cả việc bắt đầu và kết thúc đều được cài đặt bởi một dịch vụ của Jade. Chúng sẽ chứa một vài thành phần được mô tả trong phần này.

6.1.3.1 Lệnh dọc, bộ lọc và sink

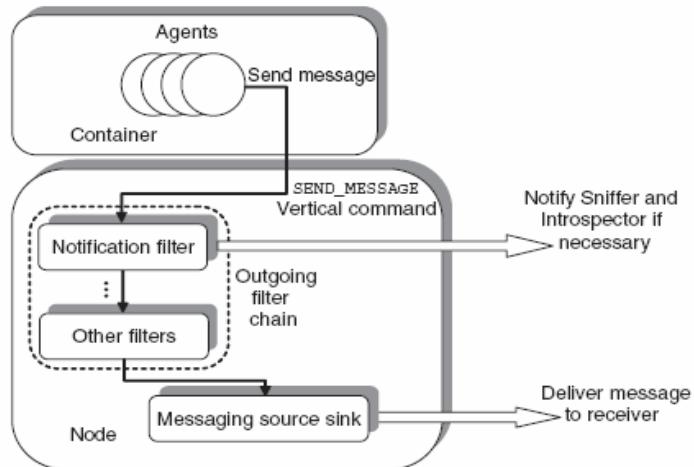
Tất cả các hoạt động mức agent được container ngầm chuyển tiếp tới dịch vụ có trách nhiệm thực hiện hoạt động đó. Ví dụ, một lỗ lực của agent để di cư tới một container từ xa được chuyển tiếp tới dịch vụ Mobility Service bởi container ngầm. Dịch vụ này có thể cài đặt trực tiếp thao tác hoặc đưa ra một lệnh dọc (vertical command). Lệnh dọc là một thể hiện của lớp jade.core.GenericCommand và nhúng vào thao tác cần được thực hiện một số tham số. Khi đưa ra lệnh dọc, việc cài đặt thao tác cần được thực hiện sẽ được giao cho một thành phần của dịch vụ được gọi là outgoing (source) sink. Tuy nhiên, trước khi đến được source sink, lệnh dọc phải đi qua một mắt xích của bộ lọc outgoing filter. Ví dụ, trong hình 6.2, khi một agent gửi thông điệp, container chuyển thông điệp đến Messaging Service. Sau đó tạo ra một lệnh dọc SEND_MESSAGE nhúng vào thông điệp tham số receiver. Outgoing filter của dịch vụ Notification Service sẽ xử lý lệnh bằng cách kiểm tra sender và receiver và đưa ra thông báo. Các bộ lọc khác của các dịch vụ khác không quan tâm đến lệnh SEND_MESSAGE thì sẽ bỏ qua nó. Cuối cùng, messaging sink sẽ phân phối thông điệp đến receiver.

Khi một dịch vụ cài đặt một hoạt động mức agent đang đưa ra một lệnh dọc, nó tạo ra một điểm mở rộng/thay đổi bên trong platform, nhờ đó các dịch vụ khác (có thể đã tạo bởi các nhà phát triển ứng dụng được miêu tả trong phần 7.2) có thể chặn lệnh lại và thêm vào xử lý khác. Một bộ lọc có thể thay đổi một lệnh và thậm chí khóa nó. Ví dụ, để chặn lệnh đứng

SEND_MESSAGE và khóa các thông điệp không tuân thủ, một bộ lọc có thể đơn giản ngăn cản toàn bộ các thông điệp mà không tuân thủ các ràng buộc để được phân phối cụ thể của ứng dụng.



Hình 6.1: Các thành phần chính trong kiến trúc lọc cộng tác phân tán



Hình 6.2: Mắt xích của bộ lọc outgoing filter

6.1.3.2 Lệnh ngang và slice

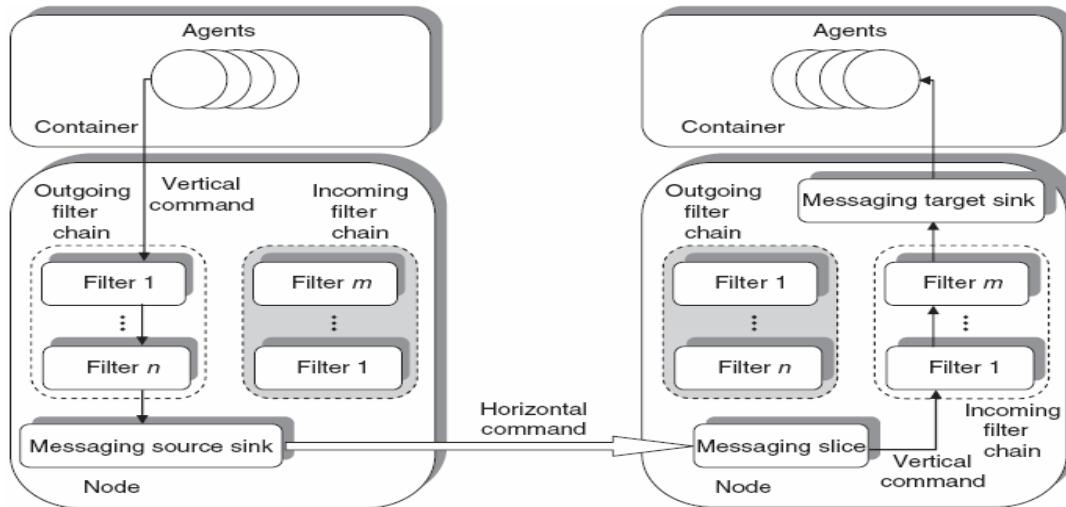
Một dịch vụ dĩ nhiên có thể được phân tán qua một vài node của nền tảng với mỗi phần cần tương tác lẫn nhau. Ví dụ, dịch vụ Messaging Service được cài đặt trên tất cả các node, chúng tạo nên một platform. Khi sink của dịch vụ thông điệp trên một node cần xử lý một lệnh đọc SEND_MESSAGE mang một thông điệp trực tiếp tới một agent tồn tại trên một container ở xa, nó bắt buộc phải dò ra nơi mà bên nhận tồn tại và chuyển thông điệp tới container nơi nhận. Hoạt động đầu tiên có thể liên quan đến liên hệ với dịch vụ Messaging Service đang chạy trên container chính để lấy được vị trí của bên nhận. Hoạt động thứ 2 là yêu cầu sự liên hệ của dịch vụ Messaging Service đang chạy trên node đích để thực sự truyền thông điệp và đưa nó vào hàng đợi thông điệp của bên nhận.

Tương tác giữa các node ở đây được thực hiện nhờ vào các phương tiện của các lệnh ngang. Giống như lệnh đọc, lệnh ngang được nhúng vào các hoạt động được thực hiện với bất kì tham số nào nó có thể có. Vẫn là lớp jade.core.GenericCommand được sử dụng để cài đặt chúng.

Thành phần, trong mọi dịch vụ, cái mà chịu trách nhiệm chấp nhận các lệnh ngang từ các nút ở xa được gọi là một *slice*. Một slice có thể thực hiện trực tiếp hành động được nhúng trong một lệnh ngang đã nhận được, hoặc là nó có thể đưa ra một lệnh đọc mới. Trong trường hợp thứ hai,

hành động thực sự được giao phó cho một thành phần dịch vụ khác gọi là incoming (hay target) sink. Tuy nhiên, trước khi tới target sink của dịch vụ, một lệnh đọc đưa ra bởi các chuyển đi của một slice, theo cách tiếp cận bộ lọc cộng tác, một mắt xích của các bộ lọc incoming filter. Mọi dịch vụ đã cài đặt trên một node có thể cung cấp một bộ lọc với cái mà chúng có thể phản ứng lại trong một dịch vụ đặc biệt với mọi lệnh đọc đã đưa ra bởi mọi slice của dịch vụ trong node đó. Có một chú ý là cả sink nguồn, đích và các bộ lọc outgoing, incoming bắt buộc phải là các thực thể khác nhau (kể cả nếu như chúng là các thể hiện của các lớp khác nhau).

Ví dụ, Hình 6.3 biểu diễn đường đi hoàn thiện được sau bởi một thông điệp ACL trao đổi giữa 2 agent. Có thể để ý rằng đường đi này có từ dạng hình chữ 'U' nơi mà các cạnh đọc tương ứng với các lệnh đọc đi qua các mắt xích của các bộ lọc outgoing và incoming, và cạnh ngang tương ứng lệnh ngang được gửi bởi source sink tới slice trên node đích. Khi một slice của dịch vụ đóng vai trò là một lệnh ngang bằng cách đưa ra duy nhất một lệnh đọc incoming và giao phó hành động thực sự cho target sink của dịch vụ, nó tạo ra một điểm mở rộng/sửa đổi trong platform để các dịch vụ khác có thể ngăn lệnh và thêm vào xử lý khác.



Hình 6.3: Ví dụ về việc phân phối thông điệp

6.1.3.3 Các bộ trợ giúp dịch vụ

Đối với các dịch vụ gắn liền với JADE, các hoạt động mức agent thường được xử lý bởi container ngầm chịu trách nhiệm biến chúng thành các dịch vụ thích hợp cho việc xử lý, như đã giới thiệu trong phần 6.1.3.1. Đối với các dịch vụ được định nghĩa bởi người dùng và đã được mở rộng, cách tiếp cận này sẽ yêu cầu sửa đổi mã của container mỗi khi 1 dịch vụ mới hỗ trợ các hoạt động mức agent được thêm vào. Do đó, kiến trúc bộ lọc cộng tác phân tán bao gồm bộ trợ giúp dịch vụ (Service Helper) trùu tượng cho phép 1 agent trực tiếp truy cập các tính năng được cung cấp bởi 1 dịch vụ. Service Helper là 1 cơ chế mà API lõi của JADE có thể được mở rộng theo cách tương thích với việc không xâm nhập và lùi. Một agent có thể lấy lại helper của 1 dịch vụ trước đó bằng phương thức getHelper() của lớp Agent. Phương thức này lấy tên dịch vụ như là 1 biến và trả lại 1 đối tượng cài đặt giao diện jade.core.ServiceHelper. Đối tượng đó phải tương thích với đúng dịch vụ helper trước khi các phương thức nghiệp vụ của nó có thể được triệu gọi.

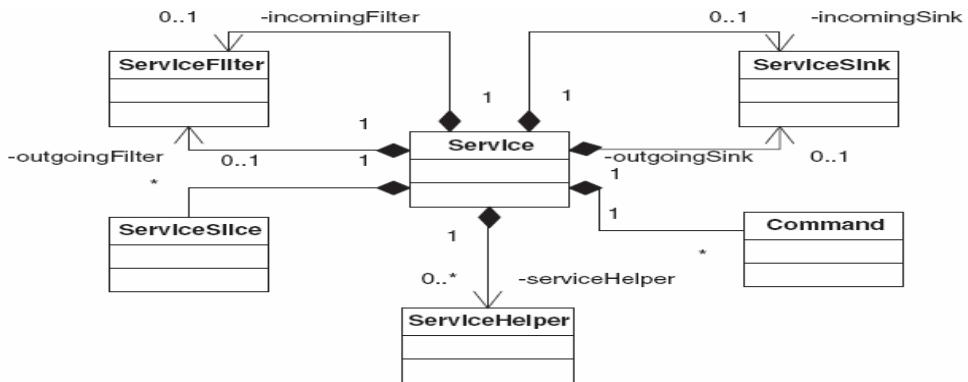
Chẳng hạn, giả sử lỗi của JADE phải được mở rộng để các agent có khả năng gửi các email, một EMailService sẽ được phát triển (chi tiết trong phần 6.2). Rõ ràng tính năng đang được gửi đi của dịch vụ sẽ làm cho không có giá trị nào thông qua một phương thức mới của lớp Agent (chẳng hạn trong trường hợp gửi các thông điệp hoặc di cư). Điều này sẽ dẫn đến việc sửa đổi các lớp lỗi của JADE. Nói cách khác, một EMailHelper tương thích sẽ phải được định nghĩa, cung cấp, chẳng hạn, phương thức sendMail(). Code tương tự như sau có thể được sử dụng:

```

1: ...
2: EMailHelper emHelper=(EMailHelper) getHelper
   (EMailService.NAME);
3: emHelper.sendMail(...);
4: ...

```

Biểu đồ lớp được mô tả trong Hình 6.4 tóm tắt các thành phần cấu tạo lên 1 dịch vụ của JADE ở mức độ khái quát. Chú ý rằng không thành phần nào trong đó là bắt buộc. Một dịch vụ nhỏ không cần có các filter, sink, slice



Hình 6.4: Các thành phần của một dịch vụ mức nhán của JADE

6.1.4 Lựa chọn các dịch vụ được kích hoạt

Khi 1 container của JADE khởi động, các dịch vụ được kích hoạt tại một node được đặc tả bởi các phương tiện của lựa chọn **-services**. Giá trị của lựa chọn này là 1 danh sách được tách biệt bằng dấu chấm phẩy (';'), với các tên lớp đầy đủ, mỗi cài đặt dịch vụ được mô tả như trong phần 6.2. Chú ý quan trọng là dịch vụ Messaging Service và dịch vụ Agent Management Service, bắt buộc trong mọi platform của JADE, luôn được kích hoạt bất chấp giá trị của lựa chọn **-services**. Bên cạnh 2 dịch vụ chính, JADE cũng kích hoạt dịch vụ Mobility Service (xem chương 5) và dịch vụ Notification Service ở chế độ mặc định. Việc chạy JADE mà không lựa chọn dịch vụ, như trình bày dưới đây, tương đương với việc không chỉ rõ dịch vụ nào

```
java jade.Boot... -services
```

```
jade.core.mobility.AgentMobilityService;jade.core.event.NotificationService
```

Nó cũng được ghi chú rằng khi thêm 1 dịch vụ để được chạy, việc chạy JADE với 1 dòng lệnh như sau

```
java jade.Boot... -services myPackage.myService
```

sẽ khởi động 1 container không hỗ trợ khả năng di động của agent và khả năng giám sát agent bởi vì khi đó sự mặc định đã bị ghi đè, dịch vụ Mobility Service và dịch vụ Notification Service không được kích hoạt nữa.

6.2 TẠO MỘT DỊCH VỤ LÔI TRONG JADE

Chúng ta đã mô tả các thành phần cấu tạo nên 1 dịch vụ lõi của JADE và minh họa các vai trò và các tương tác của chúng. Phần này sẽ giới thiệu các lớp mà cài đặt chúng và làm thế nào để tạo một dịch vụ do người dùng định nghĩa.

Dịch vụ đăng nhập đơn giản cho book-trading được sử dụng trong cuốn sách này sẽ được dùng như một ví dụ để chỉ ra tất cả các thông điệp được các agent trao đổi. Dịch vụ sẽ được xây dựng lớn dần. Các thông điệp giai đoạn đầu tiên sẽ được in ra trên các đầu ra chuẩn mô tả rằng một dịch vụ có thể cung cấp các bộ lọc đơn giản để ngăn chặn các lệnh đọc vào ra (như phần lớn các dịch vụ do người dùng định nghĩa thường làm). Trong pha thứ 2 các dịch vụ sẽ được sửa lại để in ra tất cả các thông điệp trên đầu ra chuẩn của main container. Trong pha này mô tả một dịch vụ có thể yêu cầu một mức độ hợp tác giữa các node khác nhau. Cuối cùng các dịch vụ sẽ được mở rộng bằng cách thêm vào Service Helper cho phép các agent ngay lập tức tương tác với nó.

6.2.1 Cài đặt lớp của dịch vụ

Một dịch vụ có thể được tạo ra bằng cách cài đặt giao diện jade.core.Service, hoặc thuận tiện hơn, bằng cách làm lớp con của lớp jade.core.BaseService. Cách thứ hai cung cấp việc cài đặt ngầm định cho hầu hết các phương thức của giao diện Service. Phương thức duy nhất mà phải được cài đặt là getName(), nó trả lại 1 chuỗi được dùng để định danh dịch vụ đã được cài đặt. Mặc dù điều này không hoàn toàn bắt buộc, nhưng quy tắc được khuyến nghị cho tên dịch vụ như sau :

- Sử dụng hậu tố Service trong tất cả các lớp extending BaseService
- Sử dụng tên lớp dịch vụ không có hậu tố Service chẳng hạn như tên của dịch vụ được trả lại bởi phương thức getName().

Trong khi dịch vụ khởi động, hai phương thức của giao diện Service được gọi bởi node ngầm. Phương thức init() được gọi đầu tiên một cách thụ động trước khi dịch vụ được cài đặt thực sự trong node; phương thức này có thể được coi như là một phương thức khởi tạo. Phương thức boot() được gọi sau khi dịch vụ đã hoạt động. Dịch vụ này sau đó có thể sử dụng các tính năng của ServiceManager và local container. Phương thức boot() có thể được sử dụng để thực thi, ví dụ một giao thức khởi tạo phân tán. Code bên dưới thể hiện khung của lớp LoggingService thực thi dịch vụ Logging Service đã được mô tả ở trên.

```
3: import jade.core.*;
4:
5: public class LoggingService extends BaseService{
6: // Service name
7: public static final String NAME =
"bookTrading.logging.Logging";
8:
```

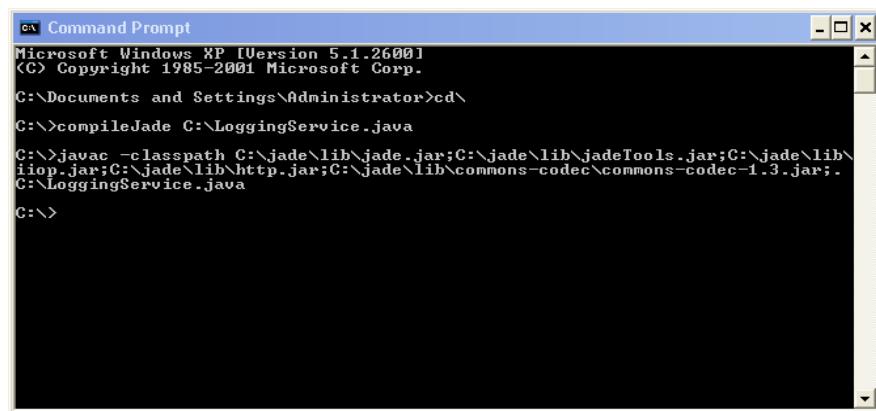
```

9: // Service parameter names
10: public static final String VERBOSE =
"bookTrading_logging_LoggingService_verbose";
11:
12: private boolean verbose = false;
13:
14: public String getName() {
15: return NAME;
16: }
17:
18: public void boot(Profile p) throws ServiceException {
19: super.boot(p);
20: verbose = p.getBooleanProperty(VERBOSE, false);
21: System.out.println("VERBOSE = "+verbose);
22: }
23: }
```

Ở dòng 18 phương thức boot() lấy một đối tượng jade.core.Profile là một tham số. Đối tượng Profile này có hiệu lực với tất cả các cấu hình khởi động của JADE đã được mô tả ở phần 4.6. Các cấu hình cụ thể của từng dịch vụ có thể được đặc tả như tham số dòng lệnh dưới khuôn dạng `-key value`. Trong trường hợp này chúng có thể được lấy lại bằng chính dịch vụ của nó thông qua đối tượng Profile được truyền như là tham số vào phương thức boot(). Ví dụ này, chúng ta giả sử dịch vụ Logging Service chấp nhận một tham số `verbose` để bảo nó in ra tất cả các trường của thông điệp (khi verbose nhận giá trị true) hoặc chỉ in ra performative (nếu verbose là false). Một quy ước đặt tên tốt được sử dụng trong JADE để tránh sự lộn xộn giữa các tham số liên quan đến các dịch vụ khác nhau, là xây dựng các tên của tham số bằng cách ràng buộc tên lớp của dịch vụ (sự thay thế những dòng gạch dưới bằng những dấu chấm) với định danh của tham số. Hằng số tĩnh `VERBOSE` được định nghĩa ở dòng 10 cung cấp một ví dụ. Ở dòng 20 thì phương thức `getBooleanProperty()` của lớp Profile được sử dụng để lấy giá trị (boolean) của tham số `bookTranding_logging_LoggingService_verbose`.

6.2.2 Khởi động dịch vụ

Lưu file LoggingService.java với nội dung như trên vào ô C. Chạy lệnh compileJade với nội dung file compileJade.bat như mô tả trong Chương 3:



Hình 6.5: Biên dịch lớp LoggingService.java

Tạo file runLoggingService.bat với nội dung:

```
java -classpath
.;C:\jade\lib\jade.jar;C:\jade\lib\jadeTools.jar;C:\jade\lib\iiop.jar;C
:\jade\lib\http.jar;C:\jade\lib\commons-codec\commons-codec-1.3.jar
jade.Boot -gui -services
jade.core.event.NotificationService;LoggingService -
bookTrading_logging_LoggingService_verbose true
```

và lưu vào ổ C sau đó chạy file này ta có kết quả:

```
C:\WINDOWS\system32\cmd.exe
WARNING: Automatic main-detection mechanism initialization failed (Error setting
        up multicast socket [nested java.net.SocketException: error setting options]). Mechanism disabled!
Jun 13, 2010 11:02:06 AM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
Jun 13, 2010 11:02:06 AM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
Jun 13, 2010 11:02:06 AM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
Jun 13, 2010 11:02:06 AM jade.core.BaseService init
INFO: Service bookTrading.logging.Logging initialized
Jun 13, 2010 11:02:06 AM jade.core.messaging.MessagingService clearCachedSlice
INFO: Clearing cache
Jun 13, 2010 11:02:11 AM jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParse
rImpl$JAXPSAXParser
Jun 13, 2010 11:02:11 AM jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://127.0.0.1:7778/acc
VERBOSE = true
Jun 13, 2010 11:02:11 AM jade.core.AgentContainerImpl joinPlatform
INFO:
Agent container Main-Container@sony-799bec6555 is ready.
```

Hình 6.6: Kết quả chạy LoggingService của ví dụ bookTrading

6.2.3 Sử dụng bộ lọc để chặn các lệnh dọc

Cho đến nay Logging Service không cung cấp bất kỳ một chức năng gì, do đó bước kế tiếp là làm cho nó thực sự ghi chép lỗi các thông điệp được trao đổi bởi các agent trong ngữ cảnh book-trading. Để làm điều này một số bộ lọc dịch vụ cần được cài đặt, đặc biệt là cần một bộ lọc outgoing filter để ghi chép lỗi các thông điệp một khi chúng được gửi đi. Các bộ lọc dịch vụ được cài đặt bằng cách làm lớp con của lớp jade.core.Filter và được định nghĩa lại các phương thức accept() và postProcess(). Cả hai phương thức nhận về một tham số jade.core.VerticalCommand. Khi một lệnh dọc đi qua một chuỗi bộ lọc, trình tự các lời gọi sau đây xảy ra:

```
Filter-1.accept()
...
Filter-n.accept()
Sink.consume()
Filter-n.postProcess()
...
Filter-1.postProcess()
```

Vì vậy một bộ lọc có thể hoạt động trên một lệnh đọc cho cả hai phương thức (accept()) và sau phương thức postProcess()) trước khi thao tác tương ứng với lệnh đọc thực sự được thực hiện bởi các service sink. Phương thức accept() trả về một giá trị boolean. Với việc trả về giá trị false, một filter có thể block một lệnh đọc.

Một dịch vụ cung cấp các bộ lọc đi và đến (nếu có) của nó bằng phương thức getCommandFilter() của giao diện Service. Phương thức này nhận một tham số chỉ hướng của bộ lọc để được trả lại. Giá trị có thể cho tham số này là Filter.INCOMING và Filter.OUTGOING. Ngay cả khi điều này là không bắt buộc, một phong cách lập trình điển hình khi phát triển các dịch vụ lõi của JADE là cài đặt các thành phần của dịch vụ như bộ lọc, sink và slice như là các lớp nội của lớp dịch vụ chính. Đoạn code dưới đây cho thấy cách mà Service Logging có thể in tất cả các thông điệp được gửi trên đầu ra chuẩn.

```
1: ...
2: private OutgoingLoggingFilter outFilter = new OutgoingLoggingFilter();
3: ...
4: public Filter getCommandFilter(boolean direction) {
5: if (direction == OUTGOING) {
6: return outFilter;
7: }
8: else {
9: return null;
10: }
11: }
12: ...
13: private class OutgoingLoggingFilter extends Filter {
14: public boolean accept(VerticalCommand cmd) {
15: if (cmd.getName().equals(MessagingSlice.SEND_MESSAGE)) {
16: Object[] params = cmd.getParams();
17: AID sender = (AID) params[0];
18: GenericMessage gMsg = (GenericMessage) params[1];
19: ACLMessage msg = gMsg.getACLMensaje();
20: AID receiver = (AID) params[2];
21: System.out.println("Message from "+sender+" to
"+receiver+":");
22: if (verbose) {
23: System.out.println(msg);
24: }
25: else {
26: System.out.println(ACLMensaje.getPerformative(
msg.getPerformative()));
27: }
28: }
29: // Never block a command
30: return true;
31: }
32: }
```

Từ dòng 17-20, lệnh đọc SEND_MESSAGE có 3 tham số đại diện cho người gửi thông điệp, thông điệp chính nó và mục đích nhận. Trên thực tế, các đối tượng ACMLMessage đã được tự đóng gói trong một đối tượng jade.core.messaging.GenericMessage.

Các lớp GenericMessage nhúng các thông tin khác ngoài thông điệp ACL thực được sử dụng bởi sink của dịch vụ Messaging Service để xử lý các trường hợp phân phối đặc biệt.

6.2.4 Cài đặt một dịch vụ phân tán trong JADE

Trong phần này ví dụ Logging Service được mở rộng để thể hiện tất cả các bản in lỗi trên đầu ra chuẩn mà không quan tâm đến container mà nó được sinh ra. Để đạt được điều này, một slice của dịch vụ Logging Service phải được thực thi. Hơn nữa, OutgoingLoggingFilter được biểu diễn trong phần 6.2.3 phải được điều chỉnh để gửi một lệnh ngang tới slice của dịch vụ Logging Service ở trên container chính mỗi lần nó chẩn một thông điệp.

Việc thực thi một slice cho mỗi dịch vụ là hơi phức tạp hơn so với việc thực thi một bộ lọc, vì cần phải phát triển ba lớp.

6.2.4.1 Giao diện ngang (horizontal interface)

Giao diện này khai báo tất cả các phương thức có thể được gọi từ một slice từ xa và phải mở rộng giao diện Service.Slice. Tất cả các phương thức trong giao diện ngang phải ném ra ngoại lệ jade.core.IMTPEException nếu một vài vấn đề mạng xảy ra tại mức IMTP, ví dụ không lấy được slice từ xa. Ví dụ giao diện ngang có thể như sau:

```
1: package bookTrading.logging;
2:
3: import jade.core.*;
4:
5: public interface LoggingSlice extends Service.Slice {
6:     public static final String H_LOGMESSAGE = "log-message";
7:
8:     public void logMessage (String s) throws IMTPEException;
9: }
```

Giao diện ngang cũng là nơi lý tưởng để định nghĩa các hằng số biểu diễn tên của các lệnh ngang mà nó có thể được đáp ứng bởi service slice. Đặc biệt phải chỉ có một lệnh ngang cho mỗi phương thức trong giao diện ngang.

6.2.4.2 Slice Proxy

Đây là lớp mà các thể hiện của nó được trao quyền đại diện cho một remote slice. Nó mở rộng từ lớp jade.core.SliceProxy và phải thực thi giao diện ngang. Khi một dịch vụ cần tương tác với một slice trên một node từ xa, đầu tiên nó phải khôi phục lại một proxy trả tới slice đó và sau đó gọi phương thức đã được yêu cầu. Mục đích chính của proxy là chuyển đổi các lời gọi phương thức thành các lệnh ngang thích hợp cái mà sẽ được gửi đến remote slice. Các lớp proxy được chủ động tải về mỗi khi cần và phải tuân theo quy ước đặt tên: chúng phải được đặt tên <servicename>Proxy. Lớp bookTrading.logging.LoggingProxy được cài đặt như sau:

```
1: package bookTrading.logging;
2:
3: import jade.core.*;
4:
5: public class LoggingProxy extends SliceProxy implements
LoggingSlice {
6:     public void logMessage (String s) throws IMTPEException {
```

```

7: GenericCommand cmd = new GenericCommand(H_LOGMESSAGE,
LoggingService.NAME, null);
8: cmd.addParam(s) ;
9: getNode().accept(cmd);
10: }
11: }
```

Dòng 7, lớp `jade.core.GenericCommand` được sử dụng để thực thi các lệnh ngang. Những lệnh này sau đó được thêm vào các tham số cần thiết và được chuyển tới các remote node. Lưu ý rằng một slice proxy được gắn vào một proxy trả tới node mà có proxy slice cư ngụ tại đó.

6.2.4.3 Cài đặt Slice

Đây là lớp thực thi giao diện `Service.Slice`. Nó chịu trách nhiệm đảm bảo dịch vụ gọi các lệnh ngang. Việc thực thi Slice không cần thực thi giao diện ngang. Lớp thực thi slice cho Logging Service được cài đặt như sau:

```

1: . . .
2: private class LoggingSliceImpl implements Service.Slice {
3:
4: public Service getService() {
5: return LoggingService.this;
6: }
7:
8: public Node getNode() throws ServiceException {
9: try {
10: return LoggingService.this.getLocalNode() ;
11: }
12: catch (IMTPEException imtpe) {
13: // Should never happen as this is a local call
14: throw new ServiceException("Unexpected error retrieving
local node") ;
15: }
16: }
17:
18: public VerticalCommand serve (HorizontalCommand cmd) {
19: String cmdName = cmd.getName() ;
20: if (cmd.getName().equals (LoggingSlice.H_LOGMESSAGE)) {
21: Object[] params = cmd.getParams () ;
22: System.out.println (params [0]) ;
23: }
24: }
25: }
26: . . .
```

Các phương thức `getService()` và `getNode()` khá đơn giản và phần thân của nó khá rõ ràng trong tất cả các lớp thực thi slice. Phương thức `serve()` là điểm trung tâm của lớp thực thi slice và được gọi đến mỗi lần một lệnh ngang được nhận. Một slice có thể phục vụ trực tiếp một lệnh ngang như trong ví dụ trên, hoặc như đã đề cập trong phần 6.1.3.2, nếu các nhà phát triển muốn các dịch vụ khác phản ứng lại việc nhận một lệnh ngang, họ có thể cài đặt phương thức `serve()` để trả về đối tượng `VerticalCommand`, đối tượng này sẽ di chuyển qua mắt xích của bộ lọc incoming filter và tới service sink. Một dịch vụ tuyên bố nó có khả năng để thực hiện việc phối hợp giữa các

node bằng cách thực thi các phương thức getHorizontalInterface() và the getLocalSlice(). Để làm được điều đó, lớp LoggingService phải được cài đặt như sau:

```
1: . . .
2: private ServiceSlice localSlice = new LoggingSliceImpl() ;
3: . . .
4: public Class getHorizontalInterface() {
5: return LoggingSlice.class;
6: }
7:
8: public Service.Slice getLocalslice() {
9: return localSlice;
10: }
11: . . .
```

Tại thời điểm này, LoggingService đã sẵn sàng để chuyển tiếp các ghi chép lỗi tới các nút của platform. Việc cuối cùng phải làm là sửa đổi OutgoingLoggingFilter như trong phần 6.2.3 để chuyển tiếp các ghi chép lỗi đến logging slice trên main container. Việc thực thi các dòng code như sau:

```
1: . . .
2: private class OutgoingLoggingFilter extends Filter {
3: public boolean accept (VerticalCommand cmd) {
4: if (cmd.getName().equals (MessagingSlice.SEND_MESSAGE)) {
5: Object[] params = cmd.getParams() ;
6: AID sender = (AID) params [0] ;
7: GenericMessage gMsg = (GenericMessage) params[1] ;
8: ACLMessage msg = gMsg.getACLMessag() ;
9: AID receiver = (AID) params[2] ;
10: // Prepare the log record
11: String logRecord = "Message from "+sender+ " to
"+receiver+": \n" ;
12: if (verbose) {
13: logRecord = logRecord+msg;
14: }
15: else {
16: logRecord = logRecord+ACLMessag.getPerformative
(msg.getPerformative());
17: }
18:
19: // Send the log record to the logging slice on the Main
Container
20: try {
21: LoggingSlice mainSlice = (LoggingSlice)
getSlice(MAIN_SLICE) ;
22: mainSlice.logMessage(logRecord) ;
23: }
24: catch (ServiceException se) {
25: System.out.println("Error retrieving Main Logging
Slice") ;
26: se.printStackTrace() ;
27: }
28: catch (IMTPException impte) {
```

```

29: System.out.println ("Error contacting Main Logging
Slice") ;
30: se.printStacktrace() ;
31: }
32: }
33: // Never block a command
34: return true;
35: }
36: }

```

Phương thức getSlice() của dịch vụ Service được sử dụng để phục hồi một proxy để đưa đến một remote slice. Phương thức này lấy tên của node mà chưa remote slice đó. Hằng MAIN_SLICE được định nghĩa trong lớp BaseService luôn báo đến slice trong container chính.

6.2.5 Tương tác giữa Agent và dịch vụ

Như đã được miêu tả trong phần 6.1.3.3, agent có thể tương tác trực tiếp với dịch vụ mà cung cấp Service Helper. Để minh họa làm sao mà một dịch vụ có thể cung cấp một helper, the Logging Service được mở rộng để cho phép agent thay đổi thuộc tính verbose của nó.

Một phong cách lập trình được khuyến cáo ở đây bao gồm định nghĩa giao diện LoggingHelper mở rộng từ lớp ServiceHelper và một lớp nội LoggingHelper của lớp dịch vụ chính. Giao diện LoggingHelper có thể là các dòng sau:

```

1: package bookTrading.logging;
2:
3: import jade.core.*;
4:
5: public interface LoggingHelper extends ServiceHelper {
6: public void setVerbose(boolean verbose);
7: }

```

Một dịch vụ cung cấp một helper bằng cài đặt phương thức getHelper của giao diện Service. Phương thức này được gọi mỗi khi một agent lấy helper cho dịch vụ đó lần đầu tiên. Điều này cho phép dịch vụ sử dụng một đối tượng helper cho tất cả agent hoặc một helper cho mỗi agent.

```

1: . . .
2: private ServiceHelper helper = new LoggingHelperImpl();
3: . . .
4: public ServiceHelper getHelper (Agent a) {
5: return helper;
6: }
7: . . .
8: public class LoggingHelperImpl implements LoggingHelper {
9: public void init(Agent a) {
10: }
11:
12: public void setVerbose (boolean v) {
13: verbose = v;
14: }
15: }
16: . . .

```

Phương thức init() được định nghĩa ở dòng 9 là phương thức duy nhất bao gồm giao diện ServiceHelper và được gọi trước khi trả về một đối tượng helper cho một agent yêu cầu nó ở lần đầu tiên. Khi sử dụng đối tượng helper cho mỗi agent, nó cho phép liên kết đối tượng helper với agent được liên kết với nó. Vì trong trường hợp này, một đối tượng helper đơn được sử dụng cho mọi agent, nên thân của nó rỗng.

CHƯƠNG 7

PHÁT TRIỂN HỆ ĐA AGENT VỚI PHƯƠNG PHÁP LUẬN MaSE VÀ JADE

Chương này bàn về vấn đề sử dụng phương pháp luận MaSE kết hợp với Jade khi phát triển các hệ phân tán trong thế giới thực. Cho đến nay có rất nhiều phương pháp luận cũng như rất nhiều khung (framework) được phát triển để xây dựng các hệ đa agent phức tạp. Một số phương pháp luận như Gaia, Tropos, Ingenias, MaSE (Multi-agent Systems Engineering)...và một số khung như TuCSoN (TUple Centre Spread Over the Network), JADE (Java Agent DEvelopment Framework), Jadex, DESIRE (Design and Specification of Interacting Reasoning components)...đã được giới thiệu. Chương này tập trung trình bày phương pháp luận MaSE kết hợp với nền tảng JADE và một case study Quản lý Hội thảo. Hệ thống này cho phép các tác giả gửi bài báo đến hội đồng và cho phép hội đồng gán trách nhiệm cho các thành viên tham gia nhận xét. Sau đó hệ thống đưa ra quyết định chấp nhận bài báo hay không. Lý do lựa chọn MaSE là vì phương pháp này khá gần gũi với những người đã từng phát triển các hệ thống theo cách tiếp cận hướng đối tượng và lý do lựa chọn JADE là vì đây là một framework dễ sử dụng, đã trở thành chuẩn công nghiệp để phát triển các hệ đa agent phức tạp và đặc biệt là tạo thuận lợi cho những người đã từng lập trình bằng ngôn ngữ Java.

7.1 GIỚI THIỆU

Lịch sử tính toán cho đến nay đã được đánh dấu bởi năm xu hướng quan trọng và đang còn tiếp diễn:

- Tính toán mọi nơi
- Kết nối lẫn nhau
- Tính toán thông minh
- Tính toán đại diện
- Tính toán hướng đến con người

Xu hướng thứ nhất, tính toán mọi nơi, nghĩa là giảm chi phí đầu tư vào khả năng tính toán bằng cách sử dụng sức mạnh xử lý vào những nơi được coi là không kinh tế. Không giống như các hệ thống máy tính thời kỳ đầu, các hệ thống máy tính ngày nay đã được kết nối với nhau để tạo thành một hệ thống phân tán lớn. Internet là một ví dụ. Hiếm có một máy tính nào sử dụng trong lĩnh vực thương mại và học thuật mà không có khả năng truy cập Internet. Ngày nay, các hệ phân tán và đồng thời là chủ đề của nhiều cuộc nghiên cứu và thực nghiệm để từ đó người ta

đánh giá lại nền tảng của khoa học máy tính và tìm kiếm những mô hình lý thuyết mới phản ánh thực tế tính toán tốt hơn: thực chất tính toán là một quá trình tương tác lẫn nhau.

Xu hướng thứ ba, tính toán thông minh, nghĩa là độ phức tạp của những tác vụ mà chúng ta có thể tự động hóa và giao phó cho máy tính đang ngày càng tăng. Khi đó, các kỹ sư máy tính có thể giải quyết được những nhiệm vụ phức tạp mà không được chỉ trong một thời gian ngắn.

Xu hướng tiếp theo là hướng đến tăng khả năng giao phó công việc cho máy. Ví dụ như hệ thống máy bay không người lái (ở đó, lời khuyên của chương trình máy tính được tin cậy hơn là những phi công dày dạn kinh nghiệm).

Cuối cùng, xu hướng tính toán hướng con người là một bước chuyển dịch lớn từ quan điểm lập trình hướng máy móc. Điều này được chứng minh ở cách mà chúng ta tương tác với máy tính. Thời kỳ đầu, tương tác được thực hiện thông qua việc thiết lập các switch trên panel của máy. Tiếp theo đó là các giao diện dòng lệnh. Sau khi có sự ra đời của giao diện đồ họa, người sử dụng có thể điều khiển thiết bị thông qua các icon một cách trực tiếp. Cùng với sự phát triển đó là các quan điểm lập trình cũng thay đổi theo: từ hợp ngữ, lập trình hướng thủ tục, lập trình hướng cấu trúc đến lập trình hướng đối tượng. Những sự phát triển đó cho phép người lập trình cài đặt phần mềm ở một mức trừu tượng cao hơn và hướng con người hơn.

Những thách thức đó đã thúc đẩy quá trình phát triển các công cụ và các cơ chế cho phép xây dựng các hệ phân tán với khả năng sử dụng lại và tin cậy cao hơn. Do đó, lĩnh vực mới các hệ đa agent đã trở thành chủ đề thu hút nghiên cứu và phát triển mạnh mẽ trong khoa học máy tính ngày nay.

7.2 MỘT SỐ ĐẶC ĐIỂM CỦA NỀN TẢNG PHÁT TRIỂN AGENT

7.2.1 Một số nền tảng hỗ trợ phát triển hệ đa agent

Có nhiều ngôn ngữ lập trình agent mà bên dưới là một platform cài đặt ngữ nghĩa của ngôn ngữ lập trình đó, ví dụ như JACK Agent Language (JAL), 3APL (An Abstract Agent Programming Language), IMPACT, AF-APL (Agent Factory Agent Programming Language). Tuy nhiên cũng có nhiều platform không phụ thuộc vào ngôn ngữ lập trình, mà thay vào đó là cung cấp sự hỗ trợ các khía cạnh như truyền thông/giao tiếp và phối hợp. Sau đây, tôi xin giới thiệu một số platform như vậy, trong đó, nổi bật nhất là JADE.

TuCSon (Tuple Centre Spread over the Network) là một framework phối hợp các hệ đa agent. TuCSon dựa trên một mô hình và một cơ sở hạ tầng cung cấp mục đích chung và các dịch vụ có thể lập trình được trợ giúp việc truyền thông và phối hợp của agent. Đây là một framework hoàn toàn dựa trên Java bao gồm các thành phần: runtime platform, tập các thư viện APIs, tập các công cụ. Lõi của TuCSon là công nghệ tuProlog là một máy Prolog được tích hợp hoàn toàn với môi trường Java. Ngày nay, TuCSon cũng được sử dụng để xây dựng các hệ thống dựa trên agent trong các dự án thuộc môi trường học tập, nghiên cứu.

Jadex là một framework để tạo các agent hướng đích tuân theo mô hình belief-desire-intention (BDI). Framework này được cài đặt thành một tầng nằm trên đỉnh của một cơ sở hạ tầng middleware như JADE và cho phép phát triển agent với các công nghệ đã có sẵn như Java,

XML. Máy suy luận của Jadex giải quyết những giới hạn của các hệ BDI truyền thống bằng cách giới thiệu các khái niệm mới như mục đích rõ ràng và cơ chế cân nhắc mục đích, làm cho kết quả của việc phân tích và thiết kế hướng goal có thể dễ dàng chuyển sang giai đoạn cài đặt. Jadex được sử dụng để xây dựng các ứng dụng trong các lĩnh vực khác nhau như mô phỏng, lập lịch và tính toán di động. Ví dụ, Jadex được sử dụng để phát triển một ứng dụng đa agent trong đảm phán lịch chữa trị ở bệnh viện.

DESIRE (DEsign and Specification of Interacting REasoning components) là một phương pháp phát triển gộp cho các hệ đa agent dựa trên kiến trúc gộp. Trong đó, việc thiết kế agent dựa trên những khía cạnh chính sau: gộp tiến trình, gộp tri thức và các quan hệ giữa gộp tiến trình và gộp tri thức. Việc đặc tả quan hệ gộp liên quan đến khả năng trao đổi thông tin giữa các thành phần và cấu trúc điều khiển để kích hoạt các thành phần. DESIRE được sử dụng cho những ứng dụng như cân bằng tải của các hệ thống chuẩn đoán và phân phối điện.

JADE (Java Agent DEvelopment Framework) như đã trình bày trong các Chương trước, Jade là một Java framework để phát triển các ứng dụng đa agent phân tán. Nó cung cấp một tập các dịch vụ sẵn có và dễ sử dụng cũng như một vài công cụ đồ họa để gỡ lỗi và kiểm thử. Một trong những mục tiêu chính của JADE là hỗ trợ khả năng phối hợp hoạt động bằng cách tuân theo một cách nghiêm ngặt đặc tả FIPA về kiến trúc cũng như cơ sở hạ tầng truyền thông. Ngoài ra, JADE cũng khá mềm dẻo và có thể thích nghi để sử dụng trên các thiết bị giới hạn về tài nguyên như PDAs và mobile phones. JADE đã được sử dụng rộng rãi trong những năm qua trong các tổ chức công nghiệp cũng như các tổ chức học thuật từ những hướng dẫn giảng dạy trong các khóa học liên quan đến agent đến các bản mẫu công nghiệp. Một ví dụ điển hình là việc Whitestein sử dụng JADE để xây dựng một hệ thống hỗ trợ ra quyết định ở các trung tâm cấy ghép nội tạng.

7.2.2 Một số đặc điểm nổi bật của JADE

Như đã trình bày ở các chương trước, JADE là một nền tảng phần mềm cung cấp chức năng cơ bản cho tầng giữa. Nó độc lập với các ứng dụng cụ thể và đơn giản hóa việc cài đặt các ứng dụng phân tán – những ứng dụng khai thác sự trừu tượng của các agent phần mềm. Một đặc điểm đáng chú ý của JADE là nó thực thi sự trừu tượng này trên ngôn ngữ hướng đối tượng Java, cung cấp một API đơn giản và thân thiện. Những lựa chọn thiết kế đơn giản hóa của JADE đều bị ảnh hưởng bởi sự trừu tượng của agent. Do đó, JADE đã được cài đặt để cung cấp cho các nhà lập trình các chức năng cốt lõi cho phép sẵn sàng sử dụng và dễ dàng tùy biến:

- Hệ thống hoàn toàn phân tán cho phép các agent cư trú trên đó, mỗi agent hoạt động như là một luồng riêng biệt, và có khả năng giao tiếp một cách trong suốt với agent khác.
- Tuân thủ đầy đủ các đặc tả của FIPA. Nền tảng JADE tham gia thành công vào tất cả các sự kiện phối hợp hoạt động của FIPA và được sử dụng như là tầng giữa của nhiều nền tảng trong mạng lưới Agentcities. Điều này đã tạo nên sự đóng góp lớn lao của nhóm JADE vào quá trình chuẩn hóa của FIPA.
- Phương tiện vận chuyển hiệu quả của các thông điệp không đồng bộ thông qua một API không bị phụ thuộc về vị trí. Nền tảng JADE lựa chọn các phương tiện sẵn có tốt nhất của

truyền thông. Khi đi qua ranh giới giữa các nền tảng, các thông điệp tự động được biến đổi từ cách biểu diễn bằng Java bên trong của JADE sang các cú pháp, cách giải mã và các giao thức vận chuyển tuân theo FIPA.

- Thực thi cả 2 dịch vụ white-page và yellow-page. Hệ thống có thể được cài đặt để biểu diễn các miền và các miền con như một đồ thị các thư mục.
- Quản lý vòng đời agent đơn giản nhưng hiệu quả. Khi các agent đã được tự động gán một định danh toàn cục duy nhất và một địa chỉ vận chuyển được sử dụng để đăng ký với dịch vụ white-page của nền tảng. Các API đơn giản và các công cụ đồ họa cũng được cung cấp để quản lý vòng đời agent vừa từ xa và vừa cục bộ, như tạo, định chỉ, phục hồi, đóng băng, tan băng, di chuyển, lặp lại và xóa.
- Cung cấp tính di động của agent. Cả mã và trạng thái của agent đều có thể di chuyển giữa các tiến trình và các máy. Sự di chuyển được tạo ra để các agent giao tiếp một cách trong suốt mà có thể tiếp tục tương tác thậm chí là trong suốt quá trình di chuyển.
- Một cơ chế đặt trước (subscription) cho mỗi agent, và thậm chí là cả các ứng dụng bên ngoài, mà muốn đăng ký với nền tảng để được thông báo về tất cả các sự kiện của platform, bao gồm các sự kiện có liên quan đến vòng đời và các sự kiện trao đổi thông điệp.
- Một tập các công cụ đồ họa để hỗ trợ người lập trình khi debug và monitor. Chúng đặc biệt quan trọng và phức tạp trong các hệ thống đa luồng, nhiều tiến trình, nhiều máy ví dụ như một ứng dụng JADE điển hình.
- Hỗ trợ các Ontology và các ngôn ngữ nội dung. Việc kiểm tra ontology và việc mã hóa nội dung được thực hiện tự động bởi nền tảng, các nhà lập trình có thể lựa chọn các ngôn ngữ nội dung và ontologies yêu thích. Những người lập trình còn có thể cài đặt những ngôn ngữ mới để thực hiện các yêu cầu ứng dụng cụ thể.
- Một thư viện của các giao thức tương tác: cái mô hình các kiểu mẫu đặc trưng của truyền thông nhằm đạt được một hoặc nhiều mục đích. Các skeleton độc lập với ứng dụng là một tập các lớp Java có sẵn và có thể được tùy. Các giao thức tương tác cũng có thể được thể hiện và được cài đặt như một tập các máy trạng thái đồng thời.
- Sự tích hợp với các công nghệ khác nhau dựa trên Web bao gồm các công nghệ JSP, Servlet, applet và Web Service. Nền tảng cũng có thể được cấu hình một cách dễ dàng để xuyên qua tường lửa và sử dụng các hệ thống NAT.
- Hỗ trợ nền tảng J2ME và môi trường không dây. JADE run-time có thể dùng cho các nền tảng J2ME – CDC và J2ME-LCDC thông qua một tập không đổi của các API che phủ cả 2 môi trường J2ME và J2SE.
- Một giao diện tiến trình bên trong (in-process) cho việc khởi chạy và điều khiển platform và các thành phần phân tán của nó từ một ứng dụng bên ngoài.

- Một nhân có thể mở rộng được thiết kế để cho phép những người lập trình mở rộng các chức năng của nền tảng thông qua việc bổ sung các dịch vụ phân tán mức nhân.

Như vậy, có thể nói JADE có rất nhiều điểm mạnh trong việc xây dựng các ứng dụng đa agent phân tán dựa trên các lựa chọn thiết kế đơn giản của sự trừu tượng hóa của agent. Phần tiếp theo, chúng ta sẽ tìm hiểu phương pháp luận MaSE để thấy được sự kết hợp của nó với JADE.

7.3 PHƯƠNG PHÁP LUẬN MaSE

7.3.1 Tổng quan về các pha trong MaSE

Những khó khăn trong việc xây dựng các hệ thống đa agent phức tạp:

- Xác định nhiệm vụ mà hệ thống phải thực hiện.
- Xác định thực thể nào trong hệ thống đóng vai trò là agent.
- Định nghĩa các tương tác giữa các agent.
- Đặc tả các giao thức phức tạp và đầy đủ
- Định nghĩa tương tác giữa hệ thống với môi trường
- Định nghĩa những hành vi liên quan đến các agent

Các phương pháp luận có nhiệm vụ giúp cho người thiết kế giải quyết các vấn đề này và phải quản lý được sự phức tạp này, ngoài ra còn phải làm dễ dàng tiến trình kỹ nghệ phần mềm bằng cách cung cấp một tiến trình nghiêm ngặt cho phép tạo ra các mô hình miêu tả các khía cạnh khác nhau của phần mềm. Federico Bergenti et al. các tác giả cuốn sách “Methodologies and software engineering for agent systems – The agent-oriented software engineering handbook” đã đưa ra một tổng kết về các giai đoạn phát triển chung cho các phần mềm hướng agent gồm:

- Phân tích
 - Miêu tả vấn đề (thu thập yêu cầu)
 - Xem xét role của các agent trong việc phân tích goal
 - Ngôn ngữ đặc tả sử dụng các khái niệm của agent
 - Công cụ hỗ trợ cho các ngôn ngữ
- Thiết kế
 - Kiến trúc phần mềm
 - Các thành phần
 - Các hành vi mong đợi
 - Cách xây dựng agent từ đầu và sử dụng môi trường phát triển agent
- Cài đặt
 - Đề xuất một số ngôn ngữ hướng agent, các thư viện, framework và các công cụ hỗ trợ.
- Test
 - Kiểm tra sự thỏa mãn yêu cầu ban đầu
 - Phần mềm có lỗi hay không?

7.3.2 Phân tích và thiết kế với MaSE

Phương pháp luận này tập trung vào các giai đoạn phân tích và thiết kế - hai trong số bốn giai đoạn phát triển phần mềm hướng agent chung.

Giai đoạn	Mô hình
1. Phân tích	
a. Xác định các Goal	Goal Hierarchy
b. Áp dụng các Use case	Use cases, Sequence Diagrams
c. Làm mịn các Role	Concurrent Tasks, Role Model
2. Thiết kế	
a. Tạo các lớp Agent	Agent Class Diagrams
b. Xây dựng các phiên giao tiếp	Conversation Diagrams
c. Lắp ráp các lớp Agent	Agent Architecture Diagrams
d. Thiết kế hệ thống	Deployment Diagrams

Tuy các bước được biểu diễn một cách tuần tự, nhưng trên thực tế, phương pháp luận này là một phương pháp lặp. Mục đích của nó là giúp người thiết kế có thể tự do di chuyển giữa các bước và các pha sao cho với mỗi lần di chuyển thành công thì có thêm các chi tiết được bổ sung vào và cuối cùng, một thiết kế hệ thống hoàn chỉnh và thống nhất được tạo ra.

Một điểm mạnh của MaSE là khả năng lưu vết những thay đổi trong suốt toàn bộ tiến trình. Mỗi đối tượng được tạo ra trong các pha phân tích và thiết kế có thể được lưu vết tiến hoặc lùi qua nhiều bước khác nhau với các đối tượng có liên quan khác.

7.3.2.1 Phân tích

Pha phân tích của MaSE tạo ra một tập các role và task miêu tả cách mà hệ thống thỏa mãn các mục đích (goal) của nó. Mục đích là một sự trừu tượng của yêu cầu và đạt được bởi các role. Thông thường, một hệ thống có một mục đích toàn cục và một tập các mục đích con. Goal được sử dụng trong MaSE vì chúng sao chép lại điều mà hệ thống đang cố gắng đạt được và có tính ổn định qua thời gian hơn là chức năng (function), tiến trình (process) hoặc các cấu trúc thông tin. Role miêu tả một thực thể thực hiện một số chức năng trong hệ thống. Trong MaSE, mỗi role có trách nhiệm đạt được hoặc giúp đỡ để đạt được goal.

Cách tiếp cận pha phân tích của MaSE là định nghĩa các goal của hệ thống từ tập các yêu cầu và sau đó định nghĩa các role cần thiết để đạt được những goal đó. Để định nghĩa role, MaSE sử dụng biểu đồ Use case và biểu đồ Sequence Diagram. Sau đây là chi tiết các bước trong pha phân tích.

Xác định các Goal

Bước này có nhiệm vụ chuyển đổi đặc tả hệ thống thành tập các goal hệ thống một cách có cấu trúc. Ngữ cảnh ban đầu của hệ thống (initial system context) là điểm bắt đầu cho pha phân tích. Đây chính là đặc tả yêu cầu của phần mềm với một tập các yêu cầu được định nghĩa rõ ràng. Hai bước con trong xác định goal là: định nghĩa goal và mô hình hóa goal.

Định nghĩa goal

Mục đích của bước này là sao chụp lại bản chất của tập yêu cầu ban đầu. Quá trình này bắt đầu bằng việc trích rút các kịch bản (scenario) từ đặc tả ban đầu và miêu tả goal của scenario đó. Goal là hiện thân của các yêu cầu then chốt của hệ thống.

Sau khi đã rút ra được các goal, thì tất cả các role và task được định nghĩa ở các bước sau phải hỗ trợ một goal. Nếu có role hoặc task nào không hỗ trợ một goal đang tồn tại, thì hoặc là role và task đó bị thửa hoặc là một goal mới được tìm ra.

Mô hình hóa goal

Bước tiếp theo là phải cấu trúc các goal đó thành biểu đồ **Goal Hierarchy Diagram**. Đây là một biểu đồ có hướng không có chu trình, trong đó các node biểu diễn các goal. Để phát triển biểu đồ phân cấp goal, người phân tích phải nghiên cứu các goal đó về tầm quan trọng và mối liên quan giữa chúng. Mỗi goal có một tầm quan trọng, một mức độ chi tiết khác nhau.

Đầu tiên, ta phải xác định goal toàn cục của hệ thống và đặt ở gốc của biểu đồ. Sau đó, chúng ta phân rã goal đó thành các goal con. Mỗi goal con phải hỗ trợ goal cha và định nghĩa “cái gì” cần phải làm để hoàn thành goal cha. Việc phân rã goal này không giống với phân rã chức năng (function). Goal miêu tả “cái gì” (what), còn function miêu tả “làm thế nào” (how). Việc phân rã goal kết thúc khi nếu phân rã tiếp ta sẽ thu được function thay vì goal.

Áp dụng các Use case

Bước này là một bước then chốt để chuyển các goal thành các role và các task tương ứng với mỗi role. Các use case được vẽ ra từ yêu cầu hệ thống và miêu tả các chuỗi các sự kiện mà những sự kiện đó định nghĩa hành vi được mong của hệ thống, chúng là những ví dụ về cách mà hệ thống sẽ cư xử. Để giúp xác định các giao tiếp thực sự trong một hệ đa agent, các use case được chuyển thành các biểu đồ Sequence Diagram. Sequence Diagram trong MaSE cũng giống với **Sequence Diagram** trong UML ngoại trừ việc chúng được sử dụng để minh họa chuỗi các sự kiện giữa các role và để định nghĩa các giao tiếp giữa các agent sẽ đóng vai trò làm các role đó. Các role được định nghĩa ở đây sẽ hình thành nên tập các role được sử dụng trong bước tiếp theo, còn các sự kiện cũng được sử dụng để định nghĩa các task và các phiên hội thoại (conversation).

Đầu tiên, chúng ta phải trích rút các **Use Case** từ ngữ cảnh ban đầu của hệ thống. Positive use case là use case miêu tả những gì xảy ra trong quá trình vận hành hệ thống bình thường. Còn negative use case thì định nghĩa lỗi hoặc thất bại.

Làm mịn các Role

Mục đích của bước này là để chuyển biểu đồ Goal Hierarchy Diagram và biểu đồ Sequence Diagram thành các role và các task tương ứng với các role đó. Các role hình thành nền cơ sở cho các lớp agent và tương ứng với các goal hệ thống trong pha thiết kế.

Thông thường việc chuyển đổi từ goal sang role là 1 – 1, mỗi goal được ánh xạ với một role. Tuy nhiên, có những tình huống mà ở đó một role có thể có trách nhiệm với nhiều goal. Nhìn chung, giao diện với tài nguyên bên trong hoặc tài nguyên bên ngoài đòi hỏi phải có một role riêng đóng vai trò giao tiếp với hệ thống. Chúng ta thường coi con người là tài nguyên bên

ngoài. Trong MaSE, chúng ta không mô hình một cách chính xác tương tác người máy, chúng ta sẽ tạo ra một role đặc biệt để đóng với giao diện người dùng. Bằng cách này, chúng ta có thể định nghĩa các cách mà ở đó một người dùng có thể giao tiếp với hệ thống mà không cần định nghĩa chính giao diện người dùng đó. Các tài nguyên khác như cơ sở dữ liệu, các tệp tin hoặc các hệ thống đã có từ trước cũng có thể cần một role riêng của nó. Định nghĩa các role được chụp lại trong mô hình **Role Model**, ở đó có chứa thông tin về các tương tác giữa các task của role và phức tạp hơn các mô hình role truyền thống như của Kendall.

Sau khi tạo các role và xác định các task, người phát triển phải chụp lại hành vi của role bằng cách định nghĩa chi tiết từng task một. Một role có thể gồm nhiều task. Mỗi task thực thi trong luồng điều khiển riêng nhưng có thể giao tiếp với nhau. Các task đồng thời sẽ được định nghĩa trong mô hình **Concurrent Task Models** và được đặc tả như là máy hữu hạn trạng thái.

7.3.2.2 Thiết kế

Pha này gồm 4 bước nhỏ sau:

- Tạo các lớp agent: gán role cho các kiểu agent cụ thể
- Xây dựng các phiên hội thoại: định nghĩa các phiên hội thoại giữa các agent
- Lắp ráp các lớp agent: thiết kế kiến trúc bên trong và các quá trình suy luận của các lớp agent
- Thiết kế hệ thống: định nghĩa số lượng và vị trí các agent trong hệ thống được triển khai

Tạo các lớp Agent

Các lớp agent được tạo từ các role được định nghĩa trong pha phân tích. Tại đây, chúng ta xây dựng biểu đồ **Agent Class Diagram** để minh họa tổ chức của toàn bộ hệ thống agent gồm các lớp agent và các phiên hội thoại giữa chúng.

Đầu tiên, chúng ta gán các role cho mỗi lớp agent. Nếu gán nhiều role thì lớp agent có thể thực hiện chúng đồng thời hoặc tuân tự. Mỗi role phải được gán cho ít nhất cho một lớp agent. Tại đây, chúng ta cũng xác định các phiên hội thoại mà các agent khác nhau tham gia vào. Ví dụ, nếu agent 1 đóng vai trò role A và agent 2 đóng vai trò role B thì phải có một phiên giao tiếp giữa agent 1 và agent 2.

Biểu đồ Agent Class Diagram cũng giống với biểu đồ lớp trong phân tích thiết kế hướng đối tượng nhưng có hai khác biệt chính là lớp agent được định nghĩa bởi các role mà nó đóng chứ không phải bởi thuộc tính và phương thức, và các quan hệ giữa các lớp agent được chụp lại dưới dạng các phiên hội thoại.

Xây dựng các phiên hội thoại

Mục đích của bước này là xác định chi tiết của các phiên hội thoại dựa trên chi tiết bên trong của các task đồng thời. Một phiên hội thoại xác định một giao thức giữa hai agent và được mô hình hóa bằng 2 biểu đồ **Communication Class Diagram**, một cho bên khởi tạo và một cho bên đáp ứng.

Bên khởi tạo bắt đầu phiên hội thoại bằng cách gửi đi thông điệp đầu tiên. Khi agent đáp ứng nhận được thông điệp, nó so sánh với các phiên hội thoại đang hoạt động của nó. Nếu phù

hợp, nó sẽ dịch chuyển phiên hội thoại thích hợp sang một trạng thái mới và thực hiện các hành động hoặc các hoạt động được yêu cầu bởi một chuyển dịch hoặc một trạng thái mới. Nếu không, nó giả sử thông điệp này là một yêu cầu mới và so sánh nó với các phiên hội thoại mà nó có thể tham gia vào cùng với agent đang gửi thông điệp. Nếu thấy phù hợp, nó sẽ bắt đầu phiên hội thoại mới. Một khi thông tin từ biểu đồ Concurrent Task Model được tích hợp vào các phiên hội thoại, người thiết kế phải đảm bảo các yếu tố khác như tính bền vững và khả năng chịu lỗi.

Lắp ráp các lớp agent

Chi tiết các lớp agent được thiết kế ở bước này. Bước này gồm hai bước nhỏ sau: định nghĩa kiến trúc của agent và định nghĩa các thành phần kiến trúc. Người thiết kế có thể lựa chọn thiết kế kiến trúc và các thành phần kiến trúc từ đầu hoặc từ những cái đã có sẵn. Các thành phần này gồm thuộc tính, phương thức, và có thể là kiến trúc con. Ở đây, chúng ta xây dựng biểu đồ Agent Architecture Diagram.

Thiết kế hệ thống

Đây là bước cuối cùng của phương pháp luận MaSE và sử dụng biểu đồ **Deployment Diagrams** để chỉ ra số lượng, kiểu và vị trí của các agent trong hệ thống. Đây là bước đơn giản nhất vì phần lớn công việc được thực hiện ở các bước trước. Người thiết kế nên định nghĩa việc triển khai hệ thống trước khi cài đặt vì các agent thường yêu cầu thông tin của biểu đồ Deployment Diagram, như hostnemt hoặc địa chỉ để giao tiếp. Biểu đồ Deployment Diagram cũng mang lại cơ hội cho người thiết kế có thể hòa nhập hệ thống với môi trường để tối đa hóa sức mạnh xử lý và độ rộng băng thông sẵn có.

7.4 HỆ THỐNG QUẢN LÝ HỘI THẢO

7.4.1 Miêu tả hệ thống

- Các tác giả có thể gửi các bài báo tới hệ thống cơ sở dữ liệu các bài báo hội thảo. Trong quá trình gửi, tác giả phải được thông báo lại là bài báo đã được nhận hay chưa và nhận được số hiệu của bài báo.
- Sau một thời gian quy định, bài báo sẽ được phân phối tới các thành viên ban chương trình – những người có trách nhiệm đọc các bài báo, hoặc liên hệ với các phản biện và đề nghị họ nhận đọc nhận xét một số bài báo.
- Người có trách nhiệm nhận xét các bài báo có thể lấy bài báo trực tiếp từ cơ sở dữ liệu và gửi bài nhận xét của họ tới điểm thu thập trung tâm.
- Sau khi hoàn tất việc nhận xét, một quyết định được đưa ra là chấp nhận hay loại bỏ bài báo.
- Sau khi ra quyết định, tác giả được thông báo về quyết định và được yêu cầu đưa ra phiên bản cuối cùng của bài báo nếu nó được chấp nhận.

Những người tham gia vào hệ thống này gồm có: tác giả bài báo (author), người nhận xét (reviewer), người ra quyết định (decision maker), người thu thập các nhận xét (review collector)....

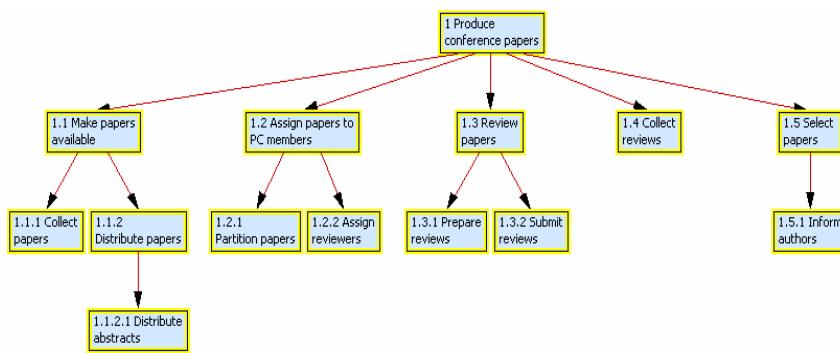
7.4.2 Phân tích

7.4.2.1 Xác định các Goal

Ví dụ về các goal được rút ra từ đặc tả yêu cầu:

- Thu thập bài báo
- Phân phối bài báo
- Gán bài báo cho thành viên PC
- Gán bài báo cho phản biện
- Gửi bài nhận xét
- Thu thập bài nhận xét
- Chấp nhận/hủy bỏ bài báo
- Thông báo cho tác giả

Biểu đồ phân cấp goal (Goal Hierarchy Diagram)



Hình 7.1: Biểu đồ phân cấp goal

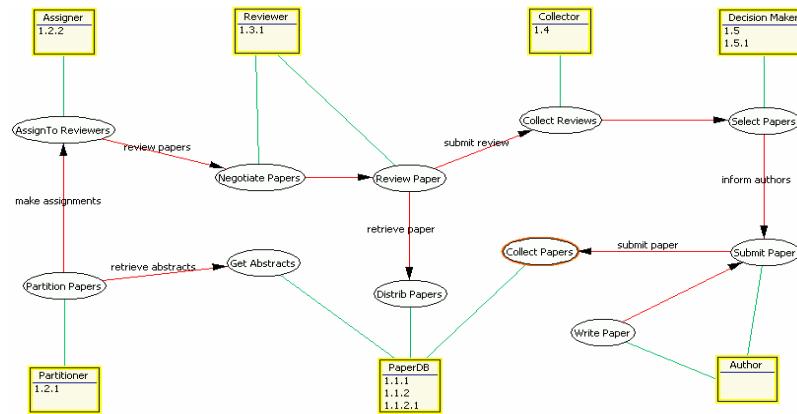
7.4.2.2 Làm mịn các Role

Tập các role tương ứng với các goal ở trên:

- PaperDB (1.1.1, 1.1.2, 1.1.2.1)
- Partitioner (1.2.1)
- Assigner (1.2.2)
- Reviewer (1.3.1)
- Collector (1.4)
- DecisionMaker (1.5, 1.5.1)

Ở đây, các goal 1, 1.1, 1.2 và 1.3 không được ánh xạ thành các role vì chúng là các goal có thể chia nhỏ được. PaperDB role được gán cho tất cả các goal tương ứng với goal 1.1, đó là 1.1.1, 1.1.2 và 1.1.2.1. DecisionMaker role được gán cho cả 1.5 và 1.5.1. Các goal có liên quan thường được kết hợp với một role. Ví dụ, “thu thập bài báo”, “phân phối bài báo” và “phân phối tóm tắt bài báo” được kết hợp với PaperDB role vì chúng có liên quan rất gần với nhau và đòi hỏi cùng một kiểu kỹ thuật truy cập.

Role Model



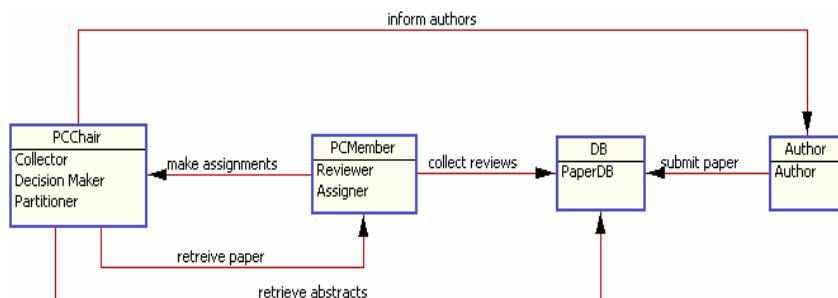
Hình 7.2: Biểu đồ Role Model

PaperDB role chịu trách nhiệm đạt được goal 1.1.1, 1.1.2 và 1.1.2.1. Do đó, để hoàn thành goal này, PaperDB role phải có khả năng thu thập các bài báo, phân phối bài báo và phân phối tóm tắt bài báo. Do đó, chúng ta tạo ra ba task có liên quan lẫn nhau: Collect Papers, Distrib Papers và GetAbstracts. Các role không nên chia sẻ hoặc trùng lặp với các task. Các task được chia sẻ nên được thay thế bằng một role riêng. Role này có thể được kết hợp thành các lớp agent trong pha thiết kế.

7.4.3 Thiết kế

7.4.3.1 Tạo các lớp agent

Agent Class Diagram



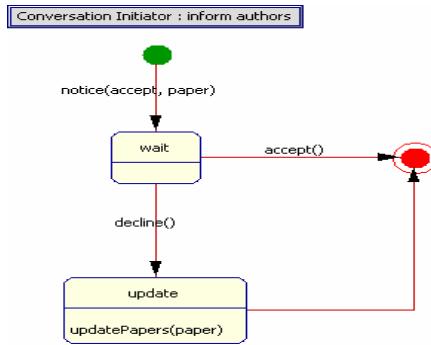
Hình 7.3: Biểu đồ Agent Class Diagram

Biểu đồ lớp agent gồm tên lớp và tập các role mà agent đó đóng. Các mũi tên xác định các phiên hội thoại và có hướng từ agent khởi tạo đến agent đáp ứng. Ở đây, PCChair đóng các role: Collector, Partitioner, Decision Maker, còn PCMember đóng các vai Assigner và Reviewer. Ngoài Author còn có một agent khác là DB cung cấp giao diện với cơ sở dữ liệu chứa các bài báo, tóm tắt bài báo và thông tin về tác giả.

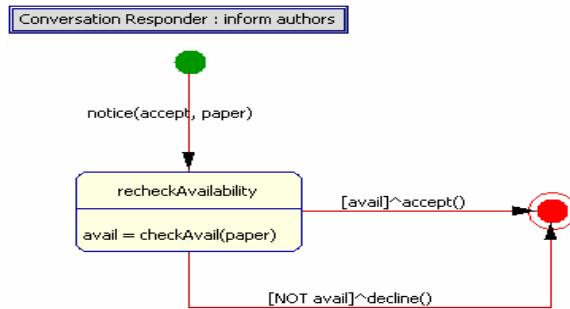
7.4.3.2 Xây dựng các phiên hội thoại

Communication Class Diagram

Inform author: Bên khởi tạo là PCChair, bên đáp ứng là Author.



Hình 7.4: Biểu Communication Class Diagram cho PCChair



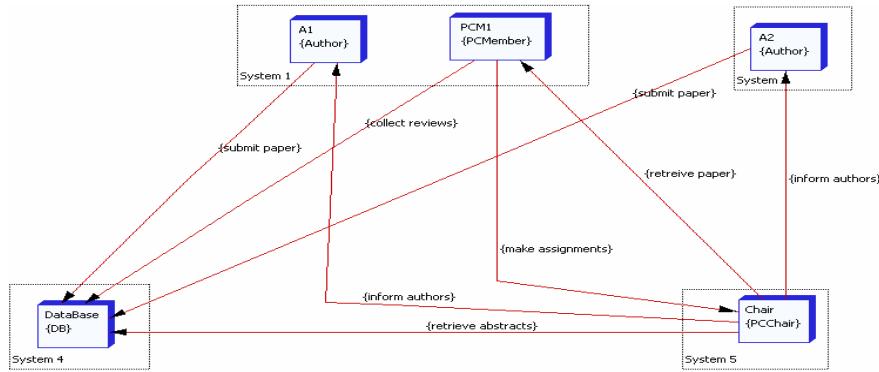
Hình 7.5: Biểu Communication Class Diagram cho Author

PCChair agent gửi một thông điệp đến Author agent để thông báo về việc chấp nhận bài báo. Sau đó, PCChair chuyển sang trạng thái chờ. Nếu Author vẫn tham gia vào buổi hội thảo, nó gửi một thông điệp chấp nhận và hoàn tất phiên hội thoại. Nếu Author không thể tham gia hội thảo, nó trả về thông điệp từ chối. Sau khi nhận được thông điệp từ chối, PCChair thực hiện hoạt động updatePapers để cập nhật danh sách những người tham dự hội thảo.

7.4.3.3 Lắp ráp các lớp agent

Agent Architecture Diagram

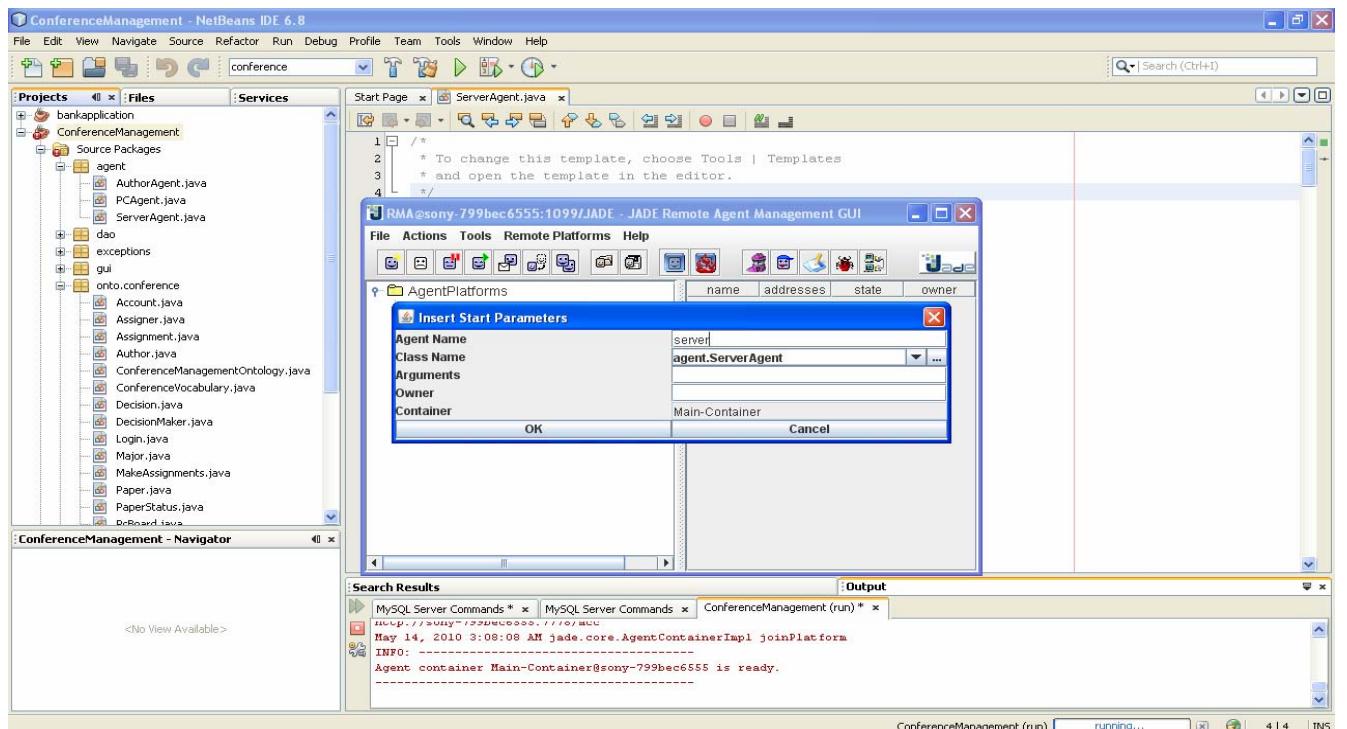
PCChair agent có 3 thành phần và được cài đặt theo kiến trúc đường ống. Thành phần Partitioner nhận các phần tóm tắt bài báo và sử dụng phương thức partitionPapers để chia danh sách các bài tóm tắt thành các tập con dựa trên nội dung. Sau đó, Partitioner gọi phương thức collectReviews của thành phần Collector, thành phần này có nhiệm vụ chờ và thu thập tất cả các bài nhận xét. Sau khi tất cả các bài báo được nhận xét, thành phần Collector gọi phương thức selectPapers của thành phần DecisionMaker, thành phần này có nhiệm vụ lựa chọn các bài báo tốt nhất và thông báo cho tác giả.



Hình 7.6: Biểu đồ Agent Architecture Diagram

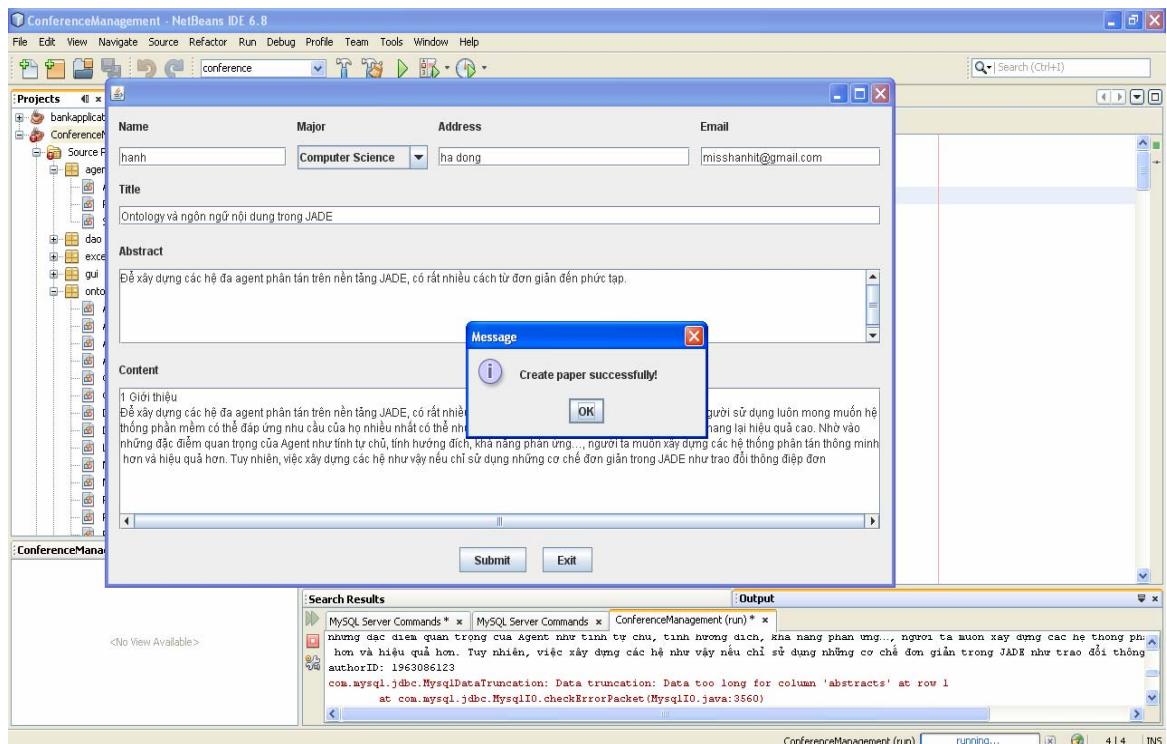
7.5 CÀI ĐẶT HỆ THỐNG

- Chạy DB agent:



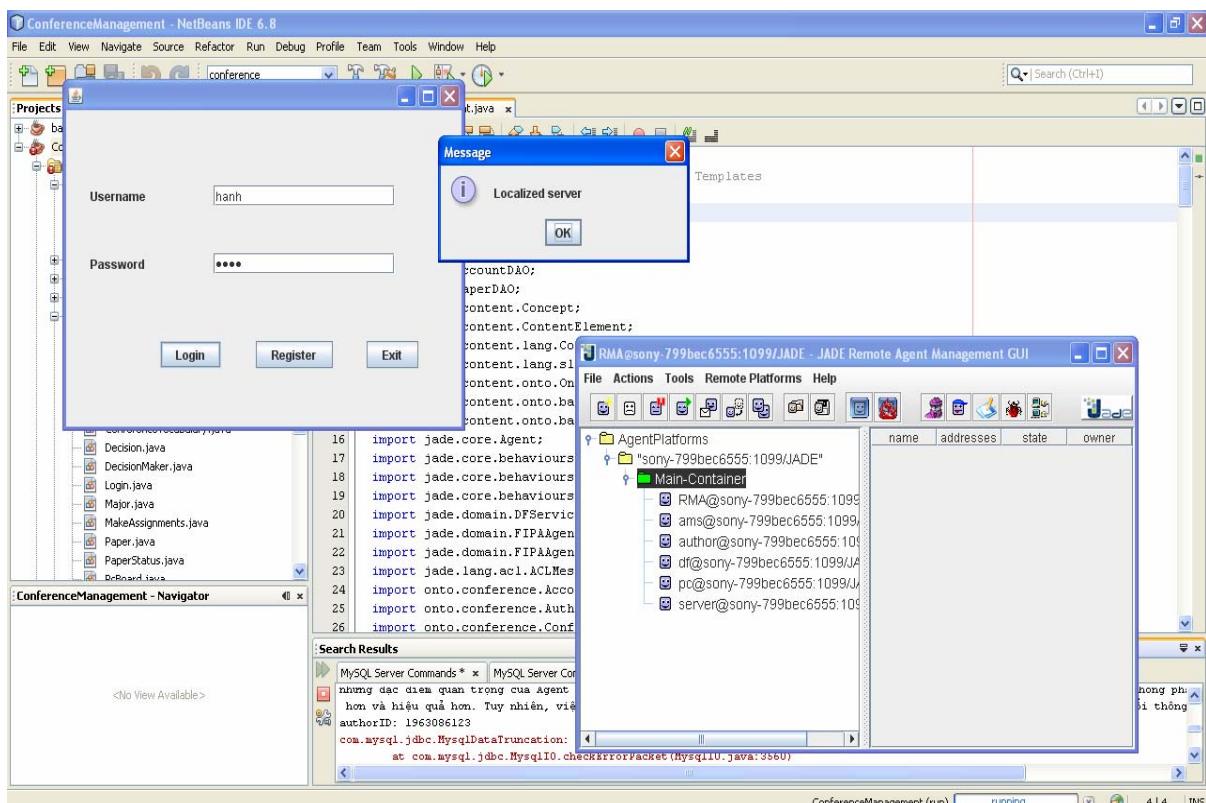
Hình 7.7: Giao diện DBAgent

- Chạy Author agent:



Hình 7.8: Giao diện Author agent

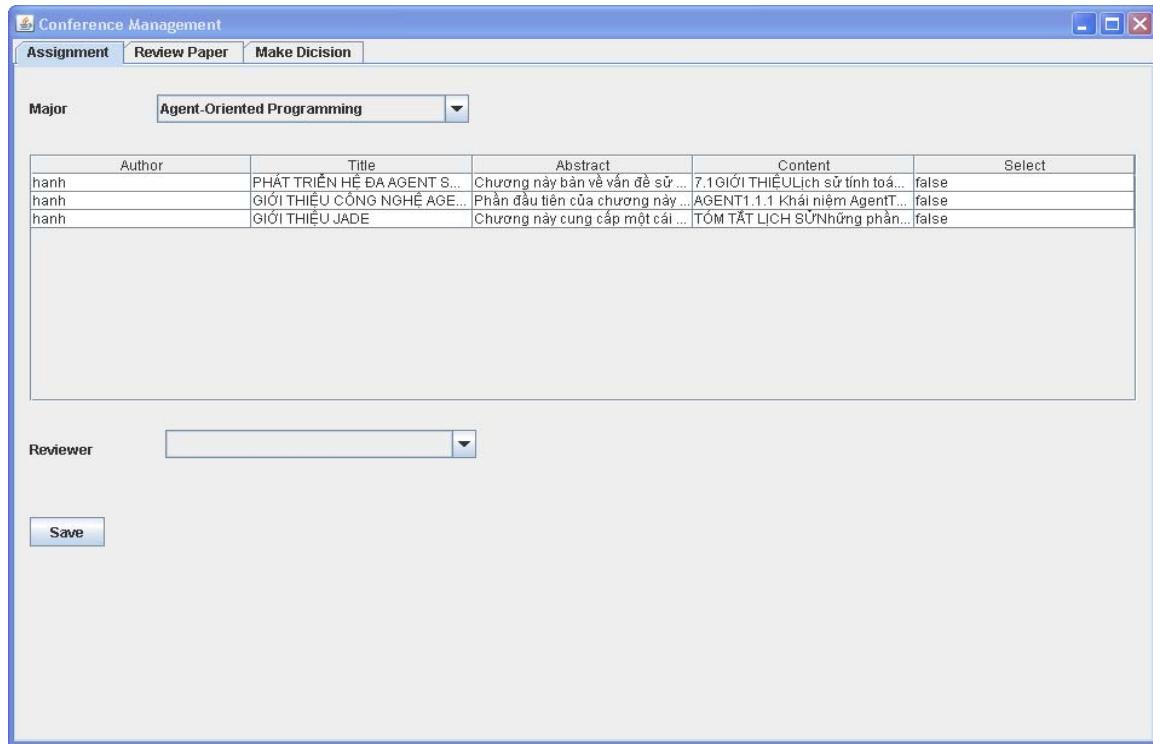
- Chạy PCMember Agent: đầu tiên phải đăng nhập (nếu chưa có tài khoản, thì tạo mới tài khoản bằng cách click nút Register)



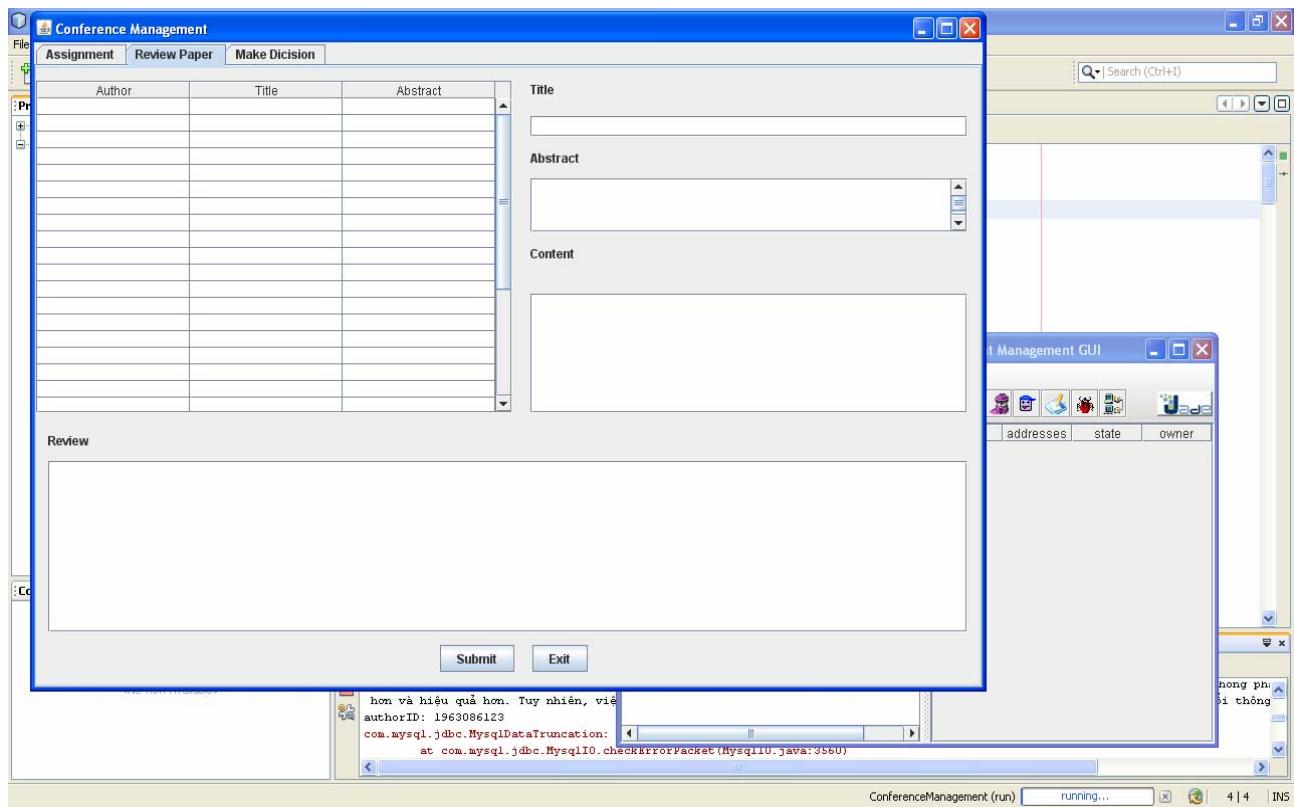
Hình 7.9: Giao diện đăng nhập của PCMember agent

Sau khi đăng nhập thành công có thể chọn các chức năng:

- Assignment: gán bài báo cho người nhận xét
- Review Paper: viết nhận xét cho bài báo
- Make Decision: quyết định chấp nhận hay từ chối bài báo



Hình 7.10: Giao diện của chức năng gán bài báo để được nhận xét



Hình 7.11: Giao diện của chức năng nhận xét bài báo

TÀI LIỆU THAM KHẢO

- [1] Fabio Bellifemine et al. *Developing Multi-Agent Systems with JADE*, John Wiley Sons, 2007
- [2] W. Branner et al. *Intelligent Software Agents: Foundations and Application*, Springer, 1998.
- [3] S. A. DeLoach, *Analysis and Design using MaSE and agentTool* 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001), Ohio, March 31-April 1, 2001
- [4] M. Wooldridge, *An Introduction to Multiagent Systems*, Published in John Wiley & Sons, 2009.
- [5] Pedro Cuesta-Morales et al., *Developing Multi-Agent System Using MaSE and JADE*, 2004