to the spirits of my parents,
who would have loved to be present
at this moment of my life.

Abstract

The global purpose of this thesis is to investigate Software Agent technology in the Network Management (NM) field. The thesis starts by surveying software agent litterature, thus providing a synthetic view of agent trends and techniques. The survey shows the richness of software agents in terms of potential applications, techniques, mechanisms and languages.

In order to determine the best way software agents are applied to NM, the thesis presents a state-of-the-art of agent-based NM applications so far developed. These applications are analyzed according to the different kinds of agent architectures and approaches. This allowed to independently assess each agent technique when applied to NM purposes. The result of the analysis shows that different kinds of software agent approaches are suitable for tackling different NM issues, and that in order for NM to get the maximum benefit of software agents, an agent approach should not be restricted to a unique kind of agents, or a limited set of agent techniques.

According to this outcome, the thesis describes a generic skill-based agent architecture targeted to NM applications. This agent architecture is derived from horizontally-layered agent architectures, and therefore inherits the same properties of modularity and equitability between agent layers. Horizontal layers are represented by skill modules that encapsulate agent capabilities and management functionaliy. Skills can be dynamically plugged into the agent without interrupting its execution. The agent brain, which encodes the basic functionality required for the agent operation, is capable of discovering the capabilities brought by loaded skills and dynamically integrating these capabilities to improve the agent behavior with new competences. The skill-based architecture defines a transparent inter-agent communication mechanism that allows the easy implementation of different agent coordination patterns. In addition to these features, the skill-based agent architecture is designed with the properties of flexibility and support for dynamism of NM functionality. The ensemble of these features is validated in the context of two case studies.

The first case study considered the implementation of hierarchically distributed NM

system in which, management domains could be dynamically reaffected according to changes in the network topology, or to the availability of domain management agents. In addition, a reliability layer is added to insure that the management tasks are reliably performed on the whole network even in the case that a subset of the domain agents become unreliable. The obtained agent system exhibits the properties of dynamism, reliability, fault tolerance, and graceful degradation.

The second case study considered the mundane task of the automatic provision of Permanent Virtual Connection services in heterogeneous ATM networks. The skill-based agent architecture is used to implement distributed agents that collaborate together to achieve end-to-end connectivity services in an efficient and cost-effective way.

In order to assess the skill-based architecture with possible other agent approaches, the two case studies are reconsidered using two of the most hyped types of agents, namely: BDI (Belief-Desire-Intention) agents, and mobile agents (MA). For BDI agents, the thesis proposes an abstract BDI agent model specially developed for NM applications. A BDI-oriented design process is also proposed and used in order to provide a different implementation of the first case study. The comparison with the skill-based agent implementation allows to show that the proposed BDI model presents powerful abstractions. The skill-based architecture provides howerver a higher degree of flexibility and ease of design.

For MAs, the thesis compares an MA approach to the second case study with the skill-based approach. The comparison considers different qualitative and performance parameters. The results of the comparison allows to pinpoint the actual potential benefits of MAs: Simplifying the design of self-contained distributed management tasks and providing a high degree of flexibility and ease of deployment. Skill-based agents still provide better performance, a sufficient degree of flexibility, in addition to being suitable for general-purpose management applications and insensitive to the complexity of management functions.

Therefore, the adopted skill-based architecture provides the best compromise to deal with the requirements of modern NM systems and approaches.

The thesis concludes with the identification of major pitfalls in the application of software agents in NM, namely the disillusion of obtaining highly intelligent management software with smart properties of learning, proactiveness and self-adaptability. The real benefit of software agents in NM relies in the ability to adapt the large set of agent techniques to tackle NM issues.

Résumé

Cette thèse traite l'application des agents logiciels dans le domaine de gestion des réseaux. Elle débute avec un résumé de la littérature sur les agents qui présente leurs différents interprétations, orientations, techniques, mécanismes et approches. Cette étude permet de découvrir la richesse des agents logiciels et leur potentiel dans la gestion des réseaux.

Afin de sélectionner la meilleur approche agent pour développer des applications de gestion de réseau, la thèse présente une étude de l'état de l'art sur les applications de gestion de réseaux basées sur les agents logiciels. Cet état de l'art permet de montrer que seule une approche d'agent générique permet d'avoir une architecture agent commune qui est à meme de supporter les différentes techniques agent pour différents domaines de la gestion des réseaux.

Sur cette base, la thèse présente une architecture agent basée sur les modules de compétence appelés *skills*. Cette architecture est développée avec l'objectif d'avoir un grand degré de flexibilité et de dynamisme, caractéristiques nécessaires pour les systèmes de gestion de réseaux modernes.

Afin de valider ces propriétés, la thèse présente deux études de cas réalisés avec un prototype de l'architecture proposée. La première étude de cas a un double objectif. Le premier objectif est de réaliser un prototype a base d'agents d'un système de gestion de réseaux distribué hiérarchiquement. La particularité de ce système est que les domaines peuvent être ré-affectés dynamiquement selon la configuration du réseau et du système d'agents. Le second objectif est d'introduire un niveau de fiabilité dans ce système. Le résultat est un système de gestion distribué qui est fiable, tolérant aux fautes, et à dégradation non drastique.

La deuxième étude de cas vise l'automatisation de la configuration des PVCs *(Permanent Virtual Connections/Circuits)* dans les réseaux ATM hétérogènes. L'utilisation de l'architecture proposée permet de concevoir un système hautement distribué à base d'agents cooperatifs, et de fournir une solution aux problèmes d'hétérogeneité.

La thèse confronte l'architecture à base de skills avec deux autres approches agent :

les agents BDI *(Belief-Desire-Intention)* et les agents mobiles. Pour les agents BDI, un modèle abstrait d'agent BDI est défini accompagné d'un processus de développement adequat. Le modèle BDI et le processus de développement proposés sont adaptés pour une utilisation orientée gestion de réseau et sont considérés pour une réimplémentation de la première étude de cas. La comparaison avec l'architecture à base de skills permet d'identifier les avantages et inconvénients d'une approche BDI dans la gestion de réseaux.

La deuxième confrontation considère une approche à base d'agents mobiles pour la deuxième étude de cas. La comparaison avec l'approche à base de skills permet d'identifier les vrais apports potentiels des agents mobiles dans la gestion des réseaux, afin de simplifier le développement de certaines applications distribuées.

Ces deux confrontations permettent d'établir que l'approche générique à base de skills présente le meilleur compromis pour répondre aux besoins de la gestion de réseaux modernes.

La conclusion de la thèse construit sur l'ensemble de ces travaux une vision permettant d'établir le vrai apport des agents logiciel dans la gestion de réseaux, et d'identifier les pièges à éviter en adoptant cette technologie, certes pas encore mature, mais bien prometteuse.

Acknowledgements

I am grateful to Jacques Labetoulle who gave me the opportunity to accomplish this thesis and was constantly prompt to provide me with a good environment to carry out my research.

I am also thankful to the members of Jury, presided by Prof. Alain Wegmann and composed of Prof. Boi Faltings, Prof. Refik Molva, Dr. Olivier Festor and Dr. Dominique Sidou. Special thanks for Refik Molva for his support and advise.

The work achieved during my thesis was the fruit of a tight collaboration inside the Nework Management Team, and cooperating with Pierre Conti and Karina Marcus was both a strong motivator, and an endless source of inspiration. Thank you very much Pierre and Karina.

This thesis was started as part of a project financed by SwissCom. A number of people from SwissCom largely contributed in different parts of the case studies and in providing vision ideas. I would like to thank all these people, especially Rolf Eberhardt and Ernesto Ruggiano.

Eurécom provides an ideal environment to accomplish world-class research, and this is owed to the personal quality of its people. First of all, I am very thankful to Dr. Jamel Gafsi, whose continuous support, encouragement, and advise were decisive factors during all the phases of my thesis. Eurécom's Doctorants are also very cooperative and I thank all those people who contributed either by discussions, assisting to presentations, or simply by sharing their knowledge. Among those, I would like to sepcially thank Matthias Jung and Sergio Loureiro, my office mates, as well as Arnaud Legout, Arnd Kohrs, Jakob Hummes, Pablo Rodriguez, and many others — good luck for all of you guys. From Eurécom also, I would like to thank all those people in the staff who were constantly prompt to help us accomplishing administrative and other tasks, especially, Arianna Mazzoni, Rémy Giaccone, Jean-Mark Calbet, Didier Loisel, David Tremouilhac, and many others as well.

Last but not least, I would like to thank my two elder brothers, who were and still

constantly present to listen to my concerns, and to provide me with all kind of support.
Thank you very much Aref and Rami.

Author's Selected Publications

[CCL00a] Morsy Cheikhrouhou, Pierre Conti, and Jacques Labetoulle. Automatic configuration of PVCs in ATM networks. In *Proceedings of Network Operations and Management Symposium (NOMS'2000)*, Honolulu, Hawaii, 2000. IEEE/IFIP.

[CCL00b] Morsy Cheikhrouhou, Pierre Conti, and Jacques Labetoulle. *Logiciel et Réseaux de Communication*, chapter Agents Intelligents et Gestion de Réseaux, pages 167–197. Observatoire Français des Techniques Avancées, ARAGO 23. Editions TEC & DOC, Paris, May 2000.

[CCLM99] Morsy Cheikhrouhou, Pierre Conti, Jacques Labetoulle, and Karina Marcus. Intelligent agents for network management: Fault detection experiment. In Morris Sloman, Subrata Mazumdar, and Emil Lupu, editors, *Distributed Management for the Networked Millennium*, IFIP/IEEE International Symposium on Integrated Network Management, pages 595–610, Boston, MA, USA, May 1999. IEEE Publishing.

[CCM00] Morsy Cheikhrouhou, Pierre Conti, and Karina Marcus. A software agent architecture for network management: Case studies and experience-gained. *Journal of Network and Systems Management*, 8(3), September 2000.

[CCOL98] Morsy Cheikhrouhou, Pierre Conti, Raúl Texeira Oliveria, and Jacques Labetoulle. Intelligent agents in network management, a state of the art. *Networking and Information Systems*, 1(1):9–38, 1998. http://www.eurecom.fr/~cheikhro/docs/StateOfTheArt.ps.gz.

[Che99] Morsy M. Cheikhrouhou. BDI-oriented agents for network management. In *Proceedings of Globecom'99*, Rio de Janeiro, Brazil, December 1999. IEEE.

[CL00a] Morsy Cheikhrouhou and Jacques Labetoulle. An efficient polling layer for SNMP. In *Proceedings of Network Operations and Management Symposium (NOMS'2000)*, Honolulu, Hawaii, 2000. IEEE/IFIP.

[CL00b] Morsy Cheikhrouhou and Jacques Labetoulle. When management agents become autonomous, how to ensure their reliability? In *Proceedings of Network Operations and Management Symposium (NOMS'2000)*, Honolulu, Hawaii, 2000. IEEE/IFIP. (Poster).

[CL00c] Morsy Cheikhrouhou and Jacques Labetoulle. When management agents become autonomous, how to ensure their reliability? In *Proceedings of the First International Conference on Artificial and Computational Intelligence For Decision, Control and Automation In Engineering and Industrial Applications*, Monastir, Tunisia, March 2000.

Contents

Summary 3

Résumé 5

Acknowledgements 7

Author's Selected Publications 9

List of Figures 17

Acronyms 18

1 Introduction 21

1.1 Current Network Management Challenges . . . 22

1.1.1 Dynamism . . . 22

1.1.2 Evolving Management Requirements . . . 22

1.1.3 Integration and Interoperability . . . 24

1.1.4 Management Automation . . . 24

1.1.5 Proactive Management . . . 25

1.1.6 Other Challenges: Adaptable Scalability and Reliability . . . 25

1.2 Recent Network Management Trends . . . 26

1.3 Software Agents: Another Trend in Network Management . . . 28

1.4 Our Thesis . . . 30

1.5 Outline . . . 31

2 Software Agents 35

2.1 What an Agent Can Possibly Be? . . . 35

2.1.1 The Agent as Representing his User . . . 36

2.1.2 The Agent as a Metaphor . . . 36

2.1.3 The Agent as a Behavior . . . 37

2.1.4 The Agent as a Situated Entity 39
2.1.5 The Agent as a Goal-oriented Software 39
2.1.6 The Agent as a Mobile Entity 39
2.1.7 The Agent as a Modeling Entity 40
2.1.8 Summary 40
2.2 Agent Architectures 40
2.2.1 A Common Agent Architecture 41
2.2.2 Deliberative Agents 41
2.2.3 Reactive Agents 42
2.2.4 Hybrid Agents 43
2.2.5 Layered Architectures 44
2.2.6 Belief-Desire-Intention Architectures 44
2.2.7 Other Agent Architectures 47
2.3 Agent Mobility 47
2.3.1 Basic Components of a Mobile Agent Framework 48
2.3.2 Strong, Weak and Memoryless Mobility 49
2.3.3 Remote Cloning 50
2.3.4 Single-hop and Multiple-hop Agents 50
2.3.5 Fixed and Dynamic Itinerary 50
2.3.6 Passive and Active Migration 51
2.4 Agent Communication 51
2.4.1 General-purpose and Standardized Agent Communication Languages 53
2.4.2 Content Languages 53
2.4.3 Ontologies 54
2.4.4 Relevant Aspects in Inter-agent Communication 54
2.5 Agent Interaction 56
2.5.1 Cooperative vs. Self-interested Agents 56
2.5.2 Reactive vs. Deliberative Coordination 57
2.5.3 Global Problem Solving 58
2.5.4 Delegation-based Cooperation (Task Sharing) 58
2.5.5 Voting 59
2.5.6 Contract Net Protocol 60
2.5.7 Market Mechanisms 60
2.5.8 Coordination without Direct Communication 61
2.6 Heterogeneous Multi-Agent Systems 61
2.7 Summary and Conclusion 62

3 Software Agents in Network Management – A Survey 63
3.1 Preamble . . . 63
3.2 Management by Delegation . . . 64
3.2.1 Background . . . 64
3.2.2 Management by Delegation . . . 65
3.2.3 Synthesis and Discussion of Management by Delegation . . . 67
3.3 Network Management with Mobile Agents . . . 68
3.3.1 Generalities . . . 68
3.3.2 Mobile Agent Frameworks for Network Management . . . 69
3.3.3 Mobile Agent Applications in Network Management . . . 73
3.3.4 Performance of Mobile Agents in Network Management . . . 77
3.3.5 Synthesis and Discussion of Mobile Agents . . . 78
3.4 Reactive, Deliberative and Hybrid Agents in Network Management . . . 79
3.4.1 Reactive Agents . . . 79
3.4.2 Deliberative and BDI Agents . . . 81
3.4.3 Hybrid Agents . . . 82
3.4.4 Synthesis and Discussion of Agent Architectures . . . 84
3.5 Agent Communication Languages for Network Management . . . 85
3.6 Multiagent Cooperation for Network Management . . . 86
3.6.1 Stand-alone Agents . . . 87
3.6.2 Coordination Without Direct Communication . . . 87
3.6.3 A Primitive Cooperation . . . 89
3.6.4 Self-interested and Negotiation Agents . . . 89
3.6.5 Market-Based Cooperation . . . 92
3.6.6 Delegation-based Cooperation . . . 93
3.6.7 Synthesis and Discussion of Agent Coordination . . . 94
3.7 Conclusion . . . 95

4 A Skill-based Agent Architecture for Network Management 97
4.1 Introduction . . . 97
4.2 Design Principles . . . 98
4.2.1 Flexibility . . . 98
4.2.2 Genericity . . . 98
4.2.3 Dynamism . . . 99
4.2.4 Architectural Principles . . . 100
4.3 Capability Skill Modules . . . 102
4.3.1 Beliefs . . . 102

4.3.2 Capabilities 102
4.3.3 Skill Declaration 103
4.4 The Brain 104
4.4.1 Belief Management 104
4.4.2 Skill Management 105
4.4.3 The Inter-agent Communication 106
4.4.4 The Skill Interface 107
4.5 Agent Communication 107
4.6 A Development Process for Agent-based Applications 109
4.7 Skill Examples: Agent Support Services 110
4.7.1 Agent Multicast Communication Skill 111
4.7.2 Directory Service Skill 112
4.7.3 Self-control Skill 114
4.8 A Polling Layer for Belief Instrumentation 115
4.9 Summary and Conclusion 116

5 The First Case Study: When Management Agents Become Autonomous, How to Insure Their Reliability? 119
5.1 Rationale 119
5.2 System Level Diagnosis 121
5.2.1 The Algorithm 121
5.2.2 The Reliability Test 122
5.3 Macro-design: Scenario and Role Identification 123
5.4 Micro-design: Skill Identification and Design 125
5.4.1 Mapping from Roles to Skills 125
5.4.2 Skill Design 126
5.5 Demonstration and Results 132
5.6 Discussion and Conclusion 134

6 Provision of Permanent Virtual Circuits in ATM Networks 135
6.1 Background and Rationale 135
6.2 Macro-design: Scenario and Agent Roles 137
6.3 Micro-design: The Agent Skills 138
6.3.1 Skills for the User Agent Role 138
6.3.2 Skills for the Switch Agent Role 138
6.4 Implementation and Results 142
6.5 Discussion and Conclusion 143

7 A BDI Approach to the SLD Case Study **147**

7.1 Preamble . . . 147
7.2 The BDI Agent Model . . . 148
7.2.1 The Deliberative Layer . . . 148
7.2.2 The Operational Layer . . . 152
7.2.3 A BDI-oriented Design Process . . . 153
7.3 Abstract Agent Design . . . 154
7.3.1 The Domain Monitoring Role . . . 154
7.3.2 The Tester Role . . . 155
7.3.3 The Testee Role . . . 156
7.4 Agent-Oriented Programming with JACK . . . 157
7.4.1 JACK Concepts . . . 157
7.4.2 Advantages of JACK . . . 159
7.5 Implementation with JACK Agents . . . 160
7.5.1 General Considerations . . . 160
7.5.2 BDI-oriented Implementation . . . 161
7.6 Discussion . . . 163
7.6.1 Impact of a BDI-oriented agent development process . . . 164
7.6.2 Impact on the agent operation . . . 166
7.6.3 Implementation Issues . . . 168
7.7 Summary and Conclusion . . . 169

8 Assessment of Mobile Agents through the PVC Provision Case Study **171**

8.1 Motivation . . . 171
8.2 Mobile Agent and Static Agent Approaches to PVC Provisioning . . . 173
8.2.1 The Case Study . . . 173
8.2.2 The Mobile Agent Approach . . . 174
8.2.3 The Static Agent Approach . . . 175
8.3 Comparison . . . 176
8.3.1 Efficiency and Performance . . . 176
8.3.2 Robustness and Graceful Degradation . . . 182
8.3.3 Ease of Development . . . 184
8.3.4 Ease of Deployment . . . 186
8.3.5 Support for Heterogeneous Environments . . . 187
8.3.6 Flexibility . . . 188
8.4 Conclusion . . . 189

9 Conclusion **191**
9.1 Summary of Major Contributions 191
9.2 Experience-gained 193
9.3 Outlook 195

Bibliography **197**

A An Efficient Polling Layer for SNMP **215**
A.1 Introduction 215
A.2 Requirements for the Polling Layer 216
A.3 The Polling Layer 217
A.3.1 General View 218
A.3.2 The Usage Parameters 219
A.3.3 Architecture of the Polling Layer 221
A.3.4 Optimizations 222
A.4 Performance Results 224
A.5 Discussion 226
A.6 Conclusion 228

B Example of Skill Declaration **231**

List of Figures

1.1 Structure of the thesis 34

2.1 Basic agent architecture 42
2.2 Agent communication layers 52

4.1 Horizontal layering 101
4.2 Our skill-based agent architecture 104

5.1 Interaction pattern between a tester and a testee 124
5.2 Interaction pattern between the manager and the tester 125
5.3 Skill interaction diagram 126
5.4 Refinement of the interaction diagram between the tester and the testee . 130
5.5 Demonstration applet 133

6.1 User interface for PVC creation 138
6.2 The testbed ATM network 142
6.3 PVC creation scenario 143
6.4 General panel of the PVC demonstration application 146

7.1 Agent's Mental Cycle 148
7.2 Agent development processes 164

8.1 PVC creation with a Mobile Agent 175
8.2 PVC creation with Static Agents 176
8.3 Mobile Agent approach compared to Static Agent approach 186

A.1 The Architecture of the Polling Layer 218
A.2 Time interval in which the polling query should be sent 222
A.3 Polling optimization with freshness periods multiple of 10 seconds 225
A.4 Polling optimization with freshness periods multiple of 30 seconds 226

Acronyms

AI	Artificial Intelligence
ACL	Agent Communication Language
AOP	Agent-oriented Programming
ATP	Agent Transfer Protocol
CIM	Common Information Model
CL	Content Language
CNP	Contract Net Protocol
COD	Code On Demand
DAI	Distributed Artificial Intelligence
DEN	Directory-enabled Network
FCAPS	The five management functional areas: fault, configuration, accounting, performance and security
FIPA	Foundation for Intelligent Physical Agents
IA	Intelligent Agent
JMX	Java Management eXtentions
KQML	Knowledge Query Manipulation Language
MA	Mobile Agent
MAD	Manager Agent Delegation
MAM	Mobile Agent Manager
MAS	Multi-Agent System
MASIF	Mobile Agent System Interoperability Facility
MBD	Model-Based Diagnosis
MbD	Management by Delegation
MCS	Mobile Code Server
MLM	Middle-level Manager
NE	Network Element
NM	Network Management
NPA	Network Provider Agent
OMG	Object Management Group
OOP	Object-oriented Programming
PA	Personal Agent
PRS	Procedural Reasoning System
QoS	Quality of Service
	(Continued on next page)

Acronyms (Cont'd)

R&D	Research and Development
RA	Reactive Agent
REV	Remote Evaluation
RMI	Remote Method Invocation
SA	Stationary (or Static) agent
SLA	Service Level Agreement
SPA	Service Provider Agent
SWA	Switch Agent
UA	User Agent
VPC	Virtual Path Connection
VPN	Virtual Private Network
XML	eXtended Markup Language

Chapter 1

Introduction

The shape of data and telecommunications networks has been dramatically changing since deregulation was introduced a number of years ago. The number of network and service providers since then has been steadily increasing and the competition between different providers is becoming increasingly tough. The impact of this competition is a clear decrease in service prices, while more value-added service packages are offered to end-users. Service providers have to insure that their services are rapidly deployed and delivered with the required quality to a large number of customers. Moreover, new types of data and multimedia services are increasingly appearing. These services need to be rapidly instrumented and efficiently managed.

The end-user, be it an individual or an enterprise, is becoming tightly dependent on the good operation of the contracted network services. Networks are becoming the lifeblood of modern organizations. Not only the organization's network is a base requirement to survive in a world-wide global marketplace, but it is also a determining factor to improve its competitiveness and leadership. Therefore, the good operation and the efficient control of the entreprise network becomes of a crucial importance. An increasing number of companies outsource the management of their networks and services in order to reduce the cost related to the purchase of very expensive Network Management (NM) tools and human expertise.

The profileration of network-intensive applications and services, including multimedia services, electronic commerce and business-to-business applications, IP telephony, application service provision, and personal communications are pushing technology to constantly overpass its own boundaries. Increased amounts of network resources such as bandwidth, storage and CPU capacity are constantly being required for new services, as well as stricter requirements of security, availability and response times. New techniques, high-performance hardware devices, and highly-customizable software versions

with enhanced functionality constantly feed enterprise networks with new capabilities. This constant evolution is posing serious challenges to the current state-of-the-art network operation and management.

We begin this introduction by surveying the most relevant NM challenges, and the recent trends that try to tackle them. The reader may however move directly to Section 1.3 that introduces the context of this thesis.

1.1 Current Network Management Challenges

1.1.1 Dynamism

Probably one of the most intricate problems that people from the whole NM field are facing is that of keeping pace with network evolution. Currently, several months (if not years) are required in order to build an operational Network Management System (NMS) using a management platform. The obtained NMS is very inflexible and hardly maintainable. Therefore, it is difficult to update the NMS to rapid changes in the structure and topology of the managed network. There are at least two problems related to the dynamic evolution of managed networks. The first problem is how to instrument new network components. In general, hardware devices are supplied with SNMP agents that provide the basic instrumentation for monitoring and quite limited control. Conversely, applications and services are rarely instrumented. In some cases, a proprietary management tool is supplied. Such tools rarely adopt a management standard to be easily integrated into an NMS.

The second problem is how to integrated changes in the deployed NMS. In general, this is a mundane task that requires to load the new MIB into the NMS, to initiate the day-to-day monitoring, to settle alarms, to handle these alarms and correlate them with the other alarms, etc. There is no mechanism that allows the NMS to automatically adapt to new changes in the network topology and services.

1.1.2 Evolving Management Requirements

Related to the dynamism of the managed network is the challenge of constantly evolving management requirements. New types of services such as e-commerce, VoIP (Voice over IP) and application outsourcing cannot be managed using the usual network management solutions, which are mostly restricted to element- and network-level management. Increasingly, new applications, software add-ons, and stand-alone tools are necessary to be able to answer these management requirements. But in order to be cost-effective,

these stand-alone tools have to interoperate with legacy management solutions. This is not obvious to achieve as the following two examples of new management requirements show.

1.1.2.1 Service-level Agreements

Market deregulation has necessitated contracts between network service providers and consumers. Service-level Agreements (SLA) [Lew99] are contracts that specify the services supplied to the customer, as well as their quality parameters and liabilities associated to the violation of the contract.

Service-level agreements require multiple functions to be included in NMSs. First, agreements should be specified and fed into the NMS using a certain formalism. Afterwards, a capability has to map the abstract agreement sepcifications into precise QoS parameters to be monitored. The NMS has then to initiate monitoring operations in order to check whether these parameters are in acceptable ranges. Finally, the NMS has to take the corrective actions and to generate the necessary alarms in the case that the agreement is violated.

Despite the issues that these functions involve, one major problem remains how to integrate SLA functions within an NMS that was not designed with such funtions. In general, SLA functions have to be supported by independent dedicated tools and cannot be plugged into an integrated NMS in a straightforward way [Lew99, pp 210–212].

1.1.2.2 Application-level and End-to-end QoS Management

Application-level management is a recent challenge for NM. One of the major problems with application-level management is that dependencies between the user applications and the network- and resource-level elements has to be properly identified. Yet the currently available instrumentation rarely goes beyond a low-level of information, and there is no common agreement on how applications should declare the services and resources on which they depend.

Another aspect of the problem is end-to-end management which is required to efficiently manage services such as IP telephony. End-to-end management requires spanning management views from different network providers. Only the combination of all these views allows to provide an overall end-to-end management view of the delivered service. This requires mechanisms for heterogeneous management applications of different network and service providers to exchange information and to cooperate, which is another challenge for today's NM.

1.1.3 Integration and Interoperability

In general, multiple independent management tools are used simultaneously. This is necessary to cover the whole, or at least the most required management functions. These management tools are often heterogeneous, i.e. developed by different parties, and it is difficult to communicate and coordinate between them. Every tool uses its own internal database and uses its own monitoring and report generation functions, which results in uselessly duplicated resources and operations. The reports generated by the different tools do not have a uniform format, which hardens the tasks of analysis and interpretation for the network administrator.

[HAN99] describes different levels of possible integration. The simplest integration occurs at the level of the user interface. Different management functions can be invoked from the same user interface, while still performed by completely independent tools. Proxies and gateways can provide increased integration between different management tools by performing such tasks as information model conversion and data change propagation. At a higher level of integration, a uniform management information base is provided. The different management tools rely on this information base and comply to the same management information model. This allows to avoid inconsistency and data duplication. Finally, the highest level of integration implies that the different tools can seamlessly interoperate together. The management services provided by one tool can be directly used by another tool or management module. With this full integration, each management module concentrates exclusively on the management functionality for which it is deployed.

1.1.4 Management Automation

Currently, a large part of the management operations provided by available NMSs are limited to monitoring. Only limited control is possible, and the administrator has to perform most control operations manually. In [Che99], the author shows that there are three reasons for the lack of automation in today's NM.

1. There is currently no powerful mechanism that allows the network administrator to specify an explicit and complete description of the expected normal behavior of the network. Disparate thresholds and alarm patterns cannot suffice to let an NMS work out to bring the network from an abnormal state back to its normal operation.

2. Similarly, there is no possibility to provide the NMS with a description of the effects of management actions. In order to autonomously initiate the adequate man-

agement operations when necessary, the NMS has to be aware of the impacts and side-effects of these management operations.

3. Current standard-based instrumentation fails to provide a complete control of the managed NEs. In many cases, network components include dedicated RPC, telnet or Web based control interfaces. Therefore, control operations have to deal with low-level details, and need to cope with heterogenenous control interfaces, which makes the generation of automatic control procedures very complex.

The dream of a self-healing network is nevertheless still tempting many industrials. Efforts to provide "plug-and-play" networks based on JINI (http://www.jini.org), and the "zero administration initiative for Windows (ZAW)" [Mic97] are proofs that management automation remains to be the ultimate target in NM, although apparently unreachable in the near term.

1.1.5 Proactive Management

Reducing the down time of the managed network strongly requires proactive management that detects, prevents, and repairs faults before they explicitly take place and seriously affect the status of the network. Proactive management is currently reduced to setting early thresholds that allow to detect the possible beginning of network anomalies such as congestions and disk failures. The problem with early thresholds is that they significanly increase the number of false alarms and the number of alarm events in general. The values of such thresholds have to strike a balance between sufficiently early detection and the number of false alarm events. Early thresholds allow to detect only a limited type of network faults.

To promote proactive management, the NMS has to be aware [OL98] of the network applications and services. In addition, it has to be able to correlate different distributed views of the network, possibly provided by several management entities.

1.1.6 Other Challenges: Adaptable Scalability and Reliability

In general, problems related to scalability in NMSs are tackled using a hierarchical organization. Such organization reliefs the central manager station from dealing with the whole processing required for management operations by pushing this processing down to intermediate entities. There are still ongoing research and standardization efforts to provide standard frameworks for such management distribution. Still the remaining problem is that the hierarchical organization lacks the flexibility required to support dynamic network configurations [HAN99, pp 117]. Adaptable scalability would make an

NMS automatically scale by changing the organization of its managers in a way to minimize management overhead according to the changing topology of the network.

With the proliferation of distributed management, a reliability problem is raised. In distributed management, such as hierarchical management, the central manager relies on intermediate entities to perform local processing of management information and to automate some management tasks. The reliable operation of the central manager, which is the most critical part of the NMS, becomes dependent on the reliabilty of the intermediate managers. There is currently no standard way to insure the reliability of these intermediate managers, while serious reliability leaks may occur if this problem is not correctly addressed.

1.2 Recent Network Management Trends

Faced to these challenges and a number of others outlined in [HAN99, pp 597–611], people in the NM R&D are continuously searching for new technologies from different domains such as networking, software engineering, and artificial intelligence. Some of these main trends are rapidly surveyed in the following paragraphs.

CORBA-based management The introduction of CORBA in NM aimed at providing a general distributed architecture for NM platforms. With the maturity of this trend, CORBA is now proposed for the integrated management of systems and applications. Management services, e.g. monitoring services, fault detection services, user interface services, etc. are built on top of CORBA. A management application can then be developed as a CORBA client based on these services. Moreover, CORBA/IDL has been adopted by the Joint Inter-domain Management (JIDM) Group (http://www.jidm.org) to federate different management frameworks. JIDM has defined gateways to allow legacy management components based on SNMP and OSI to interoperate with CORBA-based management systems.

Java-based management Another important technology trend in the NM field is based on Java. The Java Management extensions (JMX, http://java.sun.com/products/JavaManagement/) matured from past trials of JMAPI (Java Management API) and JDMK (Java Dynamic Management Kit, http://java.sun.com/products/jdmk/). JMX provide a standard for instrumenting managed resources and developping dynamic agents and management applications. They define a new simple and lightweight management architecture to make Java software manageable and to instrument Java objects. They also allow to interface with

legacy management protocols. JMX inherit the benefits of Java, including the "write once, run everywhere" feature, flexibility and dynamism. One of the main management challenges that JMX tackle is coping with highly dynamic and evolving distributed environments such as today's service-oriented networks.

Web-based management Web interfaces are introduced in NMSs to provide easy and ubiquitous access to management functionality. Management functionality can be easily updated in web servers embedded in managed resources. The network manager can access management data and run management tasks (e.g. Java applets) downloaded from the web server of the managed resource or of the NMS. The use of web technology in NM allows therefore for enhanced flexibility and homogeneity to access management functions. The WEBM (Web-base Enterprise Management) initiative of the DMTF (Distributed Management Task Force, http://www.dmtf.org) is currently promoting XML (eXtended Markup Language, http://www.w3c.org/xml) to convey management information across heterogeneous management environments.

The common information model CIM (http://www.dmtf.org/spec/cims.html) is a common data model of an implementation-neutral schema for describing overall management information in a network/enterprise environment. CIM is comprised of a Specification and a Schema. The Specification defines the details for integration with other management models (e.g. SNMP MIBs) while the Schema provides the actual model descriptions. CIM will be initially used to model management information, both for instrumentation and for management purposes. Later, it can be used to describe the management information between different management platforms, in order to provide common understanding of management information. Therefore, integration and cross-platform interoperability are the main issues tackled by CIM efforts.

Directory-enabled networks A DEN relies on a logically central repository that stores persistent network configuration data. Changes in network configuration are handled within this unique repository. The purpose of DEN is to facilitate the management of large and complex configurations of heterogeneous systems, and to track dynamic networks with frequent changes in their configuration. Therefore, DEN tackles the dynamism and scalability challenges in NM.

Policy-based Management Today, network control involves the configuration of many individual network devices, which simply would not scale over the long term. To increase their ability to respond to rapidly changing business conditions, IT managers need to

simplify and automate network control and configuration. Policy-based network management [ST94, LS97] promises to do just that by leveraging central policy repositories, allowing a variety of applications and services to share a consistent set of policies. This lets network administrators manage classes of NEs rather than individual components. Policies are expressed using high-level obligation and authorization rules, which are organized according to network domains and management roles. High-level policies are then recursivley refined until, at the lowest level, they are mapped into concrete NM operations, based on the available instrumentation.

Miscellaneous trends There are also a number of other trends. SNMPv3 is tackling security aspects in SNMP-based management. AGENTX (http://www.scguild.com/agentx/) and the Desktop Management Interface (DMI, http://www.dmtf.org/spec/dmis.html) are tackling the dynamism and flexibility at the instrumentation level, with the former directed towards SNMP, and the latter towards the PC and desktop management. Fianally, DISMAN (Distributed Management, http://www.ietf.org/html.charters/disman-charter.html) is tackling distribution and scalability by the definition of intermediate managers for SNMP.

Many NM challenge are being tackled by the surveyed trends. All of these trends rely on well defined technologies and are currently under development and experimentation by different industrials and standardization groups. Even policiy-based management, which longly has been confined within some research groups, is now gaining interest from industrials and a policy-based NM technology is expected to take-off in the near future. Each trend has more-or-less clear objectives, and tackles a specific set of NM challenges.

1.3 Software Agents: Another Trend in Network Management

Software agents (or *agents* for short) represent another hot trend in NM. Similarly to several other domains, a lot of hype surrounds software agents in NM, and a lot of promises are supposed to be expected from this trend. Yet, till now, there is only very limited interest from industrials to develop software agents for NM, and there are no clear signs of a near-term take-off of this trend. Further, there are still neither clear objectives, nor

an agreement on which challenges can be tackled by software agents, compared to the other NM trends.

We may provide a working definition of the term "software agent" before surveying the different agent aspects in Chapter 2. We may define a software agent as *an active autonomous piece of software that accomplishes high-level tasks on behalf of its user, possibly by cooperating with other agents*. Therefore, important properties of a software agent are autonomy, responsiveness, proactiveness, and social ability.

Originally, the NM community has recongized the need to introduce active management entities between managers and classical SNMP and CMIP agents. Such an entity is often called Middle-level Manager (MLM). MLMs are responsible for the accomplishment of localized management tasks, and reporting only relevant information back to the manager. An MLM is therefore an autonomous entity that performs its tasks in an independent way from the higher-level manager. It has to include the necessary logic and decision making capabilities to perform and initiate management tasks based on autonomous decisions. To make a decision, an MLM has to refer to its own localized view of the managed network. In many situations, this localized view cannot be sufficient to make decisions, and different MLMs may have to exchange information between each other. If management operations span different parts of the network, or different management capabilities, then different MLMs have to coordinate their activities and cooperate together to achieve these operations.

Therefore, activeness, independent action, autonomy, communication and cooperation are important properties of MLMs. A software agent has similar properties to those required in an MLM, hence the idea of adopting them in NM.

Software agents have been proposed in NM in order to tackle the following issues:

- **Lack of distribution:** There is a clear need in NM to push the intelligence down to the level of the managed resources. This allows to relieve the central management station from bandwidth and processing bottlenecks. Software agents are good candidates for this purpose to achieve useful localized processing on behalf of the management station [MFZH99].

- **Lack of automation:** Agents are also appropriate to relief the human network administrator from low-level mundane management tasks. They allow to raise the level of abstraction by adding intelligent processing capabilities to the management resources and applications [Mou98]. Agents can achieve a high degree of management automation based on their autonomy and ability to cooperate to solve complex management tasks [MFZH99].

- **Lack of flexibility:** Agents are able to act asynchronously, to communicate, to coop-

erate with other agents, and to be dynamically configurable. Mobile agents, which are capable of moving from one host to another during their lifetime, are useful for performing specific management tasks on an as-needed basis [Mag95]. These features allow to overcome the rigidity of today's NMSs, mostly based on an inflexible client/server architecture.

1.4 Our Thesis

Historically, Busuioc [Bus96], Magedanz [Mag95, MRK96], and Oliveira and Labetoulle [OL96] were the first to pinpoint the potential of software agents in NM and telecommunications in general. Since then, there have been a large number of publications describing works using agents to tackle a large variety of NM problems. There have been also some earlier related references such as [WT92] and [BG93]. However, despite these increasing efforts, we still encounter statements such as: "This field of research is still at its very beginning" [BWP98], and there are still no precise answer to a number of fundamental questions related to the application of software agents to NM. These questions are discussed in the following items.

- *What are the elements and concepts that compose the agent technology?*

  A rapid look to agent-based applications, and in particular NM applications, allows to easily understand that there is no two approaches that agree on how to build agents and agent-based systems. NM is a domain where rigorous approaches are required to cope with complexity, and therefore, there should be a precise definition of what concepts and constructs constitute agent technology.

  Chapter 2 of this thesis provides a global survey of the different elements that compose the software agent paradigm. The different aspects that allow to perceive agent concepts are provided as precisely as possible with regard to the current available litterature on the subject, and according to the author's point of view.

- *What is the best way that agent technology can be applied to NM?*

  As mentioned earlier, there has been a tremendous amount of research on the application of software agent techniques in NM. Chapter 3 provides an extensive survey of this research area and shows that completely different types of agents could be applied for different specific problems in NM. But in order to fully benefit from the potential of software agents, a global agent approach has to be adopted. Chapter 4 describes an agent architecture that we specially designed for global NM purposes. This architecture is based on precise concepts carefully refined from existing

agent techniques. This architecture is then experimented down to the implementation level in two case studies described in Chapters 5 and 6.

- *What agents are really useful for?*

  Currently, other NM trends that promote distribution and dynamism are strongly competing with agent-oriented trends. The original reason for considering software agents in NM was to introduce MLMs as a way to enhance distribution, flexibility and automation. However, it is still unclear what agents can bring more than CORBA in distribution, more than Java in flexibility and more than any other object-oriented approach in task automation. We propose to answer this question based on the results and discussions in Chapters 5 through 8.

- *What is really useful in agent technology?*

  If agents have clear advantages over other NM trends, then it is important to understand those agent features that are behind these advantages. We provide throughout the different chapters of this thesis several hints to show the particular aspects that are behind the real benefit of software agents.

- *If agents have such potential in NM, then how to boost their wide deployment to build agent-based NMSs?*

  Based on the experience gained from our work, we end up with our personal view of interesting research to boost agent-based NM.

In summary, the contribution of our thesis is to help clarifying the answer to the above fundamental questions about agent-based NM trends, thus providing a solid basis for future work in this direction.

1.5 Outline

The structure of the thesis is depicted in Figure 1.1, pp 34. After this introduction, Chapter 2 provides a comprehensive survey of software agents. It first presents the different existing approaches to define and interpret the agent concept. Then it continues by surveying the most important aspects, properties, architectures and techniques in the agent field. The chapter shows that different approaches exist to build agent architectures, and achieve coordination and organization within a multi-agent system.

Chapter 3 provides a state-of-the-art of agent-based approaches to NM. The chapter clearly shows the potential of software agents with all their different kinds and approaches to tackle different problems in the NM domain. It concludes that in order to

build an agent framework that is suitable for general-purpose NM, this framework should not be committed to a unique set of agent techniques. An agent-based approach to NM should therefore contemplate a *generic* agent architecture.

Chapter 4 investigates such kind of generic agent architecture, that in addition, satisfies two fundamental requirement for any new generation NM solution, namely: *flexibility* and *dynamism*. This generic agent architecture is based on modular skills. Skills can dynamically endow an agent with new capabilities and behavioral properties. The chapter details the concepts and the components of this agent architecture.

The proposed agent architecture is applied in two case studies. The first case study is presented in Chapter 5, and addresses the reliability of a distributed set of autonomous agents. These agents are each affected to a management domain, which can be redistributed dynamically according to the number of available reliable agents. Therefore, this chapter shows the ability of our proposed skill-based agent architecture to build flexible, dynamically adaptable, extensible, and fault-tolerant management applications.

The second case study, presented in chapter 6, uses a distributed set of agents that automate the configuration of permanent virtual channels in ATM networks. This case study shows that an agent-based approach allows to provide a more efficient solution than a centralized approach. The obtained agents can dynamically undertake different roles at the same time, according to their specific contexts of operation. The case study also shows how our skill-based architecture can be used to cope with heterogeneity issues.

An interesting question is how our agent architecture compares to other possible kinds of agent, and till which extent a different kind of agent may lead to a different design of the same application. For this comparison, we envisage two kinds of agents, considered as the most hyped agent approaches, namely BDI-oriented agents, and Mobile Agents. A BDI agent is an agent whose behavior is described based on mentalistic notions such as beliefs, desires and intentions, while a mobile agent (MA) is an agent with the capability to move from one host to another. (Both kinds of agents will be detailed in Chapter 2.) Chapter 7 develops an alternative agent approach based on an NM-oriented BDI model. It describes how this BDI model is applied to the first case study, leading to a different design approach. The results of this BDI design is compared to the results obtained with the skill-based architecture. Similarly, Chapter 8 considers an MA approach to the second case study and compares it, according to several quantitative and qualititative criteria, to the stationary approach obtained with the skill-based architecture. These two chapters allow to position our developed skill-based agent architecture compared to other agent approaches.

The ensemble of these works on different types of agents provided us with a cer-

tain experience-gained. Chapter 8.4 summarizes our contribution and the moral of this experience-gained. It closes this thesis with a backward view on agent technology and its real potential and pitfalls in the context of the rapidly evolving domain of Network Management.

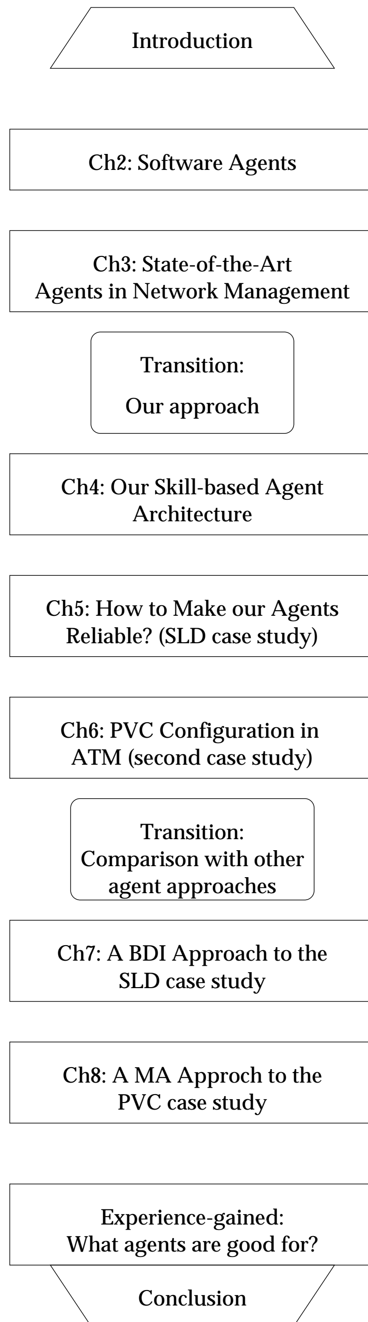


Figure 1.1: Structure of the thesis

Chapter 2

Software Agents

Putting in practice a certain theory requires a deep understanding of this theory and a clear assimilation of the definitions of its concepts. But the most immediately encountered problem in the field of software agents is precisely the definition of the "agent" concept itself. The debate is of course still open and is burstly, yet intensively triggered in discussion forums in which there are always complainers about any proposed agent definition. Those who are applying agents for concrete applications propose either a private definition tailored for the specifics of the agent application, or a general-purpose definition that is not sufficiently precise to have an *a priori* view of the type of agents to be used.

Instead of providing or adopting a certain agent definition, we preferred to review all the aspects that relate to understanding and perceiving what an agent is. Therefore, Section 2.1 presents different viewpoints, where each viewpoint allows to highlight a certain aspect of agency. The aggregation of these different viewpoints allows to have a complete picture of the agent concept.

Once the agent concept perceived, we move on to survey the different techniques developed in the context of agents and multi-agent systems. These techniques cover both the internal of the agent, as well as the interaction aspects within an agent community. This is the subject of sections 2.2 through 2.6.

2.1 What an Agent Can Possibly Be?

Though widely used, the term "agent" is still lacking a standard definition. The reason is that any proposed definition is either too general to describe precisely the agent con-

cept, or is tailored to a certain application field that focuses on a limited number of agent concepts, while neglecting the other aspects.

The goal of this section is not to provide an agent definition, but to allow the readers, especially those who are not familiar with this term, to get the whole picture of how the term agent can be used. In our opinion, the agent concept is better perceived by discussing its different aspects, rather than by struggling to provide a suitable definition. Therefore, each of the following subsections provides a particular view of the agent concept.

2.1.1 The Agent as Representing his User

It seems that the most repeated phrase "an entity that acts on behalf of its user" is mostly adopted for people who are not interested in debating the definition of an agent. This definition is indeed so general that it can be used almost for any utilization of the agent concept — and unfortunately for a lot of other concepts. Therefore, the problem with this definition is that it does not help perceiving the meaning behind the term agent.

However, what is important in this definition is that it highlights the aspect of *representing* the user. The user delegates some tasks to the agent, and the agent has to achieve these tasks in the way that its user would have achieved them, or at least wants them to be achieved.

This definition of the agent as "representing its user" is very suitable for applications such as electronic commerce and business workflows.

2.1.2 The Agent as a Metaphor

The agent can also be viewed as a metaphor. The agent metaphor is mainly an anthropomorphic mapping of human characters to a software program. Metaphors are largely used in software engineering and in all the computer-related fields in general. For example, the spreadsheet metaphor allows business people to fit a form of functional programming into a familiar medium, and the "message-passing" in the object-oriented paradigm allows to structure the activity of the developed system, while the "window" metaphor allows to provide powerful and user-friendly graphical user interfaces. Similarly, the term "agent" suggests a variety of attributes that have not generally been built into the other metaphors used in programming; attributes such as purposefulness, autonomy, and the ability to react to outside stimuli [Tra96]. "Instead of the standard metaphor of the computer as a [..] mechanical instruction follower, we substitute the metaphor of the agent that can initiate action autonomously in order to achieve a goal. Collections of agents work together to create complex behavior. [..] This is not to say that

systems cannot be built using existing tools, only that building them requires a good deal of mental contortion that may be beyond the abilities of novice programmers." (Quoted from [Tra96]).

Consequently, the agent metaphor can be used to ascribe human-like behavior (see next subsection) to software without pretending to reach the level of human intelligence, always considered as a still-unsurmountable challenge to imitate by computers. And the main reason of adopting such metaphor is to allow to model, understand and structure complex systems that are hard to tackle using other classical programming paradigms. An interesting field that uses the agent metaphor is the simulation of the behavior of complex societies. The agent metaphor is used here to designate the members of the society, which can be of various natures: insects, animals, human beings, etc.

2.1.3 The Agent as a Behavior

Probably one of the most successful approaches to defining the agent concept is based on the set of properties and behaviors that may or should be exhibited by the agent. This is adopted in [WJ95], which defines an agent as an entity that has at least the following properties: autonomy, responsiveness, proactiveness and social ability. Many references adopted this approach to defining the term "agent" and, for example, [Eur98] and [Oli98] go further by supplying a whole set of properties that agents might exhibit. Then the more of those properties a software entity exhibits, the closer it is to the agent concept. Following are the definitions of properties that are most advocated for agent software.

Autonomy The basic meaning of this property is that the agent acts independently, and evolves without direct instruction from another party. At this level, a network daemon could be considered as autonomous, while an interactive program such as a web browser is not. An increased autonomy is exhibited if the agent is capable of achieving missions that are specified at a high-abstraction level. The high-abstraction level only specifies *what* the mission is, while the burden to find *how* to achieve it is left to the agent.

But the strongest degree of autonomy requires that the agent not only achieves its missions appropriately in the normal situations, but is also able to deal with contingencies. If we refer to this degree of autonomy, a server daemon is no longer autonomous. In contrast, an autonomous agent would for example be able to detect that the machine on which it is running is overloaded, and decide deliberately to migrate to another less used machine, after taking care of updating its location entry in the location server.

Another possible characteristic of autonomous agents is that they have control over their own actions and states. A simple example is that the agent should be able to detect

that it is overloaded, and accordingly to decide to stop accepting missions or to delegate some subtasks to other agents.

Of course, an agent preferably has to be fully autonomous, i.e. acts independently, accepts abstract high-level missions, has control over its own state and deals with contingencies. However, in practice, it is neither necessary to have such high degree of autonomy, nor it is possible to achieve it in all the cases. One has to strike a balance between the required degree of autonomy and the design complexity necessary to achieve this autonomy.

Responsiveness This property means that the agent is capable of reacting to changes that occur in the environment. It implies that the agent continuously senses the evolution of its external environment (and of its own status) and acts according to these changes. The agent is perceived as a feedback controller of its environment which therefore is considered as a controlled dynamic system. This is illustrated in [Mül96] and adopted in many other agent references.

Proactiveness This property goes further compared to responsiveness and implies that the agent is not only responsive to its environment, but is also able to predict and anticipate the changes, or at least the implications of these changes. In this context, proactiveness can be understood as the ability to take the initiative, in contrast to responsiveness. A stronger meaning of proactiveness is that the agent is able to take opportunistic actions based on predicted changes in the environment, before that these changes even occur. The opportunistic behavior allows the agent to accomplish its duties with less resources or within a shorter amount of time.

Social Ability In general, agents evolve within a community. In this case, the agents have to communicate and interact with each other. This is to be contrasted with standalone applications, e.g. a Personal Digital Assistant. Social ability is a property exhibited using agent interaction mechanisms, which may range from implicitly coordinated activities to complex dialogues involving negotiations and aiming to achieve global goals.

Learning Basically, learning means that the agent is capable of using its past experience in order to improve its future behavior. Ways to implement this property may vary from simple repetitive observation of a certain behavior in the environment to heavyweight tools such as neural networks. Moreover, combined with social ability, learning may lead to multi-agent distributed and collaborative learning. For more information about learning in MASs, refer to [Wei99].

2.1.4 The Agent as a Situated Entity

An agent is not isolated from its environment. Instead, it has to keep in constant interaction with the environment in order to successfully achieve its missions. On the one hand, the agent has a certain representation of the environment and the changes that occur in it. This representation is supplied by a set of *sensors*, which capture the relevant events to the agent. On the other hand, the agent has appropriate *effectors* to act on the environment in a certain way. The agent effectors are the low-level instruments that enable the agent to execute its actions, and to actively affect the environment in a way to successfully achieve its missions.

The agent as a situated entity is closely related to the feedback control system view [Mül96] of the agent system, which was mentioned previously.

2.1.5 The Agent as a Goal-oriented Software

This view of the agent concept focuses on the nature of the missions and tasks that the agent undertakes. Agent missions are expressed in high-level goals and an agent has to work out to achieve his goals. This can be concretely understood in different complementary ways. In one way, the goal-oriented behavior means that the agent has the sense of purpose, and that its actions are not taken simply in reaction to predefined situations, but because the agent has a certain goal to achieve.

In a second way, the goal-directed behavior means that the agent has to figure out how to achieve its goals. Achieving a high-level goal may require several steps to collect the necessary beliefs about the goal, to decompose the goal into sub-goals that are easier to cope with, and to find the necessary actions that allow to achieve each (sub)goal. This means that the agent may achieve the exactly same goal in completely different ways, depending on the circumstances during which the goal is submitted to the agent.

Another aspect of goal-oriented behavior is that the agent missions may not be explicitly formulated during design time. The agent's user can dynamically affect new goals to the agent or modify existing goal expressions, which in turn affects the agent mission. Therefore, the goal-oriented behavior provides adaptability and flexibility to the agent.

2.1.6 The Agent as a Mobile Entity

A seducing idea when the agent environment is a distributed networked environment is the agent ablity to move from one site to another during runtime. We therefore obtain a specific kind of agent called a Mobile Agent (MA).

Mobile Agents will be discussed thoroughly in Section 2.3.

2.1.7 The Agent as a Modeling Entity

Inasmuch as the agent is used as a metaphor, agents are increasingly considered in software engineering as a modeling construct [Jen00, WJK99, WJ98]. Agent-based software engineering is a very recent trend and agent-oriented abstractions are not yet fully and precisely defined. However, the key abstractions when using the agent paradigm are: agents, interactions and organizations. The agent abstraction allows to decompose the system into a set of autonomous entities. Agent interactions allow to model the interactions within a distributed system at a knowledge level. And agent organization provides a means to model complex and dynamic organization structures of complex distributed systems.

2.1.8 Summary

Despite that the term "agent" lacks a standard definition, it is still possible to capture the main ideas behind the agent concept. The agent can support one or more of the aspects described above at the same time. Therefore, an agent can be used as a delegated program that represents its user, as a metaphor to understand and simulate complex systems, as a program that exhibits specific properties (autonomy, learning, etc.), as a situated entity, as goal-oriented and task accomplishing software, as a mobile program, or as an abstraction tool for modeling. It therefore appears why an agent definition that unifies all these aspects is difficult to provide. In practice, each utilization of the term agent adheres to a restricted set of these aspects. Developing an agent-based system does not require to combine all of these agent aspects. However, we believe that an agent system has to adopt at least one among these aspects to be truly characterized as an agent system.

Now that the reader is provided with a complete panorama of the possible interpretations of the term "agent", we move on to survey the different agent-oriented techniques that are available to build agent-based systems. These techniques are the subject of the remainder sections of this chapter.

2.2 Agent Architectures

There are two basic trends to build the internal agent architecture [Nwa96, WJ95]. The first trend uses artificial intelligence techniques such as logical and symbolic reasoning and expert systems. This trend allows to build *deliberative agents*, which are based on a symbolic representation of the agent's world. The deliberative agent uses its symboli-

cally represented knowledge to apply Artificial Intelligence reasoning and planning techniques. The aim is to make the agent able to produce the desired behavior to accomplish its missions. Therefore, the behavior of a deliberative agent is not explicitly encoded. Instead, the agent behavior is *inferred* at runtime.

On the contrary, the second trend rejects any kind of symbolic representation and tries to build simple agents whose behavior links perceived situations in the sensed environment directly to the actions the agent has to execute. This leads to the definition of *reactive agents*, in which the behavior of the agent is explicitly encoded.

Other architectures aim at combining the benefits of both trends by providing *hybrid agents*. The behavior of a hybrid agent is partly explicitly encoded and partly inferred using deliberation. In general, the structure of a hybrid agent is based on a *layered architecture*. One or more layers explicitly encode part of the agent behavior (the reactive behavior), while other layers deal with the deliberative behavior.

Finally, another important approach that was originally proposed in the deliberative trend makes use of human-like mentalistic notions to model the behavior of the agent. Such approaches are often called *Belief-Desire-Intention* architectures, where Belief, Desire and Intention (BDI) are the most widely used mental categories to develop such architectures. BDI agent architectures have a wide success in the agent community, mainly because it provides a suitable and powerful abstraction to model the agent behaviors.

These different kinds of agent architecture are the subject of the following sections, which begin with a basic agent architecture.

2.2.1 A Common Agent Architecture

An agent can be viewed according to Section 2.1 as a situated entity, which implies that agents evolve in an environment by "sensing" changes that dynamically occur and "acting" on this environment in a way to achieve their missions. According to this view, a basic architecture can be drawn and is illustrated in Figure 2.1.

The link from the agent percepts to the agent acts is a *decision making* process. This decision making process radically differs from an agent architecture to another. This is detailed in the following sections.

2.2.2 Deliberative Agents

Traditionally, intelligent systems are built by "giving a symbolic representation of the environment and its desired behavior, and syntactically manipulating this representation" [Wei99]. In general this means either that the agents use some theorem proving techniques, or that they incorporate planning capabilities [WJ95]. Theorem proving consists

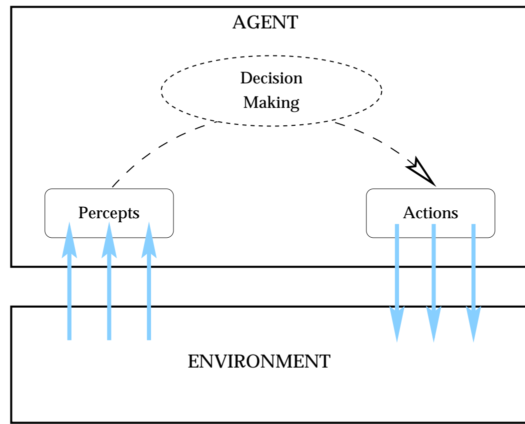


Figure 2.1: Basic agent architecture

of a set of logical formulae that translate the agent perception of the real world. The decision making process is modeled through a set of deduction rules. The deduction rules are designed in a way that if the agent succeeds to derive a formula corresponding to the execution of a certain action, then this action is the best suitable to be executed in that given context. More details about this approach and further references are given by Wooldridge in [Woo99].

Planning is another mechanism that allows for a kind of automatic programming. A planner has a symbolic description of the current state of the world and a set of actions described by their preconditions and effects. Then given a certain expression, considered as a *goal*, the planner tries to identify a sequence of actions that when executed, will bring the world from its current state to the desired target goal under the effect of the executed action plans. References on planning can be found in [WJ95].

It is nowadays admitted that these Artificial Intelligence (AI) based approaches suffer from severe problems, namely, that both planning and logical reasoning are very heavy processes and are therefore not suitable for time-constrained applications. Moreover, providing a symbolic description of a concrete problem turns out to be very complex for real-life problems. And last but not least, such applications pose serious maintenance problems.

2.2.3 Reactive Agents

In contrast to deliberative agents, reactive agents do not include any symbolic or logical reasoning in their decision making process. Instead, they tend to directly link the perceived situations in the environment to determined sequences of actions.

In principle, there is no standard architecture for reactive agents since the unique condition that a reactive agent has to satisfy is that it is not deliberative. However, the most straightforward way to provide a reactive agent is through behavior rules of the form $situation \longrightarrow action$. Such rule links a given type of situation to the reaction that the agent has to take whenever such situation occurs. In particular, *Purely Reactive Agents* as defined by Wooldridge in [Wei99] do not keep any history and therefore, take their decisions exclusively on the presently perceived status of the world.

An important concept in reactive agents is the *behavior*. A behavior specifies the way an agent responds to its environment given a certain situation. For example, a robot may be defined using two behaviors: avoid obstacles and roam randomly, where the first behavior is adopted in situations where an obstacle is detected and the second is adopted as the default behavior of the agent (i.e. in situations where no obstacles are detected).

One thesis supported by works on reactive agents is that intelligence can emerge from the collective behavior of simple entities that require no symbolic reasoning. This thesis has been applied in two distinct ways. The first way consists in building agents as composed of several layers, competence modules, or behavior specifications. Then the behavior of the agent emerges from the internal interaction of these modules. The most well known agent architectures that adopt this approach are Brooks' Subsumption Architecture [Bro91b, Bro91a] and Maes' Agent Network Architecture. A summary of such architectures and more references are found in [WJ95].

The other way to apply the emergent intelligence is to build very simple agents that interact in a way that the whole multi-agent system exhibits global intelligence. In general, such approaches are largely inspired by biological insects such as ants, which globally exhibit a special kind of intelligence based on self-organization and called *swarm intelligence*. Steel's behavior-based robots [Mül98] are an example of the application of swarm intelligence.

2.2.4 Hybrid Agents

Hybrid agent architectures mix both deliberation and reaction in order to provide agents capable of reacting promptly to changes in the environment, as well as of performing complex reasoning and planning. Again, as in reactive agent architectures, there is no unique way to build a hybrid agent. Most of the hybrid approaches follow a layered model in which the reactive layers act at a higher priority than the deliberative layers, while the deliberative layers control the operation of the reactive layers. Other approaches to building hybrid agents are based on the BDI model, which will be presented

in Section 2.2.6. The most cited example with this approach is the *Procedural Reasoning System* or (PRS) described in [WJ95].

2.2.5 Layered Architectures

Layering in agent architectures is a powerful mechanism to build modular agents with variant reactive and deliberative capabilities. Moreover, layering allows for the easy testing and maintenance of complex systems. In [Wei99], Wooldridge provides a classification of layered agent architectures based on information and control flows. In *horizontal layering*, all the agent layers are directly connected to the sensory input and action output. In this case, each layer can directly make decisions according to agent perceptions and directly execute these decisions. This might lead to incoherence problems since all the layers run concurrently. Therefore, an additional component is added to act as a *mediator* that resolves conflicts between decisions made by different layers.

We note that our skill-based agent architecture described in Chapter 4 is largely inspired by this horizontally layered model.

In another layering model, known as *vertical layering*, information and control flows are directed from one layer to the adjacent layer. In one case, known as *one pass control*, both information and control flows are transmitted from one layer to the immediately higher-level layer. Therefore, the lowest layer is the unique layer that has access to the perceptual input of the agent, while the highest layer is the unique layer that has access to the action output of the agent. In the other case, known as *two pass control*, information flows are transmitted to the higher-level layers, while control flows are transmitted to the lower-level layer. In this case, only the lowest-level layer has access to both perceptual input and action output of the agent.

Brook's subsumption architecture is an example of two pass control vertical layering architecture. Interesting mechanisms used for control flow in this architecture are *inhibition* and *suppression*. Basically, lower-level layers are attributed higher priorities. Therefore, an active layer *inhibits* the higher-level layers with lower priority. A layer that is no longer active is inhibited, and automatically gives control to the layer immediately above. Moreover, a higher-level layer can control the operation of the lower-level layer by substituting or suppressing part of the agent inputs for programmed and finite time intervals.

2.2.6 Belief-Desire-Intention Architectures

Agent theories that are based on mental categories originate from works in the Artificial Intelligence domain that focus on the understanding of practical reasoning and the

essence of actions in human behavior so as it can be mapped to intelligent software. One of the most interesting results of such works lead to the introduction of *mental categories* (or *mental attitudes*) to describe, understand and analyze the state of the agent and its past and future behavior [Bra87]. Notions such as beliefs, desires, intentions, knowledge, commitments, etc. were used for such purposes. A mental attitude is a position taken by a human or an agent towards a statement or an expression over the world. For example, an agent or a human can *believe* a statement, or *desire* to have that statement holding in the future.

Among the most popular agent architectures based on mental categories is the BDI (Belief, Desire, Intention) architecture [RG91, Bra87]. This architecture is based on the notions of **B**elief, **D**esire and **I**ntention. In fact, most of agent theories that are based on mental categories are called BDI-oriented or BDI-like theories. In general, BDI approaches are based on a set of mental categories with defined semantics and a control architecture that defines the agent's mental cycle. The *mental cycle* is the process that rationally selects its course of action based on these mental categories ([RG95] cited in [Oli98]). We provide two simple examples showing how a mental cycle may be designed.

The first example is taken from [RG91]. The agent architecture is based on the belief, desire and intention mental categories. The mental cycle is composed of three processes. The option generation process waits for events perceived as beliefs, determines which are relevant according to the current desires of the agent and generates a set of options. The deliberation process selects the set of options that the agent believes to be pertinent for achievement. Finally, the selected options are considered as intentions and an execution process works out for their achievement.

The second example is Shoham's *agent-oriented programming* language Agent0 [Sho93]. Agent0 is based on a set of mental categories composed of beliefs, commitments and capabilities. Capabilities are what the agent *can* potentially carry out, i.e. the set of actions the agent can perform. Commitments can be seen as the agent's obligations. The mental cycle is described using behavioral rules [Ret98] that map agent beliefs to directly executable commitments that invoke its capabilities.

Several mental attitudes were defined by people adopting the BDI approach. Müller [Mül96] informally defined a set of these mental attitudes. We provide in the following a description of some pertinent mental categories based on the indicated reference.

- **Belief**

  According to [Mül96], beliefs are "*the agent's expectations about the current state*

of the world and about the likelihood of a course of action achieving certain effects". Therefore, the beliefs of the agent map its perception of the world. With a more general view, beliefs might also include the perception of the state of other agents and of the agent's own state. Beliefs on the agent's own state are essential for the agent to have control over itself, while the beliefs on the statuses of the other agents are required to to reason about the opportunity of cooperating with some of them.

- **Motivation**

  The motivation attitude is introduced as a source of stimulation that leads the agent to act on the world. In [TL94], the motivation is defined as a "*driving force that arouses and directs [agent] action toward the achievement of goals*". It is important for an agent to be motivated, since motivations provide criteria to identify those situations to which the agent has to be responsive by acting on the environment. In other words, motivations allow the agent to have preferences over the status of the environment.

- **Goal**

  A goal denotes a state that the agent wants to achieve through the execution of a certain plan of actions [TL94].

- **Commitment/Intention**

  In general, the agent may have multiple possible options regarding which goal to achieve, or which course of action to adopt. In this case, the agent has to make a selection between these options, either because one, or a subset of them suffice, or because the agent is resource-bounded and cannot adopt all of them simultaneously. Accordingly, an intention is a goal or a course of action that the agent has committed to adopt. Therefore, the commitment process can be viewed as an engagement of the agent to achieve the corresponding intentions. When the agent commits to an option, it decides how long it will persist in doing the corresponding action and under which circumstances it can be dropped [SRG98]. In general, the term 'commitment' is used for engagements to the other agents, while the term 'intention' is used for self-engagements.

- **Capability**

  The agent is an active software entity. The set of actions the agent is able to perform can be classified into action types. A capability denotes the knowledge about a certain action that the agent can execute to act on the environment. This knowledge is used by the agent to decide how and when it is appropriate to execute a certain action. This definition adheres to and extends that adopted in [Ret98].

More than an architecture for building agents, the BDI paradigm is often used to model agents and their behaviors, even though these agents are not built using mental categories. For example, the database in which an agent stores the perceived environment corresponds to the agent beliefs. During this thesis, we make extensive use of the BDI terminology to describe the agent architecture presented in Chapter 4. Furthermore, we develop in Chapter 7 an abstract BDI model for NM-oriented agent-based applications, where a set of mental categories are selected and refined from those described above.

2.2.7 Other Agent Architectures

Are there still any other kind of agent architecture?

Through this section on agent architectures, we provided a survey of the most pertinent trends for building agents. More complete reviews can be found in [Wei99], [WJ95], and [Mül98] which were the basis for this survey. Two remarks should be mentioned in this context. First, applications using agent technology often use combinations of the above concepts, and in many cases add their own specific architectural features. Second, there are many other works that focus on other aspects of agency than the architecture of the agent itself. In such cases, it is very difficult to draw a clear image of the agent architecture. This is for example the general case when using Mobile Agents, which are the subject of the next section.

2.3 Agent Mobility

Work on mobile agents (MA), mobile objects and mobile code has become a hot research topic covered by many networking and software engineering conferences. Mobile code support frameworks are becoming so numerous that it is difficult to keep track of all of them. Certainly, Java is a principle factor that promoted mobility through dynamic class loading, direct support for networking and its portability. It is necessary however, to provide a certain classification of mobility approaches in order to be able to assess the application of Mobile Agents in Network Management. Our discussion is largely inspired by the classification provided in [Pic98]. We however emphasize that in contrast to many references that make use of the term "Mobile Code" as a general term for mobility, our unique concern is Mobile Agents, i.e. software agents that are in addition endowed with mobility features. Moreover, we focus mainly on those factors that affect the design of the MA. Techniques such as code pre-fetching and agent code repositories will not be considered, since they only allow to improve some performance parameters, but do not

affect the design of the agent itself. For better reviews of these aspect, refer to [Wil99] and [Pic98].

2.3.1 Basic Components of a Mobile Agent Framework

A basic component of any MA framework is an MA server that runs on each host where MAs can migrate. The role of this server is to receive agents that want to visit its host and to help transparently transferring the agents to other hosts. We adopt the same terminology as [Pic98] and we call this server "Mobile Code Server" or MCS. When an MA arrives at a site, the MCS has to provide it with an execution environment. This execution environment is in general integrated with the MCS and allows the agent to gain access to the necessary resources for its execution.

MCSs communicate together in order to reliably ensure agent transfer. An Agent Transfer Protocol (or ATP) is used for this purpose. The ATP can be based on several possible protocols such as raw sockets, HTTP, or Java RMI. The role of the ATP is to put the agent in a serialized state and to transfer it to the next MCS, where it is deserialized to resume execution.

Besides these basic functions, an agent platform may include a large set of other mobility services and facilities. Fault tolerance is one important service that insures that agents are reliably transferred and are not lost because of a crash in the system. For example, persistent saving (checkpointing) of the MA is necessary to allow fault recovery in the case that the site on which the MA is running crashes.

Many available MA frameworks include directory and location services. These services allow the agents to be aware of the existence of other agents and to tract their locations, for example to organize a rendezvous.

In order to provide an easy-to-use MA framework, it is also necessary to provide an agent development and deployment facility. In most of the cases, a set of classes and interfaces are supplied and should be integrated in the agent code in order to enable mobility. The deployment facility is an interface that allows the injection of MAs and the specification of their itinerary.

There is also an increasing need for an agent management utility that helps monitoring the operation of injected MAs and controlling their execution. For example, this may allow an agent owner to watch the progressive migration path followed by the agent.

Another important service is security, which concerns both the protection of hosts from malicious MAs, and the protection of MAs against malicious hosts. Security mechanisms have to ensure authentication, integrity, confidentiality and access control. MA security aspects are detailed in [Vig98].

The Open Management Group (http://www.omg.org) defined MASIF (Mobile Agent System Interoperability Facility), a standard aimed at enabling the interoperability of MA platforms [Obj00]. Currently, MASIF is a reference model that defines a set of common concepts and support services for MA platforms. The standardized MA services include agent management, agent transport, agent tracking and agent security. The defined MA platform components include agent system, region, place, and MAFFinder. Examples of MASIF-compliant MA platforms will be provided in the next Chapter (Section 3.3.2).

2.3.2 Strong, Weak and Memoryless Mobility

A running MA can be broadly viewed as composed of data, code and execution state. Strong mobility, as defined in [Pic98] and [FPV98], means that the agent migrates carrying all of its three parts, i.e. code, data and execution state. From a programming point of view, an agent might be running on a certain host and wants to migrate to another one. It calls a built-in migration function, e.g. `moveTo (url)` where the `url` indicates the full path to the next host. This causes the agent code and data to be transported to the new host, where the agent resumes execution right where the migration function `moveTo` has been called. This leads to a highly transparent mobility, in which the agent migrates simply by calling a determined method.

In contrast, weak mobility means that the MA migrates only with its code and data, while 'forgetting' its execution state. From a programming point of view, weak mobility requires an entry method, e.g. `onResume`, which is automatically invoked each time the agent resumes its execution on a new location. This entry method should contain the necessary code to re-establish the execution context of the agent, which is lost during the weak migration. Additionally, there might be another method (e.g. `onMigrate`) that allows the agent to prepare its migration before it is transported to the next host. Although strong mobility offers an appreciated high degree of transparency to the MA regarding the migration mechanism, weak mobility still has the advantage to be easily implemented in MA frameworks. Despite this lower degree of transparency, weak mobility still ensures one of the main advantages of mobility, namely that of conserving the history and the past data of the agent.

Two other mobile code mechanisms are defined in [FPV98], namely Code on Demand (COD) and Remote Evaluation (REV). Both COD and REV involve a piece of code that is downloaded to a certain location where it is executed. The difference between these two mechanism is strictly the standpoint of the user. COD means that the piece of code is downloaded from a remote location to be executed locally, whereas REV means that the piece of code is sent to a remote location where it is executed remotely. However,

if our standpoint is the MA itself, then both COD and REV become identical. We introduce *memoryless mobility* to group them. A memoryless MA 'forgets' its history (i.e. both its data and execution state) when it migrates to another site. Concretely, this means that each time the agent arrives at a host, it is restarted again by having the MCS call a particular entry method such as `onStart` (or simply `main`).

Interestingly, most of the existing MA frameworks use weak mobility. This is due to several reasons. The first is that most of these frameworks are based on Java, which cannot directly provide strong mobility. The reason is that Java threads are not serializable and therefore, the execution state of the MA cannot be transported. A second reason is that weak and strong mobility are equivalent from a functional point of view. The main difference with strong mobility is that it offers a higher degree of transparency to the MA programmer regarding migration aspects.

2.3.3 Remote Cloning

Another mechanism that provides a different kind of mobility is remote cloning. MA cloning is largely inspired by the `fork` process mechanism adopted in UNIX. Applied to MAs, the same principle consists in having an MA method, e.g. `cloneAt (url)` that produces an exact copy of the MA, but on a different host. The two MA instances can be distinguished for example by their respective execution hosts. Importantly, cloning is combined with one of the migration strategies and can be either strong, weak or memoryless.

2.3.4 Single-hop and Multiple-hop Agents

Single-hop agents travel to a target host, start their execution and remain there until they terminate. This kind of agent does not need any data when migrating to the target host (except maybe initialization data) and does not have to manage its itinerary. Therefore, single-hop MAs mostly compare to downloadable code, e.g. Java applets.

In contrast, *Multiple-hop agents* can travel to several sites during their lifetime. Multiple-hop MAs can perform the same repeated task for each host they visit. They can also perform different tasks, and can adapt their behavior depending on the tasks achieved in the previously visited hosts.

2.3.5 Fixed and Dynamic Itinerary

In fixed itinerary, the agent migration path is known at its creation and it does not change during the agent execution. An example of a fixed itinerary agent is an MA that has to

travel to a determined list of servers to collect data. In this example, the itinerary is in general supplied by the user of the agent. In contrast, agents that support dynamic itinerary may change their migration path during their execution. In this case, the migration is not necessarily specified at the moment the agent is created, but instead, is decided on-the-fly during agent execution. A discovery agent that is sent to discover new components in a network is a type of agent with dynamic itinerary, since its migration path can change when new components or subnetworks are detected.

2.3.6 Passive and Active Migration

Passive migration means that the migration decision is taken by another entity than the MA itself. Passive migration provides stricter control over the agent autonomy as far as its itinerary is concerned. Itinerary management is completely left to the user of the MA. Passive migration can be combined with either a fixed or a dynamic itinerary.

Active migration means that the MA takes the decision to migrate by itself. Again it can be used either with a fixed or a dynamic itinerary. With a fixed itinerary, the agent still has to decide *when* to migrate, although it cannot decide which host to visit next.

Summary

We showed in this section that although MA frameworks share the same basic infrastructure, the supported types of MAs can differ significantly. Different types of MAs can by itself significantly lead to different approaches to the same application. In our opinion, an ideal MA framework has to provide support for building mobile agents of different types and features. This allows to select the optimal kind of MAs depending on the application for which they are deployed.

2.4 Agent Communication

In the same way that the different entities of a distributed application have to exchange messages, agents need to communicate during their operation, albeit at a higher-level of abstraction. Agents may communicate using different mechanisms. We expose in the following sections different levels of communication capabilities starting from indirect communication to general-purpose *Agent Communication Languages* (ACL).

For any agent communication mechanism, three levels must be considered (Figure 2.2). Firstly, there is a need for a network-level transport protocol, which can be any suitable protocol ranging from raw TCP or UDP, to Java RMI or CORBA IIOP. Other intermediate protocols can also be used such as SMTP or HTTP. Secondly, a communication

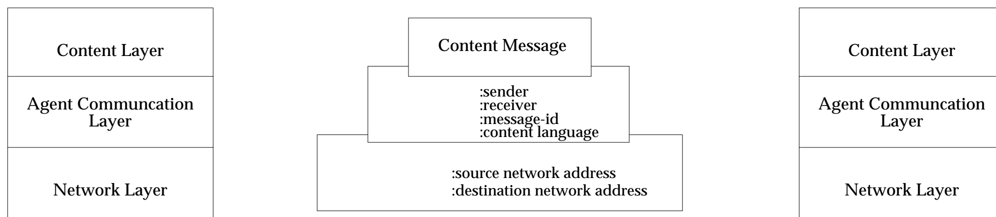


Figure 2.2: Agent communication layers

protocol is required to specify general header information concerning the message such as the names of the sender and receiver agents, the context of the message within a conversation thread and the intention behind sending the message. This is the purpose of the *Agent Communication Language*. Finally, the content of the message itself is specified using a *Content Language*. The content language should be suitable for the subject of conversation between the agents.

However, all of these three layers are not necessarily present in all the agent systems. This depends on how rich the agent communication is required to be in the target agent-based application.

Indirect Communication In some agent applications, agents may not need to communicate directly. Instead, they exchange information indirectly by modifying the status of the environment. This is the approach used in general with biologically inspired agents (see Section 2.2.3), which communicate using simulated chemical signals that they deposit on their paths.

Indirect communication is therefore a primitive kind of communication. Its main advantage is robustness and simplicity.

Ad Hoc Agent Communication Some agent applications may use proprietary communication languages. This means that the transport protocol, the message syntax and content language are tailored to meet exactly the objective of the application. Moreover, the three layers of agent communication shown in Figure 2.2 are mixed up altogether and cannot be distinguished. Such approach might be easier to implement and more efficient for a particular given agent application. Its main drawback is that it does not allow for interoperability between agents designed for different systems.

2.4.1 General-purpose and Standardized Agent Communication Languages

A structured communication protocol can be used such as KQML (Knowledge Query Manipulation Language) [LF97, FW91] or FIPA (Foundation for Intelligent Physical Agents) ACL (Agent Communication Language) [ACL99]. Such agent communication protocols are based on the speech act theory [Sea70], which attributes intentions, *communicative acts* or *performatives* to the messages exchanged between the agents. `Tell`, `ask`, `achieve` are examples of possible performatives. The advantage of KQML and ACL is that they are completely independent of the message transport protocol and the message content. KQML and ACL provide a means to structure exchanged agent messages in communicative acts with well specified semantics. Approaches based on KQML and ACL are supposed to provide a high degree of interoperability and openness in agent based applications.

For a comparative analysis of KQML and ACL and further references, the reader is referred to [LFP99].

Note also that there are many variations based on KQML and the speech act theory in general proposed for particular agent applications. For example, COOL, which is a KQML-based coordination language, proposes a set of performatives which enable negotiation and coordination in multiagent systems [BF96]. Also, JAFMAS [Cha97] extends KQML with new performatives such as `propose` and `reject` to support agent coordination.

2.4.2 Content Languages

KQML and FIPA ACL are neutral regarding the content language. This means that any general-purpose content language that allows the exchange of knowledge between agents such as Lisp, Prolog or SQL can be used. There are three major proposals for content languages in literature.

The Knowledge Interchange Format (KIF) is a declarative Lisp-like language [GF92]. The language definition includes both a specification for its syntax and another for its semantics. It allows to express simple and more complex data, logical expressions, knowledge, knowledge about knowledge and procedures. Its main advantages are readability and translatability to and from typical knowledge representation languages. KIF is often combined with KQML.

The second proposal is the FIPA's Semantic Language (SL). Originally, SL was used to formally define the semantics of ACL (Agent Communication Language) performatives. As such, SL is also appropriate for use as a content language. FIPA defines three profiles

for SL: SL0, SL1 and SL2. SL0 is the minimal subset of SL, SL1 extends SL0 by adding propositional expressions, while SL2 is another restricted subset (see [ACL99] for more details).

Another increasingly proposed content language is the eXtended Markup Language (XML) [BPSM98]. Data Type Definition (DTD) files are used to define the data types that can be used in XML messages. The main advantage of XML is that it is a state-of-the-art web-directed language. XML-based content languages are gaining increasing success in the agent community.

2.4.3 Ontologies

Both KQML and FIPA ACL contain an `ontology` field associated to the contents of the exchanged messages. The *ontology* allows to interpret the content of the message according to a certain subject or field [Gru93]. In practice, this is mainly useful in situations where confusion may arise when the same vocabulary used in agent messages can be understood with different meanings according to the topic of the dialog.

Ontologies are also useful to provide a basic shared knowledge that allows agents tackling the same application domain (e.g. network management, enterprise business workflow, etc.) to seamlessly communicate together. The ontology formally describes the semantics of the vocabulary used in a certain domain, which allows agents to correctly interpret messages related to this particular domain. More references about agent ontologies can be found in [Gru93, GF92, NW96].

2.4.4 Relevant Aspects in Inter-agent Communication

Besides the communication language aspects described above, other design issues have to be considered when agents communicate.

Synchronous/Asynchronous Messages An agent communication mechanism can be either synchronous or asynchronous. When synchronous communication is used, an agent processes the incoming messages as soon as they are received. Each agent includes a thread that listens to incoming messages, processes them, and promptly initiate replies whenever needed.

When asynchronous communication is used, the agent does not constantly listen to incoming messages. Instead, incoming messages are stored in a given repository, e.g. a mailbox, or a message queue. The agent then has to check for these messages from time to time during its execution. There is no beforehand knowledge about when a message

will be processed by the receiver agent. Asynchronous communication can be implemented using several mechanisms such as the blackboard or a mailbox.

Let's note that both synchronous and asynchronous communication do not suppose any underlying communication content language. As an example, both mechanisms can be used with a KQML-like language as well as with a proprietary application-specific language.

Note however that asynchronism is used with a different meaning in KQML and FIPA ACL (Agent Communication Language). A synchronous communication in their context means that the sender agent has to synchronously wait for the reply of the receiver. This applies for example to querying performatives. Conversely, an asynchronous communication means that the initiator of a dialog may receive asynchronous unsolicited replies from the receiver agent. This applies for example when the sender subscribes for certain change notifications, which are sent back asynchronously by the receiver agent when they occur.

Reliability In an unreliable network such as the Internet, agent messages are not guaranteed to reach their destinations. The question to be asked in this case is: Does the agent application tolerate message loss or not? The approach adopted in FIPA ACL (Agent Communication Language) is to assume that messages are delivered reliably and orderly [ACL99]. But this might not be the case in many practical applications. If message transportation is not reliable, then mechanisms that detect lost messages should be included in the agent behavior. In general, since agent communications are considered as acts, the failure to send a message, or to receive an expected reply, is considered as a failure in the agent course of action. The agent should include dedicated behaviors when it fails to achieve a certain action, and in the particular case of a communication actions, to recover from unexpectedly interrupted communication threads.

Time Aspects In many cases, the message transmission time cannot be neglected by the agents. For example, if a sender agent communicates real-time information to a receiver agent, then the receiver should be aware that this information can be out-of-date by the time it is delivered. Time aspects are not explicitly considered in FIPA. However, in time-sensitive applications agents have to cope with transmission delays (for example, using timestamps).

Direct and Indirect Communication Another important factor is that agents may not in reality communicate directly, but use an intermediate entity instead. This is the case for KQML and FIPA ACL (Agent Communication Language), which make use of *facilita-*

tors that play the role of message forwarders or *routers*. Facilitators, by their role, provide registration, naming and location services for agents in the same domain or region. In addition, they provide a suitable means to support multicast agent communication by the definition of groups of interest or agent regions.

The blackboard mechanism can also be used for indirect agent communication. The principle of the blackboard is that any agent can write messages that can be read by the other agents.

Such facilitations are certainly valuable to ensure flexibility, dynamism and support for group communications in the multiagent system. In these cases however, special care should be addressed to scalability concerns.

2.5 Agent Interaction

As soon as the agents have to interact directly, coordination problems may arise [GHN+97]. Some reasons are that agent actions can be correlated or have to be synchronized, and that agents do not own a global view of the system. Therefore, they may initiate conflicting activities if they do not coordinate their actions. The term *coordination* is used as the most general concept for agent interaction and implies that agents are able to "avoid extraneous activity by reducing resource contention, avoiding livelock and deadlock, and maintaining applicable safety conditions" [Wei99].

Several aspects of agent interaction and coordination are surveyed in the following sections.

2.5.1 Cooperative vs. Self-interested Agents

Whether agents are cooperative or self-interested widely affects the design of an agent-based application. A *self-interested agent* works in a way that maximizes its own profit without any consideration to the global benefit of the overall agent system. The profit of a self-interested can be measured using either a quantifiable metric such as an earning function, or non qualitative, such as the degree of accomplishment of the agent's own goals. A self-interested agent would not act on behalf of another agent 'for free', whereas a *collaborative agent* works for the benefit of the overall agent system.

A particular kind of self-interested agents are called hostile agents. A *hostile agent* is a self-interested agent that in addition, has its own profit increase by decreasing the other agents' profits.

Of course, there are no general rules that help choosing that kind of agents or the other. However, it is natural to use collaborative agents to model entities inside the same

party while self-interested can be used to model different competitive parties. Durfee provides a survey of cooperation mechanism for collaborative agents in Chapter 3 of [Wei99], while Sandholm provides a survey of decision making among self-interested agents in Chapter 5 of the same reference.

2.5.2 Reactive vs. Deliberative Coordination

In [CD97] (reported in [GHN+97]), Chaib-Draa distinguished between routine, familiar and unfamiliar situations to achieve coordination. *Routine situations* are the usual situations encountered in real life. *Familiar situations* are also known situations but they are less usual and are encountered only in exceptional cases. *Unfamiliar situations* are completely unknown situations for which coordination is not directly governed by social laws. The basic point in [CD97] is that coordination is easier in routine than in unfamiliar situations. Routine and familiar situations are governed by social laws that are directly derived from the real world. For example, in many countries, cars coming from the right hand direction have priority over the other cars. Such social laws can be encoded inside the agents using procedures and coordination patterns that map directly the perceived routine situation to the required coordination actions that have to be taken by the agent. If routine situations are supported by these procedures and coordination patterns, then familiar situations can be governed by reactive rules and unfamiliar situations can be handled using social knowledge and learning techniques applied on familiar situations.

The moral out of this distinction between routine, familiar and unfamiliar situations in agent coordination is that cooperation can be either reactive or deliberative. Reactive cooperation is directly encoded into the agent behavior. Therefore, it leads to efficiently executed coordination patterns, but cannot deal with unknown situations. In contrast, deliberative cooperation, although requiring sophisticated AI techniques, allows to deal with contingency situations that require the agents to infer a suitable coordination method at runtime. Coordination methods are adopted on the fly and are not necessarily known at the design time of the agent.

Note however that deliberative cooperation is very complex to design since it implies that the agents are able to collectively detect a conflict situation, and then to globally reach a consensus on what coordination scenario to be used. Reactive deliberation is much simpler to design and in most practical applications, it is quite sufficient for the application requirements, as soon as these requirements are well specified at design time.

2.5.3 Global Problem Solving

Multi-agent planning is a process during which the agents agree on a global plan that allows to achieve a common global goal. It can be either centralized or distributed. In the centralized approach, the agents send their individual plans to a central agent that checks for conflict and synchronization problems. The central agent refines the set of plans in a way they can be executed in a coordinated way.

In the distributed approach, agents build their plans by communicating together and updating their respective beliefs on the other agents. The process continues until all the agents agree on a global plan which is then executed.

The produced plan can itself be either centralized or distributed. This means that the execution of the produced plan can be either undertaken by a single agent, or by an agent system that can potentially be different from the agent system that produced the plan.

2.5.4 Delegation-based Cooperation (Task Sharing)

There are many situations in which it is important that an agent delegates tasks to other agents. The most immediate reason is load balancing, i.e. the *delegator agent* is overloaded and decides to pass some tasks to other agents supposed to be less busy. Another reason, is that agents may be specialized in accomplishing different types of tasks, and therefore, the delegator agent may simply not have the required skills to achieve the tasks, which then must be delegated to other suitably skilled agents. Moreover, delegation may be adopted simply because agents are affected to domains and that an agent cannot achieve tasks that involve another agent's domain. This type of delegation is known as domain-based delegation.

Four steps are involved in delegation:

1. **Task decomposition:** The delegator breaks a global task down into subtasks in order to determine which subtasks have to be delegated, and which it can achieve by itself. In simple cases, task decomposition can be performed at the design time of the agent application. In this case, the decomposition process is directly embedded in the agent code. In more complex cases, the agent task can be specified using an abstract expression in the form of a goal. In this case, the agent has to decompose the task at execution time, for example using planning techniques.

2. **Task allocation:** Once subtasks are identified, the delegator identifies which tasks are appropriate to be delegated. Then for each delegated subtask, a *delegatee agent* has to be designated. In most of the cases, delegatees can be easily designated using either their known capabilities or their associated resources and domains.

However, in other cases, a dynamic task allocation mechanism has to be used. One of the most used mechanisms is the Contract Net Protocol which is exposed in a forthcoming subsection (2.5.6).

It is essential to decide whether the delegatee agent has the permission to refuse the delegated task, or just has no choice other than to accept it. The former case is mostly used when agents are considered as peers. The latter case assumes that the delegator agent has a higher-level authority according to a certain agent hierarchy, that could be static or dynamic.

3. **Task accomplishment:** The delegatee agent achieves the required task, possibly by performing on its turn task decomposition and delegation recursively. In general, the delegatee agent has to notify the delegator of the progress of achieving the subtask. Depending on the delegation protocol, the delegatee may inform the delegator of such information as the time when the subtask is started, whether it is decomposed or not, and in this case, to whom the subsubtasks are delegated, the priority accorded to the subtask, etc. Such information can be essential for the delegator to synchronize between the delegatee agents, and to insure that the global task is being correctly conducted in a distributed coordinated manner. In general, it is sufficient to notify the delegator just of the final result of the subtask. In other cases, this is not necessary when the delegator may conclude the achievement of the task by observing the environment, for example by waiting for a connection to be closed or a socket to be freed.

4. **Result synthesis:** Eventually, when the subtasks are completed by delegatees, the delegator synthesizes the collected results. Note that intermediate results may continuously arrive to the delegator and therefore, the synthesizing might be a continuous process. It may also include recovering alternatives applicable in the case that a delegatee agent fails to accomplish its delegated subtask.

2.5.5 Voting

Voting is a suitable mechanism for global decision making within a multiagent system. In general terms, every agent votes for or provides certain solutions. Afterwards, and based on agent votes, one solution is selected according to defined criteria. All the agents in the election community have to abide to the adopted solution. Therefore, voting is a practical mechanism to reach consensus between a community of agents.

2.5.6 Contract Net Protocol

CNP is a widely-adopted agent interaction protocol that provides an elegant solution to the problem of resource and task allocation between agents [Smi80]. A *manager agent* wants to find candidates to perform a certain task. It sends a task announcement to the set of *contractor agents* that are likely to be interested in undertaking this task. Each contractor may submit an offer to achieve the task with a corresponding bid. The manager examines the contractors' offers and selects the best bid. Variations to CNP may allow the contractor to refuse to achieve the contracted task even when its bid is selected or allow the manager to refuse all the bids, and to formulate another more suitable announcement. This is a particular way to implement negotiation. A survey of CNP and its variations as well as a formal model are provided in Chapter 5 of [Wei99].

2.5.7 Market Mechanisms

One of the advantages of market mechanisms [Wel93] is that they avoid direct agent communications. Instead, a market institution is installed in which agents can trade goods. *Producer* or *seller agents* submit their bids in the marketplace to maximize their profits by selling their produced goods. *Consumer* or *buyer agents* submit their bid to buy goods with the lowest prices possible according to their budgetary constraints. With such market economy metaphor, agents model self-interested decision makers that act either as producers or consumers of resources. Consumers may of course use the bought resources in order to produce other goods, which can later be sold. The advantages of market-based control are the direct support of ownership and accounting of resource usage [GJVG99], and agent anonymity. Moreover, the number of agents my change dynamically within the system.

An important aspect in market-based mechanism is the equilibrium state. The purpose from using market-based agent control is to eventually converge to an equilibrium status, in which resource and task allocation is achieved in a near sub-optimal way [Wei99].

Auctioning is another market mechanism in which an *auctioneer* tries to sell a good with the highest-possible price in a community of competitive bidders. Several auction mechanisms can be used: English, Dutch and double auction. As an example, the sealed-bid first-price auction is a variation in which bidders are free to submit one bid each for the item to be sold. This item is then awarded to the highest bidder at the price of his bid.

Bargaining is a market mechanism that supports both cooperative and competitive agents. In *Axiomatic bargaining*, "desirable properties for a solution, called axioms of the bargaining solution, are postulated, and then the solution that satisfies these axioms

is sought [through the bargaining process]" [Wei99]. Therefore, agents in axiomatic bargaining are cooperative. In contrast, *strategic bargaining* makes use of competitive agents that look to maximize their own utility functions.

More references on auctioning, bargaining and market mechanisms in general can be found in [GHN+97] and in Chapter 5 of [Wei99].

2.5.8 Coordination without Direct Communication

There has been some research on achieving coordination without direct communication. This kind of coordination is achieved with the agents observing the environment and detecting the changes caused by the other agents. Consequently, an agent can deduce the activities of the other agents and coordinate its actions accordingly. Of course, only particular types of agent applications could be modeled using this mechanism. However, coordination without communication has the advantage of avoiding distribution-level problems such as synchronization and conflict resolution, which are due to inter-agent communication.

Summary

What we presented in this section on agent interaction is a representative set of coordination techniques that can be used within an agent system. An important factor that affects the way agents interact is whether agents are self-interested or cooperative. Based on this criterion, the interactions in an agent system can be designed using the many ways exposed above. Another important factor is the required ability for the agents to handle unfamiliar situations to coordinate their activities. This allows to decide whether a deliberative coordination is necessary or a reactive coordination is sufficient. A final aspect that has to be considered is agent heterogeneity. Coordination may be complicated if agents are different in their architectures, properties or behavior. A brief description of heterogeneity levels is presented in the next section.

2.6 Heterogeneous Multi-Agent Systems

In most of the cases, a multi-agent system uses a set of agents that model different types of entities with different roles. Therefore, it is often likely that agents within a MAS have different properties, roles, architectures, etc. There are different levels of heterogeneity in MASs. In a low level of heterogeneity, agents may differ only in their attributed roles and functions while still sharing the same properties and architecture. This kind of

heterogeneity is actually normal and often useful in MASs. The degree of heterogeneity increases if agents have different architectures or properties. For example, a MAS may be composed of both Mobile and Static agents, or can be built with a mixture of reactive and deliberative agents. A higher degree of heterogeneity results from agents having different cooperation and coordination strategies. For example, one may imagine a MAS which contains some agents acting as self-interested entities, and other agents with cooperative delegation-based collaboration. Finally, the highest degree of heterogeneity is obtained when agents are developed by different parties, each party using its own proprietary view of its agents. Only standardization efforts could make this full heterogeneity possible in the future. However, such high-level of heterogeneity is certainly highly appreciated in an open networked community such as the Internet and is the subject of increasing efforts.

2.7 Summary and Conclusion

This chapter provided a general view of agent concepts and techniques. Its purpose was twofold. Firstly, this chapter aimed at providing a reference terminology that will be extensively used in the forthcoming chapters. Moreover, it provided a valuable means by which existing agent implementations can be analyzed and classified systematically. For example, it becomes easy to identify the degree of mobility and the migration strategy used in a mobile agent system. Similarly, agent communication languages can be easily classified according to different levels of message structuring, and agent interactions can be easily mapped to a certain approach for agent coordination. The next chapter makes extensive use of this systematic classification to analyze a large number of agent-based applications in NM.

Secondly, the survey provided in this chapter shows that agents are not a single and precise concept. Instead, a mixture of interpretations, techniques, mechanisms, algorithms, and languages are encapsulated behind the term “agent”. This represents a proof to the richness of agent techniques and their great potential when applied in NM. On the other hand, and faced to the discrepancies in agent interpretations and techniques, it appears that there is no single way to build agent systems and that there are no guidelines that allow to select the right kind of agents.

At this stage, a precise answer to what kind of agents is best suitable for NM applications cannot be formulated. To help answering this question, we propose to survey the way software agents have so far been applied to the field of network management. Our purpose is to study which kind of agents are best suitable for NM, and in which way they have been applied. This will be the subject of the next chapter.

Chapter 3

Software Agents in Network Management – A Survey

The Network Management community is investigating a lot of new software technologies in the hope of finding suitable solutions to an increasing number of challenging problems. Agents are one amongst these investigated technologies and the number of agent-based NM applications are so tremendous and diversified that a complete survey is probably impossible. One major difficulty is that related publications and references are published in many different fields, ranging from software agents to Network Management, without forgetting distributed systems and applications, object-oriented software engineering and general networking conferences.

This chapter provides a survey of agent-based Network Management applications based on as much papers and references as could be collected.

3.1 Preamble

Our focus in this survey is mainly on the possible ways agents can be applied in NM. Our purpose is to find what agent technology is most suitable for what NM activity. In most of the described applications, we only provide an indication or a summary on the NM field for which the agent application is developed. In contrast, we analyze in more details the agent techniques used in these applications —whenever these details are supplied. This approach is different compared to the survey of Hayzeldan et al. [HB98b] who successively enumerate agent based NM applications, without separating the impacts of the different agent techniques used in each application.

We choose to handle the different aspects of agency separately. For example, when we

deal with Mobile Agents, we do not direct attention to aspects such as cooperation and communication language. If necessary, these aspects will be handled later in another section. This will lead to a modular survey which allows to objectively assess the benefit of agent techniques separately against NM applications, with minimum interference between these techniques.

One of the factors that complicates the elaboration of this state-of-the-art is that many research works only use the metaphoric concept of agency in their works. In general, related publications do not include any details on the techniques adopted for the design of their systems, for example regarding the communication language or the agent architecture. Such works adopt the agent metaphor because it provides an elegant means to naturally model a certain problem with a distributed nature. Among these works, we cite [WFFC99] which describes how agents, as modeling entities, can be used to directly map their framework that helps managing network resources by decomposing the network into a hierarchy of *blocking islands*, thus providing abstract views that highlight network bottlenecks. In this case, an agent directly models the dynamic domain defined by a certain blocking island.

The remainder of this chapter discusses agent techniques in state-of-the-art network management applications. We successively discuss management by delegation, mobile agents, reactive and deliberative agents, agent communication languages and cooperation techniques.

3.2 Management by Delegation

3.2.1 Background

People working in NM have quickly identified deficiencies in classical NM architectures and protocols. Management data is distributed among agents with very limited processing capabilities, while processing these data to extract management information is centralized in managers. The latter consequently suffer from a heavy processing load. Furthermore, management operations must be decomposed into very primitive functions mostly limited to SET- and GET-like primitives and then communicated to the agents. This is referred to as micro-management [YGY91]. Micro-management leads to heavy management traffic in the network.

To overcome this undesired situation, and in order to define a more flexible and scalable architecture with real-time management capabilities, it is tacitly admitted that management functions and operations should be dynamically carried out close to where the

managed objects are. Management by delegation (MbD) is designed following this principle.

One main concept on which MbD relies is that of an *elastic server* [Gol93]. An elastic server is an enhanced server whose functionality can be extended and reduced dynamically during run-time. A delegation protocol allows a client to pass new functions to an elastic server and then asks the server to execute these functions by instantiating them. The protocol allows to control each execution instance, e.g. to stop and resume its execution dynamically. New functionality is prescribed by a script with no *a priori* restriction on the language in which it is written.

As can be noticed, the MbD paradigm has been developed without initial consideration of software agent technologies. Nevertheless, MbD is relevant to our work at least for the following reasons:

1. Many concepts used in MbD can also be used when adopting an agent-based approach. Examples of such concepts are the delegation-based cooperation and the dynamic behavior of managers.

2. A lot of work that applies MbD concepts has been or is being conducted under the banner of "intelligent software agents".

3. We believe that this second reason was actually a driving force to consider agent technology in Network Management. Moreover, the concept of moving a script from a delegator to a delegatee is very similar to using mobile code and thereafter, mobile agent technology.

We first present works based on the MbD paradigm and how they applied the concepts of extensibility and delegation. Afterwards, we provide a synthesis and a discussion.

3.2.2 Management by Delegation

Applied to network management, elastic servers can be interpolated between managers and agents (in a similar way to middle-level managers in organizations) with similar roles to proxy agents [5]. These elastic severs are then called MbD agents [YGY91], MAD agents (Manager Agent Delegation) [YGY91], managing agents [TK97], delegation and flexible agents [Mou96] or elastic agents [GY95].

A *delegation protocol* must be defined. The manager uses the delegation protocol to upload management scripts to the flexible agents. Using the same protocol, it asks permission to instantiate some delegated script and to run it. Therefore, the management

operation prescribed in this script can execute "autonomously" in the same environment as the managed objects. The manager is able to control its execution via dedicated primitives in the delegation protocol.

A different approach than using a new delegation protocol is to make use of the existing management protocols (SNMP and CMIP) in order to make classical agents evolve towards elastic agents. In this case, the process of delegating management scripts and executing them remotely is completely handled using the management protocol that is used to manipulate a special-purpose MIB designed to enable agents to receive and execute scripts. This is the approach used by the normalizing entities, i.e. the IETF and the ISO. IETF standards through the Distributed Management working group (DISMAN) produced the Script MIB [LS99], while ISO standards produced the Command Sequencer Systems Management Function [ISO98]. An overview of the two approaches can be found in [Sch97].

Earlier, Goldszmidt [Gol93] called the management scripts –regardless of whether they are transferred using a delegation protocol or using a network management protocol– *delegated agents*. We avoid using this term, which may cause confusion with the term "delegation agent". Under the banner of intelligent agents, some research was conducted to explore the MbD paradigm for NM purposes. [KSSZ97] suggests a spreadsheet scripting environment for SNMP. The spreadsheet scripting language allows a manager to prescribe computations that can be carried out by the agent. Each cell in the spreadsheet is defined by an expression that computes a value from other given data such as MIB variables or other existing cell values. Expressions can be inserted, updated and deleted according to the manager needs.

A further improvement is depicted in [SKN97]. Each agent contains functional objects with network management functions allowing access to the management objects and communication with other agents. Therefore, a manager needs only to delegate script skeletons to invoke these management functions. When running an instance, the management functions' invocation is bound to those implemented in the functional objects. The same management function could be implemented differently on different agents, e.g. according to the network device specifics.

Other works may use different scripting environments such as the SQL-like approach presented in [ZLH96], event-driven scripts presented in [Koo95] in its divide and conquer approach and the Tcl-based scripting language in [GA97].

Alternatively, Trommer and Konopka [TK97] choose to add a rule processing unit to agents. Rules can be transmitted from managers to agents, and can endow the latter with intelligent behavior. Rules can be dynamically loaded, updated and removed, making the agents flexible yet capable of having simple and dynamic reasoning capabilities.

Finally, an interesting implementation of the MbD paradigm is that achieved by Mountzia [Mou96]. In her work, she introduces the concept of a *flexible agent* having the capability to receive scripts that define management functions and tasks. A manager or another flexible agent can therefore delegate management tasks to another flexible agent using a dedicated communication language based on KQML. In [Mou98], two delegation schemes are proposed. In *push-based delegation*, the delegator *pushes* the required functionality into the flexible agent; whereas in *pull-based delegation*, it is the flexible agent that pulls the functionality from a specified location. Flexible agents are organized hierarchically and they operate under the direction of a manager. A methodology is also proposed in [Mou98] and includes a task-to-subtask decomposition phase, a subtask description phase and a realization phase. This methodology and the proposed flexible agent concepts are applied in the field of fault management of distributed services. This application is described in [MR99].

3.2.3 Synthesis and Discussion of Management by Delegation

Management by Delegation is certainly a very interesting paradigm. All the approaches presented above provide many advantages even when low-level delegation such as spreadsheet-based delegation is used. One of the most appealing advantage is enhanced flexibility. The network administrator is no longer tightened to the management functions implemented in the NMS he uses. At runtime, the administrator can delegate new management functions and tasks to middle-level MbD agents and then invoke these functions on demand. Another advantage is the ability to bring processing capabilities close to where management data are generated, thus reducing processing overload on the management station. Finally, MbD allows to avoid the heavy management traffic due to micro-management that occurs when using a low-level management protocol such as SNMP.

The works presented above had the Management by Delegation paradigm as their starting point. For most of them the appellation "Intelligent Agent", which actually means "MbD agent", is mostly adopted to distinguish such agents from the normal management agents, i.e. SNMP and CMIP agents. Except for the work of Mountzia and recently in [GY98], the other works, although using the term 'intelligent agent', did not show any background related to software agent technology. Later in this chapter (Section 3.6), we will have another look at the concept of delegation, but at a higher-level and from an agent-oriented point of view. We will show how the agent technology provides higher-level delegation than, at the end, the simple exchange of management scripts.

Management by Delegation shares the idea of bringing the code down to where

needed with Mobile Agent approaches. On the one hand, MbD can be seen as a restriction of MAs to the remote evaluation paradigm in which code is uploaded to be executed close to data. On the other hand, MAs can be used, as noted in [Mou98], to transfer management tasks and functionality between delegators and delegatees. In our opinion, the main conceptual difference between MbD and MAs is that in MbD, the focus is on the entities that perform delegation, i.e. the delegation agents. In contrast, the MA paradigm focuses on the mobile entities themselves and does not pay much attention to the entities that create and dispatch the MAs. The main difference is therefore the standpoint from which network management is considered.

The NM research community is leaving Management by Delegation in favor of the much more powerful and flexible paradigm of Mobile Agents. The following section discusses the aspects of applying MA technologies in the context of NM.

3.3 Network Management with Mobile Agents

3.3.1 Generalities

Mobile Agents have broken out since few years, particularly in the domain of NM. Enthusiasm to mobility was motivated by such advantages as efficiency and resource saving, reducing traffic on the network, robustness, support for heterogeneity and easy software upgrades [GHN+97]. It seems that the idea of a software entity that is capable of migrating dynamically through a network attracted many researchers and industrials to investigate this technology. In the field of Network and Services Management, Magedanz [Mag95] was the first to signal the potential of MAs. Several cases and scenarios for which the use of MAs provide important benefits are described in [Mag95, MRK96]. MAs can encapsulate management scripts and be dispatched on-demand where needed. An MA can be sent to a network domain and travel among its elements collecting management data, and return with the data filtered and processed. Sending an MA for this task is a substitute to performing low-level monitoring operations and processing them centrally. If the agent is able to extract useful and concise information from raw data collected on each element, the agent size can remain small enough to save bandwidth usage. In fact, this is one of the most advanced arguments for promoting MAs.

In an interesting quantitative study on code mobility, [BGP97] presents a consequent advantage of using mobile agents; when the network administrator can only be connected via an unreliable, costly or lossy link, he can create its MA off-line, connect to the network to dispatch the agent, close the connection and then reconnect later to get his agent back with the results. This principle is actually implemented in Astrolog [SBM97]

to support mobile managers. To enhance the management flexibility, an administrator can connect to the network using his portable computer and dispatch MAs to accomplish management tasks. The connection could even be established on the basis of a GSM link. If a hand-over occurs, the dispatched MA can retrieve its sender's new location and return to him [SM98, Sah99].

The potential application of an MA to NM depends on its type of mobility. The following subsections describe the way that different types of MAs have been applied in different NM applications.

3.3.2 Mobile Agent Frameworks for Network Management

Simply viewed, an MA framework consists of two components: The Mobile Agents themselves and the nodes where they can migrate and execute. There are several appellations synonymous to nodes where MAs execute, such as Mobile Agent Server [GGGO99], Mobile Code Daemon [WPB99], agency [SSS+99, BM99], place [Zap97, BCS99], *lieu* [SM98], or site [FPV98]. We adopt the term *Mobile Code Server* or *MCS* for the remainder of this section. When the MA framework is targeted towards NM, other components are also needed. MAs performing NM tasks need to access managed resources to monitor and control them. Therefore the MA framework has to offer an interface to uniformly access these resources. This interface has also different appellations: Virtual Managed Component [WPB99], services [8], reactive tuples [CLZ99] or simply, Managed Objects [NQKA98].

NM applications require other features such as security, reliability and fault tolerance. The proposed MA frameworks differ according to the features they promote more than the others.

The Perpetuum Mobile Procura Framework The Perpetuum Mobile Procura project [PMP] developed an MA framework which offers the basic functionality required to deploy NM MAs. Every node to which MAs can migrate has to run a Mobile Code Daemon (MCD) that includes a Migration Facility and a Mobile Code Manager. The Migration Facility provides transport facilities to the agents. The Mobile Code Manager manages the lifecycle of the agents present on the MCD. The access to managed resources is supplied by Virtual Managed Components, which provide a uniform interface for the MAs to monitor and control the visited NE. In [WPB99], an MA injection client is also introduced in order to create, deploy and manage MAs in the network.

Several kinds of MAs are defined [Bie97]. Applets, servlets and extlets are single-hop downloadable components where applets are used for COD and servlets and extlets are

used for REV. Deglets are multiple-hop agents but with limited persistence, in that, the MA terminates as soon as its task is over. Finally, netlets are persistent multiple-hop MAs.

MAGENTA MAGENTA (Mobile AGENT environment for distributed Applications) [Sah99, SBM97] is targeted towards mobile user applications where users are intermittently connected to the network with unreliable and expensive connections. Places where MAs can travel are called *lieus*. The framework offers a certain degree of reliability and fault-tolerance. When a lieu is to be disconnected, it broadcasts its related information to all the other lieus so that no more MAs can migrate there. Moreover, each MA keeps a record of abruptly disappearing lieus and communicates it to the last visited lieu. The last visited lieu broadcasts the information on the crash of the disappeared lieu so that the other lieus take account of this information. In case that an MA has to travel to a disconnected lieu, it remains idle on the previous lieu until the target lieu is reconnected back. Moreover, backup copies of agents are stored so that recovery is possible in case of a single crash in the system.

MAGENTA also offers a limited support of security based on access control. Access control uses a *permit* that uniquely defines each MA deployed in the network.

JAMES JAMES is an MA platform for the management of telecommunications networks [SSS+99]. MCSs in JAMES are called agencies. The *JAMES manager* allows the network administrator to have control over all the active agents and agencies. A code server provides a central repository where the MA codes are stored and can be retrieved by the manager. Agencies have memory and disk caches for MA code. This results in remarkably better performance for agent transfer compared to other frameworks. Moreover, JAMES agents have a passive migration strategy according to which the itinerary of each agent is known at launch time. Agencies make use of this property in order to prefetch the agent code from the code server. This leads to increased performance for MA migration.

JAMES uses a checkpoint mechanism to offer fault-tolerance. Agencies states are periodically saved on a persistent medium so that a crash can be recovered and the system can be restarted from the last saved checkpoint. Moreover, agents are saved to a stable storage before being transmitted to the next destination. An agency keeps the saved internal state of an agent until the next checkpoint.

AMETAS The Asynchronous MESsage Transfer Agent System [Zap97, 8] uses the concept of a place as an MCS. Each place offers a mailbox system that allows agents to communicate asynchronously. A place can extend its capabilities by installing services

that can be accessed through a control interface. MAs can therefore access Managed resources after installing appropriate services. Another particular feature of AMETAS is that it allows to define human-agent interfaces through the definition of *user adapters*.

Finally, AMETAS uses a complex security mechanism that allows for authorization, access control, encryption and the definition of security policies.

MAMAS Security and interoperability are the key properties of MAMAS (Mobile Agents for the Management of Applications and Systems) [BCS99]. Security is ensured by secrecy, integrity, authentication and authorization. Secrecy for agent migration and communication is ensured by encryption. Integrity is ensured by a hash function that allows to decide whether an MA has been altered or not. Authentication is achieved by assigning signatures to agents. Finally, authorization is achieved through place-level and domain-level authorization and prohibition policies.

Interoperability is supplied through the support of the OMG MASIF specifications [Obj00]. This allows for the interoperability between multiple MA platforms provided they are MASIF-compliant. Moreover, a software add-on insures the interoperability with CORBA-compliant distributed applications, which allows to interoperate with legacy systems through CORBA gateways.

Places (i.e. MCDs) in MAMAS are grouped in *network localities* that model the physical resources. They can also be grouped in *administration localities* that model the different administrative responsibilities.

INCA INCA (Intelligent Network Control Architecture [NQKA98]) is an open architecture for the distributed management of multi-service networks. Among the main features of INCA are multiple code transfer schemes and agent prioritization.

INCA defines three code distribution schemes. In the case of a fixed itinerary, the code of the agent can be pushed to stations (i.e. MCSs) where the agent will migrate before even the agent is started. This *pull type code distribution* allows to obtain low response times compared to a push type code distribution. *Push type code distribution* is used when the itinerary of the agent is not known at creation time. In this case, a station that receives a new MA has first to fetch its code. Both push and pull schemes rely on a code repository which stores the code of all the MAs that can be deployed in the system. In contrast, *migration code distribution* ensures that the code travels together with the MA. This type of code distribution allows to overcome the scalability problem that may occur with the other two code distribution schemes.

Another original feature in INCA is that the network administrator can attribute priorities to MAs. Three levels of priorities are introduced. The highest level is suitable

for *network control agents* that perform configuration and fault management tasks. The medium priority level is suitable for *service management agents* that install end user services and for *network maintenance agents* that perform occasionally monitoring tasks. Finally the lowest priority is attributed to long term *network monitoring agents* that perform routine and continuous management tasks.

INCA has also other interesting features. First, it ensures reliable and fault-tolerant communications between stations by incorporating transaction capabilities based on those of SS7. Second, INCA allows the monitoring of the agent population deployed in the network. It also provides facilities to launch and control agent execution. In addition, INCA includes a *location service* and a *naming service*, which are used for inter-agent communication. Finally, managed resources are accessed via *managed objects* included in INCA stations.

Grasshopper Grasshopper is the unique commercial product that supports state-of-the-art standards (FIPA's agent standards and OMG's MASIF) in MA technology [BM99]. Agencies in Grasshopper represent MCSs, which are composed of a *core agency* and a set of one or more *places*. Places are runtime environments where agents run. The core agency offers a set of services to support agent migration and execution. The communication service that supports agent communication and migration may use a variety of protocols: CORBA IIOP, Java RMI (with possible use of SSL) and plain sockets. The registration services are included in agencies to keep track of running places and the running agents in each place. Agencies are grouped in *regions* that, by their turn, include a global registration service.

Security in Grasshopper allows to protect both agency-region interactions and agent-agency interactions. Agency-region interactions are secured using X.509 certificates and the Secure Socket Layer (SSL). This ensures integrity, secrecy and authentication. Agent-agency interactions are secured using JDK security mechanisms that allow for authentication, authorization control through access control policies, and protecting the agency and the agents themselves from unauthorized agents.

Grasshopper also includes a persistence service that allows to store agents, places and agencies on a persistent medium. The system can therefore recover in case of crashes or faults. In addition, Grasshopper uses management services in order to monitor and control agents and places. Coupled with an agent location service, the agent system can be efficiently managed.

Grasshopper is MASIF compatible. It is also interoperable with FIPA-compatible agents. This is achieved by using add-ons that implement the FIPA agent services, including communication and agent management.

3.3.3 Mobile Agent Applications in Network Management

Section 2.3 in the previous chapter provided a classification of mobility according to which part of the MA is transported when it migrates and detailed the differences between strong, weak and memoryless migration. It also mentioned that strong and weak mobility are equivalent from a functional point of view. All the functionality provided by strong mobility can also be obtained using a weak MA, albeit with more programming complexity in some cases. Therefore, and still from a functional point of view, there is no need to distinguish between strong and weak mobility in the network management context. More important however, is that memoryless mobility is used for different purposes than those for which strong and weak mobility are used. The following subsection discusses memoryless mobility, while the subsequent subsections assume either strong or weak mobility.

3.3.3.1 Memoryless and Single-hop Migration in Network Management

As a general rule, single-hop migration is useful to encapsulate a function or a service into an MA and to deploy it at a determined location, whereas memoryless mobility is efficient to deploy agents that carry out repetitive yet memoryless tasks on a list of sites. A number of applications make use of these properties.

Two symmetric ways of applying single-hop agents are described in [WPB99]. The authors suggest that in each given network element resides a Virtual Managed Component (VMC) that provides access to the NE management interface. Similarly, the manager side of the NMS requires a similar interface called the Virtual Managed Resource (VMR), which can be viewed as a remote wrapper of the VMC. The VMR can be supplied as a single-hop MA that travels from the remote NE to the NM station, and therefore, to seamlessly enable the management of newly installed NEs. In addition, this allows network component suppliers to transparently use any NM protocol which could even be a proprietary protocol — provided that the VMR has a standard interface to the NMS. The MA can also provide a customized graphical view of the managed component. In a symmetric way, the NMS may require a different management view from that supplied by the original VMC. The NMS can therefore send a single-hop MA to the NE. The MA is responsible for providing the new customized view and ensures the necessary protocol conversion if necessary.

Single-hop agents are also suitable for the implementation of health functions. In [PT99], a set of four types of MAs are identified that perform some basic management functions. Among them, the *daemon agent* is a single-hop agent sent to a network node to locally compute a health function and automatically notify the NM station when cer-

tain thresholds are exceeded. This scenario provides the advantage of saving bandwidth when compared to a remote polling based scenario and therefore, allows to have a fine-grained detail of the health function.

Other possibilities of applying MAs are suggested in [Mag95] and [MRK96]. A service or a network provider can send single-hop MAs to user end-points in order to adapt his equipment to new services. These agents can also achieve other tasks such as user accounting and capturing user requirements. Another scenario described in [HJ97] also suggests to use MAs for immediate service creation and customization.

A further work [CBD99] proposes to manage such MA-based service provision systems by using the same MA technology. Memoryless mobile single-hop agents are introduced to offer management functions covering the phases of service deployment, service utilization, service maintenance and service withdrawal. The approach is to send single-hop agents called Managed Object Agents (MOA) close to the service provision agents which are used for the deployment of telecommunication services. These MOAs provide the instrumentation for monitoring and control of service provision agents. Interface to the management system is provided by Interface Agents while Service Master Agents allow to coordinate the management activities. Finally, Notification Forwarding Agents provide a distributed notification mechanism that serves the other types of agents.

3.3.3.2 Multiple-hop Migration in Network Management

Predefined Itinerary and Passive Migration The typical and most used scenario of applying multiple-hop agents is to deploy an agent that visits a list of network nodes to locally perform management tasks. The aim is to return back with a certain report containing pertinent management information. In this context weak mobility allows the agent to perform semantic compression [BP97], to take decisions based on the past visited nodes and to bring back a report result to the management station. This scenario is proposed in many works to achieve different management activities. The simplest application is to monitor the visited nodes in order to obtain a certain performance report or to detect possible faults. This basic scenario is proposed in [PT99], which defines a *browser agent* that roams the network devices to collect MIB data, and a *verifier agent* that allows to return with a list of nodes verifying a certain condition, e.g. running a certain OS version, or having overloaded CPU. An example of this approach is described in [KH99] that uses MAs for monitoring the conformance of Service Level Agreements in enterprise networks. The paper suggests to use MAs that periodically roam the network to collect SLA monitoring results that are produced by other agents standing close to where user applications are running.

For the purpose of performance monitoring, [GGGO99] describes an MA framework and applied it to perform SNMP polling operations. The paper suggests two migration strategies for periodic polling. In the first strategy, a poller MA successively visits a given list of NEs, polls the required variables at each node, and returns back with the real-time data to the manager where it sleeps until the polling period elapses before it is launched again. In the second strategy, an MA is sent to each NE and remains there for a relatively long interval of time. Each MA performs periodic polling on the NE during that interval before returning back to the manager for off-line analysis of the collected data. For both strategies, it is up to the manager to specify the migration strategy for the MA as well as the node(s) to be visited.

NetDoctor [8] is an agent application built on top of the AMETAS (Asynchronous MEssage Transfer Agent System) agent platform (see Section 3.3.2). NetDoctor dispatches *health agents*, which roam the network and perform tests on visited nodes in order to detect anomalies. If endowed with sufficient decision making and control capabilities, then health agents can automatically initiate actions that repair the detected faults. Another similar application is described in [EDB99] in which MAs are endowed with a rule-based engine that allows to infer and diagnose possible faults on visited nodes. MAMAS (Mobile Agents for Management Applications and Systems) which was presented in Section 3.3.2, also uses the same principle for systems management [CST98].

Active Migration Combined with weak mobility, active migration enables the agent to make decisions on its adopted itinerary. This is suitable for a number of applications such as network discovery and dynamic configuration of networks and services.

In [WPB99] and [SBP98], the authors propose a scenario based either on persistent or non-persistent MAs. Persistent MAs, i.e. netlets, continuously roam the network to discover new devices and to detect removed components. Netlets are suitable for dynamic networks with frequently changing topologies and supported services. A selection process can be added to discovery netlets so that only a certain type of components are discovered, e.g. printers, SNMP-enabled devices, etc. Management applications can benefit from the results of the discovery netlets by registering to notification messages that these netlets generate when a change is discovered.

For specific discoveries in less dynamic networks, non-persistent MAs, i.e. deglets can be used instead. The deglet returns to the management application with the result of the discovery process. Therefore, non-persistent MAs allow to have a customized snapshot view of the discovered network.

In another work, Pagurek et al. [PLBS98] developed a simulation test bed for the configuration of Permanent Virtual Channels (PVC) in heterogeneous ATM networks. The

idea is to dispatch an MA that travels across the network switches, and progressively configures the PVC fragments on each switch. After configuring the PVC route on a switch, the MA travels to the next switch where it uses configuration information of the past switches to correctly configure the current one. The scenario suggests that the agent has to backtrack in case of configuration errors, undo the necessary previous configurations and find out a new path to establish the PVC [1].

Another original application of strongly-mobile multiple-hop actively migrating agents is presented in [AS95], where MAs are used to control traffic congestion in a circuit-switched network. A first class of MAs, called *parent agents*, randomly navigate among the network nodes and collect information about their utilization. By keeping track of this information, they are able to gather an approximate utilization average of the network nodes. Therefore, they are able to identify those nodes which are congested relative to this average. When a congested node is identified, a load-balancing MA (called *load agent*) is created. Its mission is to update the routing tables of the neighboring nodes (using an optimal algorithm) so as to reduce the traffic routed by the congested node.

The success of this MA application inspired other researchers to further develop the ideas behind it by using biologically inspired agents. The work of Minar et al. [KMM99, MKM99] used MAs in order to configure routing tables in a highly-dynamic radio frequency network where nodes are low-power transceivers that may move from one location to another in a two-dimensional space. This results in a peer-to-peer network in which packets require in most of the cases multiple-hops in order to reach their destinations. Nodes contain a routing table that is updated only by MAs. Each MA applies a three-step algorithm. Firstly, the MA looks at all the neighbors of the current node and decides where to go next using one of two strategies: random roaming, or oldest-node first. Secondly, the agent travels to the next node and learns about the traversed link. Thirdly, it updates the routing table of the new node using its own recent knowledge of the network. A population of such MAs allow to ensure a sufficient level of connectivity in the network.

Shoonderwoerd et al. [SH99, SHBR96, SHB97] proposed to use ant-like MAs to achieve load balancing in telecommunications networks. MAs randomly roam the network and put simulated pheromones depending on the distance from the source and the congestion of the adopted route. Basically, the strength of the deposited pheromone decreases during time and particularly when the traversed nodes are congested. Deploying a sufficiently large number of such ant-like agents allows to route calls according to the distribution of pheromones. Simulation shows that such MAs allow to significantly

[1] This PVC configuration application is of particular interest to us since in Chapter 6, we implement the same application using our own skill-based agent architecture, while in Chapter 8 we compare the two approaches.

decrease call rejects in telecommunication networks compared to other more common approaches. An improvement to this work is proposed by Bonabeau et al. in [BHG+98]. A major difference between such work and that of Appleby and Steward [AS95] is that MAs no longer roam the network in a completely random way. Instead, MAs tend to follow those places where the amount of pheromone is higher than in the other nodes.

White et al. [WBP98] applied a similar approach to a different application than routing configuration. Instead, [WBP98] describes an application that locates network faults using multiple interacting swarms of MAs. The principle is to model network services as dependencies upon resources and other services. Four types of agents are defined. *Service monitoring agents* monitor the compliance of service instances to the required quality of service. When significant changes in the service delivery are detected, a *service change agent* is sent to mark the resources on which the service depends. The service change agent marks resources in a way that increases the pheromone intensity when the quality of service has downgraded and that 'evaporates' the existing pheromone in the case that QoS parameters are reestablished. Another type of agents called *condition sensor agents* continuously roam the network and evaluate specific conditions on the visited nodes. A condition evaluated to `true` means that the node has a problem. Condition sensor agents tend to visit more often those nodes with a detected problem. This also enforces the pheromone deposited on such nodes. Nodes with strong pheromone attract a fourth type of agents called *problem identification agents*. A problem identification agent continuously roam the network, attracted by strong pheromones. It has the capability to diagnose and repair a certain pattern of problems. Other types of agents are also introduced for special purposes, e.g. to detect persistently over-utilized network resources.

It is essential to notice that in the above four applications, MAs provided a very interesting tool to model either lightweight moving entities or biologically inspired entities used for NM tasks. The MA metaphor allows to model a self-contained entity that evolves in its environment to accomplish simple tasks according to its *own view* of the network. Of course, one could model such applications using a classical set of non-mobile entities that exchange messages. But in this case, the mapping from ant-like entities would not be as straightforward and natural as with MAs.

3.3.4 Performance of Mobile Agents in Network Management

While most of MA-related research activity tried to evaluate MAs by developing frameworks and applying them for specific NM applications, some research considered other aspects of MA potentials, namely performance and scalability. The leading work in this direction is that of Picco [Pic98, BGP97]. A management task that has to be carried out

on a set of NEs and that involves multiple client/server interactions with each NE is considered. Models of the overall traffic and the traffic around the management station are computed for each of the four approaches: client/server, Code on Demand, Remote Evaluation and Mobile Agents. The impact of semantic compression is also studied in the case of mobile code approaches. The purpose of these models is to provide the network administrator with an objective quantitative criterion to decide which approach is better, given a certain management task and a certain topology of the managed network.

Liotta et al. [LKP99] propose to evaluate the potential benefits of MAs by providing quantifiable metrics based on performance and scalability. MAs are used to perform monitoring tasks in place of classical centralized polling offered by current NM systems. Performance is measured using the generated monitoring traffic and the incurred monitoring delay. Scalability measures the ability to increase the size of the monitoring problem, function of the number of monitored NEs, the polling rate and network diameter (i.e. the network size). These parameters are then computed for a combination of schemes based on monitoring models, MA organization and MA deployment patterns. Possible monitoring schemes consider an MA performing periodic polling. In one scheme, the MA is itself periodically polled by another MA or by the manager. An MA can also periodically notify the higher-level MA or the manager instead of being itself polled. Finally, the third monitoring scheme is based on event-driven alarm generation.

The MA organization can be either flat or hierarchical, and the deployment scheme can be either cloning-based or without cloning. This leads to four possible deployment patterns: flat broadcast with no cloning, flat broadcast with cloning, hierarchical broadcast with no cloning and hierarchical broadcast with cloning.

The purpose of such performance studies is twofold. First, they show how MAs may outperform classical centralized NM approaches, and that MAs are able to solve scalability and performance bottlenecks around the management station. Second, they allow to determine how MAs are best deployed for particular types of NM operations.

3.3.5 Synthesis and Discussion of Mobile Agents

Compared to our original state-of-the-art paper [CCOL98], MA applications in NM widely increased in number. MA frameworks proliferated while largely improving their features in terms of security, fault tolerance, interoperability, etc. Moreover, some MA frameworks aimed at improving the performance of MA migration through enhanced code distribution mechanisms. Other frameworks tackled the problems of MA monitoring and control. All these are signs of the beginning of MA maturity.

The usage of MAs covers many aspects in NM. Single-hop MAs are suitable for the

easy deployment of software, functionality and services. In that, single-hop MAs are used to deploy adaptable instrumentation facilities, health functions, wrappers to the managed NEs, and on-the-fly adaptors for newly installed NEs and services. In this aspect, single-hop mobility is similar to a general-purpose MbD mechanism.

Multiple-hop MAs go further and allow to perform management tasks on a set of NEs in a sequential manner. A multiple-hop MA can perform monitoring tasks on a set of NEs, and benefit from the semantic compression that it can perform while moving from one host to another. Moreover, a multiple-hop MA can encapsulate health functions that are related to a set of NEs instead of a single NE. It can also be used to perform random fault diagnosis and repair by constantly roaming the network in search of anomalies.

Active migration is often combined with a society of MAs, mostly to imitate ant-like behavior. This approach can be used to regulate the network based on the average resulting behavior of a large number of MAs. Although such kind of MA systems is particularly difficult to tune, they are very suitable for relatively dynamic networks, where more-or-less random mobility combined with a large number of MAs allows to easily detect changes related to performance, faults or configuration and to react accordingly.

Finally, there is an ongoing work to evaluate and improve the performance of MAs and to assess different MA implementations and usages, which contributes to providing expertise on how MA frameworks and applications should be designed.

3.4 Reactive, Deliberative and Hybrid Agents in Network Management

Recall from the previous chapter, that according to its architecture, an agent might be reactive, deliberative or hybrid [WJ95]. While a reactive agent acts in situation-reaction or event-reaction manner, a deliberative agent acts according to decisions taken after a reasoning process about the environment. A hybrid agent is a combination of both. The following three sections will respectively treat these types of agents, and the last section will provide a synthesis and a discussion.

3.4.1 Reactive Agents

In Section 3.3.3.2, we discussed from a mobility standpoint the work of Appelby and Steward [AS95] that uses MAs for congestion management. The agents used therein are reactive agents [AS95]. Both the parent agents and the load agents do not use any model of the network they are managing. Instead, they only observe the state of the network nodes and react accordingly. Reactive agents are particularly advantageous for

this application because MAs need to be lightweight not to overload the network by their mobility. The parent agents do not react unless they discover an overloaded node. The decision to react by creating a load agent is taken by simply comparing the currently visited node utilization with the average utilization computed during its traveling. The load agents which update the routing tables decide on the basis of an algorithm that does not imply any reasoning.

Other researches concentrate on building agent-based frameworks for network management and use the reactive architecture mainly for its simplicity. In [SR96], so called intelligent agents are applied to implement distributed management policies. Within this framework, agents are composed of a communication mechanism, a rule base, a solution base and an inference engine (hence probably the appellation 'intelligent agent'). The rule base contains rules that match perceived events corresponding to anomalies with the reactions to be taken. Reactions, called services in the referenced paper, are also described by rules within the solution base. Therefore, the reactive rules are of the form `<event-Id,reaction-Id>`. This framework was used to achieve usage-based accounting.

Another framework is described in [SC96]. A generic agent architecture is defined and built using April++, an Object-Oriented extension of April [MC94]. The agent is organized in units that communicate (*inside the agent*) via a blackboard mechanism. The units are organized as peer layers which means that they execute concurrently. The local information of the agent is centralized in an information database. At the same time, the agent has a link mediator, which is an interface to the physical components of the network, and a head that insures the communication with other agents. New functions can be added to the agent by acquiring new units. According to the classification of layered agent architectures, the agents used in this framework are horizontally layered agents. This model has been applied to provide agents that are suitable for multi-service network management. For this purpose, the agent is endowed with a customer unit to interact with the user, a service unit that honors service requests and a fault handling unit that deals with the faults that may occur.

Lissajoux et al. proposed a MAS composed of reactive agents to solve the optimization problem of antenna parameter setting in radiomobile networks [LHKC98]. Simply put, the antenna parameter setting problem consists in optimizing the coverage provided by a number of antennae, while minimizing the interference between meshes, where a mesh is a surface unit covered by a particular antenna. An acceptable solution to this problem must also satisfy other constraints, e.g. to provide sufficient handover without having uselessly large handover surfaces. In this system, antennae are modeled using reactive agents sensing and acting on their environment, which is composed of their respective meshes. Agents have a set of parameter adjusting behaviors that correspond to particular

situations on the covered meshes. These situations correspond to a lack of coverage, excessive interference, a lack of handover and excessive handover surfaces. For each of these situations, the agent reacts proportionally in order to bring back the system in an acceptable situation. In the case that none of these situation apply, the agent activates the default behavior that consists in increasing the coverage. Moreover, and in order to avoid the effect of oscillation that occurs when the neighbor agents react simultaneously to the same changes, agents use an *agitation factor* that defines the probability for an agent to react to an event. The agitation factor decreases with time, therefore minimizing the joint reaction of neighbor agents. This case study provides a proof for the ease of design of an MAS that directly models a distributed problem.

3.4.2 Deliberative and BDI Agents

Two applications can be stated as deploying deliberative agents. The first is described in [WT92], in which an interesting agent framework is applied to control a VPN service. Telecommunication services such as VPN rely on two separate layers: The logical layer maps the view of the customer's network, which is built using the physical layer that maps the actual physical network of the operator. Here, agents are used to automate the negotiation that used to occur between the service provider and the service customers when customers ask to change the service parameters or to repair network faults. A *customer agent* has knowledge about the logical structure of the VPN and its usage, while the *provider agent* knows both the logical and physical implementation of each customer's VPN. Thus, agents have a model of the external world on which they act. Precisely, a customer agent knows for each trunk (i.e. logical link) its capacity and its utilization, whereas the provider agent knows for each logical trunk the physical links upon which it is configured.

When a network fault occurs, the VPNs based on the faulty network element are affected. In this case, the customer agent first asks the provider agent to repair the fault. In many cases, a complete and immediate repair is not possible and a negotiation-based cooperation is started. Customer agents try to find intermediate solutions based on their knowledge of the utilization of their respective trunks, and suggest partial solutions to the provider agent by updating the logical structure of their VPNs. The provider agent coordinates these solutions and makes the necessary updates in order to reach an acceptable configuration. Both customer agents and provider agents use logical reasoning based on their models of the network and on their beliefs on its elements. Agents developed in this application make use of both, Shoham's Agent Oriented Programming [Sho93] and planning functions provided on a lisp-like planner called PRODIGY [FV94].

The second application is carried out within the MANIA project (Managing Awareness in Networks with Intelligent Agents) [OL95]. During his PhD Thesis, Oliveira [Oli98] developed a BDI agent-based approach to manage application-level quality of service in enterprise networks. Agents are structured in beliefs, desires, intentions, goals and commitments. A first part of the beliefs describe the real-time state of the network, e.g. 'the printing service is being highly solicited at the present time'. A second part contains the historical behavior of the network. The historical behavior is used to achieve some kind of learning about the dynamics of the network such as deducing that the printing service is highly solicited every day from 10 to 11 am. A third part of the beliefs translate the states of the services provided over the network, e.g., the minimal response time the NFS server can ever have, or the maximum client number a web server can handle at a time. Finally, the agent may have beliefs on the user application contexts, i.e. user requirements in terms of QoS.

The agent desires consist of two parts. The first part corresponds to the requests that the agent could not satisfy, such as when a certain user requires a video connection while there is no available bandwidth (according to the agent belief). The second part consists of motivation policies. The network administrator may want to motivate the agent to give a certain priority to some project members because they have a constraining deadline.

When receiving a user-application context (which describes thre resources that the user needs and the required QoS parameters), the agent translates it into a set of goals. Goals are independent from the system state. For example, a goal may state to 'actively monitor the NFS server'. These abstract goals are then mapped into intentions. Intentions take into account the current state of the system, the available means to achieve the goals (e.g., MIBs, testers, etc.) and the possible constraints that may apply.

Finally, and in the same way that the administrator could motivate the agent, he is also able to specify obligation policies that form the commitment part of the agent.

3.4.3 Hybrid Agents

As a general rule, hybrid agents are able to exhibit both reflexive, therefore prompt behavior, as well as deliberative, therefore long-term behavior. In general, such kind of agents are applied both, to perform local management functions within a network domain, as well as to perform global management functions using their deliberative capabilities. Some agent-based network management approaches followed this principle.

An ATM network is very dynamic and managing it centrally will not be appropriate because managed data quickly go out of date. But local management is not appropriate because it lacks an overall view of the network. Here, the hybrid agents are particularly

interesting. They may combine both local real-time management via the reactive behavior, while the deliberative behavior is concerned with cooperating with other agents in order to achieve global planning and coordinate high-level tasks. HYBRID, proposed by Somers [Som96b, Som96a], is an agent-based framework for the overall management of ATM networks that makes use of hybrid KQML-speaking agents. HYBRID uses a hierarchical agent system in which authority is transferred from high-level agents to lower-level agents. From an agent architecture standpoint, HYBRID provides an inheritance hierarchy of agent types. The root agent is the *basic agent* that only "supports KQML front-ends and conforms to some minimum behavior". From this basic agent, several other agent types are inherited with increased capabilities. For example, a *system agent* has "low-level skills" which are scripts called in response to events that occur in the managed system. Therefore, the system agent can be considered as reactive. The *task agent* is able to manage his concurrently running tasks. The *Intelligent agent* which is in the bottom of the inheritance exhibits both reactive and deliberative behaviors. The reactive behavior is indirectly inherited from the system agent. The deliberative behavior is supported by worldview knowledge bases, procedural skill bases, a situation assessor, and an agenda. The worldview knowledge bases include facts on the other agents, on the agent itself, and on the managed components. Procedural skill bases include low-level skills for the reactive behavior and high-level capabilities such as plans and negotiation skills. The situation assessor determines the focus of attention of planning tasks according to the current agenda and the agent goals. Finally, planning can be used in combination with skill-matching in order to plan for agent actions.

Agents in HYBRID are designed according to their roles that help deciding whether a reactive agent is sufficient or deliberative capabilities are needed. The use of a common communication language ensures that agents with different levels of intelligence are able to coordinate their activities.

Another application of hybrid agents is achieved by Wagner and is based on the concept of *Vivid agents*. Vivid agents [Wag96] comprise both a reactive and a deliberative part. However, the reactive part of the agent is not hardwired within the agent. Instead, the deliberative part may dynamically change the reactive behavior. *Reagents* are particular vivid agents with no planning capabilities.

Reagents are applied to perform the distributed diagnosis of distributed systems [FMNS97]. The network is partitioned into physical domains, each domain having its own diagnostic agent. The agent has a detailed knowledge model of its domain and minimal information about the neighboring domains, mainly the address of their respective agents. The multi-agent system applies a distributed version of the Model Based Diagnosis (MBD). The principle is that when an agent detects a fault, e.g. a lost connection,

it starts by performing a local diagnosis of its domain. This is the deliberative behavior of the agent, since it runs MBD reasoning according to the model of its domain. If it finds no local fault, then it sends the resulting observations to the neighbor agent. The latter performs the same procedure. If it finds the faulty element(s), it reports it to the first agent. If the first agent receives the fault observation from another agent, then it must ensure that the other neighbor agent is informed by forwarding the report. This is a reactive behavior in which the agent does not perform any reasoning.

In [HB98a, HBL99], Hayzelden et al. describe a heterogeneous MAS for the dynamic management of Virtual Path Connections[2] in ATM networks. Instead of using hybrid agents combining the reactive and deliberative behavior, the proposed architecture, Tele-MACS (for Telecommunications Multi-Agent Control System), relies on a set of reactive agents under the control of distinct deliberative agents. Reactive agents are located in the *control plane layer*. The control plane layer includes agents with detailed, but local knowledge and that operate at a fast time scale. Deliberative agents are located in a *management plane layer*. The management plane layer includes agents with global abstract knowledge and that operate at a slow time scale. The whole architecture is inspired from the subsumption architecture, in which competences are organized in levels where the higher levels build new functionality upon the lower levels and have control on their function. This principle is applied in Tele-MACS in an elegant way. The approach is to let deliberative agents in the management plane layer provide the reactive agents in the control plane layer with tailored views of the world using the belief suppression mechanism [Hay99].

3.4.4 Synthesis and Discussion of Agent Architectures

The first thing to note is that few papers supply the reader with information about the agent architecture. There are several possible reasons for this. The first reason is that many papers only present scenarios and have not yet reached the point of designing multi-agent systems. The second reason is that for some agent applications, the agent architecture is of major concern compared to the description of high-level interactions. Finally, in some applications where agents are very simple in functionality, there is no need to adopt a specific agent architecture that might seem as an overkill regarding the simple agent behavior.

What is interesting to note, is that reactive agents are suitable for local and real-time management operations. Moreover, a reactive architecture is the most suitable to adopt when building MAs. The reason is that a reactive architecture can be very simple and

[2] Not to confuse with the management of Permanent Virtual Channels

therefore, very lightweight compared to other common agent architectures. Layering can also be used in reactive architectures and leads to very modular agents as it is the case in [SC96]. Finally, reactive agents acting in stimulus-reaction way have been adequately used to implement management policies, for which a rule-based approach is particularly suitable.

On the other hand, deliberative agents are much more powerful from a problem-solving and logical reasoning point of view than reactive agents. This makes them require much more resources, and makes their response time rather long. For this reason, they are suitable for complex but not time-constrained tasks. Deliberative agents are more suitable to govern the strategic function of the NMS. The work of Oliveira [Oli98] showed the great potential of BDI agents to model a complex control problem in NM: managing application-level quality of service.

Furthermore, the idea of using a hybrid agent to take advantage of both architectures is very interesting. The concept of vivid agents is a good basis for implementing this idea. Vivid agents offer increased flexibility in agent design since their reactive behavior could be updated at runtime. This could be applied for example to support complex learning algorithms where the agent, still having timely reactions, has its behavior improved in time.

A very interesting substitute to using hybrid agents is a heterogeneous multi-agent architecture, i.e. a multi-agent system with both reactive and deliberative agents with the latter governing the behavior of the former. A heterogeneous agent system provides the same benefits as hybrid-agent systems. This was proven and validated by Hayzeldan's successful Tele-MACS architecture [Hay99].

3.5 Agent Communication Languages for Network Management

The underlying communication mechanism is not equally addressed in different agent-based NM approaches. While in [FMNS97] the communication protocol is not emphasized (simply because there is no need for more than simple message exchange), [RU95] has already distinguished between message syntax and semantics. In [WT92], two types of messages, namely `INFORM` and `REQUEST` messages were sufficient, whereas in [SC96], messages are identified using message patterns. Some other works define their own application-specific agent communication mechanism such as in [Bus96, BG93]. Most of the other related works have opted for KQML or FIPA ACL.

In the MANIA project [OL95] (see Section 3.4.2), agents use KQML for the message

syntax and a KIF-like Goal Definition Language to exchange management goals. Benech [Ben99] developed COBALT, which is a framework for cooperative NM applications based on a CORBA implementation of KQML [BD97]. Finally, in [Som96b], an extended version of KQML is used. The added performatives are `PROPOSE`, `COUNTER-PROPOSE`, `ACCEPT`, `REJECT` and `PROBLEM`. They are introduced to support the negotiation and tendering processes.

While in [HD98] an agent communication language inspired from FIPA ACL is used and is implemented using LISP, the work in [AAC+99] uses a subset of FIPA ACL combined with XML as a content language. Moreover, a mailbox mechanism is used in [AAC+99], which makes the agents to communicate asynchronously. In [BBB+00], it is suggested that the combination of FIPA ACL (Agent Communication Language) with XML results in a powerful mechanism to tackle interoperability in NM and service deployment. Another approach is adopted in [HB98a], which uses multiple levels of abstractions. At a design level, the agents exchange FIPA messages. At an implementation level, these FIPA ACL messages are mapped to messages with simpler syntax. In their turn, these messages are mapped to Java RMI. An asynchronous mailbox communication system is used also.

Finally, indirect communication through the environment is implemented in some applications. Particularly, those applications using ant-like agents use the concept of a simulated pheromone that the agents can either deposit, or evaporate. Another work described in [CLZ99] suggests the use of Managed Objects modeled using tuple spaces in order to enable agent communication.

Therefore, there is currently no consensus for a common agent communication mechanism. Every agent-based approach to NM adapts inter-agent communication to its own specific needs, according to whether this application targets interoperability, efficiency and performance, or support for complex agent negotiation and coordination.

3.6 Multiagent Cooperation for Network Management

Cooperation is one of the most important properties in MASs [DFJN97]. In NM, cooperation could have sensible benefits in distributed NMSs. We describe in the following sections cooperation mechanisms used in agent-based network management approaches, progressively starting from no, or simple coordination towards more elaborated cooperation.

3.6.1 Stand-alone Agents

According to the taxonomy of Nwana [Nwa96], an interface agent does not cooperate, but is mainly learning to smartly assist its user. An application of an interface agent as defined in [Mae94] for network supervision is presented in [EDQD96]. The agent has to process a large amount of alarms and events by filtering them and relating each notification to its context. Such interface agent is completely stand-alone.

For this application of interface agents, an on-line knowledge acquisition is adopted. Using the chronicle model (see [Gha94]), the agent is able to perform temporal reasoning on the notified events and to automate management tasks. A chronicle is a set of (correlated) events terminated with an action that must be taken when the chronicle is matched. A chronicle is expressed using two formula types. The `hold` formula expresses that an attribute holds its value over some interval. The `event` formula expresses a discrete change of an attribute value.

While "looking over the shoulder" of the human administrator, the agent *Chronicle Recognition System* identifies chronicles and associates each one to the action taken by the human network administrator. A *Learning System* evaluates the identified chronicle by comparing it to the known ones. A first step would be to store the chronicle in an unconfirmed chronicle base. After reaching a certain consolidation threshold, or being confirmed by the human, the chronicle is transferred to a confirmed chronicle base. Therefore, in this application, the agent learns from the user either by observing him or by receiving explicit instructions.

Another application using stand-alone agents is described in [dRW97]. In order to provide a framework for proactive fault management, a rule-based agent implemented using Prolog is used to detect and diagnose network faults. Rules are derived from baseline values on network parameters obtained by monitoring the network behavior over a long period of time.

It is easy to notice that individual stand-alone agents are not widely used, certainly because network problems are distributed in nature, and therefore, modeling them with agents is likely to lead to cooperating agent systems. Interestingly, the conclusion of [EDQD96] mentions that cooperation between the chronicle recognition interface agents is planned for future study.

3.6.2 Coordination Without Direct Communication

As described in Section 3.3.3.2, the MAS used in [AS95] is based on agents that do not directly communicate. An agent may be aware of the other agents only by observing the system state. Each time a parent agent visits a node, it leaves some information therein

such as its identity, its age and the date of its visit. By reading this information in every visited node, the parent agents are aware of their number in the network. If this number is excessive, then the youngest parent agent will terminate. Furthermore, in determined nodes, static processes continuously check if all the parent agents are still alive and did not crash. If a very long period has elapsed from the last visit of a certain parent agent, then the static process will conclude that this parent has crashed and will replace it.

In addition, when a load agent is launched on a node, it creates a record with its identity and its start time. When it finishes updating the routing tables, it returns to the first node and registers its same start time again and then terminates. Therefore, any parent agent that visits that node is able to know whether the load agent is still active or not. It is also able to detect that the load agent has crashed if the start time has not been written for the second time after a long period. In this case, the parent agent creates another load agent to replace the first crashed one. The application is therefore a good example that shows how the overall multi-agent system goes beyond the capabilities of all the reactive agents when taken individually.

Similarly, the other applications [BHG+98, SHB97, WBP98, MKM99] using biologically inspired agents use the concept of pheromone to let the agents coordinate their actions. This relies on a simple idea, namely that agents are more attracted to nodes where the concentration of pheromone (or a certain type of pheromone to which particular agents are sensitive) is higher. This is the unique coordination mechanism that allows to obtain an intelligent emergent coordinated behavior out of the agent population.

The application of Lissajoux et al. [LHKC98] for the antenae parameter setting (Section 3.4.1) also uses indirect communication between agents. Agents, that model antennae in this application, can sense only their covered meshes which supply them by all the information they need for their operation. The mechanism used to coordinate the actions of the agents is to use an *agitation factor* that makes the probability of an agent to respond to a certain situation decrease over time.

When there is no direct communication between agents, a lot of distribution-related problems are avoided. Firstly, the coherence problems which result, say from an agent sending messages to a no-longer-existing agent are radically avoided. Furthermore, all the agents equally share the same information and no mechanisms are needed to preserve coherence. Secondly, the agents are able to coordinate their tasks in a simple way. These properties give the MASs thus designed a high degree of robustness and graceful degradation. These features are rarely considered in other applications.

3.6.3 A Primitive Cooperation

In Section 3.4.3, we presented an agent-based system implementing a distributed version of Model-Base Diagnosis [FMNS97]. In the case of a fault, the agent performs local diagnosis and if it finds no faulty element, it asks the next agent to perform diagnosis in its corresponding domain. One may consider this as a simple reactive cooperation for at least two reasons. The first reason is that the underlying communication mechanism is itself primitive. The second is that this cooperation is hard coded, and the agents have not decided how to carry it out. However, and referring to [DFJN97], agents are cooperating since they participate in achieving a global (though implicit) goal which is to identify the faulty element in a large network.

3.6.4 Self-interested and Negotiation Agents

Self-interested negotiation agents seem to be very suitable for some network management activities, especially for service management. Most of the papers dealing with service management refer to this kind of agents. Magedanz [Mag95] shows that these agents are adequate for the provision of high-level services in an open electronic market of telecommunications services. A good starting point for covering this trend is the FIPA Network Management and Provisioning application draft [FIP97], which defines three kinds of agents, namely the Personal Agent (PA), the Service Provider Agent (SPA), and the Network Provider Agent (NPA). The latter is responsible for the provision of the network resources and elements which are necessary for the service implementation. Based on these network resources, the SPA is responsible for the provision of the services with the expected quality. The PA is a kind of Personal Digital Assistant [Mae94] that assists the user in defining his requirements for the application needs with regard to the user's preferences. It then has to negotiate these requirements with different SPAs in order to find out the best provider with the best service in terms of maximum quality and minimum cost. When a connection is about to be established, the PA has to commit the local resources of the user and to configure his equipment according to the established connection.

The SPA has to catch the user requirements, and to identify the necessary services and to map the user requirements into these services parameters. It then negotiates with the NPAs to select the best network provision, again in terms of maximum quality with minimum cost. Feedback with the PA may occur in order to conclude with the best arrangement both with the network provider and the user.

Finally, the NPA gets the SPA's specifications and translates them into network re-

quirements in terms of bandwidth, jitter, etc. It may also have to negotiate with other NPAs, mostly in the case that multiple network domains are involved.

Most of the work based on agent negotiation makes use of such scenario (or similar ones) in which the interacting parties include Customer Agents and Service Agents. According to [HB99], Busuioc's work was among the earliest to use agents in telecommunications problems. [Bus96] describes a flat agent architecture for the management of services in a mixed mobile-fixed network. The Contract Net Protocol is used as a negotiation-based trading mechanism to enable agent cooperation. Agents involved in this architecture are Customer Agents (PA in the FIPA terminology), which ask for the establishment of services from Service Agents (i.e. SPA in the FIPA terminology). Service Agents may themselves negotiate among each other according to their attributed roles and domains.

Calisti et al. [CWF99] proposes a software architecture that integrates FIPA scenario with existing TINA and TMN frameworks. In [CF99b], FIPA compliant agents are also suggested to automate the negotiation between multiple NPAs in order to provide a dynamic solution for the inter-domain demand allocation (IDA) problem. IDA consists of the establishement of cross-domain connectivity with specified QoS. The network provider interworking (NPI) paradigm [CF99a, CFF99] relies on a distributed set of agents that model different network domains. These agents apply distributed constrain satisfaction in order to provide an efficient solution to the IDA probem.

Ruzzo and Utting [RU95] introduce User Agents to define customized services. New services can be customized by supplying the agent with the user policy. The UA is then able to make the best choice between what can be achieved as suggested by the fall-back of the called agent, and what the service provider offers. The caller and the called agents may also negotiate in order to reach an agreement that satisfies both users policies, i.e. preferences and constraints. To support user mobility, end-point agents are also introduced. Mobile-user agents have to negotiate with end-user agents to get the permission to access the corresponding end-points and establish communications.

Another similar work by Griffeth and Velthuijsen [GV93] uses negotiating agents to handle feature interaction problems in value-added telecommunication services [GV94].

Other scenarios suggest implementing new services in agents. [HJ97] suggests that while services are user-customized by PAs, new services can be created within service agents in a service-level agent environment. Service agents encapsulate network specifics and each one is dedicated to one service that can be constructed over other service agents. Similarly, [MRK96] presents a scenario where MAs are sent into an agent environment on the services host, where they can offer new services by using the legacy

services and/or by exploiting other mobile agents' facilities. In both scenarios, the agents that have to provide new services have to negotiate with the already existing agents in order to come out with an agreement to implement these new service.

Negotiation is also adopted in HYBRID (Section 3.4.3) for managing ATM networks. Agents are organized within a hierarchy of authorities. Each authority is responsible for some delegated resources and exports "performance indices" [Som96b]. When an authority (or precisely, a service agent inside the authority) receives a user request with a certain Service Level Agreement, it commits the necessary directly managed resources and then decides which of the sub-authorities is best eligible to route the request through. This decision is based on a *tendering process* based on the performance indices declared by sub-authorities. Moreover, a *negotiation process* is initiated between the service agent and the *customer agent* each time the SLA parameters for a given service need to be refined, e.g. when a network failure occurs. The tendering process is supported by a set of four performatives: `PROPOSE`, `COUNTER-PROPOSE`, `ACCEPT`, and `REJECT`. The negotiation process also uses four performative: `PROBLEM`, `COUNTER-SOLUTION`, `ACCEPT`, and `REJECT`. Both processes are based on finite-state machine modeled dialogues layered on top of the extended KQML.

The approach adopted in [HD98] uses negotiation for bandwidth allocation of Virtual Path Connections in ATM networks. The idea is to have agents associated to physical links that distribute bandwidth to agents associated to VPCs using those links. (This is similar to the model proposed by Hayzeldan et al. [HBL99, Hay99, HB98a].) Agent negotiation is based on a communication language similar to FIPA ACL. In the case that a VPC is congested, links agents (i.e. agent associated to physical links) have to agree on the amount of bandwidth to be allocated. This amount cannot exceed the spare bandwidth on the most used link. This negotiation is based on `INFORM` messages followed by a `REQUEST` message from the link agent with less spare bandwidth. In the case that there is a link with no spare bandwidth to be reallocated for the congested VPC, one of the VPCs on this link has to be rerouted through other links. Again, negotiation takes place between the VPC agents (i.e. agent associated to VPCs) in order to determine the VPC with the most allocated bandwidth. In this case bidding is started between the VPC agents using the `PROPOSE` performative, i.e. each agent broadcasts a `PROPOSE` message containing its bid. The agent responsible for the VPC with the highest amount of allocated bandwidth will produce the highest bid and will therefore commit to rerouting.

It can be noticed that negotiation within a group of agents is costly especially when multiple stage negotiation is allowed. This is due to the negotiation messages, which have to be broadcasted to all the agents participating in the negotiation. Such problem should be carefully handled as it might lead to scalability and synchronization issues.

3.6.5 Market-Based Cooperation

In the context of ATM, where management decision-making must be very prompt, Gibney et al. [GJ97, GJ98, GJVG99] demonstrate how the use of *single-shot sealed bid-type* auction can be advantageously used for Connection Admission Control and call routing. In this restricted type of auction, an agent must commit to his offer as soon as this offer is selected in an auction. This yields to an efficient system in which decisions can be made in a limited number of steps compared to the case where negotiation is tolerated. Three types of self-interested resource-bounded decision-maker agents are used. *A link agent* models a link in the network and is a seller of the bandwidth available on that link. *A path agent* is responsible for bandwidth allocation of an end-to-end path. Path agents act as bandwidth buyers from link agents and connection sellers for *call agents.* Call agents represent the interests of the callers (network users). Two market institutions are installed. In the *link market*, path agents compete to gain the right of using slices of bandwidth from individual links. In the *path market*, path agents try to sell slices of bandwidth on the entire paths in order to establish calls required by the call agents. Interestingly, agents are responsive to the success ratio of their bidding and adapt their proposed prices accordingly: When a resource is scarce, bidders increase their proposed prices to maximize the chance of buying and vice versa. Furthermore, path agents are proactive: They buy bandwidth from links in advance to handle the predicted connection requests that call agents would ask for. The obtained MAS exhibits similar performance to the centralized approach. But the main advantage according to the authors is that the approach results in a scalable and responsive system in which the local action of agents is combined with the social interactions using a marketplace metaphor. However, from our point of view, there is still the need for an additional entity that models the marketplace (market institution) since agents cannot achieve transactions by themselves. This can be viewed as a weakness point, at least in terms of scalability in the probable case that leads to a centralized entity.

Miller et al. [MKH+96] describe a bandwidth resource allocation system for a video server based on an auction market mechanism. The system is based on two segregated entities: the *auctioneer* and the *bandwidth deliverator*. The auctioneer accepts bids from clients indicating the required QoS parameters and the offered price. The bandwidth deliverator allocates the necessary resources to clients with winning bids.

Unlike the above approaches, Kuwabara et al. use an equilibratory market instead of bidding and auction mechanisms. In [KINS96], they study a market-based approach for distributed network resource allocation in which “resource prices are determined by their associated seller based on the demand for the resource”. The principle is that the re-

source seller increases the price as resource demand increases, and lowers it as resource demand decreases. Symmetrically, resource buyers decrease the amount of resources required as the price increases and vice versa. Another key assumption in the equilibratory market is that communication can only occur between a buyer and a seller, i.e. a seller never communicates with another seller and a buyer never communicates with a buyer.

As can be noticed, all the above works rely on a centralized institution to model the marketplace. In [WFFC99], it has been noted that such centralization leads both to scalability and synchronization problems in the MAS. Jennings and Arvidsson paid attention to this problem and proposed to use distributed auctions. In [JA99], a *cooperative market* is proposed to allocate processing capacity in Intelligent Networks. *Quantifier agents* associated to Service Control Points sell tokens representing service request processing load to *allocator agents* associated to Service Switching Points, thus allowing them to accept service requests. The process of token allocation uses a three step auction algorithm. Firstly bids are accepted from quantifier and allocator agents. Secondly, the auction process is run in a way that maximizes a common good expressed by a gain-to-cost ratio. Finally, token assignments are reported to Allocators. In order to avoid the scalability problems of a centralized auctioneer, each quantifier agent runs its own auction, and each allocator splits its bids into many parts that are submitted to different quantifiers.

3.6.6 Delegation-based Cooperation

As opposed to the MbD paradigm presented in Section 3.2 which only deals with script (or at the best, functionality) delegation, we address here the delegation between software agents. This is a higher-level delegation because agents delegate abstractly specified tasks instead of low-level operations.

The agent framework described in [SR96] (see Section 3.4.1) uses the delegation of management policies expressed in logical rules. High-level policies are delegated to agents. These policies are then mapped into lower-level policy rules and stored in the agent's rule base.

The MANIA project (see Section 3.4.2) also makes use of high-level delegation, where tasks are specified using abstract high-level goals using the Goal Definition Language . An agent may fail in achieving a goal mainly because it lacks the necessary resources. It therefore delegates that goal to another agent. Importantly, only the goal and not the associated intended plan is delegated. The delegatee agent rebuilds the entire intentions for that goal, and these intentions may differ from those built by the delegating agent due to different beliefs, commitments and skills of the two agents.

Therefore, agent-based delegation makes use of higher-level concepts than the sim-

ple delegation of management scripts. Agent-based delegation relies on management policies, or abstract goals, without specifying how policies need to be refined and executed, or goals to be achieved. This results in more concise yet efficient and expressive communications between the agents.

3.6.7 Synthesis and Discussion of Agent Coordination

We covered in this section all the levels of agent interaction ranging from stand-alone agents to delegation-based cooperation. We showed that even though an important advantage of agents results from their interaction, stand-alone agents can be used to develop interface agents that learn to automate alarm correlation and fault repair. They also can be used to individually learn network usage patterns to proactively detect performance degradations.

Indirect communication and simple coordination can be sufficient in other cases, for example to take advantage of the emergent behavior of a large society of agents. This kind of basic coordination was applied for distributed fault diagnosis and to globally regulate the network behavior.

An important approach to agent coordination is self-interested and market-based coordination. Both of these approaches use agents that represent different independent, possibly competitive parties. Such agents have been applied in a variety of cases, namely, service provision, service customization, solving the feature interactions in telecommunications services, resource allocation and reallocation, etc. Moreover, market-based approaches enable the open trading of services. They also allow to provide auto-regulated environments for service and resource allocation and trading.

In addition, we showed how agent delegation can be used to specify cooperation problems at a high-level of abstraction based for example on goals and policies. This is very interesting in NM when communication between cooperative parties is expensive or sensitive to performance and traffic bottlenecks.

Lastly, we notice that in all the cited applications, cooperation mechanisms are completely encoded inside the agent behavior. This translates the practical point of view of using agent technology in network management applications, since deliberative cooperation is hard to design, implement and verify in a real application. Therefore, a more pragmatic approach would be to endow the agents with the necessary cooperation patterns that they may use when appropriate situations are encountered.

3.7 Conclusion

After having shown in Chapter 2 that the term "agent" hides a multitude of approaches and techniques to build agent systems, we proposed in this chapter to survey agent applications in the NM field, in order to help deciding which kind of agents is best suitable for NM purposes.

We started by surveying the Management by Delegation paradigm. We showed that the aim from this paradigm is to enhance the distribution in NMSs, and to introduce a certain degree of flexibility in achieving NM tasks. The idea of bringing processing capabilities close to the managed NEs was certainly one of the major motives that pushed the research of software agents in NM. MAs inherit many of the properties introduced by MbD. In that, single-hop MAs are used for the seamless deployment of processing capabilities, such as health functions, protocol wrappers, network services, etc. Multiple-hop MAs are proposed for different NM tasks that involve a group of NEs sequentially visited by MAs. Both passive migration and active migration are used. Active migration is used when fully autonomous MAs are required, as it is the case of biologically-inspired agents. Therefore, different combinations of mobility types are deployed for different kinds NM applications.

We can draw similar conclusions for the other aspects of software agent. Different types of agent architectures are used for different management purposes. The experimented architectures range from very simple reactive agents to complex hybrid agents with a dynamic reactive behavior controlled by a deliberative layer. The work of Hayzeldan perfectly illustrates the suitability of using reactive agents for local prompt control and deliberative agents for strategic long-term control.

Similarly, different types of agent communication mechanisms are experimented in different contexts. Some works favor performance and use proprietary communication messages, while others favor high-level structured communications for expressiveness and interoperability purposes.

Finally, a rich variety of agent interaction, coordination and coordination mechanisms are used to tackle different NM problems. Simple coordination mechanisms based on a small set of exchanged messages, as well as sophisticated cooperation and market-based mechanisms are all useful in NM and no single approach can be advantageously used in all the situations.

We generalize by claiming that in order to take the most advantage of software agents in NM, the whole set of agent techniques should be considered. Adopting an agent approach that is restricted to a certain aspect of agent techniques, and therefore excluding a whole range of other agent aspects, will largely reduce the potential benefits of software

agents in NM. A particular type of software agents may be advantageous for a certain type of NM problems, while not being necessarily suitable for other NM purposes.

Consequently, an agent approach targeted to NM should not commit to a specific type of agents. Instead, a NM-oriented agent architecture has to be open to provide and deploy different types of agents for different NM tasks. Agents should be endowed with the necessary capabilities according to the management requirements. The agent architecture should be sufficiently *generic* to allow to organize agent interactions in the most efficient and suitable way for the target NM application.

It appears that the solution to successfully use software agents in the overall field of NM, is to contemplate a generic agent architecture that is open enough to provide customized agents, and that does not commit beforehand to any particular type of agents, thus excluding the other types.

The purpose of the next chapter is to further motivate the choice of a generic agent architecture, and to propose such NM-oriented agent architecture.

Chapter 4

A Skill-based Agent Architecture for Network Management

"... a more advanced implementation should allow for new methods, procedures and strategies to be dynamically added to the agent or automatically 'imported' by the agent itself following an autonomous decision." [Bus96]

4.1 Introduction

Networking technology is evolving at an amazingly rapid pace. As the network is becoming a sensitive integral part of any modern company, the network also becomes an important factor that affects competitiveness and productivity. Therefore, the role of an NMS is becoming increasingly crucial to insure the constant reliability of the corporate network and to satisfy the contracted service level agreements. Given that business strategies have to frequently accommodate with the market fluctuations, a successful NMS imperatively needs to be easily adaptable to reflect the changing business strategies. Therefore, the first concern with our agent architecture is a high degree of *flexibility* and *dynamism* of agent based management applications.

On the other hand, the previous chapter showed that different agent techniques are suitable for different NM tasks and problems. Therefore, any agent architecture that is committed to a specific type of agent architectures and properties would potentially limit its field of application to a certain kind of NM problems. Accordingly, our approach is to build a generic agent architecture in which agents are potentially open to incorporate any agent-oriented technique depending on the management application to be dealt with.

Therefore, another important concern we had when designing our agent architecture is *genericity* to support multiple agent-oriented techniques and mechanisms.

The purpose of this chapter is to develop these requirements and to detail the agent architecture which we designed for NM purposes. We also provide the skeleton of agent-based application developement lifecycle. This is followed by some illustrative examples of building primitive agent support services.

We note that although our agent architecture is not BDI-oriented, we make use of BDI terminology to help better describing its components. As pointed out in Section 2.2.6, attributing mental categories to agents is a natural approach to easily describe agent architectures.

4.2 Design Principles

This section elaborates on the implications of the identified requirements: flexibility, genericity and dynamism.

4.2.1 Flexibility

In a flexible management system, the network administrator should be able to seamlessly initiate new management operations and introduce modifications on currently running tasks. This is necessary to be able to reflect the enterprise business strategies and adapt the network accordingly. For example, an abuse in using network resources may lead the network administrator to set up a usage based accounting. The accounting function may not have been planned in the initial requirements for the management system. In this case, the administrator has to be able to introduce the new accounting function and, importantly, without disturbing the other management activities. This requirement imposes a highly modular approach in which the management know-how can be encapsulated into modules. Moreover, modules should be *loosely-coupled* in a way that allows the agent to acquire new management capability modules at runtime, without stopping its execution.

4.2.2 Genericity

We showed in the previous chapter how different agent techniques have been successfully applied for NM purposes. However, it became clear that a unique set of agent techniques cannot suffice by themselves to all the fields and functional areas of NM. Therefore, any agent architecture that is based only on a limited selection of agent concepts would have potential limitations when applied to NM.

A successful agent framework should allow to use the right agent oriented technique for the right management application. Take agent communication as an example. It is obviously preferable to have a built-in ACL that allows for high-level communications between the agents, e.g. *à la* KQML. However, one should be able to use another communication language when required. For example, a particular management application that requires intensive agent communication would require a more efficient and concise language than KQML. In other situations, an agent system may include particular agents that support two or more agent communication mechanisms to interoperate with another different agent system.

In hierarchically-organized network management, an agent can delegate tasks to agents under its authority simply using 'orders'. This is the way most of the management systems are currently organized. However for some tasks, delegation is better supported using negotiation. The delegator agent negotiates with the delegatee agent on the way the task has to be achieved in order to maximize a certain utility function. Suitably, this applies to the task of monitoring a particular component (or set of components) in the network, for which the delegator has to choose the monitoring agent that generates the least polling traffic. The advantage of a generic agent architecture is the ability to support both types of delegation: order-based, and negotiation-based.

As for the flexibility requirement, genericity requires agent capabilities to be encapsulated in highly loosly-coupled modules that the agent can acquire dynamically at runtime.

4.2.3 Dynamism

Consider a typical network composed of routing and switching devices, server machines, and end-user client hosts. Such networks dynamically evolve in their topology, hardware, and services and applications software. Moreover, network users evolve in the way they make use of network services according to their changing usage profiles. Network performance patterns and resource usages can dramatically change as user profiles evolve and new services and applications are installed.

An NMS has to cope with such evolution. Particularly, for an agent-based NMS this has two implications on the agent architecture. The first is that an agent must have a common repository of all its beliefs on the managed network. As the network behavior evolves, relevant events are captured by changes in the agent beliefs and the agent can adapt its behavior according to these changes. Not only the beliefs themselves may evolve, but the agent has also to integrate new kinds of beliefs, or *belief templates*, dynamically without interrupting its operation.

The second implication is that the agent activity has to dynamically adapt to network changes. The agent tasks has to be notified of relevant changes in the agent's beliefs. Consequently, a belief notification mechanism based on a publish-subscribe-like pattern has to be introduced. The publish-subscribe pattern allows to have multiple observers that are automatically *notified* when relevant changes in the agent beliefs occur. This notification mechanism allows to propagate changes on the network structure, topology or configuration, in a way that management tasks can accommodate their execution accordingly. These notifications have to be available both inside each agent as well as between distant agents, i.e. an agent may coordinate its activities according to the beliefs of the other agents.

In the following, we describe how these requirements are mapped to our agent architecture.

4.2.4 Architectural Principles

Both flexibility and genericity back the fact that the agent does not have to tight to its initial role and functionality for which it was built. Instead, the agent can acquire new capabilities that allow to have new kinds of behavior, and to perform different management activities than those for which it was designed. This requires that agent capabilities be encapsulated into strongly moduled pieces of software. Each module provides the agent with the competence and skill to perform a new task or set of tasks. Hence, we introduce the "*skill*" as an encapsulation of the necessary components that define a new functionality for the agent.

Support for adaptability and dynamism requires that the agent can acquire new skills dynamically at runtime without interrupting its ongoing activities. The runtime integration of the new skills suggests that the agent has a core structure to enable this feature. Hence, we introduce the "*brain*", which is a common component that ensures the basic functions of an agent. If we refer to the basic agent architecture drawn in Section 2.2 (see Figure 2.1), then we conclude that the basic functions of the brain are to manage the agent beliefs and to tract its actions. In our case, the brain has in addition to manage the dynamic integration of skills, and their possible interactions.

The agent brain only provides a part of the agent default behavior. Decision making and agent behaviors are rather provided by the skills. Skills offer a modular approach to provide the agent with new features related to a certain Network Management (fault detection, monitoring, etc) or agent-oriented aspect (agent communication language, learning, etc). These features require the definition of new belief templates and new *ca-*

pabilities, where the capabilites encode the know-how related to a certain management activity or agent behavior.

Skills have to dynamically and seamlessly integrate into the agent brain. Therefore, the brain has to be able to *discover* the content of a skill at runtime. This is the requirement of loose coupling between the skills and the agent brain. Loose coupling requires a declarative approach by which the skill 'declares' the belief templates and the capabilities it defines. The brain uses this declared information to integrate skills dynamically and to insure the role of the mediator between them.

Another aspect that allows skills to integrate dynamically into the agent is that all skills must have equal access to the brain. This means that skills can *a priori* equally manipulate the agent beliefs and cause the agent to initiate actions. The most suitable approach for this kind of equitable access is horizontal layering, where in our case, layers are assimilated to skills. This is illustrated in Figure 4.1 adapted from [Woo99]. The characterizing property of horizontal layering in agent architectures is that all the layers have equal access to the agent sensory input and effectory output.

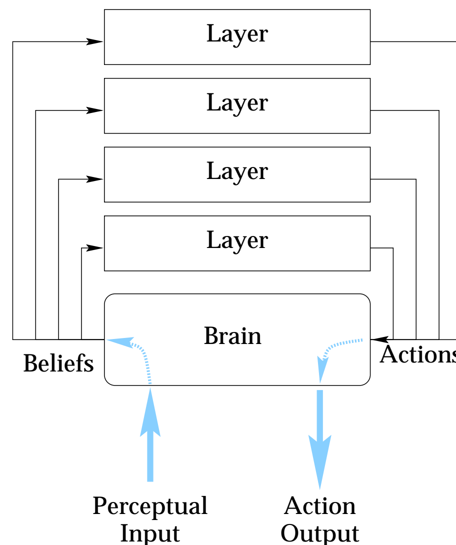


Figure 4.1: Horizontal layering

The added value of our approach compared to horizontal layering is that layers can be dynamically added to and removed from the agent. This is essential to provide the required levels of dynamic flexibility and genericity analyzed in the previous sections.

The final aspect for our agent architecture is that a KQML-like communication mechanism is supplied as a built-in protocol inside the agent brain. This communication mechanism is pragmatically developed in order to provide basic interactions between

the agents, particularly notification-based and delegation-based interactions. Of course, an agent system is not tightened to this communication mechanism and new skills implementing other communication mechanisms can be developed if need be.

Java is adopted as the implementation language, which leads to a portable agent prototype framework.

The following sections detail these different aspects of our agent architecture.

4.3 Capability Skill Modules

A *Capability Skill Module* (or *skill* for short) is a software module that can be instantiated into the agent to provide it with a new functionality or behavior and to allow the agent to ensure a certain management role or part of a role. It defines new information elements called *Beliefs* and the necessary *Capabilities* that bring the required know-how to achieve new management operations or to insure agent roles.

4.3.1 Beliefs

In the context of network management, agent beliefs hold the management information the agent has, as well as information about the other agents it knows about. For example, the status of a switch that the agent is managing can be expressed as a belief:

```
(switch :name sw301 :ipAddr 193.55.63.48 :status Ok).
```

As can be noticed we adopt a Lisp-like syntax in which the first item indicates the type of the belief or the name of the *belief template*. Attributes are preceded by semicolons and followed by their respective values.

Each skill must define the beliefs related to its management functionality. For example, a skill that manages ATM switches must define and instrument the necessary beliefs on the managed switches, including the statuses of the virtual paths and virtual channels created. The skill is responsible for ensuring the coherence and the integrity of the beliefs it generates. In addition, it is the skill that defines the complete syntactic and semantic properties of its beliefs.

The beliefs brought by the skills are all made available to the agent's brain and are centralized in the *Belief Database* presented later in Section 4.4.

4.3.2 Capabilities

A *capability* is a type of action that the agent may invoke from a certain skill to achieve a certain task or operation, or to provide the agent with a certain behavior. A particular

kind of capabilities allow the agent to interact with the managed network and can be either *sensors* or *effectors*. Sensors allow to capture the status of the network typically by using a management protocol such as SNMP. They provide the agent with up-to-date beliefs on the status and behavior of the managed network. Effectors allow to affect the status of the network and to configure its managed components.

In general, a capability may require the help of another capability within the same or another different skill. In addition, a capability may involve either a long-term activity or a function-like action. For example, a capability that allows to monitor the job queue of a certain printer and constantly refreshes the corresponding belief is a continuous long-term activity, whereas, a capability that allows to add a route entry on a certain router involves only a time-limited activity that terminates as soon as the configuration is completed.

4.3.3 Skill Declaration

To allow the Brain to make use of a loaded skill, a declarative interface is provided within each skill. This interface allows the brain to discover the beliefs and capabilities offered by each skill. For the beliefs, only the syntax aspects including the name of the belief and its attributes are declared. For the capabilities, several pieces of information are provided, namely:

- the preconditions that must be satisfied before it can be invoked;
- the other capabilities that it may invoke for its different execution instances. These prerequisite capabilities can be defined either by the same or by another skill;
- the beliefs that are used as input for its execution instances. These beliefs can be provided by other skills;
- the beliefs that are produced or updated during the achievement of the invoked capability.

In the implemented agent framework, a proprietary Lisp-like syntax is used for skill declaration, as illustrated in the example provided in Appendix B. A future improvement would be use XML instead of this proprietary language.

4.4 The Brain

The Brain offers basic and innate facilities necessary for the agent operation. In Figure 4.2, we distinguish the four functional parts of the agent's brain: Belief management, skill management, inter-agent communication and skill interface.

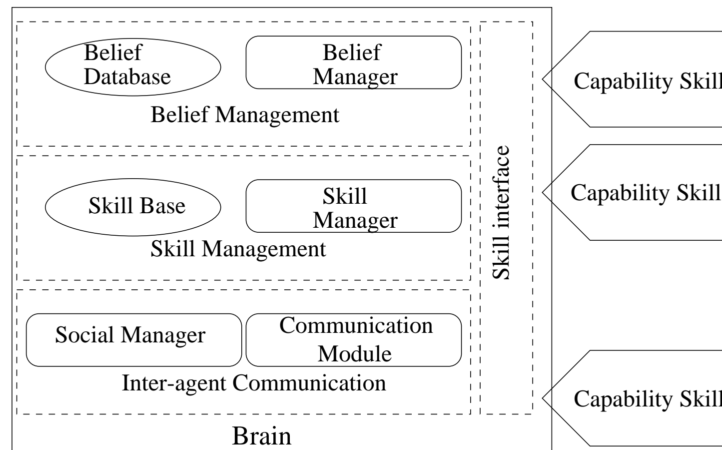


Figure 4.2: Our skill-based agent architecture

4.4.1 Belief Management

The *Belief Database* centralizes all the beliefs produced by the skills loaded into the agent. In this way, beliefs can be accessed concurrently from any skill. The *Belief Manager* allows the skills to query and modify the agent belief database. Besides the usual database operations that allow to retrieve, delete, create and modify the beliefs, the belief manager allows the skills to *subscribe* as listeners to specific changes in the database. For example, a skill that is responsible for the monitoring of the printing service would be interested in each new printer that is introduced in the network. Therefore, the printing skill should subscribe for the creation and deletion of beliefs corresponding to printing devices (which could be supplied by a discovery skill for instance). The belief management services can be summarized in the following items:

- Belief querying service using the `ask` primitive.
- Belief manipulation services using the `insert`, `delete`, and `update` primitives.
- Belief subscription services using the `subscribe` and `unsubscribe` primitives.
- Belief notifications services following subscription requests.

4.4.2 Skill Management

The *Skill Base* holds the declarative information describing the belief templates and capabilities brought by each skill known to the agent. This knowledge allows the *Skill Manager* to help the agent making efficient use of the functionality offered by each skill. In that, the services offered by the skill manager are enumerated in the following items:

- The skill manager links belief changes and action execution requests to their respective skills. For belief changes, we mentioned earlier that it is up to the skill to insure the coherence and integrity of its beliefs (i.e. beliefs that are instances of the belief templates defined by this skill). Therefore, changes to such beliefs caused by other skills has to be notified to the corresponding skill to take integrity-maintenance actions when necessary.

  For action execution requests, the skill manager looks in the skill base to determine which skill defines the capability that allows to execute this action. The action execution request is then transmitted to that skill.

- When an action, say action A, requests the execution of another action, say action B, the skill manager automatically subscribes the skill of action A to the belief modifications caused by action B. This allows action A to have a strict control of the evolution of task B, and to use the results produced by the latter task as soon as these results are produced. To tranparently set the necessary subscription, the skill manager directly invokes the subscription methods in the belief manager. This is only the default behavior of the skill manager, and can be turned off if deemed unnecessary.

- The skill manager keeps track of the tasks performed by the agent. Each task is associated to the capability from which it is instantiated, and therefore to the corresponding skill.

- The skill manager is also responsible for the loading and unloading of agent skills. The corresponding methods are respectively `loadSkill` and `unloadSkill`.

  Skill loading is achieved either locally or cooperatively with the other agents. The skill manager first searches for the required skill in the local repository of the agent skills. If this search fails, the skill manager sends search requests to the other agents. Another agent may have the requested skill available and can therefore serialize its code to the requesting agent. Again, this is the default behavior, and can be modified by designing a special skill that intercepts the skill loading query and performs

the skill search process in a different way. The Directory Service skill presented in Section 4.7 is an example of such skill.

Once a skill is loaded, it may require some initialization operations, for example to generate primitive beliefs for its operation. Every skill is endowed with a `warmup` method for this purpose. The skill manager calls this method as soon as the skill is successfully loaded.

- Skills can also be unloaded from the agent. In order to correctly stop the skill activity and to remove its corresponding beliefs, every skill has a disposal method. The skill manager calls this method before a skill can be successfully unloaded.

Therefore, the high-level role of the skill manager is to coordinate the agent tasks and to provide systematic associations between belief changes and the running tasks. This is made possible by the declarative information provided by the skills and stored in the skill base.

4.4.3 The Inter-agent Communication

The Brain offers inter-agent communication facilities that allow skills from different agents to interact in a transparent way. The *Communication Module* is responsible for sending to and receiving requests from the other agents, whereas the *Social Manager* holds such information on the other agents as the host on which they run as well as their addresses. Therefore, skills only deal with the symbolic names of the distant agents with which they interact, and they are not aware of distribution-related details in the agent system.

The communication module provides a set of methods that allow to send communication messages to remote agents. Each method corresponds to a certain performative (see Section 4.5). The name of the agent is resolved by the social manager and transformed into a network address (URL).

The communication module receives messages from the other agents. According to the message performative, the message is forwarded either to the belief manager, or to the skill manager. In the particular case of belief query (i.e. `ask` performative), the communication module submits the query to the belief manager and automatically initiates a reply with the result to the remote agent.

Many network protocols can be supported by the communication module including raw TCP, UDP, HTTP and SMTP. This allows to adapt the used communication protocol according to the size and the nature of the information to be exchanged between the agents.

The records maintained by the Social Manager can be mapped into beliefs in the belief database. These beliefs allow the skills to make use of the knowledge on the other agents. Particularly, one may build a registration skill that can be loaded into an agent to make it ensure the role of an agent name server. They also allow an agent to ask another agent for the list of all the agents the latter knows about.

4.4.4 The Skill Interface

The *Skill Interface* is responsible for the dynamic plugging of skills during the agent operation. It also ensures the important role of managing the Skill-Brain interaction. The skill-to-brain interactions can be either belief queries, which are forwarded to the belief management part of the brain; task invocations and skill loading orders, which are forwarded to the skill management part; or communication acts, which are forwarded to the communication part.

The brain-to-skill interactions are either:

- request for the skill declarative information, which is required for skill loading by the skill manager,
- request to warmup the skill after being successfully loaded,
- notifications for relevant belief changes,
- capability invocation to initiate the execution of some tasks,
- or a request for skill disposal before the skill is unloaded.

In each of these cases, the skill interface is responsible for forwarding each interaction from the brain to the right skill.

4.5 Agent Communication

Compared to KQML and FIPA ACL (Agent Communication Language), our approach to agent communication is more pragmatic. First, we adopted a set of performatives that are directly useful for NM purposes. This is a bottom-up approach driven by requirements, opposite to the top-down approach of Benech [Ben99], who applied KQML *as a whole* to NM. The communications requirements in NM are to efficiently exchange information, and to delegate management tasks.

Second, we provided a set of performatives to directly manipulate the beliefs of the agent. Such performatives do not exist in ACL (Agent Communication Language), and

KQML contains only one of them (`insert`). However such direct belief manipulation is very practical, especially when there is an authoritative relationship between some agents, e.g. a hierarchical authority.

Moreover, we preferred to be explicit in the meaning of the defined performatives. While FIPA ACL (Agent Communication Language) uses the `request` performative to ask for action execution, we call it `requestAction`, since a `request` in its general meaning may potentially imply an information request, an action request, or a service request.

Finally, the `ask` performative is synchronous in our case. This means that the corresponding method in the communication module of the brain blocks until a reply is received from the remote agent. This might seem to be inefficient at first glance. However, a synchronous query is easier to use and we rarely encountered cases in which an `ask` query requires to be asynchronous.

Accordingly, we defined a set of ten performatives:

- **tell:** Allows an agent A to inform another agent B of a subset of A's beliefs.
- **ask:** Allows an agent to query the beliefs of another agent. Replies are contained in `tell` messages.
- **insert, delete, update:** Allow to respectively add, remove or modify a belief or a set of beliefs in the receiver's belief database.
- **subscribe:** Allows an agent A to inform another agent B that A requires to be notified of specific changes in B's belief database. In this case, B must send `tell` messages for each relevant change in its beliefs. A subscription messages contains a special field to specify which belief manipulation (insertion, deletion, update) are interesting for the subscriber.
- **unsubscribe:** Allows to stop the effect of a preceding `subscribe` message.
- **achieve:** Allows to make the receiver reach a certain state on the managed network. The content of an `achieve` message is an abstract goal that yields the agent to look for the necessary actions to achieve it. Since we did not include any real planning in the agent's brain, an `achieve` query should be completely handled by the involved skills if need be.
- **requestAction:** Allows to delegate the execution of a specified action to the receiver agent.
- **stopAction:** Allows to stop an action that has been launched due to a preceding `requestAction` message.

Like in ACL (Agent Communication Language), every message contains a number of fields that specify the context of the agent conversation (`conversastion-id`) and the message references (`message-id`, `in-reply-to`, `reply-with`). Some performatives require optional fields such as the `on-event` field in the `subscribe` performative, which specifies when notifications should be sent (periodically, following a belief suppression, etc). In contrary, there is no support for ontologies. The reason is that ontologies are suitable only for knowledge sharing, i.e. for an open agent system where agents may deal with different subjects that might use the same terms for different concepts defined in different domains. In our case, we have only one context: Network Management.

Finally, we note that in the following of the manuscript, we make use of a message syntax that is similar to that used in KQML and ACL (Agent Communication Language), although we use a slightly different way to encode these messages to transmit them over the network.

4.6 A Development Process for Agent-based Applications

This section provides an overview of the agent development process suggested for our skill-based agent architecture. Mainly, the process is divided into three phases: macro-design, micro-design and agent deployment.

1. **Macro-design**

   In this phase, the developer identifies the major agent roles needed for the system to be developed. The *role* abstraction is introduced because a single agent may assume different roles, and a single role may be undertaken by a set of interacting agents. Moreover, the roles of an agent can be changed dynamically during its lifetime according to the skills that this agent has at a certain moment. The role abstracts either a management functionality, a coordination functionality, an overall agent behavior, or an activity that the agent has to insure. In general, agents are conceived as peer-to-peer entities but with different roles: each agent is specialized in some aspect of the overall system. But roles can also have different relationships than peer-to-peer relationships. For example, a master role might be attributed to an agent that provides this agent with a certain authority over a set of other agents.

   In the absence of a role specification model, we suggest to specify roles using simple text. However, once the agent roles are identified, the behavior of the agent system can be described using interaction scenarios between these roles (or agents supposed to assume these roles).

2. **Micro-design**

   This phase is a refinement of the Macro-design. It focuses on the skills that implement the identified roles instead of the agents themselves. A set of skills must be designed to ensure each of the roles identified in the first phase. In general, skills have to be defined based on the beliefs and capabilities of other skills.

   A skill is completely specified when all its beliefs and capabilities are fully defined. This specification includes the integrity constraints that must be ensured for the beliefs and the possible relation of these beliefs with other beliefs from other skills. It also includes the relationship between each capability and its prerequisite beliefs and capabilities. The output of the micro-design allows to refine the scenarios described in the first phase to go down to the details of the interaction between the skills themselves.

3. **Implementation and Deployment**

   The skills can be written without paying attention to distribution-related issues (thanks to the brain inter-agent communication facilities). The problem of agent distribution among the available network hosts can be handled just after the skills are developed. At this stage, parameters such as CPU load balancing, response time and bandwidth usage can be optimized by placing the agents appropriately throughout the network. Furthermore, the attribution of skills can be handled and tailored during the operation of the agents.

These three development stages have been used in the two case studies that we implemented using our agent architecture. This will be illustrated in Chapters 5 and 6 respectively.

4.7 Skill Examples: Agent Support Services

In many cases, a multi-agent system requires some basic services that improve its flexibility and its ease of use. In this section, we show how agent support services can be developed in skills that can be loaded into particular agents to make them undertake organization roles. We first consider a skill that supports multicast communications between groups of agents. The multicast service is very useful to dynamically organize the agents in groups, and to enable private communications inside these groups. Next, we present a directory support skill that allows the registration of agents and the declaration of available skills. This service is important when agents are created and killed dynamically during the operation of the overall agent system. Finally we present a self-control

enabling skill that produces beliefs on the agent itself. This service is useful to the agent to have control over its own status and actions.

4.7.1 Agent Multicast Communication Skill

Group communications in agent systems are very useful in situations where a group of agents need to exchange broadcast messages among themselves. For example, in an agent-based NMS, a group of agents may be responsible for the management of the printing service in the whole network. These agents may need to commonly exchange information related to the network printers without involving the other agents. Another example is that of a group of agents that ensure low-level data monitoring. These agents may want to privately allocate monitoring taks in a transparent way regarding the other agents. Furthermore, given that agent roles can be dynamic, agents may join and leave particular groups randomly during their execution. We propose to provide a simplified skill that helps managing agent groups. We call this skill *Multicast Support Skill*.

4.7.1.1 Belief Templates

The multicast support skill requires beliefs that describe the currently defined agent groups, and beliefs that state the agents members of each group. These beliefs can be respectively written as:

```
(multicastGroup :name printingManagers
                :privacy open )

(multicastGroupMember :group printingManagers
                      :agent subnetwork1printingManager )
```

A `multicastGroup` belief has two attributes that respectively indicate the group name and whether messages from non-member agents are allowed or not (`open` vs. `closed` privacy). A `multicastGroupMember` belief specifies the group name and the name of an agent that belongs to this group. There are as many beliefs for a certain group as agents belonging to this group.

4.7.1.2 Capabilities

The only capability that the multicast support skill has to provide is to broadcast messages to the members of a certain group. The `broadcast` capability requires the name of the group and the message to be broadcasted. In the example of the printingManagers

group, an agent, say *centralManagementAgent* may want to propagate a message asking for those printers that have a `down` status, i.e. `(ask :content (printer :status down))`. If we suppose that the agent having the role of multicast message router is *mcastRouter-Agent*, then the message to be sent by the centralManagementAgent is:

```
(requestAction
      :sender centralManagementAgent
      :receiver mcastRouterAgent
      :content (broadcast
                    :toGroup printingManagers
                    :message (ask :content (printer :status down))
               )
)
```

4.7.1.3 Using the Multicast Support Skill

Let's suppose that *mcastRouterAgent* is the agent that insures the role of the multicast router. In order to create a new multicast group, an agent may just send an `insert` message with the name of the group and the privacy policy:

```
(insert :content (multicastGroup :name systemManagers :privacy closed))
```

Similarly, in order to subscribe to a group, an agent would send an `insert` message with its name and the name of the group specified in a `multicastGroupMember` belief expression. Conversely, in order to leave a multicast group, an agent has to send the same content but in a `delete` message.

An interesting way to use the multicast support skill is to have a particular agent send welcome information to agents that newly join a certain group. The welcoming agent has just to send a `subscribe` message to the mcastRouterAgent asking to be notified on beliefs of the form: `(multicastGroupMember :group systemManagers)`. This allows the agent to be notified whenever a new agent joins the systemManagers group, and can therefore send initialization information to this agent.

4.7.2 Directory Service Skill

The directory service is an essential component for any MAS where agents can be dynamically added and removed from the system. In such dynamic MAS, it is necessary that agents declare their existence and their roles in the system. This information can then be accessed by the other agents in the system and agents can be added and removed

from the agent system dynamically. The directory service can also be used in order to retrieve information concerning the skills that agents can fetch from skill repositories or other agents.

The directory service skill requires belief templates for the agents and for the skill. The belief template for agent description has the syntax illustrated by the following example:

```
(dsAgent :name Hulk
         :address lys.eurecom.fr
         :port 6622
         :status running
         :creationTime 2000:03:07, 15h15'10"35
         :role "Domain monitoring of components")
```

The `role` attribute can be used to textually describe the global role of each agent.
The belief template for skills has the following syntax:

```
(dsSkill :name SnmpMonitoring
         :repository http://skillserver.eurecom.fr
         :description "Polling of MIBs from SNMP agents")
```

Another third belief template is defined in order to let agents declare their respective skills:

```
(dsAgentSkill :skill SnmpMonitoring
              :agent Hulk
              :loadedAt <...>)
```

This last belief template is essential when agents have to be retrieved according to their loaded skills. This is very useful because the agent capabilities are defined through its acquired skills.

The way the directory skill is used is basic. When an agent is created and wants to register, it sends an `insert` message to the directory server agent with the content field set to a `dsAgent` belief. To unregister, a `delete` message is used. Similarly, when an agent wants to declare that it has acquired a new skill, an `insert` message is sent using the corresponding `dsAgentSkill` belief. Finally, an `ask` message is used to query the directory service agent for currently existing agents (`dsAgent` beliefs), or for agents having certain skills (`dsAgentSkill` beliefs). If an agent wants to locate the repository server where a skill can be found, an `ask` message with the corresponding `dsSkill` belief has to be used.

In the case that agents are not considered to be necessarily reliable, an agent that previously registered in the directory server may crash without properly unregestering.

This may cause coherence problems in the agent system. For such situations, the directory service agent has to actively check whether the registered agents are still alife or not. Since this may not be mandatory in all agent systems, this feature is implemented using two capabilities that can be invoked when necessary. These two capabilities are `(dsCheckAgentsOnce)` and `(dsCheckAgentsPeriodically :period 5mn)`. The first capability performs a single-shot check and the second one performs a periodic check. The check is performed using a simple `ask` query and assumes that the tested agent no longer exists if no reply is returned after a certain timeout.

Of course, the checking can go further by identifying the skills acquired by each agent and therefore updating the `dsAgentSkill` beliefs. This can be achieved by targetting the `ask` performative to the beliefs of each agent on its acquired skills. However, such beliefs are not available by default. This is one among the reasons to develop the *self-control skill* described in the following section.

4.7.3 Self-control Skill

This skill produces beliefs on the agent itself. For example, it produces beliefs on the currently plugged skills, on the available capabilities, on the running tasks, on the agent activity, etc. This is a valuable skill to provide the agent with control over its own status. Recall from Section 2.1.3 that self-control is an important feature to achieve agent autonomy. The self-control skill provides the agent with different belief templates.

- The `selfSkill` belief template is instantiated for each skill that the agent has acquired. For example, if the agent has acquired the multicast support skill, then the corresponding belief is:

```
(selfSkill :name mcastSupportSkill
           :status loaded)
```

- The `selfCapability` belief template is instantiated as the capabilities of a recently acquired skill are being discovered by the brain. For each capability, a belief is generated similar to the following, corresponding to the `broadcast` capability of the Multicast Support Skill:

```
(selfCapability :name broadcast
                :skill mcastSupportSkill
                :attribute toGroup
                :attribute message )
```

- Similarly, as the brain is discovering the belief templates supplied by a recently acquired skill, `selfBeliefTemplate` beliefs are generated such as the following example:

```
(selfBeliefTemplate :name multicastGroup
                    :skill mcastSupportSkill
                    :attribute name
                    :attribute privacy)
```

- Finally, as the agent initiates actions, `selfAction` beliefs are accordingly generated, similarly to this example:

```
(selfAction :capability broadcast
            :parameterValue (attribute :name toGroup :value printingManagers)
            :parameterValue (attribute :name message :value (ask <...>))
            :startedAt 2000:02:25, 15h52'12"22
            :status terminated)
```

The capabilities defined by the self-control skills allow to selectively enable the tracking of the `selfSkill`, `selfCapability`, `selfBeliefTemplate`, and `selfAction` beliefs inside the agent. These capabilities are:

```
(selfControlTasks)
(selfControlSkills)
(selfControlBeliefs)
(selfControlCapabilities)
```

The services thus provided by the self-control skill are not only useful for the agent itself, but also can be useful for the other agents. A particularly interesting example is when an "agent manager" needs to be introduced in order to monitor and control the agent activities throughout the multiagent system. In that, recall how a directory service agent would need such self beliefs in order to feed its own directory that can then be shared by all the agents.

4.8 A Polling Layer for Belief Instrumentation

Agent-based NM applications rely on a large set of beliefs on the network status and behavior. These beliefs, referred to as *network beliefs*, need to be created and updated from the currently existing instrumentation, mostly using SNMP agents. Many NM skills

define network beliefs that have to be filled using polling operations. The duplication of polling functionality in many skills results in a large number of polling threads, thus leading to heavy CPU usage. Moreover, the heavy monitoring activity required for high-level and sophisticated management functions need to be rationalized in order to optimize polling operations and network resources.

Consequently, SNMP polling operations are encapsulated into a highly optimized module called the *polling layer*. Agent skills use the services of the polling layer to instrument the necessary network beliefs using a higher-level API than the instantiation of polling threads and the handling of SNMP polling packets. In addition, the polling layer uses a small number of polling threads for the instrumentation of all agent network beliefs.

The polling layer is based on sophisticated optimization mechanisms, and provides a powerful interface for NM applications. Details about the polling layer do not fit in the context of this chapter, and are therefore thoroughly presented in Appendix A.

4.9 Summary and Conclusion

Our agent architecture is different from other NM-oriented agent approaches. First, our skill based architecture is not restricted to a single type of NM applications. Most of the agent applications surveyed in the previous chapter are designed to tackle specific NM issues and are therefore not reusable for other types of management applications. Our approach is open to the overall NM functional areas and is designed to answer global NM requirements instead of specific applications. Management competences can be developed in skill modules which allow agents to undertake new management roles and functions.

Second, our agent architecture is not restricted to certain techniques of software agents. Instead, it is designed in a way that new agent features can be encapsulated into skills and supplied to the agents, thus providing them with new behaviors and capabilities. The behavior of each agent can be customized, and each agent can be endowed with the necessary intelligence to achieve its roles. This is also different from the agent approaches presented in the previous chapter, which commit to a certain type of agents, thus inevitably limiting the potential application of the resulting agents. We have also shown that our architecture offers the basic agent services to allow agent communications, while more sophisticated services can be provided by dedicated skills. Therefore, our approach allows to customize the organization of the agent system exactly as needed, which is opposed to other approaches that impose a certain type of agent organization,

for example around a facilitator agent, or within administrative groups that constrain inter-agent communications.

Our architecture supports the developement of highly dynamic management applications. Any task instantiated from a skill capability can subscribe to relevant changes in the agent belief database, and dynamically adapt to these change. Hence, a domain-based monitoring task can automatically integrate the monitoring of newly added elements in the domain and stop the monitoring of the removed elements. This support for dynamism is not constrained to a single agent, but can occur between different agents as well. The communication mechanism allows an agent to transparently adapt its behavior according to the beliefs of another agent. On-the-fly skill plugging while the agent is running is another feature that promotes dynamism. An agent can therefore acquire new management functions and new agent capabilities without interrupting its operation. This is important in future generation NMSs that have to support frequent updates to the management functionality.

Concrete case studies using this agent architecture will provide enhanced evidence of these advanced properties. These case studies are the subject of the next two chapters.

Chapter 5

The First Case Study: When Management Agents Become Autonomous, How to Insure Their Reliability?

The first case study has two distinct aspects. The first is an NM aspect that, in a simplified way, shows how a global management task can be dynamically distributed amongst a set of agents, using domain-based delegation. The second aspect is rather a fundamental problem in using autonomous agents to achieve critical NM tasks. If agents are deployed to undertake sensitive management responsibilities, then how to be sure that these agents remain reliable? This chapter proposes to answer this question.

5.1 Rationale

The problem of the reliability of management agents has not been a major issue in classical management paradigms. As a matter of fact, such management agents do not undertake important management responsibilities and all the decisions were taken at the manager level. Therefore, there was no real danger that a particular agent performs uncontrolled management operations that might compromise the overall operation of the network. Moreover, the management protocols employed to communicate with these agents intrinsically ensured the reliability of the agent. For example, the SNMP protocol

mainly uses confirmed communications for SET operations, or polling-based monitoring in which each `GET_REQUEST` query expects a `GET_RESPONSE` reply.

However, these conditions are no longer ensured when a distributed NMS is based on highly autonomous agents. Autonomous management agents are capable of making high-level decisions. They have the authority to execute sensitive management operations without direct control from the network administrator. Therefore, it is not straightforward to detect the unreliability of an autonomous agent that might make wrong management decisions, or might not even react to critical changes in the network. Such abnormal behaviors may compromise the overall security and performance of the managed network.

Since the agents themselves make use of the managed network, which is fault prone, autonomous management agents may become unreliable due to network or system faults. Therefore, an agent-based NMS must be able to promptly detect the possible unreliability of its intelligent management agents. When an agent is detected to be unreliable, the other agents should be able to cooperate together in order to ensure the management tasks that were previously assigned to the unreliable agent. Therefore, the agents must be capable of dynamically undertaking new management tasks during their operation.

The purpose of this first experiment with our skill-based agent architecture is to build a prototype of an agent-based NMS in which autonomous agents, affected each to a distinct domain of the network, are able to mutually test each other and to detect the agents that become unreliable due to network, system or service failures. These so called *domain agents* can dynamically redistribute their management tasks in the case of an agent failure, so that all these tasks continue to be performed.

We use a distributed diagnosis algorithm to detect the unreliability of the autonomous agents. This algorithm is based on *System Level Diagnosis* (SLD) [Ber96] which will be presented in the next section. As an example of a distributed management task the agent system may achieve, we choose the monitoring of NEs for fault detection purposes. Each autonomous agent is affected to a network domain composed of a dynamic set of NEs that the domain agent has to monitor.

After detailing the principle of SLD, we present in the subsequent sections the way we proceed to identify the required agent roles and correspondent skills. Each identified skill is then presented, with the intra/inter-agent interaction involved. The experimental results and conclusions about this case study close the chapter.

5.2 System Level Diagnosis

Consider a system made up of many interdependent components, where each component may or not be faulty. If we suppose that these components mutually test the defectiveness of each other, and that there is no *a priori* knowledge on whether each given component is faulty or not, then is it possible to deduce out of these mutual tests which components are faulty and which are faultless?

In this situation, the difficulty is that the test results of two different components could be conflicting. For example, a given component could be marked as faulty by a first testing component and as fault-free by a second testing component. As a matter of fact, all the components are considered as peer entities, and hence, there is no reason to believe in the test results of a certain component when confronted to another's results.

System Level Diagnosis specifically tackles this issue.

5.2.1 The Algorithm

SLD applies on a set of entities testing each other mutually. In the case that these entities are agents, the result is a matrix where each line represents the beliefs of a certain agent on the reliability statuses of the other agents. It is not mandatory that each agent tests the reliability of all the other agents. SLD theory stipulates that if the total number of agents is n, then it is sufficient that each agent be tested by $\lfloor(n-1)/2\rfloor$ distinct neighbor agents. The aggregation of all the test results is called the *syndrome* and can be expressed using a *syndrome matrix* that serves as input to SLD. For example, if we have five agents respectively denoted from A to E, we may obtain the following syndrome matrix:

Syndrome Matrix	Agent A	Agent B	Agent C	Agent D	Agent E
Agent A's beliefs	-	Ok	not Ok	?	?
Agent B's beliefs	?	-	not Ok	Ok	?
Agent C's beliefs	?	?	-	not Ok	Ok
Agent D's beliefs	Ok	?	?	-	Ok
Agent E's beliefs	Ok	Ok	?	?	-

There are many variations of the SLD algorithm, which are discussed in [Mar98]. The adopted algorithm starts by finding a first agent which reliability is certain. For this purpose, it is assumed that no more than half of the number of the agents are unreliable. The reason is that it has been shown in [PMC67] that the number of unreliable agents should not exceed $(n-1)/2$, otherwise the system cannot be diagnosed. Based on this assumption, it is proven that there exists at least one agent sequence of a minimum length

of $(n+1)/2$, such that the first agent is believed to be reliable by the second agent, the second agent is believed to be reliable by the third agent, and so on. It can be intuitively understood that since the length of the sequence exceeds the maximum tolerated number of unreliable agents, the sequence necessarily contains at least one reliable agent, and therefore the first agent is necessarily reliable.

Once a reliable agent is identified by constructing such an agent sequence, the algorithm continues by considering the reliability test results of this agent and chaining through the reliable agents until all the agents are determined to be reliable or not. In our example, agent A is necessarily reliable, since a possible sequence is $< A, D, E >$. (A is reliable according to D, and D is reliable according to E.) Agent A's beliefs imply that agent C is unreliable and that agent B is reliable. Since B is reliable, agent B's beliefs can be considered recursively, thus implying that agent D is also reliable, and so on. The result of SLD is a boolean vector indicating the reliability status of each agent. In the example, this vector indicates that agents A,B,D and E are reliable, while agent C is unreliable.

A formal description of this algorithm and the proofs on which it is based are given in [Mar98] where several references on SLD can also be found.

5.2.2 The Reliability Test

An essential question in this case study is how to attest that an agent is reliable or not. What test has an agent to perform in order to attest that, according to its own beliefs, the tested agent is reliable or unreliable? The answer to this question tightly depends on the tasks that the agents are performing. A rigorous approach would be to check all the competences of the agent, possibly by asking it to simulate some decision makings related to predefined situations. Of course, this is not feasible both because it would be very complex to perform such tests when the agent ensures multiple roles, and because the agent may acquire new skills dynamically. If a new skill is loaded in the testee agent, the tester agent may not have the knowledge to test the new skill.

Conversely, an effortless approach consists in sending a simple `ask` query to an agent and assuming that it is reliable as soon as a `tell` message is returned. However, this approach is not accurate in many situations. For example, the agent may contain a thread that is responsible for keeping its beliefs updated. An uncaught exception that occurs in this thread will make it stop, thus leading to outdated beliefs. In this situation, the agent still continues to reply to `ask` queries with outdated beliefs, while it has to be considered unreliable since its decisions are based on outdated information.

Our approach is consequently to strike a balance between these two extreme testing

methods. We assume that an agent would perform reliably as long as its beliefs are always kept up-to-date. Therefore, the reliability test consists in having the tester agent establish in its own belief database a representative selection of the testee's beliefs. Then the testee agent is considered to be reliable as long as its representative beliefs are coherent with the tester's corresponding beliefs. In this case study, the management task that the agents are performing is the monitoring the NEs in their respective domains. Accordingly, the representative beliefs can simply be the tested agent's beliefs on a certain number of NEs assumed to be selected according to a given criterion.

In summary, a tester agent, say agent A, initiates by its turn the monitoring of selected network elements in the testee agent, say agent B. Periodically, agent A asks agent B about its beliefs on these NEs and compares them to its own beliefs. If agent B's beliefs are not coherent with those of A, then agent A will consider that B is unreliable.

5.3 Macro-design: Scenario and Role Identification

Consider a network with a reasonably large size. To be effectively managed using distributed agents, this network may be partitioned into domains, each domain being managed by one corresponding agent. We call such agent a *domain agent*. To simplify the case study, we choose a sample management task to be performed by the domain agents, that of performing fine-grained and detailed monitoring of the sensitive NEs in their respective network domains in order to come up with the high-level global operational status of each element. This identifies the first role in the agent system, which is referred to as the *domain monitoring role*.

The *fault management role* is introduced as a management function that launches the monitoring activity and collects the monitoring results from the domain agents. This role has no incidence on the other agent roles and is present only to demonstrate the results of this case study, which will be described later in this chapter.

As the network topology is supposed to be dynamic, new NEs can be added and others removed at runtime. The number of domain agents is also supposed to be dynamic, since domain agents could be launched and killed at runtime, e.g. to scale to the size of the managed network. The management activity should not be interrupted in such situations, and therefore, agent domains should be dynamically reassigned, hence the *domain management role*. The domain management role consists in equitably distributing the monitoring activities to dynamically scale to the size of the network and the number of available domain agents.

As can be noted, the domain monitoring, fault management and domain manage-

ment roles are independent of the SLD mechanism, and can be insured completely without it.

SLD requires three roles to be defined. An agent that is testing the reliability of another agent has to assume the *tester role*, whereas the agent being tested has to undertake the *testee role*. In addition, another agent has to consolidate the syndrome matrix and to run the SLD algorithm each time this matrix is updated. This role is the *SLD manager role*.

In order to simplify studying the interactions between these roles, we group them into two macro-roles, namely the *master role* and the *domain role*. The domain role brings together the tester role, the testee role and the domain monitoring role. The master role groups the other roles.

Three major interaction patterns govern the agent system. The first pattern occurs between a tester agent and a testee agent. According to the reliability test presented in the previous section, the tester agent asks the testee for a selection of NEs in the testee's domain. In order to keep the tester updated with this selection in case of a domain change, the tester agent has to subscribe for this information. Afterwards, the tester has to subscribe for the testee's beliefs on the statuses of the selected NEs. In this way, the testee agent has to send the necessary updates whenever the status of a NE changes. This interaction pattern is summarized in Figure 5.1.

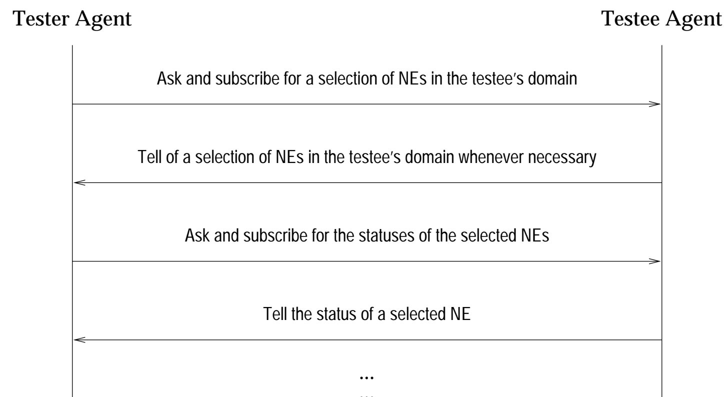


Figure 5.1: Interaction pattern between a tester and a testee

The second interaction pattern is initiated by the manager agent to ask a domain agent to test the reliability of another agent. The manager then subscribes for the belief upon the reliability of the testee agent. Therefore, the tester domain agent sends updates whenever it performs the reliability test on the testee agent. This is illustrated in Figure 5.2.

Note how during the greyed portion of the tester agent's thread, the tester agent runs

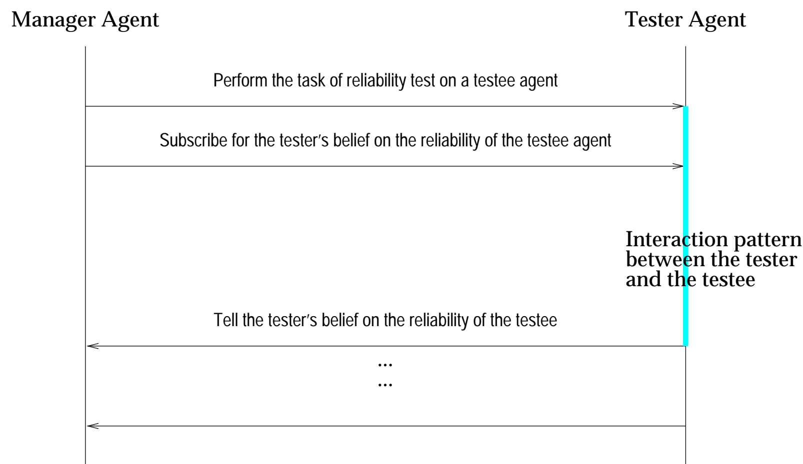


Figure 5.2: Interaction pattern between the manager and the tester

the interaction pattern of Figure 5.1 with the testee agent, in order to setup the reliability test.

The last interaction pattern occurs between the manager and the domain agents when a domain reassignment is performed. In this case, the manager sends the necessary domain updates to each domain agent.

These interaction patterns will be refined in the following section.

5.4 Micro-design: Skill Identification and Design

5.4.1 Mapping from Roles to Skills

The agent roles identified in the previous section are supported by a set of six skills as described in the following.

- The monitoring role is mapped directly to a *monitoring skill*.
- The tester and the testee roles are combined together into an *SLD reliability test skill* or *SLD skill* for short. The reason that these two roles are combined into the same skill is practical since all the domain agents are at the same time testers and testees of each other.

  The monitoring skill and the SLD skill together implement the domain role.
- The SLD manager role is supported by the *SLD manager skill* denoted as *SLDM*.
- The domain management role is directly mapped to the *domain management skill*.

- The fault management role is introduced, as noted in the previous section, for demonstration purposes. This role is mapped to a *fault management skill*. Another skill, the *interface skill* is similarly introduced for demonstration purposes to provide GUI facilities to access the agent system.

  The SLDM skill, the domain management skill, the fault management skill and the interface skill implement the master role.

The remainder of this section provides the design and implementation details of these skills.

5.4.2 Skill Design

We base our description on the skill interaction diagram presented in Figure 5.3. The diagram shows the skills required for the master agent role and those required for the domain agent role. It also shows the belief notification flow between these skills. The bold arrows represent inter-agent communication acts, while the thin arrows represent skill interactions inside the same agent.

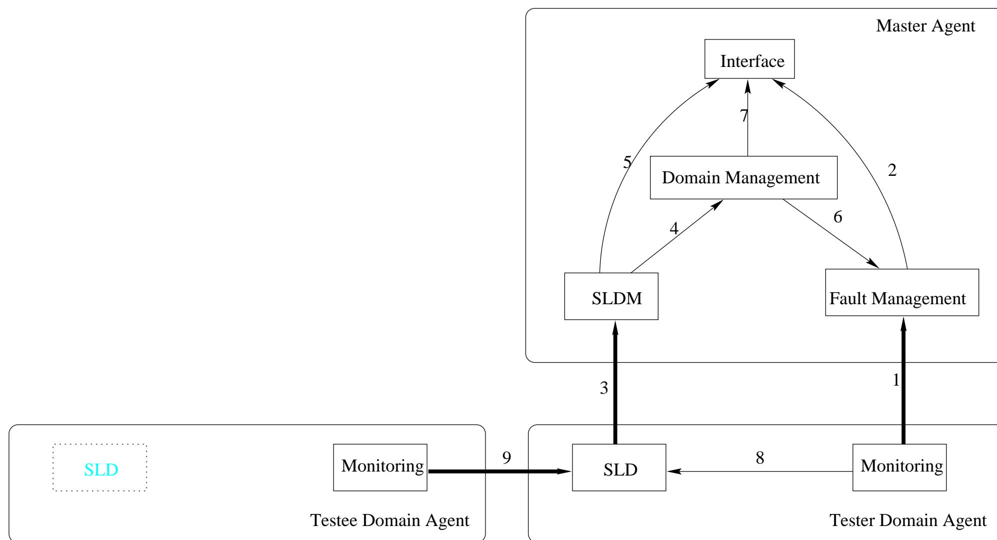


Figure 5.3: Skill interaction diagram

In Figure 5.3, we first note the aggregation of the skills in a way that maps the agent master and domain macro-roles. Let's insist however that being a master does not require a dedicated and fixed agent. As a matter of fact, thanks to the possible dynamic loading of skills, any agent in the agent system can potentially ensure the role of the master by loading the necessary skills, i.e. the domain management skill, the fault management skill, the interface skill and the SLDM skill. It is even possible that an agent

ensures at the same time the role of the master agent as well as the role of a the domain agent.

5.4.2.1 The Monitoring Skill

The monitoring skill is responsible for the instrumentation of the necessary beliefs about the monitored NEs. This information is then used by the SLD skill (⑧ and ⑨) and the fault management skill ①.

The monitoring skill uses the services of the polling layer (Section 4.8). It initiates periodical polling on each monitored NE and produces a summary of its global status. This results in a `networkElementStatus` belief such as the following:

```
(networkElementStatus :ne hub101 :globalStatus Ok)
```

The monitoring skill defines the `(monitor)` capability accepting as a unique argument the name of the NE to be monitored. A `requestAction` for this capability leads to a continuous activity to monitor the requested NE. This activity is responsible for the creation and update of the `networkElementStatus` belief corresponding to this NE.

Based on the `(monitor)` capability, the `(domainMonitoring)` capability automatically performs the monitoring of all the NEs in an agent's domain. The NEs belonging to the agent's domain are declared using beliefs such as `(inMyDomain :ne hub101)`. When invoked, `(domainMonitoring)` produces a continuous activity that not only initiates the monitoring of all the NEs included in the domain, but also subscribes to changes in the agent's domain and updates the monitoring activity accordingly. It stops the monitoring of NEs that are removed from the domain, and initiates the monitoring of the added NEs.

5.4.2.2 The Domain Management Skill

The domain management skill performs the assignment of NEs to domain agents to ensure that all the NEs are continuously monitored and are equitably distributed amongst the domain agents. The domain management skill acts also as a simplified directory service for the domain agents and for the managed NEs. Each domain agent is represented by a `domainAgent` belief and each NE is represented by a `networkElement` belief. The domain management skill is based on two capabilities: `(assign)` and `(manageDomains)`

The `(assign)` capability performs the NE assignment to domain agents. If performed for the first time, it produces `inDomain` beliefs such as the following, which states that `hub101` is affected to domain agent `Ironman`:

```
(inDomain :ne hub101 :domainAgent Ironman)
```

As the network structure evolves, new NEs and services can be added or discovered, while others can be removed. In these cases, when the (assign) capability is invoked, the resulting action performs incrementally by creating or deleting the corresponding inDomain beliefs.

The (manageDomains) capability represents a persistent action that continuously watches for domain agents and NEs that are newly introduced or removed and invokes the (assign) action to update the domain distribution. The domain manager (i.e. the master agent) has to inform the concerned domain agents of such changes in the domains decomposition, which has lead to two possible design choices.

In the first choice, it is up to each domain agent to send a subscribe performative on the beliefs concerning the NEs in its domain. This will cause the domain manager agent to send tell messages for each pertinent change concerning each agent's domain.

In the second choice, it is up to the domain manager agent to send belief updates, whenever necessary, to domain agents using the insert and delete performatives. The difference with the first choice is that insert and delete performatives are authoritative and assume that the domain agent has no chance to negotiate or refuse changes in its domain. In contrast, the subscribe performative in the first choice leaves the possibility to the domain agent to send an unsubscribe message and receive no more domain updates. Moreover, a domain agent is not obliged to consider the tell messages sent by the domain manager agent, due to the informative semantics of the tell performative.

Our implementation adopts the second choice, since it better models the authority of the domain manager over the domain agents.

5.4.2.3 The SLD Skill

A domain agent starts the testing of another domain agent whenever a requestAction message is received with a content similar to: (sldTest :agent Hulk) where the (sldTest) is the capability that enables the SLD testing, and Hulk is the name of the agent to be tested in this case.

As described during the macro-design phase, the first step consists in finding a selection of NEs in the testee's domain. This can be achieved using one of two possible scenarios.

In the first scenario, it is the tester agent that determines by itself the NEs to be selected from the testee's domain. This implies that the tester is continuously updated with all the NEs in the testee's domain. Moreover, it implies that the tester agent has the know-how to select the representative NEs in the testee agent's domain. The main advantage of

this scenario is that it reflects the passiveness of the testee role, which is therefore, should not necessitate additional capabilities in principle.

In the second scenario, it is the testee agent that selects the representative NEs and maintains a belief for this purpose. Any other agent that has to perform the SLD testing has to subscribe for the belief corresponding to this selection. This scenario has many advantages over the first one. Firstly, only the selected NEs are communicated to the tester agents. This selected communication has a great advantage over the first scenario, in which all the NEs in the testee's domain have to be communicated for each tester agent. Secondly, the NE selection is performed only once for each tested agent, whereas in the first scenario, the selection is performed as many times as the number of the tester agents. Finally, the second scenario ensures that two testers of the same tested agent will base their tests on the same selected NEs, which is not explicitly guaranteed in the first scenario. Consequently, the second scenario is adopted for this case study.

The implication of this choice is that the SLD skill needs to define for the testee role the belief template `(selectedNetworkElements)` to hold the NE selection. Moreover, the SLD skill defines the `(provideSelection)` capability that establishes this selection and keeps it updated when the agent domain changes.

Hence, the abstract interaction pattern between the tester and the testee (Figure 5.1) is now refined as presented in the three first messages in Figure 5.4. Note how in this case, a subscribe interaction pattern based on the `requestAction-subscribe-tell` performatives is used rather than the authoritative pattern based on the `insert-delete` performatives previously adopted in the domain management skill. This choice is justified by the peer-to-peer relationship between the domain agents versus the authoritative interactions between the master agent and the other domain agents.

Once the tester agent receives the selected list of NEs on which the test will be based, it invokes the `(monitor)` capability of the monitoring skill on these NEs. At the same time, it sends `subscribe` messages for the testee's beliefs about them. The testee agent sends the belief updates about the selected NEs to the tester agent, which translates them into local `believedNetworkElementStatus` beliefs. For example:

```
(believedNetworkElementStatus :by-agent Hulk :ne hub101 :globalStatus Ok)
```

translates Hulk's belief on `hub101` in the tester agent belief database. The testee agent compares these beliefs to its own ones on a regular basis. The comparison is carried out by the `(performSldTest)` capability, which performs the SLD test. This capability accepts the name of the testee agent as a parameter and generates the corresponding reliability belief, e.g. `(sldStatus :agent Hulk :reliability Ok)`.

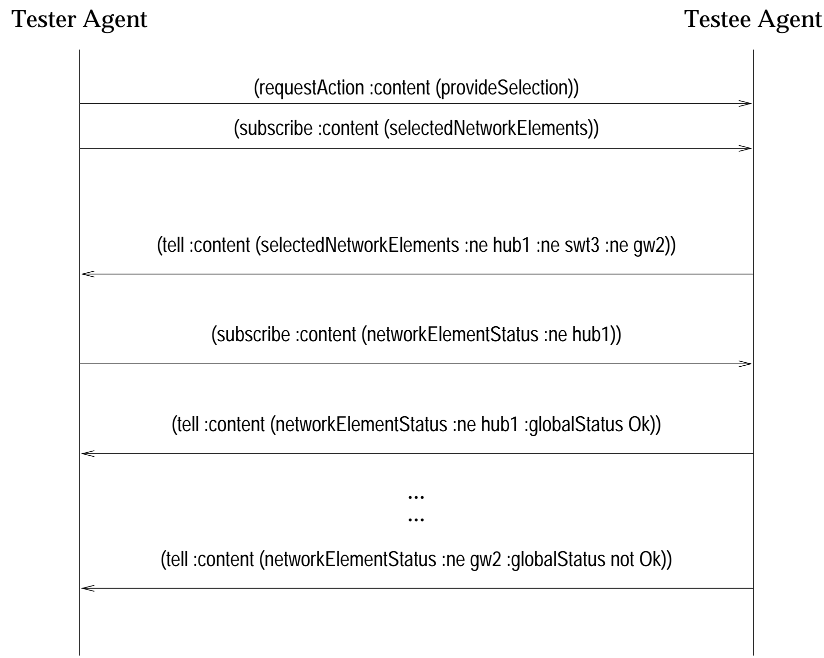


Figure 5.4: Refinement of the interaction diagram between the tester and the testee

5.4.2.4 The SLDM Skill

The SLDM skill defines a unique capability, called `sldDiagnose`. The actions caused by the invocation of this capability are explained in the following paragraphs. Importantly, we recall that the whole SLD mechanism is provided as an ad hoc mechanism to insure the reliability of the agent-based system. This means that the other skills used for the normal management operations in our case study should not be aware of the existence of the SLD skills. This principle poses very interesting design aspects to be discussed in the following paragraphs.

The Test Graph The `(sldDiagnose)` capability does not accept any arguments. Instead, the list of agents on which the SLD mechanism is to be run are directly provided by the domain management skill, i.e. the `domainAgent` beliefs . The first action that the `sldDiagnose` capability performs is to generate the *test graph*, i.e. the graph that indicates which node will be tested by which other node. This test graph has to ensure that it provides a syndrome matrix sufficiently consistent to run the SLD algorithm. The domain agents are then affected to the nodes of the test graph. In the implemented system, this affectation is performed in a static way. Future improvement could consider several possible affectation criteria, e.g. to minimize the overhead traffic caused by SLD.

Enabling the SLD Testing Once the test graph is generated, the master agent sends `requestAction` messages to the tester agents to invoke the `(sldTest)` capability of the SLD skill on each tested agent. At this stage, the master agent may subscribe for the `sldStatus` beliefs thus generated by each tester agent. This allows to receive the updates of their `sldStatus` beliefs asynchronously. However, the problem with this approach is that tester agents may not send their updates at the same time, and the resulting syndrome matrix may become incoherent.

Therefore, we adopted another approach in which, the master agent periodically sends `ask` messages to the tester agents. Using this pull-based approach leads to a coherent syndrome matrix since all the updates are received at the same time and not asynchronously as when using the subscription mechanism.

In addition, the pull-based approach allows to detect those agents with serious operation problems (e.g. crash) that prevent them from sending the results of their reliability tests. If a tester agent does not reply to the `ask` query within an appropriate timeout, it is directly marked as unreliable in the syndrome matrix.

Agent Diagnosis The SLD algorithm is applied on the syndrome matrix, which is consolidated periodically. The output of this operation determines the unreliable agents, if any exist.

In the case that an agent is detected to be unreliable, it is necessary to perform a domain reassignment in which the NEs that have been monitored by this agent are distributed to the other reliable agents. The question is therefore how to let the domain manager skill ignore the existence of the unreliable agent when it performs the reassignment of the involved NEs.

To this question, we adopted a technique borrowed from vertically layered agent architectures such as Brook's subsumption architecture (see Section 2.2.5). The *belief suppression technique* consists in providing an incomplete view of the world to a given layer, in a way that this layer acts within a limited field of action. This technique applies very well in our case. In order for the SLDM skill to trigger a domain re-affectation that excludes unreliable agent, it deletes its corresponding `domainAgent` belief. The deletion of this belief is detected by the `(manageDomains)` capability of the domain management skill, which therefore invokes the `(assign)` capability. This interaction is labeled ④ in Figure 5.3.

In order to keep trace of the unreliable agent until it recovers back, the SLDM skill keeps a record of it in an `unreliableDomainAgent` belief holding the name of this agent. If later the agent recovers, this belief is converted back to a normal `domainAgent` belief,

which again causes the domain reassignment, this time by considering the recovered agent.

5.4.2.5 Fault Management Skill

The fault management skill delegates the domain monitoring tasks to domain agents and subscribes for the generated beliefs in a way that the statuses of all the NEs in the network are collected by the master agent (① in Figure 5.3). In the demonstration, this data collection allows to show how the monitoring activity is not interrupted when SLD is activated despite the unreliability of a domain agent.

5.4.2.6 Interface Skill

The purpose of the interface skill is to allow multiple access to the master agent in order to watch and control the demonstration scenario. Using a Java-enabled browser, users that connect to the master agent URL download an applet provided by the interface skill. The aspect of this applet is drawn in Figure 5.5. The left side of the panel shows the monitored statuses of all the NEs in the managed network, each NE being associated to the domain agent to which it is affected. The right side of the applet is used both to control the demonstration process via the control buttons (in the upper panel), and to dump the beliefs that are continuously received by the master agent (in the lower panel).

To supply the connected applets with the necessary displayed information, the interface skill subscribes for all the required beliefs from the fault management, the domain management and the SLDM skills. (This is shown by ②, ⑤ and ⑦ in Figure 5.3.) All the connected applets are notified when pertinent changes in the beliefs are detected, which allows them to update the information displayed in the graphical panel accordingly.

If a user clicks on one of the control buttons in his applet panel, this action is transmitted to the interface skill that map it to the corresponding capability invocation in the master agent. In this way, the user can control the demonstration process through his applet.

The typical demonstration process is detailed in the next section.

5.5 Demonstration and Results

Eurécom's network was used as a testbed for the experiment. It was partitioned into five domains, each of which was affected to one domain agent, as indicated in Figure 5.5. In addition, we used another agent to undertake the master role.

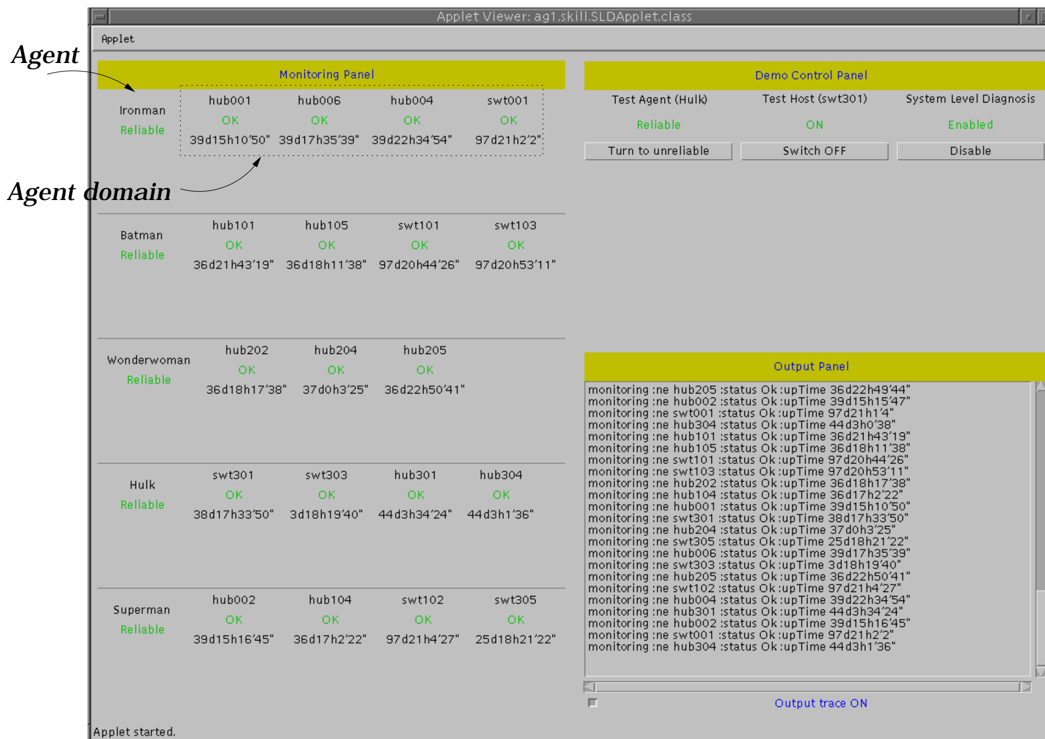


Figure 5.5: Demonstration applet

The experiment was composed of a twice-repeated process: first without the SLD mechanism, and then with the SLD mechanism enabled. At the first stage, the monitoring was launched on the five domain agents, which started to forward the results to the master agent. We caused the crash of a domain agent (this agent was *Hulk* in the demonstration). The crash was not detected by the agent system, and the monitoring of the network elements in Hulk's domain was no longer ensured. The results displayed on the applets about Hulk's domain were outdated and even incorrect: one network element in Hulk's domain was switched off and this was not reported on the connected applets.

At the second stage, the SLD mechanism was enabled. As previously, the monitoring was launched on the domain agents and the applet started displaying the monitoring results. Afterwards, we cause a crash in Hulk's monitoring thread. In this case, the crash was detected as soon as the master agent received the SLD reliability beliefs from the domain agents and ran the SLD algorithm. The SLDM skill updated the beliefs describing the set of the available domain agents. The brain notified the domain management skill, which performed a domain re-distribution. The network elements that had been previously monitored by *Hulk* were affected to the other domain agents in a balanced way. Therefore, the connected applets continued to receive the monitoring information reli-

ably despite the failure of a part of the agent system. This property introduced an aspect of fault tolerance into the network management system.

5.6 Discussion and Conclusion

The starting point of this case study is that distributed management agents, as soon as they are endowed with autonomous decision making, have to be checked for their reliability. The SLD mechanism is a solution to this reliability issue, as well as an interesting case study from an agent-oriented programming point of view. SLD is indeed implemented as a voting mechanism (Section 2.5), where voters are the domain agents. The master agent makes the reliability decision based on the collected votes, and all the agents abide to this decision.

Moreover, different interaction patterns are used to provide an elegant design of the agent system. Different reasons are considered to select the right interaction pattern. In that, NE selection algorithm was incorporated in the testee agent rather than in the tester. The reason is to improve efficiency by reducing the communication overhead. Alternatively, the `requestAction-subscribe-tell` performatives were used for peer-to-peer delegation, while direct `insert-delete` performatives are used to reflect the authority of the domain management (master) agent over the domain agents. This shows the ability of our agent architecture to generate appropriate solutions to different design issues and to model different types of relationships.

Another important aspect in this case study is that the SLD mechanism and its related skills can be viewed as an add-on to the basic management tasks for which the agent system was in principle designed. The SLD skills are designed in a way to be transparent to the monitoring, fault management and domain management skills. This shows how functionality and behavior can be added to the agent system without disturbing its operation. Moreover, through the use of the belief suppression mechanism, we showed how our agent architecture, with its equitable horizontal layering approach, can also support a hierarchical control between skills in the same way as the SLDM skill controlled the domain management skill operation.

This case study also showed the capacity to build dynamically adaptable management applications using our agent architecture. The domain management skill perfectly shows this property since it is designed in a way that each change in the network structure is immediately handled. The introduction or the suppression of a domain agent (or an NE) directly causes a domain redistribution to integrate this change in the management application.

Chapter 6

Provision of Permanent Virtual Circuits in ATM Networks

The second case study implemented using our skill-based agent architecture deals with the configuration of Permanent Virtual Channels (PVC) in ATM networks. From an ATM operator stand point, this case study is very useful to automate the provision of PVCs, which is a tedious task further complicated in heterogeneous ATM networks. The presentation of this case study requires an ATM background which can be additionally found in [PLBS98] and [TB94].

6.1 Background and Rationale

Virtual Channel Connections (VCC) are unidirectional links that connect end-to-end points in ATM networks. There are two main types of VCCs in ATM networks: Switched Virtual Channels and Permanent Virtual Channels. Switched Virtual Channels (SVC) are established automatically using ATM signaling protocols UNI and PNNI [Bla98]; whereas Permanent Virtual Channels (PVC) are established by the human operator on a switch-by-switch basis. PVCs can be useful in many cases. For example, most of the existing ATM switches cannot offer SVCs with all the standardized traffic contracts (e.g. [FOR97]). Currently, most of the ATM hardware can only support SVCs with either UBR or CBR traffic. Moreover, PVCs are permanent connections. They can be established permanently between two end-hosts in order to have at any time QoS-guaranteed and immediately available connection. PVCs are more suitable for continuously-used connections with fixed QoS, since there is no waiting time due to the establishment of the connection using the ATM signaling protocol. Finally, an ATM equipment may not support a com-

plete implementation of the ATM signaling protocol. In fact, current implementations of signaling mechanisms are not completely compatible between switches from different providers. Some ATM equipment providers implement their own proprietary signaling mechanisms such as SPANS [FOR98]. Therefore, SVCs cannot be established in a heterogeneous ATM network and the only way to establish end-to-end connections is to use PVCs.

The establishment of a PVC is not a simple task. Firstly, a physical end-to-end route between the source and the destination must be selected. The route is a non-empty list of nodes. Each node is a triplet that identifies a switch and its input and output ports that are going to be used. There might be several physical routes and one of them must be chosen.

Virtual channels have to be routed via Virtual Paths (VPs). Therefore, the next step is select the VPs through which the PVC will be bundled. Several strategies can be adopted leading to different levels of optimization and implementation difficulties. The approach generally adopted by ATM operators consists in creating VPs between each consecutive switches on the route. Though this shows to be far from being an optimal solution in terms of the number of supported VCCs, it is an easy and quick way of establishing PVCs.

The final step is to attribute VCIs (Virtual Channel Identifiers) on each switch and to create the necessary entries in the switch routing tables. On each switch, two VCIs are needed, one for the input port and another for the output port. The output VCI of an intermediate switch must be the same as the input VCI of the next switch. In the case that local VPs are created on each switch to transport the PVC, the outgoing VPI (Virtual Path Identifier) of a switch must also be the same as the incoming VPI of the next switch. If one of these constraints is not satisfied, then data transmitted on this PVC will not reach its destination. These are the source of configuration errors that are hard to diagnose.

But what actually complicates even further the task of PVC creation is the problem of equipment heterogeneity. Until now, most of the ATM hardware providers elaborate their own management interfaces based on proprietary MIBs. In most of the cases, proprietary configuration and administration interfaces are also supplied. Even if SNMP is supported as a management protocol on most of the ATM switches, each provider uses a different MIB. Therefore, the human network operator must know all these management interfaces to be able to appropriately create an end-to-end PVC.

Finally, the different fragments of a PVC must be configured with the same Usage Parameter Control (UPC) contract. The UPC is a set of parameters that specify the required quality of service in ATM. The UPC must be defined using exactly the same parameters on each switch . If a UPC parameter is wrongly configured at a switch, then the whole

traffic on the PVC could be affected. Again, troubleshooting such abnormal behavior is particularly hard.

In summary, the establishment of a PVC between end systems needs to take into account a lot of parameters and has to satisfy some constraints that are hard to verify especially in a heterogeneous environment. Therefore, a management application that allows to automate the provision of PVCs is of a great help for ATM network operators to rapidly provide connectivity services to their customers.

We proceed, as in the previous case study, by identifying the agent roles involved in the PVC creation scenario. Afterwards, we develop the necessary skills for these roles and specify their interactions. This is followed by details on the implementation and deployment of the agent system.

6.2 Macro-design: Scenario and Agent Roles

We consider an ATM network operator that offers end-to-end connections to its customers. Two distant users may want to establish a connection channel, for example to initiate a video-conference session. The network users are represented by User Agents (UA), while the network operator is represented by Switch Agents (Switch Agent (SWA)) that configure the ATM network switches. The role of the UA is to capture the user requirements and to map them into PVC parameters including the destination and the quality of service. The UA may automatically suggest the best suitable quality of service required for the type of connection that the user wants to establish. Once the parameters of the PVC are fully determined, the UA sends the request to the nearest SWA on the operator side. This SWA collaborates with the other SWAs in order to configure the end-to-end PVC. If the PVC is successfully created, the UA is informed of the adopted PVC identifiers and may automatically configure itself the user equipment to make use of the new PVC.

The role of the SWAs is to configure the switches in the ATM network. They accept the PVC creation requests from the UAs and cooperate together in order to satisfy these requests. Each switch in the ATM network has a dedicated and unique SWA to configure it and to coordinate its configuration with that of the other switches. Any SWA can accept requests from a UA to establish a PVC, and in this case, it becomes the responsible vis-a-vis of the demanding UA for the coordination of the PVC creation process. We call such an SWA, a *Master Agent* regarding that particular PVC. The other SWAs involved in the creation of that PVC are called *Slave Agents*. The Master and the Slave are sub-roles of the SWA role. We emphasize that the same SWA can be at the same time the Master Agent for a particular PVC, and a Slave Agent for another PVC.

In summary, this scenario defines two roles: the user agent role and the switch agent role. The switch agent role is further decomposed into two dynamic sub-roles: the master role, and the slave role. These two sub-roles are associated to particular PVC creation requests.

6.3 Micro-design: The Agent Skills

6.3.1 Skills for the User Agent Role

According to the description of its role, the UA behavior can be implemented using two skills. The *Contract Negotiation Skill* is responsible for sending PVC requests to the SWA and negotiating the service contract as well as the price. The *User Interface Skill* is responsible for capturing user requests for a connection establishment and to formulate them for the Contract Negotiation Skill.

The implementation of this case study provides a simplified version of these two skills. The User Interface Skill is only composed of a Graphical User Interface that allows the user to specify the destination of the connection and to select a class of QoS (Figure 6.1). This request is then forwarded to the Contract Negotiation Skill which, in its turn, delegates the task of PVC establishment to the SWA that manages the ATM switch to which the user is connected.

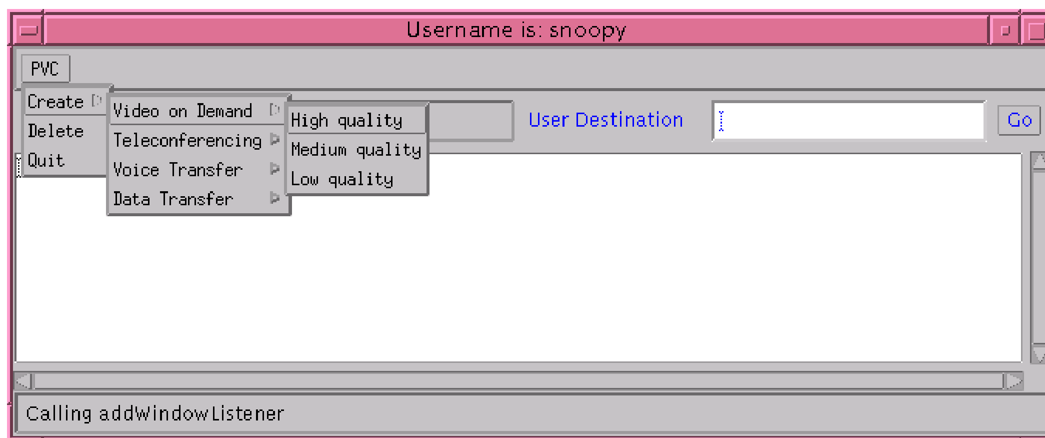


Figure 6.1: User interface for PVC creation

6.3.2 Skills for the Switch Agent Role

The switch agent role is ensured by four skills: the *Switch Skill*, the *Slave Skill*, the *Master Skill* and the *Topology Skill*.

- **The Switch Skill**

  A switch agent is responsible for PVC configuration operations of the ATM switch. The *Switch Skill* is therefore designed to provide services to create and delete VPs and VCs using the SNMP management protocol. There is a switch skill for each type of ATM switches. For example, the current implementation runs on FORE ATM switches, and therefore, a *FORE ATM Switch Skill* is developed. As a matter of fact, the switch skill is the unique part of the whole system that should be adapted for each product family. However, this should be a temporary situation until standard ATM management MIBs (e.g. [AT94]) are deployed in future ATM devices.

  The capabilities defined in the switch skill for the creation of VPs and PVCs are illustrated by the following usage examples:

```
(createIncomingVP :port 4 :vpi 5 :bandwidth 1200 :pvcId lys-212)
(createOutgoingVP :port 1 :vpi 3 :bandwidth 3000 :pvcId lys-212)
(createLocalPVC       :inputPort 4 :inputVpi 5 :inputVci 10
                      :outputPort 1 :outputVpi 1 :outputVci 2
                      :upc VHQ
                      :pvcId lys-212)
```

  The `pvcId` parameter allows to unambiguously identify an end-to-end PVC creation request. If the creation succeeds, the `pvcId` uniquely identifies the created PVC, for example during the deletion request. In the switch skill, this parameter is used to associate a VP or local PVC creation request to the result of the creation process which is reported in a dedicated belief. For example, the creation report for an incoming VP has the following format:

```
(IncomingVpCreationReport :pvcId lys-212 :result error :reason ...)
```

  There are also equivalent capabilities to delete VPs and local PVCs on a switch. These capabilities are: `deleteIncomingVP`, `deleteOutgoingVP`, and `deleteLocalPVC`.

  In addition, the switch skill maintains beliefs on the status of the current VPs and PVCs existing on the ATM switch. An example of a belief on an existing incoming VP is:

```
(incomingVP :port 4 :vpi 5 :allocatedBandwidth 1200 :usedBandwidth
1100)
```

The `allocatedBandwidth` parameter specifies the amount of bandwidth allocated to the incoming VP, while the `usedBandwidth` tells the amount of bandwidth that it is already allocated to PVCs routed through this incoming VP. Such information is used by the *Slave Skill* to decide on the local parameters for the establishment of a new PVC.

Finally, the switch skill maintains beliefs on the physical ports of the ATM switch. A belief on a physical port holds the value of the maximum bandwidth supported on the port as well as the total allocated bandwidth to the created VPs. An example of such belief is:

```
(port :number 4 :maxBandwidth 50000 :allocatedBandwidth 12000)
```

- **The Slave Skill**

  The *Slave Skill* is responsible for the local configuration operations that create or delete a PVC fragment on the switch managed by the corresponding SWA. For example, it is up to the slave skill to decide whether to create a new local VP in order to convey the PVC within, or to use an already existing VP with sufficient bandwidth available. Also, it determines which VPI/VCI couples are to be assigned to the newly created VPs and PVCs.

  Before concretely creating a PVC fragment on a switch, the slave skill is first asked to check for resource availability. This minimizes the number of cancellation that are due to insufficient resources on a certain switch. Despite this reservation step, PVC creation errors may occur during the configuration process. Therefore, the capabilities defined in the slave skill are respectively: `reserveLocalPVC`, `createLocalPVC`, `cancelPvcReservation` and `cancelPvcCreation`.

  An example of a PVC reservation request is the following:

  ```
  (reserveLocalPVC :inPort 4 :outPort 1 :upc VHQ :pvcId lys-212)
  ```

  The `upc` field specifies the class of service required for the PVC. In this case, `VHQ` stands for High Quality Video.

  Note how at this stage, the VPI and VCI parameters are not specified in the reservation request. It is up to the slave skill, while checking for resource availability, to decide of the VPI/VCI values, both at the input and output ports.

  The beliefs generated by the slave skill report the results of reservation and creation of local PVCs. These reports can be expressed using the `LocalPvcReservationReport` and `LocalPvcCreationReport` belief templates.

  Finally, the `deleteLocalPVC` capability is introduced to remove a PVC fragment on a switch.

- **The Master Skill**

  The *Master Skill* is responsible for the global supervision of the PVC establishment. Once a physical end-to-end route is found between the source and destination end-systems, the master skill contacts the slave SWAs on that route in order to ask them to perform the necessary operations to create the PVC. It is also responsible for handling creation errors that might occur during this process.

  The master skill defines two capabilties: `createEndToEndPVC` and `deleteEndToEndPVC`. An example of a creation request is:

  ```
  (createEndToEndPVC :source lys :dest nelke :upc VHQ :reqId 45)
  ```

  The source and destination end-points are specified using their symbolic names. The master skill relies on the topology skill to determine their addresses and to find the physical route between them.

  For each end-to-end PVC creation request, the master skill maintains an `EndToEndPVC` belief containing the progression status of the creation process. Here is an example:

  ```
  (EndToEndPVC  :source lys :dest nelke :upc VHQ
                :status underReservation
                :reqId 45 :pvcId lys-212)
  ```

  The status of the end-to-end PVC can be one among the following possible statuses: `underReservation`, `underCreation`, `created` and `underDeletion`.

- **The Topology skill**

  Finally, the *Topology Skill* helps the master skill to identify a physical route between the source and the destination. The physical route identifies which switches must be traversed by the PVC. For each switch, it determines the input and the output ports that shall be used. Here is an example of such belief:

  ```
  (EndToEndPvcRoute     :source lys :dest nelke
                        :switch forerunner :inport 4 :outport 1
                        :switch forele :inport 5 :outport 3)
  ```

  This belief specifies that to reach `nelke` from `lys`, the PVC can be routed from port 4 to port 1 through the `forerunner` switch, then from port 5 to port 3 of the `forele` switch.

Finding a physical route is of a minor concern for us since we are more interested in the PVC configuration problem than with the routing issues. Yet the topology of the network is hard-coded inside the topology skill source, since the experimental network on which we run the application is not large.

6.4 Implementation and Results

The experimental ATM network of *Eurécom* is used as a testbed for this case study. The ATM network is composed of two FORE ATM switches. Four machines are interconnected via these switches. Figure 6.2 shows the topology of this ATM network and the names of the agents affected its different components.

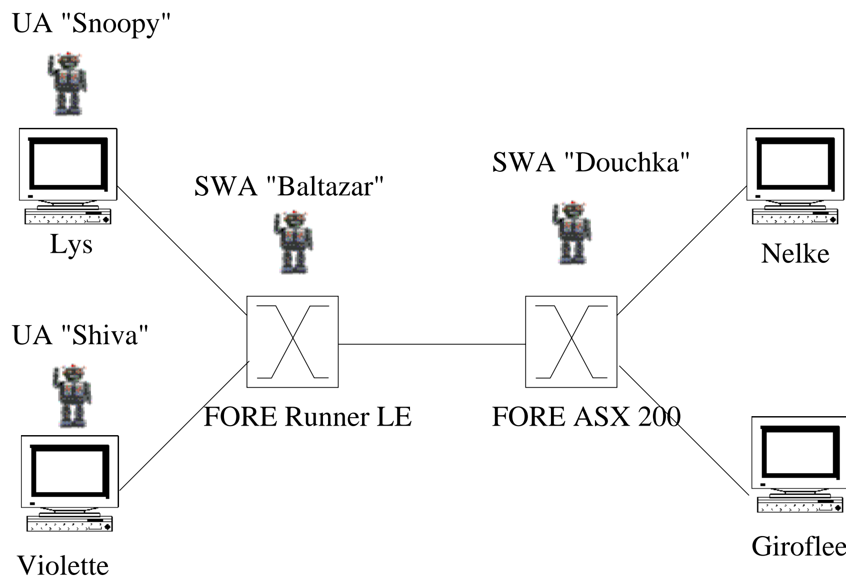


Figure 6.2: The testbed ATM network

The creation of an end-to-end PVC requires three major steps which are coordinated by the master skill. These steps are detailed in Figure 6.3 that shows the interactions between all the skills of a user agent (on the left) and two switch agents (respectively identified as “Baltazar” and “Douchka”).

1. **Finding a physical route.** The master skill queries the topology skill for a physical route that links the source to the destination. The topology skills replies with the set of switches to be traversed and which input and output ports to be used on each switch on the route. The interaction between the master and topology skills is performed through the second and the third messages in Figure 6.3.

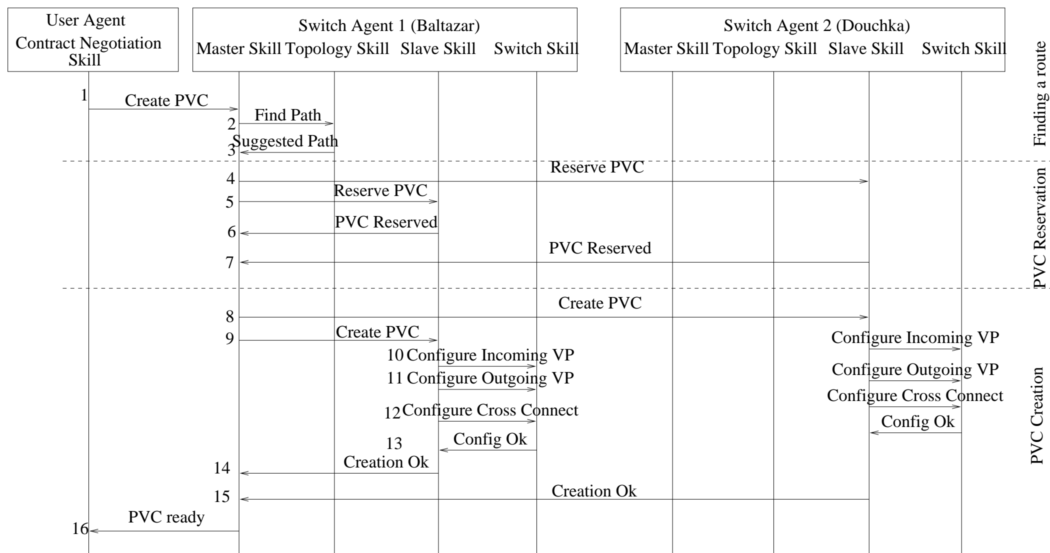


Figure 6.3: PVC creation scenario

2. **PVC reservation.** In this phase, the master skill asks the slave skills on the switch agents (including its own agent) to reserve the PVC (messages 4 and 5). Each slave agent then determines whether it is possible or not to accept the PVC (messages 6 and 7). If it can be accepted, then all the parameters (e.g. VPI/VCI values) of the PVC are determined during this reservation phase.

3. **PVC creation.** If the master SWA receives positive acknowledgments from the other SWAs, then the PVC can be effectively created. Again, the master skill sends creation commitments to the switch agents (messages 8 and 9). The slave skills in the switch agents execute these commitments using the services of the switch skill (messages 10 through 13).

   Finally, when all the creation positive acknowledgments are received, the master skill can send back a message to the UA indicating that the PVC is created and can be used (message 16).

Figure 6.4 (page 146) shows the aspect of the demonstration application developed for this case study.

6.5 Discussion and Conclusion

One important design principle that we adopted for the implementation of this case study is to push the managerial intelligence down to the level of the managed network

components. The user agent stands close to the end-user equipment, and the switch agent stands close to the managed ATM switch. The main reason for adopting this principle is to improve performance and avoid centralization. Only a few concise messages are exchanged between the agents that we developed for each PVC to be established (messages 1, 4, 7, 8, 15 and 16 in Figure 6.3). Moreover, while the design of the SLD case study lead to some kind of central control, this case study is fully distributed. The unreliability of an SWA does not affect the whole agent system, but only the corresponding ATM switch. Of course, it would be interesting to integrate SLD with the PVC provision agent system, in a way that if an SWA crashes, the closest reliable SWA directly undertakes the management of its ATM switch.

Skills in this case study are organized using vertical control. In the highest-level, the master skill controls the topology and the slave skills. The slave skill in its turn controls the switch skill, which is considered as the lowest-level skill. This leads to multiple abstraction levels to the PVC configuration problems. The first level of abstraction provides a homogeneous view of possibly-heterogeneous ATM switches. The second level controls the localized PVC configuration by considering individual switches. And the third level manages the global view of end-to-end PVCs. Our agent architecture allowed therefore to cope with the configuration problems due to switch heterogeneity.

Another property of our skill-based architecture is that skills can be developed without caring about distribution-related issues. A skill is developed as if it will be invoked within the same agent, but can be deployed in a remote agent and invoked remotely in a transparent way. This appears when the master SWA invokes capabilities of remote agents in exactly the same way as it invokes its own capabilities. All the distribution related aspects are handled by the agent's brain. This is another argument that our architecture provides a powerful support for building distributed applications.

All the interactions in this case study rely on a delegation model. This delegation model is based on the combination of `invokeAction` and `tell` performatives. Although not sophisticated from an agent coordination point of view, it shows to be efficient and the number of exchanged messages is kept to minimum.

This delegation model remains valid as long as the SWAs are supposed to belong to the same ATM network provider. If we consider multiple providers that have to cooperate together to achieve cross-domain connections, then SWAs from different parties may be self-interested and may have to negotiate to reach agreements in terms of cost and QoS. This scenario leads to another interesting problem for possible future work.

Last but not least, this case study illustrates how agents can assume multiple dynamic roles simultaneously. An SWA can be at the same time the master for a certain PVC creation request, and a slave for another PVC.

From the SLD and PVC case studies, we wanted to show the different advantages related to the dynamic flexibility and genericity of our agent architecture. However, there are still open questions as how our architecture compares to other most-used agent approaches. Moreover, among the advantages we obtained during these case studies, which are attributed to the agent-oriented approach as a whole, and which others are attributed to the specifics of our agent architecture? We propose to answer such questions in the following chapters.

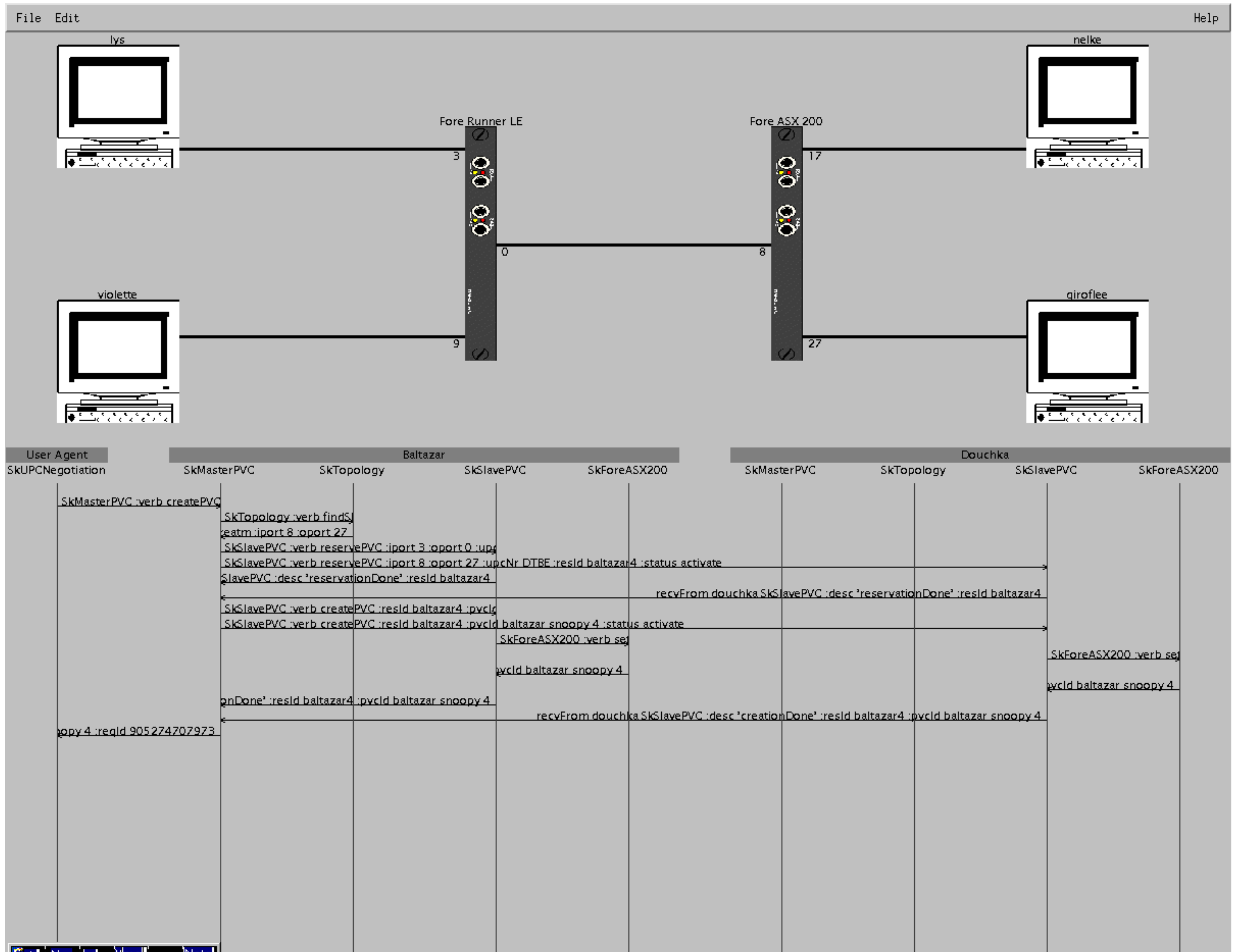


Figure 6.4: General panel of the PVC demonstration application

Chapter 7

A BDI Approach to the SLD Case Study

7.1 Preamble

In chapter 5, we used our skill-based agent architecture in order to build a reliable management system based on autonomous software agents. This allowed to perceive how an agent-based implementation to this case study could be developed. We noticed that during the development process, we made an intensive use of the specific constructs defined in our agent architecture, e.g. skills and belief templates. An interesting point to raise is the extent to which the approach changes if we use another agent architecture based on different constructs and concepts.

The work that we describe in this chapter considers a BDI approach to the same SLD case study of Chapter 5. Our contribution is manifold. We first developed a BDI-oriented agent model which is refined from several related work on BDI agent architectures and which is adapted to NM applications. Then we used this agent model to develop a BDI abstract design of the domain agent role in the SLD case study. Afterwards, we implemented this abstract design using an existing BDI-oriented agent development framework. Finally, based on the obtained results, we discuss several aspects related to the application of a BDI-oriented design, emphasizing the differences with the skill-based agent architecture.

7.2 The BDI Agent Model

The structure of a BDI agent architecture can be devided in two separate layers: the *deliberative layer* and the *operational layer*. The deliberative layer provides an environment in which the agent mental state can evolve. The mental state is organized according to a set of mental categories controlled by a mental cycle. The mental cycle rationally allows the agent to select and adapt its actions according to its mental state. The actions that the deliberative layer selects are then executed and controlled in an *operational layer*, which provides the action execution environment for the agent.

The deliberative and operational layers that we devised for our abstract BDI-oriented agent architecutre are described in the following sections.

7.2.1 The Deliberative Layer

The mental categories used in the proposed BDI agent model are *motivations*, *goals*, *intentions*, *beliefs* and *capabilities* (Figure 7.1). In the following, we recall the semantics of these mental categories and we refine them according to our particular NM-oriented BDI agent model. Afterwards, we describe the different processes included in the mental cycle.

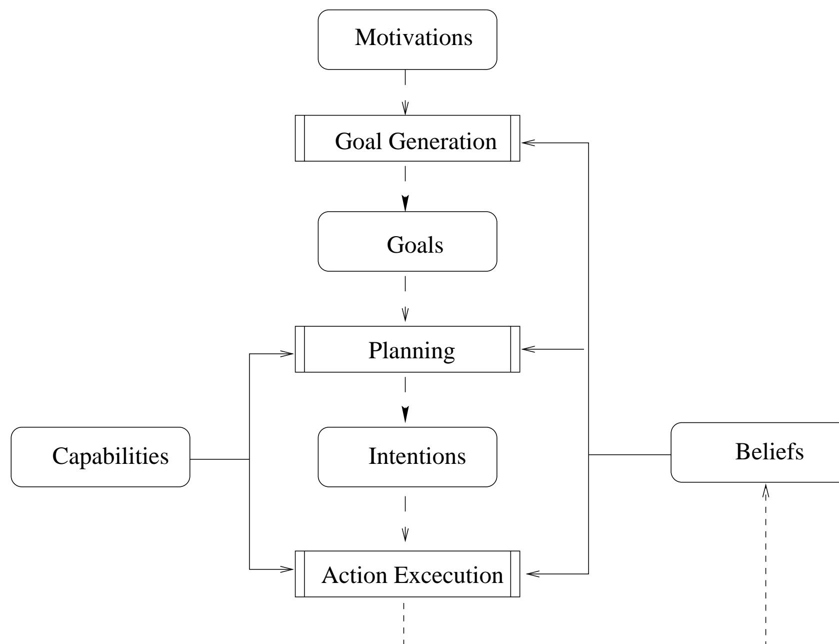


Figure 7.1: Agent's Mental Cycle

7.2.1.1 Mental Categories

- **Beliefs:** They reflect the agent's perception of the external world. We use a simple relational notation to express them. Each belief is a relation tuple having a determined number of parameters or fields. For example, the belief "host 'esteron' is down" can be written as follows:

  `(Host :name esteron :status down).`

  One or more of the belief fields can be designated as a key that allows to identify the belief unambiguously. Beliefs can be asserted and retracted from the belief database.

  Belief querying can be done using logical variables. For example, assuming the above belief is asserted in the agent belief database, then the statement:

  `(Host :name esteron :status ?s)`

  holds with the value of variable `s` bound to 'down'. To avoid ambiguity when using variables, we use `?var` to bind the variable to some field value, and `$var` to use the value to which variable `var` has been bound.

  Finally, logical expressions can be combined together using the usual logical operators. For example:

  `(Printer :name ?p)` **and** `(OutOfOrder :host $p)`

  holds when variable `p` is bound to the name of a printer that is out of order.

- **Capabilities:** They describe the actions that the agent can perform, mainly to interact with the external world. In some way, the execution of an agent action could be seen as an instantiation of a certain capability. The actions that the agent can perform can be either primitive actions, or composite actions organized in *plans*. There are four types of primitive actions.

  1. *Sensors.* They are actions that the agent performs to perceive the external world. Part of the agent's beliefs are provided and maintained through the activation of its sensors. Therefore, a sensor provides a mapping from the changes and events that occur in the network to structured beliefs.
  2. *Effectors.* They are actions that affect the status and behavior of the managed network, for example by changing the configuration parameters of an NE.
  3. *Reactors.* They allow the agent to perform defined actions when specified situations occur on the network. The agent can use reactors to have prompt re-

actions to events that may occur. Precisely, a reactor links a plan of actions to a situation expressed on the agent beliefs.

4. *Calculators.* These are a particular kind of actions that allow to compute or deduce new beliefs from others. For example, suppose that the printing system status is Ok only if the statuses of all the printers in the network are Ok. A calculator can be used to maintain the printing system status belief by continuously checking the statuses of all the printers in the system.

- **Motivations:** Motivations are the essence of agent actions. A motivation lets the agent have preferences towards certain states of its managed network. It can be viewed as an expression that the agent continuously tries to satisfy. When a motivation is violated, the agent tries to figure out the reason of the violation, then generates goals that are believed to help satisfying the motivation when achieved (Figure 7.1).

  At this stage, abstraction is made on which goal generation mechanism is better suited for the agent. This deliberation process should be chosen at a later stage in the agent development according to the application needs. For example, goal generation can be done by comparing the motivation expression to the current beliefs. By analyzing the resulting differences, the agent can determine what goal is to be generated.

- **Goals:** A goal denotes a state that the agent wants to achieve through the execution of a certain plan of actions [TL94].

  Similarly to the goal generation process, abstraction is made on which planning method should be used. At a later stage in the agent development process, it will be decided whether an embedded planning system or a simple search in the agent's plans is better suited for the application requirements.

  We identify two kinds of goals: *achievement goals* and *maintenance goals*. The action plan generated for an achievement goal should only achieve the goal at some moment in time, whereas the plan generated for a maintenance goal should ensure that the goal expression is hold continuously.

  Finally, negative expressions can be used in goal expressions. This can be used for example to cause a preceding achieved goal to be cancelled (or unachieved). Of course, the planning process that is to be chosen has to take into account whether closed world assumption is assumed or not.

- **Intentions:** An intention is an action or a plan of actions that the agent decides to execute in order to achieve a certain goal. Therefore, the intentions corresponding

to a goal are a description of how to invoke a set of agent capabilities. This description is used by the operational layer to perform and control the execution of the generated plan (Figure 7.1).

7.2.1.2 The Mental Cycle Processes

The mental cycle relies on three processes: goal generation, planning and action execution.

- **Goal generation** is a strategic process that constantly watches the belief database to check whether the agent motivations are satisfied or not. The purpose of the goal generation process is to generate the necessary goals as soon as one or more of the agent motivations are violated. The generated goals depend on many factors and may differ even for the same type of motivation violation. Firstly, the generated goals may depend on the reason that caused the motivation to be violated. Different reasons may cause a motivation to be no longer satisfied and the generated goals depend on the actual reason. Secondly, the generated goals may depend on the agent beliefs. This means that even for the same reason that caused a motivation violation, the goal generation process may generate different goals according to the status of the network. Finally, the generated goals may depend on the status of the agent as far as the already generated goals and the running agent actions are concerned. For example, the goal generation process may take into consideration the goals that have been previously generated due to other motivation violations.

  Goal generation can also provide the agent with a certain degree of pro-activeness. Typically, goal generation may anticipate motivation violations before they even occur. This allows to generate preventive goals, which have the advantage of minimizing the overall time during which motivations are violated.

  Therefore, the goal generation process may range from very simple reactive rules associating to each kind of motivation violation the necessary goals to generate, to a complex reasoning process requiring situation analysis and opportunity detection. Learning algorithms can also be incorporated. Obviously, not all the NM applications require the same goal generation algorithm. Yet in this abstract model, the goal generation process is left unspecified until the agent development process reaches a sufficiently advanced stage that allows to select a suitable goal generation algorithm.

- **Planning** allows to generate the action plans that achieve the generated goals. In the general case, the generated plan depends on many factors. One important fac-

tor is the set of capabilities that the agent has at the time the goal is generated. Another important factor is the present beliefs of the agent, which may lead the planning process to select one particular plan among many possible others. Finally, the generated plan may depend on the actions that the agent is already running.

As in the case of the goal generation process, the planning process may range from a simple match between each type of generated goal with a predefined plan to execute, to an embedded planning tool. Similarly, and since there is no one-fits-all planning algorithm for all the NM applications, abstraction is made on which planning process to use until a late stage of agent development that allows to suitably choose a planning algorithm is reached.

- **Action execution** is the process that actually executes the agent actions. This process may also range from a very simple sequential execution of agent action plans to a complex execution environment supporting parallel threads and synchronization mechanisms. The action execution process may also provide support for non-deterministic or uncertain actions that may not execute successfully for example. Again, this is left unspecified until the developer is able to decide of the level of complexity required for the action execution process.

7.2.2 The Operational Layer

The agent Operational Layer is an integrated environment within which the actions intended by the agent are executed. The deliberative layer delivers intentions (or action plans) decided at the planning process to the operational layer. This latter has to initiate and control their execution. Therefore, the operational layer uses the capabilities description to correctly instantiate and launch the action execution. In addition, the operational layer ensures the proper update of agent beliefs following the execution or the completion of an action.

In the case of a sensor, the agent intentions can specify its activation by executing the primitive `startSensor` with the sensor parameters. For example, if the agent wants to activate `BandwidthUsageSensor` between two adjacent points, it has to execute:

```
(startSensor BandwidthUsageSensor :source host1 :dest host2).
```

This will cause the corresponding belief to be created and maintained in the agent belief database. To stop a sensor, the agent should execute

```
(stopSensor BandwidthUsageSensor :source host1 :dest host2).
```

This causes the corresponding belief to be removed from the agent belief database.

Similarly, calculators and reactors can be started and stopped using the respective

primitives `startCalculator`, `stopCalculator`, `startReactor` and `stopReactor`. For example, an emergency reactor can be used to kill user processes that are excessively using a machine resources. It can be launched on host 'dahlia' using:

```
(startReactor KillConsumingProcesses :host dahlia).
```

Of course, there should exist beliefs in the agent telling which are the resource consuming processes on the target host.

Finally, effectors can only be executed in a one-shot way. For example, the kill process effector can be invoked as follows:

```
(effector KillProcess :host dahlia :pid 4585).
```

7.2.3 A BDI-oriented Design Process

The agent development process proposed for our skill based architecture presented in Chapter 4 can no longer be used as-it-is with this BDI-oriented agent model. The two agent approaches have different modeling constructs: skills, capabilities and belief templates in the first, and motivations, goals, intentions in the second. However, we still adopt a top-down process for the BDI approach.

The first stage is identical to that prescribed in Chapter 4 and consists in identifying the different agent roles. Interaction scenarios are then drawn using high-level interactions between the agents.

In the second stage, we associate the necessary motivations to agent roles. Each role is specified using a certain number of abstract expressions that define how the role is properly ensured by the agent. Violations to these expressions mean that the role of the agent is no longer properly ensured. The agent has to constantly watch these expressions.

In the third stage, each motivation is studied to determine the possible situations in which it is violated. For example, if the motivation is expressed as an $(A \wedge B)$ proposition, then it is violated in one of the three situations: A does not hold and B holds, B does not hold and A holds, and both A and B do not hold. Then each situation should be analyzed in order to identify the adequate goals to be generated. It is by studying the goal generation for each violation possibility of each motivation that the appropriate goal generation process can be selected. This stage is referred to as *motivation violation analysis*.

The final step consists in analyzing the planning process. Each goal that might be generated due to a motivation violation is associated to either an algorithm that generates the required action plan, or directly to a predetermined action plan. The result of this stage determines the suitable planning process to be used in each agent's mental cycle. This stage is referred to as *goal achievement analysis*.

7.3 Abstract Agent Design

The role identification phase is identical to that described in Chapter 5 and therefore, we simply reuse the same defined roles. The purpose here is mainly to experiment the BDI approach and not to reach a fully operational agent system. Therefore, this work restricts to the domain agent roles, i.e. it considers only the roles of tester, testee and domain monitoring. The implementation of the domain agent role is sufficient to perceive the main concepts of a BDI agent approach.

The specification of these roles using the abstract agent model is described in the following sections.

7.3.1 The Domain Monitoring Role

Goal Generation To make an agent ensure the role of domain monitoring, it must be motivated to have the status belief for each of the NEs in its domain. We use the same belief templates as in Chapter 5: an `inMyDomain` belief specifies an NE in the agent's domain, and a `networkElementStatus` belief specifies the status of a NE. Accordingly, the domain monitoring motivation can be expressed as:

`(inMyDomain :ne ?e)` $\Leftrightarrow$ `(networkElementStatus :ne $e)` *(M1)*.

This motivation is formulated as a $p \Leftrightarrow q$ expression, which is violated either when p holds and q does not hold, or when q holds and p does not hold. The first case occurs when a new NE is added to the agent's domain. This causes the agent to create the following goal:

`(achieve (networkElementStatus :ne $e))` *(G1)*.

The second case in which *(M1)* is violated corresponds to an NE that is removed from the agent's domain. For this situation, the generated goal is:

`(achieve (not (networkElementStatus :ne $e)))` *(G2)*.

Intentions To achieve goal *(G1)*, the agent needs a sensor to perceive the status of the NE. The `StatusMonitoringSensor` is defined for this reason. To start it on an NE, the agent must execute:

```
(startSensor StatusMonitoringSensor :ne $e)
```

which will create and maintain the missing `networkElementStatus` belief, thus leading to the satisfaction of the motivation.

Goal *(G2)* can also be easily achieved, assuming a closed world assumption for the `networkElementStatus` beliefs, by executing:

`(stopSensor StatusMonitoringSensor :ne $e)`.

7.3.2 The Tester Role

Goal Generation An agent A ensures the tester role regarding a testee agent B when it has a belief on the reliability of agent B. As in Chapter 5, such belief is written as `(sldStatus :agent B :status reliable)`. Therefore, to make agent A ensure the tester role, it should simply be motivated with:

`(sldStatus :agent B)` *(M2)*.

Satisfying this motivation requires the generation of multiple successive goals. Firstly, the tester should have a belief containing a set of, let us say, three representative elements in agent B's domain. Therefore, the following goal has to be generated:

`(achieve (selectedNetworkElements :agent B :element1 ?elt1 :element2 ?elt2 :element3 ?elt3))` *(G3)*

Once this goal is achieved, or if it is already achieved, the next step is to have agent B's belief on `elt1`, `elt2` and `elt3`. For each of these elements, a goal

`(achieve (believedNetworkElementStatus :agent B :ne $elt))` *(G4)*

is generated, where the belief `(believedNetworkElementStatus :agent B :ne $elt :status DOWN)` tells that agent B believes that the status of `elt` is down.

Afterwards, agent A has to start monitoring the selected NEs. This is done exactly in the same way as for the domain monitoring role, i.e. by generating goal *(G1)*.

Finally, if all the above goals are achieved, then all the beliefs required to deduce agent B's reliability status are available. The goal:

`(achieve (sldStatus :agent B))` *(G5)*

can be generated and successfully achieved.

Later in time, the domain of agent B might change. One (or more) of the three elements, let us say `elt1`, that have been chosen in the `selectedNetworkElements` may be replaced by another element, e.g. `elt4`. When agent A get informed of the change, the motivation becomes violated since the status belief on `elt4` is missing and, hence, the SLD status of agent B cannot be maintained. Consequently, agent A will successively generate the following goals:

- `(achieve (not (believedNetworkElementStatus :agent B :element $elt1)))` *(G6)*.
- `(achieve (not (networkElementStatus :ne $elt1)))`, which makes the agent stop the `StatusMonitoringSensor` on `$elt1`.

- (achieve (believedNetworkElementStatus :agent B :ne $elt4)).
- (achieve (networkElementStatus :ne $elt4)).

Intentions To achieve goal *(G3)*, we use the same approach as in Chapter 5, i.e. the tester agent asks the testee to establish a list of selected NEs, then subscribes for them. The way this approach is mapped in this context consists of two steps. In the first step, the tester agent recursively delegates the same goal *(G3)* to the testee agent. The way the testee agent achieves *(G3)* is described in the next subsection. In the second step, the tester agent sends a subscription message for the selectedNetworkElements belief of the testee agent. As soon as the testee agent replies with a tell containing the selectedNetworkElements belief, the tester inserts it into its belief database. Hence goal *(G3)* becomes achieved.

Goal *(G4)* can be achieved by sending a subscription message to agent B, asking it for the status of the required NE. Each time agent B sends a tell message with a networkElementStatus belief on a selected NE elt, agent A converts it to a (believedNetworkElementStatus :agent B :ne $elt :status ...) belief.

Finally, goal *(G6)* is achieved in the opposite way. The tester agent first unsubscribes for the (networkElementStatus :ne $elt1) belief in agent B, then retracts the corresponding believedNetworkElementStatus from its belief database.

7.3.3 The Testee Role

The testee role is a passive role, in the sense that there is no need to motivate the testee agent to make it ensure the role. In fact, the testee agent only replies to the queries sent by the tester agent, and achieves what it is asked to do. For example, when the tester receives the message containing goal *(G3)*, it integrates the goal in its mental cycle and tries to achieve it. The achievement of this goal is simply done by activating the corresponding calculator TopThreeImportantElements. This persistent calculator creates the belief (selectedNetworkElements :agent B) and updates it whenever the domain of agent B evolves.

In addition, the testee agent must be able to manage multiple subscriptions originating from different testers. For example, suppose that there is another agent, agent C, that is also testing agent B. If later in time, agent C is no longer motivated to test agent B, then agent C should unsubscribe on agent B's belief (selectedNetworkElements :agent B), and then tells that agent B no longer needs to have this belief achieved. However, agent B should be aware that agent A still needs the belief and therefore, it should not stop maintaining it.

Although this problem frequently occurs in distributed agent-based design, we still do not provide a high-level solution to tackling it. Instead, we postpone its resolution to the implementation phase.

The implementation is based on an agent programming framework that supports many interesting agent concepts. This Java-based agent programming framework is called JACK. The next section provides an overview of JACK concepts, while the section after details how JACK was applied to implement the proposed BDI abstract design.

7.4 Agent-Oriented Programming with JACK

7.4.1 JACK Concepts

JACK [BRHL99, Age98] is an agent programming language based on Java. It extends the Java object-oriented language by providing agent programming constructs such as beliefs and plans. Agents written in JACK language are first compiled into Java, then they are compiled and executed like any other Java program.

The constructs that JACK defines to support agent programming are *plans*, *events*, *belief databases* and *agents*.

Belief Databases Beliefs in JACK are stored in relational databases. JACK allows to easily define belief relations and offers facilities to make them generate the necessary events, for example when a new belief is added, updated or removed.

It is also possible to define database queries that can be used either to retrieve the agent beliefs or can be used in the logical expressions of goal statements.

Events JACK defines several types of events. The simple `Event` is a general-purpose event that can be used for several reasons, mainly to signal database changes that are relevant for the agent operation. For our special purpose, we call the events that are generated due to a belief change as *database events*.

`MessageEvents` are used for the inter-agent communications. They are the mechanism that enables agents to communicate in JACK. To send a message event, e.g. of type `SampleMessageEvent`, the built-in primitive `@send` is used as follows:

```
@send (receiverAgent, sampleMessageEventInstance)
```

where `receiverAgent` is the name of the agent to receive the message, and `sampleMessageEventInstance` is a created instance of `SampleMessageEvent`.

In JACK, every type of message event should have a plan that can handle received instances of this event.

Another important type of events for the agent BDI behavior is the `BDIGoalEvent`. This type of events is handled by the agent not as simple events, but rather as goals that the agent has to achieve. Such events are raised using *goal statements*. Goal statements take a database query expression and a `BDIGoalEvent` instance as arguments. JACK offers the following goal statements:

- **@achieve**, which first checks whether the belief query holds on the current belief state or not. If it holds, then the statement successfully returns. If it does not hold, the instance of the BDI goal is posted. This causes the agent to look for a suitable plan capable of handling this goal. A plan that handles such a `BDIGoalEvent` has to execute a sequence of actions that bring the agent beliefs into a state in which the belief query succeeds.
- **@insist**, which is similar to the `@achieve` statement, except that the agent checks again the query against its belief state after the plan is executed.
- **@test**, which posts the BDI goal only if the query could not be evaluated to true or false in the current beliefs. The handling plan should then provide actions that allow to determine whether the query holds or not.
- **@determine**, which is used to find a unification of the belief query for which the plan that is executed when the BDI goal is posted succeeds. This means that the agent tries all the possible query unification, until the executed plan succeeds for a particular unification.

Plans Plans are pre-established action procedures that the agent can use to perform a certain task or to respond to events that raise during its execution. Each plan handles a unique event type. When an event is posted, the agent first checks the plan preconditions, and if they match the current status of its belief, the plan is included within the set of possible plans that can be executed. If there are many possible plans, only one plan is selected.

The plan can contain many kinds of actions. Database actions allow to manipulate the beliefs of the agent by asserting and retracting beliefs. In addition, it is possible to query the databases to search for a particular belief, or a set of beliefs.

The plan may contain goal statements. When a goal statement is encountered during the plan execution, the agent first tries to achieve the goal before continuing the execution of the plan.

A plan can also post events locally in the same agent, or can send message events to other agents. Synchronization is also supported through the use of the `@waitfor` statement, which waits until a certain condition becomes true.

Finally, a plan can contain raw Java expressions and can call methods in the agent class.

The Agent An agent is defined as containing a set of plans that handle the events posted or received from the other agents. It also has a set of database instances that contain its beliefs during its execution. When the agent is running, it continuously performs a loop in which it waits for an event to occur, search for a suitable plan that processes the event or achieves the goal, and then executes that plan. The plan execution may lead to the generation of new events that are processed either synchronously or asynchronously depending on how the event was generated.

7.4.2 Advantages of JACK

In our application, JACK was chosen mainly for two reasons. The first reason is that it directly supports BDI concepts. For example, the belief database uses the same relational model we use in our abstract agent model. Moreover, JACK allows for the definition of goal-oriented behavior. It is an open agent framework, since it does not impose a particular way of goal generation or goal achievement mechanisms. Instead, these issues can be fully taken under control with Java programming. For example, database queries are defined at the same time the database itself is defined. Therefore, a database can be endowed with arbitrary complex queries if necessary. Similarly, one can define as many goal events as needed, and associate these goals to any expression that can be formulated using the database queries.

The second reason is that JACK is a lightweight framework compared to other agent languages that offer heavyweight reasoning capabilities for example. The use of plans is particularly advantageous in a domain such as network management where most of the performed tasks have a procedural nature.

Another reason that contributed to choosing JACK is that it is inspired by PRS (Procedural Reasoning System), which was used in several concrete applications [GI90, SRI] including applications in the network management field.

Finally, JACK has the advantage of being fully implemented in Java, thus enabling the easy integration of external Java classes and components.

7.5 Implementation with JACK Agents

7.5.1 General Considerations

While mapping our agent specifications detailed in Section 7.3 to JACK concepts, some aspects are translated in a general way, while others are translated in a case-by-case basis. This section presents the generally-used translations.

7.5.1.1 Agent Communications

The only means of communication in JACK are *MessageEvents*. However, `MessageEvents` do not offer built-in facilities to handle subscriptions and goal integration into the agent's mental cycle. Therefore, to map agent communication messages in JACK, our implementation used normal MessageEvents that are handled with specific plans. These plans take into account the performative semantics of the corresponding message. For example, the message sent to an agent to subscribe for the status of an NE is mapped into a `SubscribeNetworkElementStatusMessage`. This message is handled by a plan that first checks whether the required belief exists or not, then adds the subscriber agent to a list of subscribers which is stored as a private attribute in the agent Java class. The subscribers list is indexed according to the NEs. When a change occurs in the `networkElementStatus` database, the subscribers list is checked to find if there are subscribers on the changed NE. If so, an appropriate message is sent to all the NE subscribers.

7.5.1.2 Motivations Handling

In our implementation, we simulated the motivational behavior using goal events and database events. Each motivation is implemented using a goal event that activates the motivation, and another goal event to discard it from the agent's mental cycle. Theses goals are handled by plans that lead to the motivation satisfaction, or respectively, to its removal. To make the agent detect that the motivation is violated, special database events are defined and triggered when violation situations occur. These events are also handled by dedicated plans that are able to resolve the problem. Obviously, the plans that handle a motivation violation are triggered only when the motivation is active in the agent's mental cycle.

7.5.1.3 Operational Layer

JACK does not explicitly support the definition of agent capabilities. Instead, we used JACK concepts to have the same function as that of the operational layer. In that, the

different calculators are replaced by database-event driven plans. For example, the `TopThreeImportantElements` calculator is replaced by a plan that handles an event corresponding to the addition or removal of an NE in the agent's domain, and re-computes the three selected NEs, then updates the corresponding belief accordingly. Reactors could be implemented in a similar way.

The `StatusMonitoringSensor` was implemented in a separate class and runs in its own execution thread. Special methods are defined in the agent class to start and stop a sensor within a plan definition. A sensor can update the agent's beliefs by invoking belief database methods. Effectors can be implemented in a similar way.

7.5.2 BDI-oriented Implementation

7.5.2.1 Implementation of the Domain Monitoring Role

The motivation *(M1)*, which makes the agent ensure the domain monitoring role is replaced by two goal events: `StartDomainMonitoringGoal` and `StopDomainMonitoringGoal`.

In JACK, it is not possible to specify composite expressions, such as *(M1)*, in an `@achieve` statement. For this reason, a new belief `(DomainMonitoring :is enabled)` is created when the domain monitoring is enabled. Therefore, motivating the agent to ensure the domain monitoring is equivalent to the following JACK statement:

```
@achieve ((DomainMonitoring :is enabled), StartDomainMonitoringGoal).
```

The plan that handles this goal simply loops on all the NEs listed in `InMyDomain` beliefs and starts the monitoring sensor for them. Similarly, the plan that handles `StopDomainMonitoringGoal` does the same loop and stops the monitoring sensor on all the NEs in the agent's domain. This plan is invoked as a response to such statement:

```
@achieve ((not (DomainMonitoring :is enabled)), StopDomainMonitoringGoal).
```

To ensure that new NEs added to the domain are included in the domain monitoring, and that the monitoring stops on the NEs that are removed from the domain, the `InMyDomain` database generates an `InMyDomainUpdate` event whenever an NE is added or removed from the agent's domain. This event is handled by a plan that accordingly starts or stops the monitoring sensor on that NE.

7.5.2.2 Implementation of the Tester Role

Similarly to the domain monitoring role, to motivate an agent A to ensure the role of a tester for a testee agent B, the following achieve statement should be used:

```
@achieve ((sldStatus :agent B), SldOnAgentGoal).
```

This statement generates the goal event `SldOnAgentGoal`, which makes the agent be a tester of agent B. The plan that handles this goal event generates equivalent goals to those described in Section 7.3.2.

Getting the Three Representative Elements. The following JACK statement allows to have a selection of three elements in the testee's domain:

```
@achieve ((selectedNetworkElements :agent B), SelectThreeElementsGoal).
```

In each agent, there are two distinct plans that deal with goal `SelectThreeElementsGoal`.

First, the plan for the tester role simply sends a `SubscribeThreeSelectedElementsMessage` to the testee agent. The semantics of this message combine both, to `achieve` goal *(G3)* and to `subscribe` for the resulting `selectedNetworkElements` belief. Afterwards, the plan waits until the reply of the testee agent is received and processed. The testee agent replies with a `TellChangeInSelectedNetworkElements` message event containing the three selected NEs. A plan in the tester agent handles this event by creating the corresponding `selectedNetworkElements` belief. The same process occurs when there is a change in the three selected NEs in the testee domain.

Second, the plan for the testee role finds out the three candidate NEs and generates the corresponding `selectedNetworkElements` belief. The content of this belief is then sent to the tester agents. To ensure that this belief is maintained, another plan catches the `InMyDomainUpdate` database event to update the `selectedNetworkElements` belief if necessary.

Getting the Testee's Beliefs on Network Element Statuses. The next step consists in obtaining the testee agent's beliefs on the three selected NEs. This step is initiated by the following statement:

```
@achieve ((believedNetworkElementStatus :agent B :ne $elt),
BelievedNetworkElementStatusGoal).
```

The plan that handles this goal sends a `SubscribeNetworkElementStatusMessage` to agent B, and waits until the reply is received and processed. However, JACK does not allow to express explicit parallel tasks in plans. Therefore, the tester agent would not be able to send the subscribe message for the second NE unless the reply for the first NE is already received and processed. This causes serious performance degradation since the tester agent simply hangs waiting for communication replies, while it is possible theoretically to send the subscription for the three selected NEs at the same time, and wait for the replies asynchronously. This is what was indeed done in the current implementa-

tion. Instead of the above synchronous `@achieve` statement, the subscription message is directly sent as the following:

```
@send (B, SubscribeNetworkElementStatusMessage).
```

Since communication in JACK is asynchronous, the three subscription messages can be sent and replies processed asynchronously without blocking the initial plan.

The subscription message is handled in the testee agent by registering the sender as a subscriber to the NE status and sending back a `TellNetworkElementStatus` message. Another plan that handles the database event `NetworkElementStatusChange` (which is generated each time a belief `networkElementStatus` changes) ensures to notify all the subscribers for an NE status of changes that may occur.

Later on, when the tester agent receives the `NetworkElementStatusChange`, the handler plan maps it into a `(believedNetworkElementStatus :agent B :ne $elt :status ..)` belief.

We refer back to the problem of managing multiple subscriptions by agent B, mentioned in Section 7.3.3. If agent A unsubscribes in agent B for the three selected NEs, then agent B should not stop maintaining this belief if another agent C is still subscribed. For this purpose, the agent Java class contains a table of all the subscriptions for the `selectedNetworkElements` belief. When this table becomes empty, the agent stops maintaining the `selectedNetworkElements` belief and removes it from its database.

Computing the Testee SLD Reliability Status After launching the sensors on the selected NEs in agent B's domain, agent A has all the necessary information to deduce the reliability of agent B. The `ComputeSld` plan is triggered whenever a change in the status of a selected NE is updated. The plan compares the new status with the last recorded belief received from the testee agent. By combining this comparison with the results of the other selected NEs, the reliability status of the testee agent is deduced and used to update the `agentSldStatus` belief.

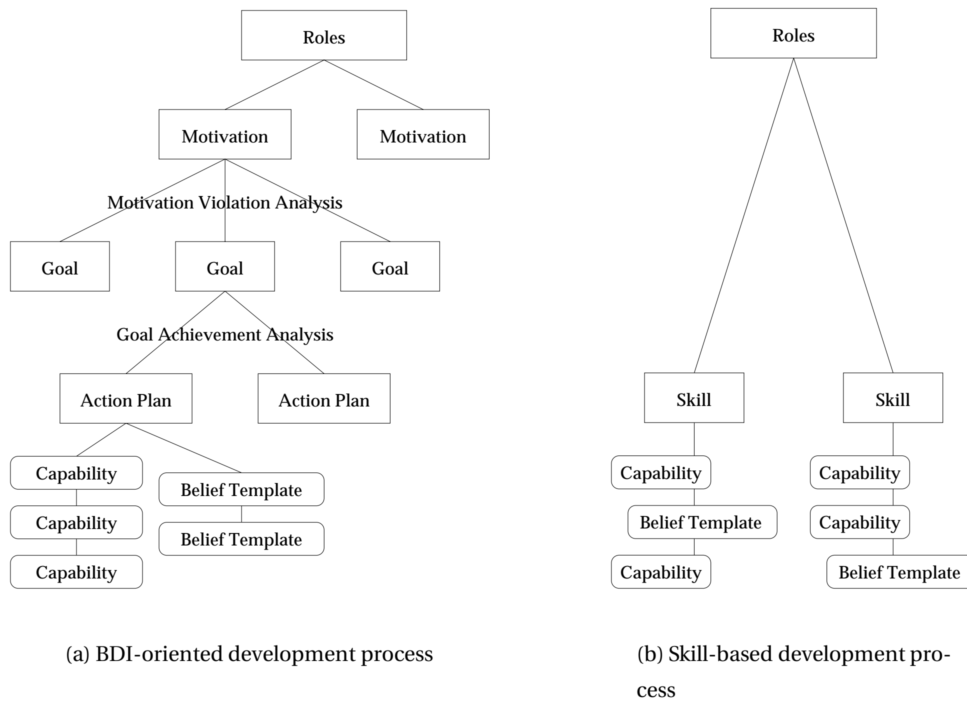
7.6 Discussion

The previous sections described how the SLD case study was reconsidered, down to the implementation level, using a BDI agent approach. We propose to discuss the BDI approach according to three aspects. The first aspect concerns the impact of the BDI approach on the agent development process, and the suitability of the BDI constructs for NM applications. The second aspect concerns the impact of a BDI-oriented develop-

ment on the behavior and functionality of the implemented agents. The third aspect concerns implementation issues.

7.6.1 Impact of a BDI-oriented agent development process

Figure 7.2, depicts the abstraction refinement tree used in a BDI-oriented development process, and that used in a skill-based development process, respectively presented in Sections 7.2.3 and 4.6. The BDI-oriented development process (Figure 7.2(a)) includes multiple abstraction levels and requires more refinement stages than the design process of the skill-based agent architecture (Figure 7.2(b)).



(a) BDI-oriented development process

(b) Skill-based development process

Figure 7.2: Agent development processes

These additional refinement stages yield to a more fine-grained agent design in the case of a BDI model. In addition, a BDI agent approach directs the designer's attention to different aspects than the skill-based approach.

- The mapping from role description to abstract motivations is a formalization process. This formalization is a solid and reliable basis to derive the other lower abstraction levels. However, its main issue is to faithfully transcribe the semantics of

an informal role into formalized motivations. Moreover, complex roles might be difficult to formalize.

- The motivation violation analysis allows to determine how to satisfy the agent motivations, and how to react to situations where they are violated. Given that the agent motivations are formulated using logical expressions, the motivation violation analysis determines exactly all the relevant belief changes that the agent has to watch, and to which it has to react. Therefore, the obtained design is efficient, since only the relevant belief changes are analyzed.

  The obtained design is also robust. Robustness is the ability of a software to perform correctly even under sever conditions of operation, e.g. overloaded CPU, network problems, erroneous input, etc. The motivation violation analysis allows to pinpoint situations of erroneous input data or severe operational conditions that potentially can prevent the agent from correctly insuring its roles.

  The output of the motivation violation analysis process is the set of goals that has to be generated and achieved in response to motivation violations. Goals are themselves abstract expressions.

- The purpose of the goal achievement analysis phase is to determine how the agent can achieve generated goals. The advantage of goal achievement analysis is to design action plans with a clear purpose for each plan. This purpose corresponds to a particular goal expression. Therefore, the developer of a plan concentrates its design on how to achieve the target goal, assuming that the plan is correctly executed.

The general conclusion is that a BDI-oriented agent design is purpose-oriented, where each refinement of the abstract design corresponds to more precise purposes. This is to be compared to the task-oriented design of the skill-based approach, in which, the agent designer focuses on the tasks that the agent has to execute. A purpose-oriented design focuses on the reason or the purpose of executing a certain task. This shift in abstraction from *how* to execute a task, to *why* to execute a task affects the design of the agent, as well as the way it operates when the agent is running.

The counterpart of a BDI design is complexity, which clearly appears through the large number of motivation and goal expressions obtained in Section 7.3. BDI-oriented design of agent applications tend to be lengthy since it requires multiple refinement stages. Of course, this complexity could be reduced using software developement assistance tools, which may automate certain tasks of motivation violation and goal achievement analysis. The main issue to be faced before such tools can exist is a common uni-

form BDI model for agent applications. Such uniform model is far from today's reality in the agent field.

7.6.2 Impact on the agent operation

Tractability. The purpose-oriented design of the agent consists in having a clear goal expression for each action plan. The impact on the agent operation is that the agent activity is perceived as goal-oriented. There is an explicit reason for each action that the agent executes. Each executed action is related to a plan definition, each plan is executed to achieve a generated goal, and each goal is generated to satisfy a violated motivation. Each motivation is eventually related to a role. Therefore, each action that the agent initiates is related to a certain purpose that can be perceived at different abstraction levels, ranging from the associated plan, to the global role of the agent. We conclude that a BDI approach not only allows to provide a better agent design, but also allows to efficiently track the actions of the agent. Needless to emphasize how this property is important in NM to be able to track the management operations initiated by an NMS.

Adaptability. Another aspect of a BDI implementation concerns the adaptability of the agent. We showed during the skill-based implementation of SLD in Chapter 5 how the resulting agent system was adaptable to changes in the NE domain distribution. Similarly, in this BDI implementation, NEs can be dynamically inserted and removed from agent domains and the agent system automatically adapts to these changes. In the skill-based agent architecture, the adaptability property was related to the subscription/notification mechanism implemented therein. In the BDI approach, adaptability is provided using another mechanism: motivation violation/goal generation. This motivation violation/goal generation mechanism is more abstract than the subscription/notification mechanism. The agent activity has no longer to handle low-level belief subscriptions and notifications. Instead, these are abstracted into the goal generation process, and are handled automatically according to the agent's motivations. Moreover, the goal generation process watches only the belief changes that are relevant to the active motivations. If a motivation is removed, the goal generation no longer needs to watch the related belief changes.

Autonomy. The BDI approach allows to have enhanced agent autonomy. The agent missions are specified using abstract expressions of motivations and goals. Motivations and goals specify only what is required from the agent. This is to be compared to the direct task invocation used in the skill-based architecture. Of course, this aspect of au-

tonomy would be even more powerful if the designer of the BDI-oriented agent does not have to explicitly develop plans that directly achieve agent goals. A more powerful approach would be to endow the agent with a planning capability that allows to derive a suitable plan for each generated goal. In our case, a planning tool is not required since the required plans could be easily developed. Moreover, planning has to be used only when necessary since planners are heavyweight processes that are difficult to design.

Similarly, the motivation violation analysis allows to detect contingency situations that the agent may encounter. Moreover, the designer of the agent can provide multiple plans to handle the same goal. If a plan fails to achieve the goal due to unpredicted errors, the planning process of the agent's mental cycle could select another suitable plan to achieve the goal.

Proactiveness. As explained earlier in this chapter (Section 7.2), goal generation can support a proactive agent behavior. By generating preventive goals before a motivation is about to be violated, the agent can decrease or avoid the time during which a motivation is violated. In NM, proactiveness allows the agent to be prepared to situations where necessary control operations have to be performed.

It is important to note that a BDI approach does not automatically lead to proactive agents. What we are claiming is that our proposed BDI model provides support to include the necessary mechanisms to endow the agents with a proactive behavior. These proactiveness mechanisms are not part of the BDI model, but have to be studied as additional agent features.

There is no single mechanism to provide proactive agents. One possibility is to include a learning engine that observes the behavior of the agent beliefs to discover belief change patterns that predict a motivation violation. Such a technique is already used in NM, for example to proactively detect disk failures, network congestion, etc. The goal generation process can therefore generate preventive goals that avoid the impact of a motivation violation.

Another possibility to obtain a proactive behavior is to endow the goal generation process with an additional engine that continuously assesses both the risk implied by a motivation violation, and the impact of this violation on the global behavior of the network. Motivation violations that lead to the most serious damage to the network are considered with the highest priority, and preventive goals can be generated to reduce the probability and the risk of such violations. A typical scenario is that of the management of a sensitive network server such as the network file server. The assessment of the risk of a dysfunction of this server shows that all the other network services are seriously affected. Moreover, possible dysfunction of the server may raise due to disk or CPU over-

load. Therefore, a preventive goal that the agent might generate is to prohibit users from accessing the server host for a different reason than using the network file server.

7.6.3 Implementation Issues

The mapping of the BDI abstract design into a JACK implementation showed how the implementation can be different from the design. Even though JACK supports many agent concepts such as goals, plans, beliefs, etc., many parts of the design could not be properly mapped during implementation.

- **Unstructured communication**

  One major issue with JACK was that communication is `MessageEvent` based and not speech-act oriented such as KQML. The implemented communication in this case study lacks structure, and the semantics of the exchanged messages are completely encapsulated in their respective handling plans. Many plans had to handle subscription semantics, while subscription could be handled once-for-all if using a KQML-like language.

  This shows the advantage of using a high-level ACL (Agent Communication Language) compared to a non-structured communication mechanism. An agent ACL (Agent Communication Language) allows to shift the developer of agent-based applications from the low-level tasks of implementing inter-agent facilities, such as belief subscription and notification.

- **No support for motivations**

  Another issue is that motivations had to be implemented using plans, where each plan simulates the goal generation process for a particular motivation. This is contrary to the idea of goal generation, which is to have a unique process instead of multiple plans. A unique process for goal generation is simpler to maintain.

- **No support for composite goal expressions**

  As it is not possible to specify a composite goal expression within a JACK goal statement, we had to define an otherwise-useless belief template that replaces the composite goal expression with a unique simple expression. This leads to additional plans to maintain the coherence with the newly created belief.

The moral from this implementation is that adopting a particular agent framework implies to be tightened to the concepts defined in this framework. This emphasizes the reason for which we opted to develop our own skill-based agent architecture for NM,

instead of using an existing agent framework. This also explains why appliers of agent technology tend to build their agent support architecture from scratch, instead of reusing existing agent frameworks.

7.7 Summary and Conclusion

The contribution of this chapter is threefold. First, we proposed a BDI abstract agent model suitable for NM purposes. This proposed mental cycle can support different mechanisms for goal generation, goal planning and action execution. We also provided a multiple-stage BDI-oriented development process that uses different abstraction levels to specify the agent behavior. Second, we used this abstract BDI model to provide a BDI-oriented design of the SLD case study, different than the design proposed in Chapter 5. The BDI-oriented design is then implemented using JACK, an agent development tool that supports BDI agent concepts. Third, we discussed the usefulness of a BDI approach, and compared it in relevant aspects to the skill-based architecture that we proposed.

Interestingly, using different agent architectures, skill-based and BDI-based, lead to completely different modeling approaches. While with the skill-based agent architecture, the main focus was directly on *how to* let the agent achieve its role, the BDI-based approach added a further abstraction layer that analyzed *why* the agent performs actions when undertaking its roles. This is achieved through the mapping from agent roles to abstract motivations, then analyzing the different situations in which these motivations are violated. Understanding the essence of agent actions by analyzing its motivations lead to a deeper understanding of the case study. Moreover, this BDI approach allows to obtain a more robust agent design provided that a careful motivation violation analysis is performed.

The BDI approach leads to purpose-oriented agents that initiate actions according to explicitly formulated goals and motivations. In addition, the proposed BDI model can integrate enhanced mechanisms for goal generation and planning that improve agent autonomy and endow it with a proactive behavior.

We however maintain that the skill-based architecture excels with its dynamic skill-loading feature to support flexibility and genericity. Providing an equivalent flexibility using modular skills in a BDI approach seems to be a challenge, since the different mental categories are strongly interrelated in the mental cycle. This strong interrelation hinders a modular approach similar to our skill-based architecture.

Accordingly, we believe that our skill-based architecture succeeded to provide a com-

promise between NM requirements that strongly promote flexibility and dynamism, and agent concepts that at the same time that they provide strong means of abstractions, are also difficult to adopt in a general context.

Another conclusion, that is going to be emphasized in the next chapter, is that an agent-oriented methodology and development process strongly depends on the type of agents to be used.

We are interested in a similar comparison of our skill-based architecture with a Mobile Agent approach. The next chapter considers an MA implementation of the PVC case study and compares it to the implementation provided in Chapter 6.

Chapter 8

Assessment of Mobile Agents through the PVC Provision Case Study

Mobile agents are an interesting metaphor for the PVC case study. An MA is a natural way of modeling a unique ATM operator that sequentially connects from a switch to the next one in order to configure PVC fragments. In this chapter, we propose to discuss an MA approach to the PVC case study and to compare it to the skill-based approach described in Chapter 6. As far as mobility is concerned, a skill-based agent is considered as a Stationary Agent (SA). Therefore, the comparison to the MA approach is focussed on the mobility aspect.

8.1 Motivation

Mobile agents, considered as autonomous programs capable of moving from one host to another during their execution, have been experimented in several management applications such as the optimization of monitoring traffic, configuration management and performance monitoring and control (see Chapter 3, Section 3.3). Despite the hype that surrounds MA technology, still only few references in the literature assess on concrete basis the real advantages and shortcomings of using MAs. Proponents of MAs advocate the enhanced flexibility and the amelioration of performance and efficiency compared to classical approaches based on a central manager and a rigid set of management pro-

tocols. In that sense, MAs allow to reduce the generated management traffic by moving the client knowledge close to servers resources and accessing them locally [BP97].

In contrast, opponents of MAs argue that agent migration is likely to generate in many cases a lot of traffic and to cause serious control problems in MA-based management systems, such as load-balancing and resource access control. They assert that there is no application that can be designed with MAs and cannot be designed with classical approaches. Consequently, the added-value of MAs is not clear enough to justify its deployment in industrial NM applications.

Likewise, proponents of SAs advocate their innate support for cooperative management tasks and the high-level communication protocol they use. SAs are likely to reduce the generated management traffic since they exchange only concise and high-level messages and do not require recurring interactivity with each other. In contrast, opponents claim that SAs lack the flexibility that MAs offer thanks to their migration capability.

In reality, there are neither available criteria nor the required expertise that help answering the question of whether to adopt autonomous migrating agents instead of distributed cooperating stationary agents. Arguments in favor of one approach can easily be counter-balanced by arguments in favor of the other one. To our knowledge, the only relevant work that provides an objective way to evaluate the cost of selecting the MA approach in an NM problem is that described in [BP97]. Although the results reported therein are original and useful, they are limited to traffic-based criteria, namely the global generated traffic and the traffic generated around the central management station.

The thesis of Papaioannou [Pap00] is another work that objectively discusses “the argument for mobility”. The thesis evaluates the use of MAs and mobile objects to build and integrate software systems that support distributed manufacturing enterprises. Papaioannou uses real-work case studies to assess mobile code abstractions [PE00]. These case studies are oriented towards building manufacturing information systems. We propose in this chapter a similar approach specific to the NM context, and we use assessment criteria relevant to NM software.

This chapter indeed provides a comparison between the two agent approaches, mobile and stationary, that goes beyond the obviously important but not unique criterion of generated traffic, to consider other quantitative and qualitative parameters. In order to have credible and arguable comparison results, we use a representative case study as a basis to evaluate the strengths and weaknesses of both agent approaches. In Chapter 6, we have implemented an SA-based approach to the PVC provision case study, for which MAs seem to be a suitable approach as well. Therefore, we propose to base our comparison on this case study. Interestingly, it appeared that an MA-based implementation

of the PVC provision problem has been already accomplished by the Perpetuum Mobile Procura group [PMP], and published in [PLBS98].

Since the algorithm used in [PLBS98] is not exactly the same as we used in Chapter 6, we produced an adaptation of the SA approach in which the PVC reservation phase is removed. In this way, the predominant difference between the MA and SA approaches is reduced to the mobility factor, which leads to a more focussed and consistent comparison.

This case study is representative of a suitable NM application for both approaches. A traditional approach would consist in a centralized management station from which the human administrator directs the multiple management queries to the consecutive ATM switches in order to create the PVC. As far as the number of exchanged messages is concerned, both MAs and SAs are used to localize these queries close to the switches, thus avoiding the multiple configuration queries to be transmitted over the network links. MAs are suitable for this problem because they may be used to accomplish distributed tasks by travelling across a set of hosts and interacting locally with these hosts. SAs are suitable because they are used to accomplish distributed tasks by cooperating with each other. Both approaches are proposed to overcome centralization and micro-management issues, typical for classical NM paradigms.

Although this comparison is based on a particular case study, most of the results obtained and discussed in Section 8.3 can be extrapolated to the general use of SAs and MAs in the NM context.

8.2 Mobile Agent and Static Agent Approaches to PVC Provisioning

8.2.1 The Case Study

We reuse the same scenario adopted in Chapter 6 (Section 6.2). Every end-user is represented by a User Agent (UA) that captures the user request for an end-to-end connection and maps it to PVC parameters. These parameters are transmitted to the ATM network side in a PVC creation request. For the SA approach, the ATM network operator is represented by SAs that accept this request and cooperate to achieve it. For the MA approach, it is necessary (although it was not mentioned in the original paper [PLBS98]) to introduce an additional entity that accepts the requests of the UA and creates the necessary MAs. We refer to this entity as the *Mobile Agent Manager (MAM)* . Of course, there is no restriction that prevents from having multiple MAMs deployed throughout the ATM

network. In this case, the UA contacts the nearest MAM, which then launches an MA in response to its request.

8.2.2 The Mobile Agent Approach

8.2.2.1 The Mobile Agent Framework

In any Mobile Agent framework, there is a need for a server that can receive MAs and allows them to move to other sites in the network. In the considered MA framework this server is called the Mobile Code Daemon or MCD. When the MCD receives an MA, it prepares the necessary resources for its execution and keeps track of relevant information. When an MA wants to migrate to another site, the MCD is responsible for contacting the MCD of that site and to transfer the agent code and state. If the transfer process ends successfully, the local copy of the MA is then destroyed.

When an MA is running on a host, it can access the managed resources in a uniform way via a Virtual Managed Component (VMC). The VMC provides an abstract bridge between the MA and the managed resources, in such a way that the MA can handle any NE without having to know the details of its specific management interface. The framework offers the possibility to upload new VMCs or new versions of VMCs at any time. All MA transfers are performed using the Java RMI (Remote Method Invocation) mechanism.

8.2.2.2 PVC Provision with Mobile Agents

As indicated in Section 8.2.1, the scenario begins with the UA sending a request to create an end-to-end PVC to a certain destination. The MAM receives the UA request and feed the PVC parameters to a newly created MA.

The way the MA establishes a PVC is depicted in Figure 8.1. The created MA travels to the ATM switch to which the source user host is connected. The MA carries the user requirements in terms of traffic quality of service. It starts by establishing the cross-connect on the first switch, where it determines the VPI/VCI values for the incoming and outgoing virtual channel link. Once the cross-connect is established on a switch, the MA migrates to the next switch on the route carrying the additional VPI/VCI couple used for the outgoing virtual channel link of the previous switch. These values are then used to create the incoming virtual channel link in the current switch.

There is only one physical link that the MA traverses to migrate from an ATM switch to the next one on the PVC route.

The process of cross-connect creation is iterated for every switch on the route between the two end-hosts until the last switch is reached and configured. Then the MA

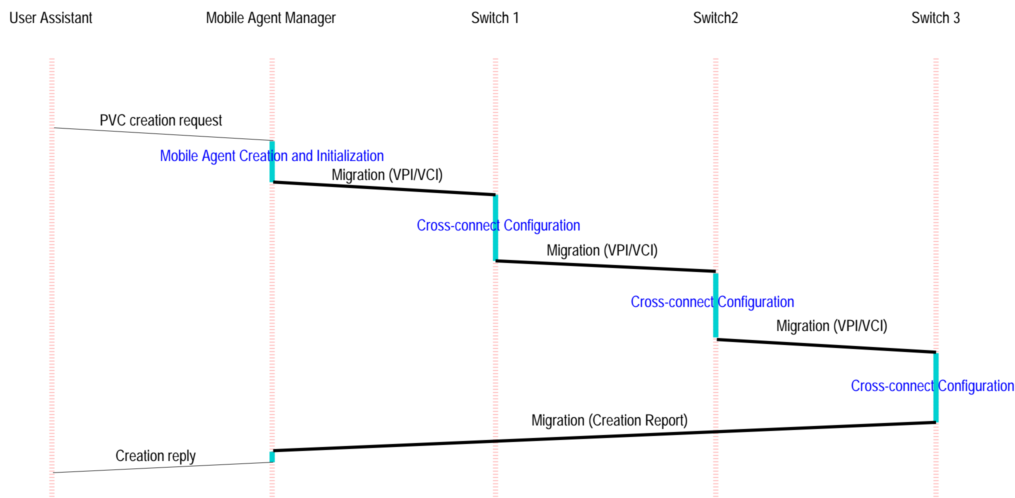


Figure 8.1: PVC creation with a Mobile Agent

returns back to its MAM, which in turn informs the requesting UA of the successful creation of the PVC. Of course, it is also possible, and even more efficient, that the MA sends a report message to the MAM and terminates on the last switch, instead of traveling back from there.

8.2.3 The Static Agent Approach

As explained earlier, we adopt a modified algorithm for PVC creation than the one used in Chapter 6, in order to closely match the algorithm of the MA approach. There are two differences between the algorithm in Chapter 6 and the one used in this section. The first difference is that there is no reservation process, and the second is that the master agent does not send direct messages to the slave agents. Instead, messages are relayed in cascade between consecutive SAs, as depicted in Figure 8.2.

We recall that using SAs for PVC provision requires running an agent on each ATM switch. The agent that receives the request to establish a PVC will be the responsible for the creation process vis-a-vis the PVC requester. The first step to do is to check for the available resources and VPI/VCI values on the switch. This allows the SA to determine which are the cross-connect parameters that can be used, both for the incoming and outgoing virtual channel links. The outgoing virtual channel VPI/VCI values can then be transmitted to the SA in the next switch on the route to establish the next cross-connect. This can be performed even before the first SA starts configuring the cross-connect on the switch.

Similarly to MA migration, a message exchanged between two consecutive SAs tra-

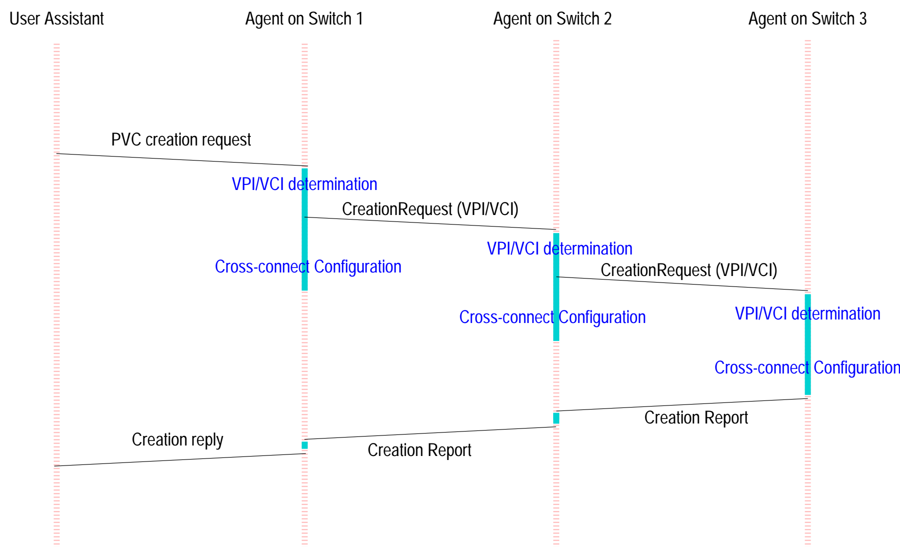


Figure 8.2: PVC creation with Static Agents

verses only a single physical link, since neighbor switches are directly connected to each other.

Each SA has to report back to the preceding SA whether the local cross-connect has been correctly created or not. An SA does not report the result until it receives the report of the SA on the next switch of the route. In this way, an SA's report implicitly includes the reports of all the SAs situated beyond on the PVC route.

In contrast to the MA approach, the SA approach does not require an agent manager that initiates the configuration operations of PVCs. Every SA is capable of accepting an end-to-end PVC request from the user.

8.3 Comparison

8.3.1 Efficiency and Performance

Criteria. Usually, the first criterion to consider to confront two approaches is performance. We provide in this section a quantitative comparison based on the implementations described in Sections 8.2.2 and 8.2.3. In order to get comparable analytical expressions, we do bot consider any possible optimization or improvement (e.g. code prefetching for MAs) to the implemented scenarios.

We use analytical expressions instead of measurements, since analytical expressions

allow to pinpoint the parameters and ratios that should be considered for comparison. In addition, there had been no experimental measurements available for the MA approach.

Comparison. The performance parameters considered in this comparison are the generated traffic and the response times. We study for each criterion the ratio of the results obtained for both MA and SA approaches, and we use the generated traffic and the response time ratios as a way to conclude which approach offers better performance than the other.

8.3.1.1 Generated Traffic

- **Traffic generated by MAs.** The overall traffic incurred by the MA approach to create a PVC includes the request of the UA to the MAM, the migration of the MA from the MAM to the first switch, then to the consecutive switches until the last one, the migration from the last switch back to the manager and the reply of the manager to the UA. If N is the number of switches to configure (which we denote by S_1, S_2, etc.) and D_X is the number of links traversed to reach X from the MAM, then the traffic T_{MA} generated is:

$$T_{MA} = (M_{req} + M_{rep})D_{UA} + M_0 D_{S_1} + M_N D_{S_N} + \sum_{i=1}^{N-1} M_i \qquad (8.1)$$

  where M_0 is the size of the MA when leaving the MA manager and M_i is the size of the MA when leaving switch S_i. We recall that the physical distance between two consecutive switches is always equal to a single link. M_{req} and M_{rep} respectively denote the traffic generated by the **req**uest message sent by the UA to the manager and the **rep**ly message sent by the manager to the UA.

  In order to simplify Equation 8.1, we assume that $M_{req} = M_{rep} = R$ and that the size of the MA changes very little during its lifetime and can be considered as a constant that denoted as τ. Therefore, Equation 8.1 becomes:

$$T_{MA} = 2RD_{UA} + (D_{S_1} + D_{S_N} + N - 1)\tau \qquad (8.2)$$

- **Traffic generated by SAs.** The overall traffic incurred by the SA approach to create a PVC includes the request of the UA to switch S_1, the two messages exchanged by each two consecutive switches, and the reply of S_1 to the UA. Since the distance between the UA and the first ATM switch, as well as the distance between two consecutive switches is reduced to a single link, the total traffic T_{SA} generated by SAs is

therefore:

$$T_{SA} = M_{req} + M_{rep} + \sum_{i=1}^{N-1} (Req_i + Rep_i) \tag{8.3}$$

where Req_i represents the traffic generated by the request message sent by SA on switch S_i to the SA on switch S_{i+1} and Rep_i represents the traffic generated by the reply message sent by the SA on switch S_{i+1} back to the SA on switch S_i.

Again, in order to simplify the expression of T_{SA}, we assume that all the exchanged messages are of the same order of magnitude and that they can be considered as equal to a constant R (same R as defined for Equation 8.2). This leads to:

$$T_{SA} = 2NR \tag{8.4}$$

To compare Equations 8.2 and 8.4, we introduce α as the ratio between the size of an MA and that of an exchanged message between SAs, i.e. $\tau = \alpha R$. The ratio between T_{MA} and T_{SA} can then be written as:

$$\frac{T_{MA}}{T_{SA}} = \frac{\alpha}{2} + \frac{2D_{UA} + \alpha(D_{S_1} + D_{S_N} - 1)}{2N} \tag{8.5}$$

It appears that $\frac{T_{MA}}{T_{SA}}$ can be written as the sum of $\frac{\alpha}{2}$ and of a constantly-positive term that decreases towards zero when N grows large (assuming that D_{UA}, D_{S_1} and D_{S_N} are constant). Two results can be concluded. The first is that that as soon as the the size of the MA exceeds twice the size of a message, it is sure that MAs generate more traffic than SAs. The second is that the ratio $\frac{T_{MA}}{T_{SA}}$ continuously decreases when N grows large and converges to a lower limit of $\frac{\alpha}{2}$. To get an idea of the order of magnitude, the average size of messages can be set pessimistically to 100 Bytes, while the size of an MA can be set optimistically to 5 KBytes, which produces an α of 50. In this case, when N grows large, we obtain that $T_{MA} = 25\, T_{SA}$. This means that in the best cases — when N is very large — the MA approach generates 25 times the amount of traffic generated by the SA approach.

Consequently, the bottleneck for the MAs to outperform SAs is clearly the size of the MA.

8.3.1.2 Response Time

- **Response time in the MA approach.** The response time RT_{MA} in the MA approach for the configuration of a PVC is the sum of the following elements:
  - R, the time for the UA to send the PVC request to the MAM;
  - I, time needed to create and initialize the MA;
  - $S_0 + T_0$, time for the MAM to serialize the MA and to send it to switch S_1;

- $W_i + P_i + C_i + S_i + T_i$, respectively, time for the MA to be deserialized and spawned, to compute the VPI/VCI values, to configure switch S_i, to be serialized again and to be transmitted to the next switch (or to the MAM for $i = N$); and
- $W_0 + K + L$, time for the MAM respectively to deserialize and spawn the MA, to extract the PVC configuration report and to send the reply back to the UA.

To simplify the expression of RT_{MA}, we assume that the serialization and deserialization times are identical, i.e. $S_i = S$ and $W_i = W$ for $i = 0..N$. Also, P_i and C_i for $i = 1..N$ can be considered the same for all the switches and equal to P and C respectively. We may in addition consider that the transmission time of the MA is the same between all the consecutive switches and therefore, $T_i = T$ for $i = 1..N - 1$. In this case, if we consider that $R = L$, we obtain that:

$$RT_{MA} = (W + S + P + C + T)N + (2R + W + S + I + T_0 + T_N - T + K) \quad (8.6)$$

- **Response time in the SA approach.** The response time RT_{SA} in the SA approach is sum of:
  - $2Q$, the time for the UA to send the request to the SA_1 (i.e. the SA on the first switch) and to receive the reply back from it later at the end of the PVC configuration;
  - P_i, time respectively for agent SA_i to compute the VPI/VCI values. We assume that $P_i = P$ for $i = 1..N$;
  - $t_i + t'_i$, time for agent SA_i to send the request to the next SA, and for SA_{i+1} to send back the reply for agent SA_i. We assume that $t_i = t'_i = t$ for $i = 1..N - 1$;
  - C, time for the last agent SA_N to locally configure switch S_N; and
  - r_i, time for agent $SA_{i,i=1..N-1}$ to prepare the reply to be sent to the preceding SA (or the UA in case that $i = 1$). We assume $r_i = r$ for $i = 1..N$.

  This leads to:

$$RT_{SA} = (P + 2t + r)N + (2Q + C - 2t) \quad (8.7)$$

Let's introduce the following parameters:

- We define β as the ratio of the total MA migration time over the transmission time of an SA message. Therefore, $S + T + W = \beta t$.
- Let's also suppose that the switch configuration time C can be expressed using the SA report generation time r. We define λ as: $C = \lambda r$, where $\lambda > 1$ since $C > r$. As a matter of fact, it is likely that the configuration of a switch requires multiple

interactions with its SNMP agent, while the time for an SA to decide of the reply back to the preceding agent requires only to check whether the configuration of the PVC on the next switch succeeded or not.

Using these two parameters, the response time ratio between MAs and SAs can be expressed as:

$$\frac{RT_{MA}}{RT_{SA}} = 1 + \frac{((\beta - 2)t + (\lambda - 1)r)N + (S + W - T + 2t - \lambda r + \Gamma)}{(P + 2T + r)N + (2Q + \lambda r - 2t)} \tag{8.8}$$

where $\Gamma = 2R - 2Q + I + K + T_0 + T_N$. Γ is a strictly positive constant.

Therefore, the only chance for the MAs to obtain a better response time than SAs is that $(\beta - 2)t + (\lambda - 1)r$ be negative, while N is sufficiently high. This implies that β should verify:

$$\beta < 2 - \frac{(\lambda - 1)r}{t} \tag{8.9}$$

Accordingly, a necessary, but certainly not sufficient condition for MAs to outperform SAs is that $\beta < 2$. However, according to measurements in [IHM99] and [SSS+99], the total migration time required for an MA is of the same order as $100ms$, while the transmission time required for a message is of the same order as $10ms$ in a local network environment. This leads to $\beta = 10$. Within a local network, it has been observed that the bottleneck for MA migration is the serialization/deserialization and spawning times and not the transmission time *per se*. This bottleneck is even more emphasized in a high-speed network such as an ATM network.

To get an idea of the ratio between MA and SA response times, we consider that all the message transmission times are equal to $t = 10ms$. A realistic value of β would be 10. Moreover, we suppose that the transmission time of the serialized MA in an ATM network is comparable to that of a simple message, i.e. $T = 10ms$. In addition, we assume that $C = P = 500ms$ and that $I = K = r = 100ms$ ($\lambda = 5$). This leads to:

$$\frac{RT_{MA}}{RT_{SA}} = 1 + \frac{0.59N - 0.11}{0.62N + 0.5} \tag{8.10}$$

For these particular values, we obtain that the MAs spend approximately 1.6 as much time as SAs when $N = 1$. The ratio increases with N. When N grows large, the response time of MAs converges to nearly twice the response time of SAs.

8.3.1.3 Discussion

It could have been intuitively easy to foresee that SAs would have a better performance than MAs. However, the fact that numerically, an MA generates 25 times the traffic gen-

erated by SAs can be important for a network provider. In the case of PVC provision, the amount of traffic generated by MAs that are launched only now and then will not affect the high-capacity links of ATM networks. However, many other management applications are sensitive to the amount of traffic they generate, and therefore, an SA implementation is by far to be preferred over an MA one.

Similarly, the response time is a decisive factor because it implies the time that the usually-impatient end-user is waiting while the connection is being established. This is a competitive factor — although not unique — that would lead an ATM network provider to opt for an SA-based solution.

An interesting question is whether it is potentially possible to improve the MA migration strategy in a way that MAs could outperform SAs (or at least have equivalent performance). Regarding the generated traffic, SAs put the barrier too high according to Equation 8.5. In order for the MA implementation to generate less traffic than the SA implementation, it is at least necessary that the average size of the MA does not exceed in the worst cases twice the average size of an exchanged message between SAs. Of course, such challenge is not obvious to overcome. Even with an MA code caching mechanism, the MA's data and status are still larger than a simple message.

Regarding response time, we emphasize that the most influencing factor is the serialization and deserialization times S and W. In our case, we assumed that the serialization and deserialization processes take much more time than the transmission. This is justified by the fact that transmission only occurs between two consecutive switches through a direct high-capacity link. Serialization and deserialization times can be improved in the future by adopting a more efficient RMI implementation, and by improving the speed of Java programs when using Java-enabled chips. Alternatively, let's extrapolate this case study to a similar configuration problem that is implemented on a wide-area network with long delays. In this case, the message transmission time t can be large enough to outweigh the $(\lambda - 1)r$ factor in Equation 8.9. Similarly, it is also possible that the transmission times T and t take larger values than the serialization/deserialization times S and W to have Equation 8.9 verified. Consequently, the ratio $\frac{RT_{MA}}{RT_{SA}}$ can take smaller values than 1 when N is sufficiently high, hence a better response time for MAs.

We argue however that even in this extreme situation, we could provide an SA algorithm that incurs a better response time than MAs. In fact, if instead of the cascading reports of PVC creation results, only the final SA could send back a report to the first SA. This leads to exactly an equivalent algorithm as used with MAs. In this case, it can be easily proven that SAs provide a better response time than MAs even in a long-distance high-capacity network, although SAs will lose their advantage in reliability which is discussed in the next section.

One of the most important reasons of adopting MAs and SAs in this case study is decentralization. Both approaches succeed to overcome major problems in centralized NM, namely, micro-management, bandwidth bottleneck around the central manager, and the overall performance. However, as discussed previously, an MA-based implementation hardly can outperform an SA-based one, as demonstrated by this analytical comparison.

8.3.2 Robustness and Graceful Degradation

Criteria. In this section, two important criteria for analyzing distributed applications, and in particular, Network Management Systems are discussed. We consider two questions:

- Which approach shows to be more robust in the presence of invalid inputs or stressful environmental conditions?
- Following a partial crash of the agent system, which approach can still operate without drastic degradation?

Comparison. In general, providing a robust application requires to include additional verification tests and exception handling in the application code to handle input anomalies and critical situations in the environment. These tests lead to a sensible increase in the size of the agent code. At first sight, MAs seem to be sensitive to this factor since it directly affects their performance and increases the traffic incurred by MA migration. An SA is more tolerant to an increase in its code size since it does not affect the network traffic.

However, in the case of SAs, robustness may require verification tests to be performed on information that is distributed amongst the agents. This is likely to happen since an SA is supposed to have only a local and partial view of the overall data and system parameters. Therefore, SAs may need to exchange messages when they have to check whether the input data along with the environment parameters are correct or not. Distributed checking may therefore introduce significant communication overhead as well as an additional design time. On the contrary, if the MA is designed to be *self-contained*, i.e. in a way to contain all the information related to the task it has to perform, robustness can be achieved in a simple way by including internal verification tests inside the MA own code.

This is applicable in the case of preventive tests to insure software robustness. To study the degree of fault tolerance of each approach, we consider the behavior of the MA and SA systems in the cases of different types of crashes.

In the MA system, crashes may occur either on the MCD (see Section 8.2.2), or on the MA itself. In the former case, no more PVCs can be routed via that switch. However, other PVCs that do not use the corresponding switch can be established without any trouble. Unless an enhanced mechanism for the detection of Mobile Code Server failure is used, the MA will not be aware of a server failure until it intends to migrate there.

In the latter case, i.e. when an MA crashes while trying to create a PVC, configuration incoherence may occur in the ATM network. For example, the PVC creation could be stopped before the MA reaches the final destination. In this case, there will be Virtual Paths and Channels allocated on a switch, but not allocated on the next one. This yields to uselessly allocated resources on some switches and alternatively may even cause potential errors during the creation of future PVCs.

An analogous reasoning can be formulated for the case of SAs. It is possible that an SA on an ATM switch crashes. This makes that switch unmanageable, which is equivalent to the crash of an MCD in the MA approach. Furthermore, if the SA crashes while it is under the establishment of a PVC, its preceding agent which is located right before on the PVC route will detect it (either because of a connection loss, or due to a timeout). This crash information can be returned back to the first SA on the PVC route. An alternative route can then be computed and another PVC can be created instead. In addition, the SAs inherently implement a commit/rollback mechanism which guarantees the consistency of the configuration operations and ensures the safe cleaning in the case of failures.

The clean-up process following a failure in the creation of a PVC is more difficult to implement when the MA approach is adopted. The difficulty with MAs is that *all* the information related to the PVC creation process is lost when the MA crashes. Therefore, it is necessary to retrieve the PVC parameters used by the MA before the clean-up can be initiated. Retrieving these parameters is not a simple task and may require to search into log files or MAs' traces.

In summary, MAs can easily include preventive mechanisms to enhance robustness by implementing preventive control policies when designed in such a way to contain all the necessary information for the achievement of their tasks (self-containment). Although MAs are more sensitive to an increase in their size due to the verification tests, they can counterbalance the communication overheads and design difficulties that arise when using SAs. On the other hand, SAs offer a greater ability of recovery since each SA holds part of the information regarding the PVC configuration information which can be used to clean-up a PVC when its creation fails. Finally, both approaches have the same degree of graceful degradation in the case of a partial crash of the system.

8.3.3 Ease of Development

Criterion. In this section, we discuss the differences between the two approaches from a development methodology point of view. We compare the way the case study was designed in each approach and draw conclusions on which approach can be developed more easily.

Comparison. The SA approach is based on distributed computational entities cooperating together to achieve the required functionality. This is significantly different from the MA approach in which an MA is considered as a unique computational and decisional entity despite its ability to migrate. In fact, while the design of the case study using SAs dealt with issues related to distributed computing, its MA counterpart was analogous to monolithic computing. SAs inherently include the difficulties of distributed programming [Lyn96]. Therefore, the design of the MAs was much simpler than the design of the SAs. Some of the problems that can be encountered using SAs and that may be transparent to MAs are enumerated below.

- **Partial views:** One of the major issues is that each SA has only a limited view of the end-to-end PVC parameters. Therefore, a unique SA is *never* able to take a decision related to that PVC without being in consensus with the other agents. We can illustrate this by considering the SA on the Switch 2 in Figure 8.2. This SA cannot establish whether the PVC is fully created or not, simply because it has no information to establish whether the SA on Switch 1 succeeded in creating its PVC fragment or not.

  In contrast, an MA is, at each time, capable of stating whether the PVC has been fully created or not. The reason is that the MA holds all the information related to the PVC to be created. In addition, the MA does not need the help of any other agent to accomplish its task.

- **Time uncertainty:** Another related idea that was originally stated in [Hub97] is that MAs can be used to overcome time uncertainty caused by distributed processing. This statement holds in our experiment. For instance, an SA issuing a request for a PVC creation (or deletion) to another SA cannot be guaranteed a prompt response. This causes potential uncertainty inside both SAs: the first agent does not know when it will receive the reply, while the second agent does not know when its reply will reach the first agent. In fact, if we wish to implement a more robust solution based on SAs, a mechanism such as one based on timeouts should be used to help an SA deciding whether it still has to wait for a reply or not. With MAs, this problem

does not exist since it is the same entity that performs all the necessary operations and there is no need for inter-agent communication.

- **Inter-agent coordination:** An obvious issue for SAs is task allocation. It is essential to affect tasks to SAs in a way that minimizes the dependencies between the agents and avoids concurrent access to shared resources. Moreover, tasks have to be coordinated and synchronized to ensure that they are carried out with the right sequence order. In the considered case study, task synchronization was not an important issue because the PVC creation is mainly sequential. In other cases, task synchronization may be costly both in design time and in inter-agent communication overhead. Again, the MA approach allows to transparently avoid such problems. The MA performs all the tasks relative to the PVC creation by itself. Instead of delegating the execution of a certain task to an agent running on a remote switch, the MA travels to that switch and performs the required task.

- **Intra-agent design:** An SA may receive simultaneous PVC requests. Therefore, it is necessary to protect the sensitive parts of the agent code from concurrent access, for example to ensure that the affected VPI/VCI values are different for each new PVC. Consequently, the SA has to be able to ensure the coherence and the integrity of the data it has. In addition, it is necessary to determine a way by which the Static Agents can refer to a certain PVC in a non-ambiguous way when exchanging its related configuration parameters. This lead to the introduction of a PVC identifier when adopting the SA design. The PVC identifier allows to uniquely refer to its PVC in any Static Agent that has taken part in the configuration of that PVC.

  MAs provide an abstraction to such problems. Each MA deals only with a unique PVC and therefore it does not have to handle concurrent demands. Moreover, with MAs there is no need to attribute a unique reference to each created PVC.

We may analyze more deeply the difference between the two approaches in the case of PVC creation. For every PVC creation a certain number of tasks have to be performed. The principle in the SA approach is to handle the problem by attaching a fixed task to each agent. Therefore, each SA always performs the same task but for different PVCs. In contrast, the MA approach attributes all the tasks to the same Mobile Agent, and assigns a PVC to each MA instance. This is illustrated in Figure 8.3, which shows that MAs provide a horizontal approach, while Static Agents provide a vertical approach. The simplicity of the MA design comes from the fact that it is much simpler to handle a serie of tasks related to the same activity than to coordinate parallel tasks related to different activities. Therefore, MAs provided a better abstraction for the problem of PVC provision.

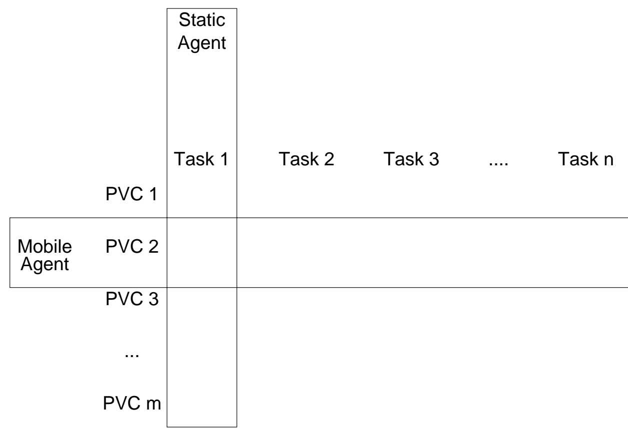


Figure 8.3: Mobile Agent approach compared to Static Agent approach

From the above reasoning, it appears that MAs would lose their benefit of simplicity and ease of development if they were built in a vertical task-based distributed way instead of a horizontal way. In fact, this approach was studied in [BPW99] that describes another approach to the problem of PVC provision with cooperative MAs. Not only this approach is not simpler than the MA approach considered here, but it also showed to be inefficient according to [BPW99].

We conclude that MAs offer an elegant concept that can be used in distributed applications. They have the advantage of compacting distributed processing within the same entity, that virtually can be considered as centralized from a design point of view. These concepts perfectly apply to the PVC case study.

8.3.4 Ease of Deployment

Criterion. After comparing MA and SA approaches from a design and development point of view, we consider in this section the cost incurred by the deployment of each type of agents.

Comparison. The first phase in the deployment of MAs consists in running an MCD on each switch to enable MA migration. In the SA approach, this phase is equivalent to running an agent brain on each switch.

The second phase in the MA approach consists in uploading the necessary Virtual Managed Component (VMC) software. This is equivalent to skill plugging in the SA approach. However, for SAs, it is necessary to plug-in all the skills including those ensuring

the PVC management functions. In contrast, with MAs, the PVC management functions are included into the MA and no initial deployment is required.

Consequently, once the agent system is started, the major difference between MAs and SAs is that the functionality coded into an MA can be immediately invoked as soon as it is developed; whereas the functionality coded as skills for SAs must first be deployed (i.e. plugged into all SAs) before it can be invoked. Deployment is therefore facilitated in the case of an MA system.

8.3.5 Support for Heterogeneous Environments

Criterion. Networks are in general composed of a heterogeneous set of software and hardware elements. Moreover, they are continuously evolving over time due to software and hardware upgrades, capacity dimensioning and changes in user profiles.

This section discusses how the MA and the SA approaches deal with the problem of the heterogeneity of ATM switches and evaluates the cost of supporting a new type of switches with a different management interface.

Comparison. In the MA approach, the MA interacts with VMCs to manage switch resources. The VMC offers an abstract view of the switch and acts as a bridge between the MA and the managed resources. Therefore, there is no need to change the MA code to support a new type of ATM switches. What is needed is to develop the VMC and to deploy it to the MCD of the new switch. The process of dynamic download of VMC code is supported by the MA platform (Section 8.2.2).

In the SA approach, a new switch skill (see Section 6.3) containing the necessary capabilities to manage the new switch must be developed and integrated inside the SA. The switch skill offers a common abstract management view of the new ATM switch. Similarly to the VMC in the case of the MA, the new switch skill can be dynamically integrated into the SA.

Hence, both approaches offer the same degree of ease of portability and support for new types of ATM devices.

There is however a noticeable difference between the two approaches. In the MA approach, the VMC is *not* part of the agent, but part of its supporting runtime environment, whereas in the SA approach, the switch skill *is* of the agent. Of course, the switch skill module could be separate from the SA and implemented as a separated entity on the ATM switch. However, there was neither an advantage, nor a necessity in introducing a separate VMC in the case of SAs. In contrast, if there were not a homogeneous manage-

ment interface provided by VMCs, MAs would have to include the knowledge to handle switch heterogeneity, hence a sensible increase in the MA size.

MAs are therefore not suitable to directly using low-level management protocols and MIBs. They require an additional component that offers a higher-level and a homogeneous access interface regardless of device heterogeneity.

8.3.6 Flexibility

Criterion. Flexibility is the ability to adapt to new, different or changing management requirements. From a practical perspective, it means for the network administrator not to be tightened to a certain management protocol or technology when trying to achieve the required management operations. For example, an SNMP agent is not flexible since it is usually impossible to extend its functionality or enrich its instrumentation capability.

Comparison. At first sight, both MA and SA approaches seem to be equivalent in flexibility. Both approaches allow to easily extend the functionality of the agent system. The MA approach allows developing new MAs with new or improved management functions. It also allows improving the instrumentation of the switches by uploading new versions of VMCs into the switches. Similarly, the SA approach allows to plug new skills with new or improved management functions into the already running SAs and then to dynamically invoke these new functions.

However, there is an important difference that can be shown through the following scenario. We assume that we want to replace the algorithm of PVC creation in the ATM network — e.g. in order to improve the bandwidth allocation algorithm. We suppose that the new version of the algorithm is already developed for both approaches, and we need to replace the old version.

In the SA approach, a new skill version is provided and should be plugged into the SAs. If we suppose that some PVCs are under establishment, then it is not directly possible to guarantee that the same skill version is used. The same PVC creation process could start using the old version and meanwhile the new version is uploaded inside the SAs and will be used for the remainder of the PVC route. This may lead to incoherent configurations if the two skill versions are not compatible. Consequently, there should be a mechanism that ensures version coherence during the establishment of end-to-end PVCs.

The MA approach naturally avoids such problems. In fact, the code of an MA responsible for the creation of a certain PVC does not change during the whole process. During its life cycle, the MA has either the old version or the new version and it is sure that the same version of the algorithm is used until the PVC creation process is terminated.

This example shows how MAs can help preserving coherence while updating the application and therefore, offer enhanced flexibility to the network administrator. The SA approach would require an offline time during which the new skill is upgraded into all the SAs before the PVC provision application could be made operational back again.

8.4 Conclusion

The starting point of the work described in this chapter is that both Mobile Agents and cooperative Static Agents have interesting potentials in dealing with NM problems. Expertise to assess both approaches is therefore urgently needed, and a case study based comparison between MAs and SAs is of a great help in this direction.

Although they prove to be less efficient than SAs, MAs offer other interesting advantages. From a design point of view, they offer a great transparency to distribution-related issues such as information distribution, task synchronization and coordination, time uncertainty and concurrence. An MA is indeed a compact piece of software that *is designed monolithically and acts distributedly*. In addition, the case study pinpoints that the reason behind this advantage is mainly the property of *self-containment*, i.e. the MA is supplied with all the necessary information and know-how to perform its management duty without any help from another agent. If we observe the qualitative criteria used for our comparison, we find out that self-containment is also behind the enhanced flexibility of MAs over SAs, and the better support for robustness. In addition, the advantage of MAs over SAs in the case of ease of deployment is due to the fact that the MA does not require initial code deployment, since it directly encapsulates the know-how to achieve its management task.

In turn, this shows that the main bottleneck of MAs is inter-agent communication, which is exactly the strong point of SAs. In fact, not every NM application can be designed through independent tasks that can be encapsulated in self-contained MAs. In many cases, the distributed nature of NM problems requires to have interacting entities that exchange information flow and task flow messages. SAs are specially fitted to such situations. In addition to the direct support for high-level communication, SAs have a clear advantage over MAs in terms of efficiency and performance. In that, they can execute tasks in parallel thus improving response time, and can exchange concise messages thus reducing the generated management traffic. Another advantage is that they are much less sensitive than MAs to an increase in their size, and can encapsulate arbitrarily complex functionality.

Concerning the other qualitative criteria, MAs and SAs have comparable satisfactory degrees. For MAs, this is due to mobility and self-containment. For SAs, this is due to the

dynamic skill loading capability provided by our skill-based agent architecture. As mentioned in Chapter 4, dynamic skill loading is not an inherent property of software agents as perceived in the agent community, but rather is a design choice that we adopted in answer to NM requirements. The work described in this chapter shows that the skill-based approach provides a balanced compromise between fully-mobile agents and inflexible distributed SAs. The flexibility and ease of deployment exhibited by MAs are replaced by dynamic skill loading, which provides a sufficient degree of flexibility, while still allowing to build highly efficient management applications. Of course, our architecture excels by providing a general purpose support for building a large choice of distributed NM applications, instead of being particularly suitable for a specific range of NM applications.

Chapter 9

Conclusion

Network Management is facing many challenges due to the recent evolutions in networks and networked applications and services. There are many new trends in NM that try to tackle these challenges. In general, these trends make use of well-established techniques from other computing fields (e.g. CORBA, XML, Java, and web technology), which they adapt to NM problems.

Software agents are another trend in NM that differs from the other trends at least in two aspects. The first aspect is that agent technology contrarily to the other approaches, is not yet well defined. Debates are still ongoing regarding the definition of an agent, and the different possible types of agent architectures and systems. The second aspect is that it is still unclear from an NM standpoint what challenges can be addressed by software agents, and whether it is really opportune to deploy them in management systems.

The purpose of my thesis was to help drawing the way agents can be successfully applied in NM software.

9.1 Summary of Major Contributions

Our first contribution is to show that the term “agent” hides a multitude of interpretations, applications, techniques, and algorithms that are proposed by the agent research community. The perception of the agent concept differs according to whether it is adopted as a modeling construct, a metaphor, a decision-capable entity, etc. There are also a large set of techniques to build individual agents, to make them communicate, and to organize their interactions within agent communities. These are among the reasons that there is still no concensus on a unique agent approach.

The existing agent-based approaches to NM either apply a specific type of agents, or

target a specific management application (or both). Our second contribution is to show that NM can benefit from nearly all agent techniques, and that an agent approach that is restricted to a single type of agents cannot be advantageously used in all the NM areas. Therefore, a successful agent approach to NM should necessarily contemplate a generic agent architecture.

One of our major contributions is a generic skill-based agent architecture that provides a high degree of flexibility and support for dynamism, essential properties in future generation NM products. This architecture does not commit to a specific organization of the agents, yet agents can dynamically acquire the necessary capabilities to insure management functions, and to adapt their individual and social behaviors.

Reliability is a crucial issue if autonomous agents are to be used to achieve high-level critical NM tasks in an independent way from the other management components. Critical network problems can occur if agents do not make management decisions reliably. We studied this problem in the case of a domain-based distribution of management activities, where domains can be dynamically reaffected to agents in the case of a change in the network configuration or the number of deployed agents. Our contribution in this first case study is therefore to implement a reliable system, in which unreliable agents are automatically detected based on a variation of SLD (System Level Diagnosis). Further, the taks of an unreliable agent are automatically undertaken by the other reliable agents. The case study showed how agents allowed to model different types of interaction patterns that exactly reflect the organizational constraints in the NM application. Agent techniques such as belief suppression were easily adopted in this case study, thus showing the genericity of our skill-based agent architecture.

PVC configuration in ATM networks is a tedious task, and we proposed an agent-based system to automate this task in the second case study. With a centralized approach, PVC configuration requires a large number of SNMP queries to be sent to the ATM switches over the network. An agent-based approach performs these queries locally (or next) to the ATM switch. Only concise messages are exchanged between the agents, thus considerably reducing traffic overhead. In addition, a suitable vertical skill design allows to bring a practical solution to switch heterogeneity.

In order to have a complete assessment of the skill-based agent architecture, we proposed to compare it with other most-promoted agent approaches. Another contribution of this thesis is an abstract BDI-oriented mental model for NM applications, adapted from other agent BDI approaches. We used this abstract mental model to design and implement the SLD case study according to the BDI-oriented design process that we proposed. This BDI-oriented implementation was compard to the skill-based approach. Similarly, an existing MA approach to the PVC configuration case study was compared to

a suitably adapted version of the skill-based approach. These comparisons showed that the skill-based architecture provided a good compromise regarding flexibility, genericity and dynamism compared to the BDI and mobile approaches. We showed that a BDI approach is powerful from a design point of view and have the potential to encapsulate sophisticated agent features, but lacks the flexibility required for NM applications. We also showed that the MA approach is an elegant way to easily apprehend distributed applications. However, MAs are less efficient than stationary agents. They are suitable only for particular kinds of distributed applications, and they can lose their advantages with cooperative scenarios.

9.2 Experience-gained

Despite the large spectrum of agent techniques, which are surveyed in Chapter 2, agents did not yet crystallize in an appropriate technology that could be truly called *software agent technology*. The approach adopted in this thesis is therefore to dig its own way to use software agents in NM. An agent architecture is proposed out of the available agent techniques to fit the requirements of next generation management systems. This architecture left out buzzwords used by agent proponents and proposed a concrete view of software agents with suitable abstraction constructs supported by an implemented framework. What are considered as buzzwords are those commonly-repeated phrases such as “an autonomous task-achieving” and “capable of intelligent decisions”, while till now there is no clear way to endow a software with intelligence and how to make this software autonomous. More important when applying agents is which techniques should be used to provide a modular agent architecture and which other techniques should be used to design the inter-agent organization and interaction in a way to achieve the requirements of efficiency, performance, and the easy modeling of real-life interactions. Thus, the applications developed based on our agent architecture focussed on using the right agent techniques and models to handle the case studies. The genericity of our agent approach allowed to easily incorporate these techniques and models.

Agent properties, such as autonomy, responsiveness, proactiveness, social ability, etc. should not to be considered as aquirements resulting simply from adopting agent-oriented techniques. The reality is that these properties can be interpreted differently by different people, hence the attempt of Chapter 2 to provide as accurate definitions as possible to these properties. The reality is also that these properties should be considered only as potentially higher standards required for agent software, and, nevertheless, a well-defined technology to reach such requirements is doubtlessly still lacking. The direct implication is that a property-based view of software agents (see Section 2.1.3) is

not the immediately viable approach to apply software agents to NM. There is a need for more research to understand how properties such as autonomy, responsiveness and proactiveness, can be suitably achieved in NM agent-based software. The discussion in Chapter 7 provided hints about how to incorporate some of these properties in a BDI-like agent architecture. These hints could be used as a basis for further work in this direction.

We adopted in this thesis an approach that uses agents mainly as a new software modeling paradigm. Agents as a modeling paradigm is a recent trend in the field of software agents and references about it began to appear recently, e.g. [Jen00]. In order to use software agents as a modeling paradigm, it is important that precise agent-oriented abstract modeling constructs be defined. An abstraction is supposed to highlight some aspects of the modeled system, while hiding irrelevant details. (For example, the object abstraction highlights the interface of an entity while hiding its implementation details.) The agent constructs that we defined in Chapter 4 were beliefs, capabilities, skills, roles, and communication acts. These abstracts showed to be appropriate to build agent applications following a development process. We defined this development process as a suite of multiple refinement stages. The defined constructs, together with these refinement stages, are however specific to our skill-based agent architecture. Other agent approaches define different constructs and require different agent-oriented development processes. This appeared clearly when considering the BDI approach in Chapter 7 and the MA approach in Chapter 8. In these two chapters, we showed how the design of the same case studies using different agent approaches lead to completely different designs, although the obtained systems are equivalent from a functional point of view. This is to say that viewing the agent as a modeling paradigm requires a strong commitment to a particular type of agents with well defined constructs. Different types of agents are suitable for different kinds of problems. Our skill-based agent architecture showed a good compromise to tackle different kinds of NM applications.

The view of agents as metaphors is helpful to derive a high-level solution to complex problems. The SLD case study is largely inspired by such metaphoric view. This can be viewed in the reliability test where an agent compares the beliefs of another agent with its own beliefs. The metaphoric view also appears in the way agent interations were designed, with a peer-to-peer relationship between domain agents, and an authoritative relationship between the manager and the domain agents. The scenario of the PVC case study is also completely based on an agent metaphoric view. In Chapter 6, agents as static entities allowed to derive the solution of a single agent responsible for the PVC creation cooperating with the other agents, which act under its authority. Alternatively, in Chapter 8, MAs allowed to derive the solution of a single MA that creates the whole PVC by itself by migrating from a switch to another. This scenario imitates a human manager

connecting sequentially to a switch after the other. Therefore, the agent metaphoric view is interesting in the NM context. Its unique danger is that a developer is tempted to forget that in many cases, simple solutions could be found without the agent metaphor. These solutions can even be more efficient and simpler to implement. Agent-metaphoric solutions tend to target high-levels of abstraction, thus neglecting efficiency and scalability issues. This pitfall is often observed with MA approaches.

The promise of software agents in NM was to enhance the development process of management software, and to improve the functionality of management systems with increased automation, proactiveness and self-regulation. This thesis showed the potential of agents to provide flexiblity, dynamism, adaptability, support for multiple distributed organizations, and support for scalability to NM software.

Research on how software agents can improve automation and proactiveness in management systems was not covered in this thesis. As mentioned earlier, till now, there is no clear technique that allows to incorporate such features in software agents in a uniform way. In general, AI techniques such as planning, problem solving and learning mechanisms should be used. The disappointment during the previous decades with the concrete results of research in AI will hinder the use of such AI techniques in the practical deployment of software agents. This situation is likely to remain until good arguments appear that show how AI could succeed with software agent applications, while it had failed with some other types of applications.

9.3 Outlook

Current standardization efforts in the agent community are lead by FIPA and the OMG. The priority of FIPA standards is to provide an agent platform that allows for heterogeneous agents to communicate based on common agent services and the standard communication language ACL (Agent Communication Language). Recently, FIPA efforts are being empowered with ongoing efforts to provide reusable agent interaction patterns described using scenario diagrams. Interaction patterns will be very useful to easily and flexibly incorporate coordination mechanisms within agent-based systems. A valuable effort for the application of software agents in NM would be to provide a library of such interaction patterns that can be directly used in complex management applications. These interaction patterns could be classified according to the type of management function to be implemented.

Furthermore, there are emerging efforts to extend UML (Unified Modeling Language, http://www.rational.com/uml) for agents [OPB99, OPB00, (see http://www.jamesodell.com)]. These standard UML extensions will further encourage

agent developers to provide general reusable patterns to describe both agent internal processing, and inter-agent communications. References about this subject are still scarce, but are expected to emerge in the near future.

A pattern-based approach to apply agents in NM has a great potential to encourage agent-based management systems. Interaction patterns are certainly of a great use in this context. Even more important are systematic mechanisms that allow agent software to be highly autonomous, responsive, proactive and self-adaptable. Therefore, we believe that agent patterns that describe internal agent architectures as well as behavioural patterns such as autonomy and proactiveness patterns targeted to NM would be the killer argument for the deployment of a real software agent technology.

Bibliography

[AAC+99] Matthew Addis, Paul Allen, YewBie Cheng, Mike Hall, Mark Stairmand, Wendy Hall, and David DeRoure. Spending less time in internet traffic jams. In PAAM99 [PAA99], pages 193–209.

[ACL99] Agent communication language. http://www.fipa.org/spec/fipa9712.pdf, 1999.

[AG98] Sahin Albayrak and Francisco J. Garijo, editors. *Proceedings of the Second International Workshop on Intelligent Agents for Telecommunications Applications, IATA'98*, number 1699 in Lecture Notes in Artificial Intelligence, Berlin, June 1998. Springer.

[Age98] Agent Oriented Software Pty. Ltd., Carlton, Victoria, Australia. *JACK Intelligent Agents User Guide*, 1998. http://www.agent-software.com.au.

[Alb99] Sahin Albayrak, editor. *Intelligent Agents for Telecommunication Applications*, number 1699 in Lecture Notes in Artificial Intelligence, Stockholm, Sweden, August 1999. Springer.

[AS95] S. Appleby and S. Steward. Software agents for control. In P. Cochrane and D. Heatley, editors, *Modelling Future Telecommunication Systems*. Chapman & Hall, 1995.

[AT94] M. Ahmed and K. Tesink. Definitions of managed objects for ATM management version 8.0 using SMIv2. RFC1695, August 1994. ftp://ftp.nus.sg/pub/docs/rfc/rfc1695.txt.

[BBB+00] Andrzeij Bieszczad, Patrik Biswas, Walter Buga, Manu Malek, and Hai Tan. Addressing interoperability in network management with intelligent agents. In *Proceedings of Network Operations and Management Symposium (NOMS'2000)*, Honolulu, Hawaii, 2000. IEEE/IFIP.

[BCS99] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. An open secure mobile agent framework for systems management. *Journal of Network and Systems Management*, 7(3):323–339, September 1999.

[BD97] D. Benech and T. Desprats. A kqml-corba based architecture for intelligent agents communication in cooperative service and network management. In *Proceedings of the International Conference on Management of Multimedia Networks and Services MMNS'97*, Montreal, Canada, July 1997.

[Ben99] Dominique Benech. *Interaction Frameworks for Distributed and Cooperative Paradigms of Intelligent Systems and Networks Management*. PhD thesis, Université Paul Sabatier de Toulouse III, November 1999. http://www.irit.fr/~Dominique.Benech/.

[Ber96] Guy Berthet. *Extension and Application of System-level Diagnosis Theory for Distributed Fault Management in Communication Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Lausanne, CH, 1996.

[BF96] Mihai Barbuceanu and Mark S. Fox. The design of a coordination language for multi-agent systems. In *Intelligent Agents III. Agent Theories, Architectures, and Languages*, pages 341–355. Springer, 1996.

[BG93] M. Busuioc and D. Griffiths. Co-operating intelligent agents for service management in communications networks. In *Proceedings of CKBS-SIG*, Keele, UK, 1993.

[BGP97] Mario Baldi, Silvano Gai, and Gian Piertro Picoo. Exploiting code mobility in decentralized and flexible network management. In *Proceedings of the First International Workshop on Mobile Agents*, Berlin, Germany, April 1997. Available at http://www.polito.it/~picco/papers/ma97.ps.gz.

[BHG+98] Eric Bonabeau, Florian Henaux, Sylvain Guérin, Dominique Snyers, Psacale Kuntz, and Guy Theraulaz. Routing in telecommunications networks with ant-like agents. In Albayrak and Garijo [AG98], pages 60–71.

[Bie97] Andrzej Bieszczad. Application-oriented taxonomy of mobile code. Technical Report SCE-97-13, Systems and Computer Engineering, Carleton University, June 1997.

[Bla98] Uyless Black. *Signaling in broadband networks*, volume ATM: Volume II. Prentice-Hall, 1998.

[BM99] C. Bäumer and T. Magedanz. Grasshopper - a mobile agent platform for active telecommunication networks. In Albayrak [Alb99], pages 19–32.

[BP97] Mario Baldi and Gian Pietro Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In R. Kemmerer and K. Futatsugi, editors, *20th International Conference on Software Engineering (ICSE'97)*, Kyoto (Japan), 1997.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. World Wide Web Consortium (W3C), February 1998. http://www.w3.org/TR/REC-xml.

[BPW99] John Boyer, Bernard Pagurek, and Tony White. Methodologies for PVC configuration in heterogeneous ATM environments using intelligent mobile agents. Submitted to MATA'99, 1999. ftp://ftp.sce.carleton.ca/pub/netmanage/jb-mata99.ps.gz.

[Bra87] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[BRHL99] Paolo Busetta, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents - components for intelligent agents in Java. AgentLink News Letter, January 1999. White paper, http://www.agent-software.com.au.

[Bro91a] Rodney A. Brooks. Intelligence without reason. A.I. Memo 1293, Massachusetts Institute of Technology, April 1991.

[Bro91b] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence Journal*, (47):139–159, 1991.

[Bus96] M. Busuioc. Distributed intelligent agents – a solution for the management of complex services. In *Proceedings of IATA'96*, Budapest, Hungary, August 1996.

[BWP98] Andrzej Bieszczad, Tony White, and Bernard Pagurek. Mobile agents for network management. *IEEE Communications Surveys*, September 1998.

[CBD99] Sang Choy, Markus Breugst, and Mohit Datta. Management issues of a mobile agent-based service environment. *Journal of Network and Systems Management*, 7(3), September 1999.

[CCOL98] Morsy Cheikhrouhou, Pierre Conti, Raúl Texeira Oliveria, and Jacques Labetoulle. Intelligent agents in network management, a state

of the art. *Networking and Information Systems*, 1(1):9–38, 1998. http://www.eurecom.fr/~cheikhro/docs/StateOfTheArt.ps.gz.

[CD97] B. Chaib-Draa. Connection between micro and macro aspects of using agent modeling. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[CF99a] Monique Calisti and Boi Faltings. Distributed constrained agents for allocating service demands. In *Proceedings of the CP-AI-OR'99 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, February 1999.

[CF99b] Monique Calisti and Boi Faltings. A multi-agent paradigm for the inter-domain demand allocation process. In *Proceedings of DSOM'99, Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, 1999.

[CFF99] Monique Calisti, Christian Frei, and Boi Faltings. A distributed approach for qos-based multi-domain routing. In *Proceedings of AiDIN'99, AAAI-Workshop on Artificial Intelligence for Distributed Information Networking*, 1999.

[Cha97] Deepika Chauhan. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation.* PhD thesis, ECECS Department, University of Cincinnati, 1997. http://www.ececs.uc.edu/~abaker/JAFMAS/index.html.

[Che99] Morsy M. Cheikhrouhou. BDI-oriented agents for network management. In *Proceedings of Globecom'99*, Rio de Janeiro, Brazil, December 1999. IEEE.

[CLZ99] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Network management based on mobile agents using programmable tuple spaces. In PAAM99 [PAA99].

[CST98] A. Corradi, C. Stefanelli, and F. Tarantino. How to employ mobile agents in systems management. In *Proceedings of the third International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology*, pages 17–26, London, UK, March 1998.

[CWF99] Monique Calisti, Steven Willmott, and Boi Faltings. Supporting interworking among network providers using a multi-agent architecture. In *Proceedings*

of NOC'99, European Conference on Networks and Optical Communications, 1999.

[DFJN97] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(3), 1997. available at http://www.elec.qmw.ac.uk/dai/pubs/fomas.html.

[dRW97] Marco Antonio da Rocha and Carlos Becker Westphall. Proactive management of computer networks using artificial intelligence agents and techniques. In *Integrated Network Management V: integrated management in a virtual world*, pages 610–621, San Diego, California, USA, May 1997. IFIP/IEEE, Chapman & Hall.

[EDB99] M. El-Darieby and A. Bieszczad. Intelligent mobile agents: Towards network fault management automation. In Sloman et al. [16].

[EDQD96] Babak Esfandiari, Gilles Deflandre, Joël Quinqueton, and Christophe Dony. Agent-oriented techniques for network supervision. *Annals of Telecommunications*, 51(9-10):521–529, 1996.

[Eur98] Agent based computing, a booklet for executives. European Insitute for Reserach and Strategic Studies in Telecommunications, September 1998.

[FIP97] Application design test: Network management and provisioning. http://drogo.cselt.it/fipa/spec, June 1997.

[FMNS97] Peter Fröhlich, Iara Móra, Wolfgang Nejdl, and Michael Schroeder. Diagnostic agents for distributed systems. In *Proceedings of ModelAge97*, Sienna, Italy, Junary 1997. Available at http://www.kbs.uni-hannover.de/paper/96/ma1.ps.

[FOR97] ForeRunner ATM switch configuration manual. http://www.fore.com/products/manuals.htm, March 1997.

[FOR98] SPANS: Simple protocol for ATM network signaling. http://www.mi.infn.it/INFN/atm/articles/spansfore.ps, 1998.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[FV94] Eugene Fink and Manuela Veloso. Prodigy planning algorithm. Techical report CMU-CS-93-123, Carnegie Mellon, School of Computer Science, 1994.

[FW91] T. Finin and G. Wiederhold. An overview of KQML: A knowledge query and manipulation language, 1991. Available through the Standford University Computer Science Dept.

[GA97] Garry Grimes and Brian P. Alley. Intelligent agents for network fault diagnosis and testing. In *Integrated Network Management V: Integrated Management in a virtual World*, pages 232–244, San Diego, California, USA, May 1997. IFIP, Chapman & Hall.

[GF92] Michael R. Genesereth and Richard E. Fikes. *Knwoledge Interchange Format, Version 3.0.* Logic Group, Computer Science Departement, Stanford University, January 1992. http://logic.stanford.edu/sharing/papers/kif.ps.

[GGGO99] Damianos Gavals, Dominic Greenwood, Mohammed Ghanbari, and Mike O'Mahony. Using mobile agents for distributed network performance management. In Albayrak [Alb99].

[Gha94] Malik Ghallab. Past and future chronicles for supervision and planning. In Jean Paul Haton, editor, *Proceedings of the 14th Int. Avignon Conference*, pages 23–34, Paris, June 1994. EC2 and AFIA.

[GHN+97] Shaw Green, Leon Hurst, Brenda Nangle, Pádraig Cunningham, Fergal Somers, and Richard Evans. Software agents: A review. Technical report, Trinity College Dublin and Broadcom Éireann Research Ltd, May 1997. Available at http://www.cs.tcd.ie/research_groups/aig/iag/pubreview.ps.gz.

[GI90] M. P. Georgeff and F. F. Ingrand. Real-time reasoning: The monitoring and control of spacecraft systems. In *Sixth IEEE conference on Artificial Intelligence Applications*, Santa Barbara, CA, USA, March 1990.

[GJ97] M. A. Gibney and N. R. Jennings. Market based multi-agent systems for ATM network management. In *Proc. 4th Communications Networks Symposium*, Manchester, UK, 1997. http://www.elec.qmw.ac.uk/dai/projects/agentCAC/.

[GJ98] M. A. Gibney and N. R. Jennings. Dynamic resource allocation by market-based routing in telecommunications networks. In Albayrak and Garijo [AG98], pages 102–117.

[GJVG99] M. A. Gibney, N. R. Jennings, N. J. Vriend, and J. M. Griffiths. Market-based call routing in telecommunications networks using adaptive pricing and real bidding. In Albayrak [Alb99], pages 46–61.

[Gol93] Germán Goldszmidt. Distributed system management via elastic servers. In *IEEE First International Workshop on Systems Management*, pages 31–35, Los Angeles, California, April 1993.

[Gru93] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In *International Workshop on Formal Ontology*, March 1993.

[GV93] N. D. Griffeth and H. Velthuijsen. Win/win regotiation among autonomous agents. In *Proceedings of the 12th Internationl Workshop on Distributed Artificial Intelligence*, pages 187–202, Hidden Valley, PA, 1993.

[GV94] N. D. Griffeth and H. Velthuijsen. *Feature Interactions in Telecommunications Systems*, chapter The negotiating agents approach to untime feature interaction resolution, pages 217–235. IOS Press, Amsterdam, 1994.

[GY95] Germán Goldszmidt and Yechiam Yemini. Distributed Management by Delegation. In *The 15th International Conference on Distributed Computing Systems*. IEEE Computer Society, June 1995.

[GY98] Germán Goldszmidt and Yechiam Yemini. Delegated agents for network management. *IEEE Communications Magazine*, 36(3), March 1998.

[HAN99] Heinz-Gerd Hegering, Sebastien Abeck, and Bernhard Neumair. *Integrated Management of Networked Systems - Concepts, Architectures, and Their Operational Application*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[Hay99] Alex Louis Gill Hayzelden. *A Multiple-Agent System Approach for Resource Configuration in Communication Networks*. PhD thesis, Departement of Electronic Engineering in Queen Mary and Westfield College, London, 1999. http://www.agentcom.org/papers/hayzelden-thesis99.pdf.

[HB98a] Alex L. G. Hayzelden and J. Bigham. Heterogeneous multi-agent architecture for ATM virtual path network resource configuration. In Sahin Albayrak and Francisco J. Garijo, editors, *Intelligent Agents for Telecommunication Applications*, pages 45–59, Cité La Villette, Paris, France, July 1998. Springer.

[HB98b] Alex L.G. Hayzelden and John Bigham. Software agents in communications network management: An overview. Submitted to Knowledge Engineering Review. http://www.agentcom.org/papers/hayzelden-web.pdf, 1998.

[HB99] Alex L.G. Hayzelden and John Bigham. Agent technology in communications systems: An overview. *Knowledge Engineering Review*, 14(4):341–375, 1999.

[HBL99] Alex L.G. Hayzelden, John Bigham, and Zhiyuan Luo. A multi-agent approach for distributed broadband network management. In PAAM99 [PAA99].

[HD98] Jim Hardwicke and Rob Davison. Software agents for ATM peformance management. In *Networks Operation and Maintenance Symposium (NOMS98)*, pages 314–321, New Orleans, USA, February 1998.

[HJ97] G. Hjálmtýsson and A. Jain. An agent-based approach to service management - towards service independent network architecture. In *Integrated Network Management V: integrated management in a virtual world*, pages 715–729, San Diego, California, USA, May 1997. IFIP, Chapman & Hall.

[Hub97] Oliver J. Huber. Multimedia services based on agents. In *IBC Intelligent Agents Conference*, Café Royal, Londres, March 1997. Available from http://www.rennes.enst-bretagne.fr/~aber/oliver/index-fr.html.

[IHM99] Leila Ismail, Daniel Hagimont, and Jacques Mossière. Evaluation of the mobile agent technology: Comparison with the client/server paradigm. In *Proceedings of OOPSLA'99, the Int. Conf. on Object-Oriented Programming, Systems and Applications*, 1999.

[ISO98] Command sequencer. International Standard ISO 10164-21, 1998.

[JA99] Brendan Jennings and Åke Arvidsson. Co-operating market/ant based multi-agent systems for intelligent network load control. In Albayrak [Alb99].

[Jen00] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):227–296, 2000.

[KH99] Graham Knight and Reza Hazemi. Mobile agent-based management in the INSERT project. *Journal of Network and Systems Management*, 7(3):271–293, September 1999.

[KINS96] Kazuhiro Kuwabara, Toru Ishida, Yoshiyasu Nishibe, and Tatsuya Suda. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, chapter Market-Based Approach for Distributed Resource Allocation and Its Applications to Communication Network Control. World Scientific Publishing, 1996.

[KMM99] Kwindla Hultman Kramer, Nelson Minar, and Pattie Maes. Tutorial: Mobile software agents for dynamic routing. *Mobile Computing and Communications Review*, 3(2):12–16, 1999.

http://nelson.www.media.mit.edu/people/nelson/research/routes-sigmobile/.

[Koo95] Richard Kooijman. Divide and conquer in network management using event-driven network area agents. In *ASCI Conference*, Heijderbos, Nederland, May 1995. available at http://netman.cit.buffalo.edu/Doc/Papers/koo9505.ps.

[KSSZ97] P. Kalyanasundaram, A.S. Sethi, C. Sherwin, and D. Zhu. A spreadsheet-based scripting environment for snmp. In *Fifth IFIP/IEEE International Symposium on Integrated Network Management IM'97*, volume 5, pages 752–765, San-Diego, California, USA, May 1997. Chapman & Hall.

[Lew99] Lundy Lewis. *Service Level management for Enterprise Networks*. Artech House, 1999.

[LF97] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, Maryland, USA, February 1997. http://www.cs.umbc.edu/kqml/papers/.

[LFP99] Yannis Labrou, Tim Finin, and Yun Peng. Agent communication languages: The current landscape. *Intelligent Systems*, 14(2):45–52, March/April 1999. http://www.csee.umbc.edu/~jklabrou/publications/ieeeIntelligentSystems1999.pdf.

[LHKC98] T. Lissajoux, V. Hilaire, A. Koukam, and A. Caminada. Genetic algorithms as prototyping tools for multi-agent systems: Application to the antenna parameter setting problem. In Albayrak and Garijo [AG98], pages 17–28.

[LKP99] Antonio Liotta, Graham Knight, and George Pavlou. On the performance and scalability of decentralised monitoring using mobile agents. In Rolf Stadler and Burkhard Stiller, editors, *Active Technologies for Network and Service Management, Proceedings of the tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM'99*, number 1700 in Lecture Notes in Computer Science, Zurich, Switzerland, October 1999. IFIP/IEEE, Springer.

[LS97] E.C. Lupu and M. Sloman. Towards a role-based framework for distributed systems management. *Journal of Network and Systems Management*, 5(1), 1997.

[LS99] D. B. Levi and J. Schönwälder. Definitions of managed objects for the delegation of management scripts. RFC 2592, 1999. http://www.ietf.org/rfc/rfc2592.txt.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[Mae94] Pettie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, 1994.

[Mag95] Thomas Magedanz. On the impacts of intelligent agents concepts on future telecommunication environments. In *Third International Conference on Intelligence in Broadband Services and Networks*, Crete, Greece, October 1995.

[Mar98] Karina Marcus. Improving reliability of intelligent agents for network management. Technical Report 98-045, Institut Eurécom, December 1998.

[MC94] F. G. McCabe and K. L. Clark. April: Agent process interaction language. Technical report, Dept. of Computing, London, UK, November 1994. available from http://www-lp.doc.ic.ac.uk/˜klc/.

[MFZH99] Jean-Philippe Martin-Flatin, Simon Znaty, and Jean-Pierre Hubaux. A survey of distributed entreprise network and systems management paradigms. *Journal of Network and Systems Management*, 7(1), March 1999.

[Mic97] The ”zero administration” initiative for windows. http://www.microsoft.com/windows/platform/info/zawmb.htm, 1997. White paper.

[MKH+96] M. S. Miller, D. Krieger, N. Hardy, C. Hibbert, and E. D. Tribble. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, chapter An Automated Auction in ATM Network Bandwidth. World Scientific Publishing, 1996.

[MKM99] Nelson Minar, Kwindla Hultman Kramer, and Pattie Maes. *Software Agents for Future Communications Systems*, chapter Cooperating Mobile Agents for Dynamic Network Routing. Springer-Verlag, 1999. ISBN: 3-540-65578-6, http://www.media.mit.edu/˜nelson/research/routes-bookchapter/.

[Mou96] Maria-Athina Mountzia. Intelligent agents in integrated network and systems management. In *Proceedings of the EUNICE'96 Summer School*, Lausanne, Switzerland, September 1996.

[Mou98] Maria-Athina Mountzia. A distributed management approach based on flexible agents. *Journal of Interoperable Communication Networks*, 1998. Special Issue on Telecommunications Service Engineering, Baltzer Science Publishers.

[MR99] Maria-Athina Mountzia and Gabi Dreo Rodosek. Using the concepts of intelligent agents in fault management of distributed services. *Journal of Network and Systems Management*, 7(4):425–446, December 1999.

[MRK96] T. Magedanz, K. Rothermel, and S. Krause. Intelligent agents: An emerging technology for next generation telecommunications? In *INFOCOM'96*, pages 464–472, USA, March 24-28 1996. IEEE.

[Mül96] Jörg P. Müller. *The Design of Intelligent Agents - A Layered Approach*. LNAI State-of-the-Art Survey. Springer, Berlin, Germany, 1996.

[Mül98] Jörg P. Müller. Architectures and applications of intelligent agents: A survey. *The Knowledge Engineering Review*, 13(4):353–380, 1998.

[NQKA98] Jan Nicklisch, Jürgen Quittek, Andreas Kind, and Shinya Arao. INCA: An agent-based network control architecture. In Albayrak and Garijo [AG98].

[NW96] Hyacinth Nwana and Michael Wooldridge. Software agent technologies. *BT Technology Journal*, 14(4):68–78, 1996. http://www.labs.bt.com/projects/agents/publish/papers/report5.ps.gz.

[Nwa96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, October/November 1996. Available at http://www.cs.umbc.edu/agents/introduction/ao/.

[Obj00] Mobile agent facility formal specification, Janury 2000. OMG TC Document cf/00-01-02.

[OL95] Raúl Oliveira and Jacques Labetoulle. Intelligent agents : a way to reduce the gap between applications and networks. In J. D. Decotignie, editor, *Proceedings of the First IEEE International Workshop on Factory Communications Systems - WFCS'95*, pages 81–90, Leysin, Switzerland, October 4-6 1995. Available at http://www.eurecom.fr/˜oliveira/wfcs/wfcs.ps.gz.

[OL96] Raúl Oliveira and Jacques Labetoulle. Intelligent agents: a new management style. In *Proceedings of the Distributed Systems and Operations Management Workshop - DSOM'96*, L'Aquila, Italy, October 1996.

[OL98] Raul Oliveira and Jacques Labetoulle. Mania: Managing awareness in networks through intelligent agents. In *Submitted to IFIP/IEEE Network Operations and Management Symposium – NOMS'98*, New Orleans, Louisiana, USA, February 1998.

[Oli98] Raúl Oliveira. *Gestion des Réseaux avec Connaissance des Besoins : Utilisation des Agents Logiciels*. PhD thesis, École National Supérieure des Télécommunications, 1998.

[OPB99] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. http://www.jamesodell.com/Rep_Agent_Protocols.pdf, 1999.

[OPB00] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending uml for agents. http://www.jamesodell.com/ExtendingUML.pdf, 2000.

[PAA99] *PAAM´99. The Pratical Application of Intelligent Agents and Multi-Agent Technology*, London, April 1999. The Practical Application Company Ltd.

[Pap00] Todd Papaioannou. *On the Structuring of Distributed Systems: The Argument for Mobility*. PhD thesis, Loughborough University, 2000.

[PE00] Todd Papaioannou and John Edwards. Towards understanding and evaluating mobile code systems. Under review. http://www.luckyspin.org/Docs/Papers/aa-mas2000.pdf, May 2000.

[Pic98] Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino, 1998.

[PLBS98] Bernard Pagurek, Yanrong Li, Andrzej Bieszczad, and Gatot Susilo. Network configuration management in heterogeneous ATM environments. In Sahin Albayrak and Francisco J. Garijo, editors, *Intelligent Agents for Telecommunication Applications - IATA'98*, number 1437 in Lecture Notes in Artificial Intelligence, Paris, France, July 1998.

[PMC67] F.P. Preparata, G. Metze, and R.T. Chien. On the connection assignement problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16(6):848–853, 1967.

[PMP] Perpetuum mobile procura project. http://www.sce.carleton.ca/netmanage/perpetuum.shtml. Department of Systems and Computer Engineering, Carleton University.

[PT99] Antonio Puliafito and Orazio Tomarchio. Advanced network management functionalities through the use of mobile software agents. In Albayrak [Alb99].

[Ret98] Agent builder: An integrated toolkit for constructing intelligent software agents. http://www.agentbuilder.com, September 1998.

[RG91] Anand S. Rao and Michael P. Georgeff. Modelling agents within a BDI architecture. In R. Fikes and E. Sandewall, editors, *Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, April, 1991.

[RG95] Anand S. Rao and Michael P. Georgeff. BDI agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 321–319, S. Francisco, CA, June 1995.

[RU95] Mike Rizzo and Ian A. Utting. An agent-based model for the provision of advanced telecommunications services. In *Proceedings of TINA '95*, Melbourne, Australia, 1995.

[Sah99] Akhil Sahai. *Conception et Réalisation d'un Gestionnaire Mobile de Réseaux Fondé sur la Technologie d'Agent Mobile.* PhD thesis, Université de Rennes 1, January 1999.

[SBM97] Akhil Sahai, Stéphane Billiart, and Christine Morin. Astrolog: A distributed and dynamic environment for network and system management. In *Proceedings of the 1st European Information Infrastructure User Conference*, Germany, February 1997. Available at http://www.irisa.fr/solidor/doc/pub97.html.

[SBP98] Cheryl Schramm, Andrzej Bieszczad, and Bernard Pagurek. Application-oriented network modeling with mobile agents. In *Networks Operation and Maintenance Symposium (NOMS98)*, New Orleans, USA, February 1998. Poster.

[SC96] Nikolaos Skarmaeas and Keith L. Clark. Process oriented programming for agent-based network management. In *ECAI96 Workshop on Intelligent Agents for Telecommunication Applications (IATA96)*, Budapest, Hungary, August 12 - 16 1996.

[Sch97] Jürgen Schönwälder. Network management by delegation - from research prototypes towards standards. *Computer Networks and ISDN Systems*, 29(15):1843–1852, 1997.

[Sea70] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1970.

[SH99] Ruud Schoonderwoerd and Owen Holland. *Software Agents for Future Communications Systems*, chapter Minimal Agents for Communications Networks Routing: The Social Insect Paradigm. Springer-Verlag, 1999. ISBN: 3-540-65578-6.

[SHB97] Ruud Schoonderwoerd, Owen Holland, and Janet Bruten. Ant-like agents for load balancing in telecommunications networks. In *Proceedings of the First International Conference on Autonomous Agents*. ACM Press, 1997. http://www-uk.hpl.hp.com/people/ruud/abcAA97.ps.

[SHBR96] R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz. Ant-based load balancing in telecommunications networks. *Journal of Adaptive Behavior*, 5(2):169–207, 1996. http://www-incl.hpl.hp.com/people/ruud/thesisRuud.ps.Z.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, pages 51–92, 1993.

[SKN97] Motohiro Suzuki, Yoshiaki Kiriha, and Shoichiro Nakai. Delegation agents: Design and implementation. In *Integrated Network Management V: integrated management in a virtual world*, volume 5, pages 742–751, San Diego, California, USA, May 1997. IFIP, Chapman & Hall.

[SM98] Akhil Sahai and Christine Morin. Enabling a mobile network manager through mobile agents. In *Proceedings of Mobile Agents'98*, Lecture Notes on Computer Science, Stuttgart, Germany, September 1998.

[Smi80] Reid G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[SML99] Morris Sloman, Subrata Mazumdar, and Emil Lupu, editors. *Integrated Network Management VI: Distributed Management for the Networked Millennium*, IFIP/IEEE International Symposium on Integrated Network Management, Boston, MA, USA, May 1999. IEEE Publishing.

[Som96a] Fergal Somers. HYBRID: Intellligent agents for distributed ATM network management. In *Proceedings of IATA'96*, Budapest, Hungary, 1996.

[Som96b] Fergal Somers. HYBRID: Unifying centralised and distributed management for large high-speed networks. In *Networks Operation and Maintenance Symposium (NOMS96)*, Kyoto, 1996.

[SR96] P. Steenekamp and J. Roos. Implementation of distributed systems management policies: A framework for the application of intelligent agent technology. In *2nd International Workshop on Systems Management*, Toronto, Ontario, Canada, June 1996. IEEE.

[SRG98] Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter Formal Methods in DAI: Logic-Based Representation and Reasoning, pages 331–376. The MIT Press, 1998.

[SRI] Artificial Intelligence Center, SRI International, Menlo Park, CA, USA. *Procedural Reasoning System User's Guide*.

[SSS+99] Luis Moura Silva, Paulo Simões, Guilherme Soares, Paolo Martins, Victor Batista, Carlos Renato, Leonor Almeida, and Norbert Stohr. JAMES: A platform of mobile agents for the management of telecommunication network. In Albayrak [Alb99], pages 77–95.

[ST94] Morris Sloman and Kevin Twidle. Domains: A Framework for Structuring Management Policy. In Morris Sloman, editor, *Network and Distributed Systems Management*, pages 433–453, Imperial College of Science Technology and Medicine, m.sloman@doc.ic.ac.uk, 1994. Addison-Wesley Publishing Compagny.

[Sta96] William Stallings. *SNMP, SNMPv2 and RMON, Practical Network Management*. Addison-Wesley, USA, 1996.

[TB94] K. Tesink and T. Brunner. (Re)Configuration of ATM virtual connections with SNMP. *The Simple Times*, 3(2), August 1994.

[TK97] Markus Trommer and Robert Konopka. Distributed network management with dynamic rule-based managing agents. In *Integrated Network Management V: integrated management in a virtual world*, pages 730–741, San Diego, California, USA, May 1997. IFIP/IEEE, Chapman & Hall.

[TL94] T. J. Timothy and D. Long. Goal creation in motivated agents. In Michael Wooldridge and N. R. Jennings, editors, *Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages*. Springer, 1994.

[Tra96] Michael David Travers. Programming with agents: New metaphors for thinking about computation. http://xenia.media.mit.edu/˜mt/diss/index.html, May 1996.

[Vig98] Giovanni Vigna, editor. *Mobile Agents and Security*. Number 1419 in Lecture Notes in Computer Science. Springer Verlag, August 1998. ISBN: 3540647929.

[Wag96] Gerd Wagner. Vivid agents – how they deliberate, how they react, how they are verified. In W. Van de Velde and J.W. Perram, editors, *Agents Breaking Away, Proc. of MAAMAW'96*, 1996. Available from http://www.informatik.uni-leipzig.de/˜gwagner/.

[WBP98] Tony White, Andrzej Bieszczad, and Bernard Pagurek. Distributed fault location in networks using mobile agents. In Albayrak and Garijo [AG98], pages 130–141.

[Wei99] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, MA, 1999.

[Wel93] M. P. Wellman. A market oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[WFFC99] Steven Willmott, Boi Faltings, Christian Frei, and Monique Calisti. *Software Agents for Future Communications Systems*, chapter Organization and Coordination for Online Routing in Communications Networks. Springer-Verlag, 1999. ISBN: 3-540-65578-6.

[Wil99] Uwe Wilhelm. *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-resistant Hardware*. Thesis number 1961, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 1999. http://lsewww.epfl.ch/ wilhelm/Papers/thesis.pdf.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[WJ98] Michael Wooldrige and Nicholas R. Jennings. Pitfalls of agent-oriented development. In Michael Wooldridge Katia P. Sycara, editor, *Proceedings of the*

Second International Conference on Autonomous Agents, pages 385–391, St. Paul, Minnepolis, USA, May 9-13 1998. ACM Press, New York.

[WJK99] Michael Wooldrige, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. In O. Etzioni, J. P. Muller, and J.Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*, pages 69–76, Seattle, WA, USA, May 1999. ACM Press.

[Woo99] Michael Wooldridge. Intelligent agents. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, MA, 1999.

[WPB99] T. White, B. Pagurek, and A. Bieszczad. Network modeling for management applications using intelligent mobile agents. *Journal of Network and Systems Management*, pages 295–321, September 1999.

[WT92] Robert Weihmayer and Ming Tan. Modelling cooperative agents for customer network control using planning and agent-oriented programming, 1992.

[YGY91] Yechiam Yemini, Germán Goldszmidt, and Shaula Yemini. Network Management by Delegation. In *The Second International Symposium on Integrated Network Management*, pages 95–107, Washington, DC, April 1991.

[Zap97] Michael Zapf. Design paradigms in agend-based systems. In *Distributed Applications and Interoperable Systems*, pages 101–107, Cottbus, Germany, 1997. Chapman & Hall. ftp://www.vsb.cs.uni-frankfurt.de/pub/papers/1997/dpiabs.pdf.zip.

[ZHGW99] Michael Zapf, Klaus Herrmann, Kurt Geihs, and Johann Wolfgang. Decentralized SNMP management with mobile agents. In Sloman et al. [16].

[ZLH96] Simon Znaty, Michel Lion, and Jean-Pierre Hubaux. Deal: A delegated agent language for developing network management functions. In *First International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1996.

Appendix A

An Efficient Polling Layer for SNMP

A.1 Introduction

The Network Management community is faced with an increasing shift in functionality requirements in Network Management Systems (NMS). It is no longer sufficient to monitor the status of the physical components of the network and to display a red-colored icon when the number of errors increases on some device's network interface. Instead, the focus is currently shifted towards service and application-level management [1, 2, 3] which is closer to the network user perception than the component-level management. In addition, the network administrators are increasingly demanding integrated information models capable of describing the managed network in a high-level way (e.g. the Common Information Model [4]). The principle is that the network administrator is interested in managing a logical abstract entity regardless of where and how to extract the management information related to that entity .

Both of these factors are expected to impose an increased load on the monitoring activity of the NMS. If the monitoring of physical components requires only sparse polling to be performed now and then, the application- and service- targeted monitoring requires a lot of data to be gathered and with more frequent update. There are more data to collect because each entity in the network information model needs to have all of its attributes up-to-date each time information is solicited by a management application. The polling frequency needs to be high enough not to miss relevant changes or degradations in the application's behavior or in the server's availability.

This sensible increase in the monitoring traffic on the managed network could be

intolerable. The management traffic should be kept to a minimum compared to data traffic, especially in the case of fault diagnosis or intensive network activity.

To anticipate this evolution, we developed a Polling Layer that optimizes the number of polling packets sent over the network. Management applications can dynamically request multiple variables to be polled while the polling layer ensures that only the minimal number of polling queries are sent over the network. In the case that multiple management applications are running in the same management system, these applications can start and stop the polling of any management data without caring about what data are being polled by the other applications.

Our Polling Layer is developed to be used with the Simple Network Management Protocol (SNMP) [5] which is by far the most used management protocol in corporate networks and in the Internet community in general. Unless a new management protocol and instrumentation is agreed on to replace SNMP (which is unlikely to be the case at least in the near future), there is no other standard that allows us to collect management data with less cost than SNMP.

Section A.2 continues with an enumeration of the requirements that the polling layer has to fulfill. Section A.3 presents the polling layer architecture that allows us to fulfill these requirements. It also details the different optimization mechanisms. Section A.4 presents a performance assessment by considering the number of polling packets sent with and without using the polling layer. It shows an important gain of polling packets as the polling activity gets more and more intensive. Section A.5 provides a qualitative discussion of the polling layer, mainly by comparing it to other related approaches. A conclusion is provided in Section A.6.

A.2 Requirements for the Polling Layer

The shift from a component-level management information model to a service-level model will affect the requirements of the management applications running upon an NMS. Obviously, a service-oriented management function would require more data to be collected from the SNMP agents than a simple component-oriented application. However, other factors have to be considered also:

Monitoring-Intensive Applications Increasingly, management applications are supposed to be more aware and predictive about the status of the network services. Therefore, they are becoming more and more monitoring-intensive, i.e, requiring more data to be collected with increased precision. Therefore, the polling layer has to make efficient

use of polling packets. This can be achieved by including multiple variables within the same polling packet whenever this is possible.

Multiple Management Functions In an NMS that offers many functions, it may happen that several management functions require the same management data to be collected from the network. Hence, an optimized polling layer should be able to combine polling requests coming from different management applications if these polling requests concern the same data to be collected.

Intensive Polling It is possible to have polling peaks during certain periods. This may happen when many management applications are running at the same time, or when the network administrator is performing an intensive management activity to diagnose a certain misbehavior of the network services and applications. The polling layer should be able to handle these peaks in the polling activity and to keep the generated monitoring traffic as low as possible.

Polling Characteristics The polling characteristics may differ according to the usage to be made of the polled value. Management applications should be able to specify the "quality" of the polling that they need for each of the variables monitored. Different usages of variable values may require different polling characteristics. The polling characteristics will be detailed later in section A.3.2.

A.3 The Polling Layer

Modern Network Management Systems and platforms rely on an information model of the managed network. Basically, this model is composed of objects with attributes (or properties) that have to be instrumented out of collected managed data from the network. In order to fulfill the requirements mentioned in the previous section, we propose to provide a separate Polling Layer that allows us to efficiently poll data from the network. The polling layer is introduced between the management applications and the SNMP-managed network components. Its role is to provide the object attributes at the application level with their respective values in order to have these attributes regularly updated with fresh values. The aim is to give the network administrator a transparent and integrated view of the managed resources regardless of where and how the relevant information is gathered.

The remainder of this section details the design of the polling layer as well as the different optimizations performed.

A.3.1 General View

The polling layer has two interfaces, as shown in Figure A.1. The lower interface interacts with the SNMP agents deployed in the network elements. As such, the polling layer assumes the manager role vis-a-vis those agents. It is responsible for the polling of the required variables on these agents and communicating the received values to the object properties in the application-level layer at the upper interface. Each instrumented object property has an associated "*Image*" of the corresponding polled variable that fits exactly the usage to be made of that property in the management application. The main concern is that the management application uses the instrumented object in a transparent way as if it were the object itself that performs the update of its attributes.

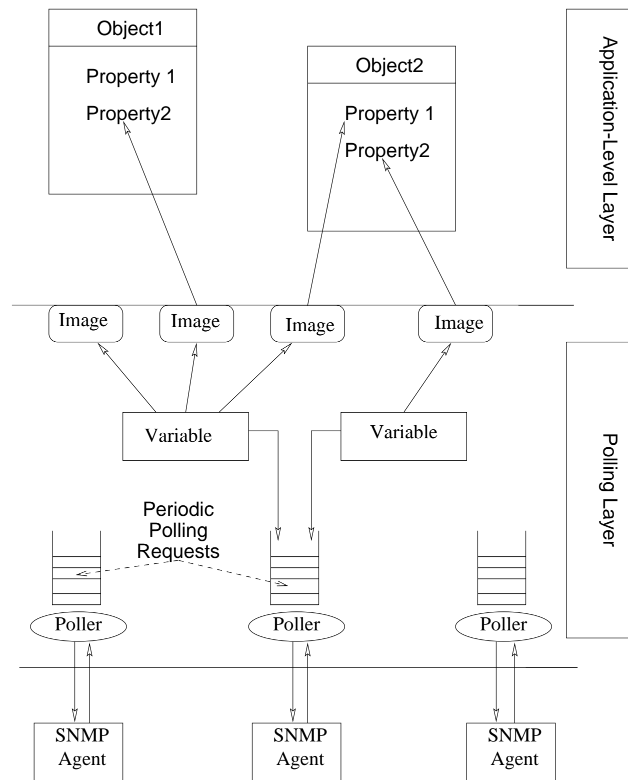


Figure A.1: The Architecture of the Polling Layer

The concept of an "image" is introduced to reflect that the same variable on an SNMP agent can be used in different manners, possibly by different applications, and therefore, should be polled with different strategies or periods. The way an image reflects its associated variable is described using a certain number of parameters called the "*Usage Parameters*". These parameters are detailed in the following section.

A.3.2 The Usage Parameters

The way a variable is polled depends on the usage to be made of its value. For example, an instrumented object's attribute used to perform precise statistical computations on a network parameter does not need the same type of polling as another attribute that is solicited only occasionally by the management application. For this reason, one of the main functions of the polling layer is to provide usage-adapted *images* of the variables polled from the SNMP agents. Each object attribute that requires a variable with a certain usage type is associated with an *image* that is exactly suitable for that usage. The polling layer is responsible for the permanent compliance of this image to the requirements of the instrumented attribute.

We identified a set of parameters that describe how a variable is polled.

- **Freshness Interval** Δt**:** It is important for an object property to have a fresh value available each time this value is required for a management task. Depending on several parameters, the freshness interval for an attribute may vary from a few seconds to several hours. This interval is to be decided according to, for example, the importance or the precision of the task involving this data.

- **Freshness Precision** ε**:** This is a precision over the freshness interval Δt. Since there is no guarantee on the time at which an SNMP query is processed by an agent, it is not possible to have the exact polling period between each consecutive pollings. The freshness precision allows us to specify when a polling request can be accepted or not. In general, if a polling query is supposed to be achieved at time $t = n.\Delta t$, then the polled value is accepted only if the SNMP agent processes the query in the interval $[t - \varepsilon, t + \varepsilon]$.

- **Offset** δ**:** In order to maximize the chance of having many variables polled at the same time, it is necessary to refer to the same time origin for all the pollings. All the polling requests are synchronized to this origin, which we denote by t_0. However, there are cases in which it is necessary to add an offset δ to t_0.

  Consider for example a large polling period of, let us say, one day, but for which the polling has to be performed at a specific hour, e.g. 8 am. If t_0 refers to the midnight hour, then it is necessary to specify an offset of eight hours in order to have the polling occur at the desired time.

  The offset can also be used for a more sophisticated reason. Take a large network of 10,000 nodes that have to be polled each day. If all the pollings are synchronized to t_0, then each day the management application will have to send 10,000 polling

requests at the same time. In addition to the processing overhead due to the emission and the reception of polling packets and their replies, the management station will be overwhelmed with a burst of replies within a short period and the network may be congested, causing packet losses. This "Feedback Implosion" problem can easily be overcome by dividing the one day period in, let us say, 1440 time slots separated by a one-minute sub-period. At each slot, no more than five network nodes are polled. In this case, the network bandwidth usage as well as the processing time are spread out over the polling period.

- **Update Method:** This parameter tells us whether and when the image should update the object attribute with the new value. Three update strategies are identified:

  1. In some cases, the attribute prefers to have the freshest possible value. The image has to update the attribute's value whenever a new value is recorded, even if the attribute still has a fresh value with respect to the required freshness interval. This kind of update is useful in many cases in order to profit from the variable pollings achieved for other images. This kind of update strategy is called *eager update*.

  2. In other cases, the attribute specifically requires its value to be updated only each Δt period. In this case, even if an intermediate value is obtained, the attribute is not updated. This kind of strategy is useful when it is necessary to perform arithmetic calculations on a set of attributes, and therefore, the values of the attributes should all be considered at the same time to have a coherent result. This strategy is called *periodic update*.

  3. Finally, the attribute may not be required to be updated unless it explicitly asks for it. This can be suitable for attributes that are only rarely solicited. For example, the operating system version running on a machine can be rarely required. When it is required, even if there is no available fresh value, it can be fetched by initiating a separate request. This kind of update strategy is called *lazy update*.

- **Availability:** This parameter allows us to select one of two refreshment strategies. In the first strategy, the application's attribute is supplied with a fresh value at least as soon as the last polled value is out-of-date with respect to Δt. In the second strategy, the attribute will not be refreshed unless explicitly accessed by the management application. In this case, the last polled value is provided unless it is out-of-date. If the last polled value is out-of-date, then a polling is initiated to retrieve a new fresh value.

While the first strategy allows the application to have a prompt fresh value each time the attribute is accessed, the second strategy may introduce more delays in the application's response time while ensuring that no polling is generated unless explicitly needed.

A.3.3 Architecture of the Polling Layer

The structure of the polling layer is shown in Figure A.1. Its components are the *images*, the *periodic polling requests*, the *variables* and the *pollers*.

A.3.3.1 Images

The image holds a representation of the polled variable that conforms to the usage parameters specified. The role of the image is to verify whether the polled values need to be themselves communicated to the object attribute or not. The image has also to constantly verify whether its usage parameters are satisfied or not. If they are violated, then the management application is notified with a warning telling the reason for the violation: network-level problem, SNMP-related error, etc.

A.3.3.2 Periodic Polling Requests

In order to provide the images with fresh values that meet their usage parameters, periodic pollings have to be performed. The characteristics of periodic pollings are encapsulated into *Periodic Polling Request* objects. A periodic polling request holds specific values for the polling period Δt, the freshness precision ε, and the offset δ.

A.3.3.3 Variables

A *variable* instance in the polling layer is associated with a MIB variable polled from a predetermined SNMP agent. The variable performs a first step of polling optimization by merging the usage parameters of its different images and producing the minimum number of periodic polling requests to be achieved by the poller. If the variable has already enough images whose periodic polling operations can satisfy the newly submitted image, the variable does not ask for a new periodic polling operation. This optimization process is described in Section A.3.4.

A.3.3.4 Pollers

The poller is viewed as a server that performs periodic polling operations submitted by the variables. It is responsible for the actual sending of SNMP polling packets. By execut-

ing an optimization algorithm which is presented in the next subsection, it can group as many polling requests as possible within the same sent packet. This optimization algorithm is described in the following section.

A.3.4 Optimizations

A.3.4.1 Poller-Level Optimization

A Poller accepts two kinds of polling requests: periodic polling requests, i.e. pollings that have to be achieved each period, and on-demand requests, i.e. single pollings to be achieved only once. A periodic polling can be identified by a tuple $\Pi = (\Delta t, \varepsilon, \delta, Oid)$, where Oid is the full MIB Object Identifier of the variable to be polled. The purpose of the poller is to include the maximum number of variables within a minimum number of SNMP PDUs. This is achieved by choosing the suitable time to send the PDU. For example, in the case of Figure A.2 in which three pollings are ready to be sent, the darkened time zone is the only time interval in which the three polling requests can be included in the same packet. The time interval $[\alpha, \beta]$ within which the poller must send the next polling request is calculated as follows.

Firstly, given the current time t, the next polling interval for a periodic polling $\Pi_i = (\Delta t_i, \varepsilon_i, \delta_i, Oid_i)$ is the interval $[\alpha_i, \beta_i]$ where $\alpha_i = \delta_i + \lceil \frac{t}{\Delta t_i} \rceil \Delta t_i - \varepsilon_i$ and $\beta_i = \delta_i + \lceil \frac{t}{\Delta t_i} \rceil \Delta t_i + \varepsilon_i$.

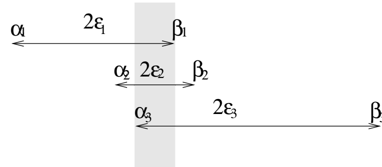


Figure A.2: Time interval in which the polling query should be sent

Secondly, the time at which the request should be sent cannot be later than $\beta = Min_i\{\beta_i\}$. For example, in the case of Figure A.2, $\beta = \beta_1$. Therefore, all the pollings having $\alpha_i < \beta_k$ could be included in the same polling request. Consequently, the polling query should be sent at $\alpha = Max_i\{\alpha_i, \alpha_i < \beta_k\}$, that is $\alpha = \alpha_3$ in the case of Figure A.2. Preferably of course, the polling query is sent at the beginning of the time interval.

The poller automatically includes the `sysUptime` MIB-II variable, which is essential if cumulative variables are being polled.

A.3.4.2 Variable-Level Optimization

The Variable is responsible for generating and submitting the required periodic pollings to the poller in order to satisfy the usage parameters of its different images. Thanks to the explicit specification of the usage wanted from an image, the variable is capable of generating just the minimum number of periodic polling requests. As a matter of fact, the variable perceives a new image as a tuple $(\Delta t, \varepsilon, \delta, U, A)$ where U is the update method and A is the availability of the image value.

What can be noticed, is that except in the case that a periodic update is required, the image could benefit from pollings having shorter periods of freshness Δt. In fact, using a more frequent polling will only refresh the image more often. Contrarily, if a periodic update is required for the image, then a new periodic polling request should be submitted to the poller in order to have the value of the variable refreshed at the exact period. Consequently, a set of images requiring either no update, or update for any new variable value, can be satisfied with a unique periodic polling request with a period equal to the smallest freshness period required by these images. These optimizations hold if the availability parameter A requires a prompt fresh value each time the image is accessed.

If the availability A does not require a prompt value for the image, then no periodic polling is necessary. Each time the image value is required, the image checks whether its last recorded value is still fresh or not. If it is not fresh, the image asks to perform an on-demand polling.

A.3.4.3 SNMP-Level Optimization

This is a simple optimization that allows us to remove duplicated variables that are already included in the polling PDU. This optimization is necessary because the same variable may be required by two distinct images.

A.3.4.4 Table Handling

In the first version of SNMP (SNMPv1), it was not possible to perform bulk retrieval of variables from an agent [5]. The only possible way to retrieve tables and to perform a “walk” on a MIB is by sending successive `GetNextRequest` packets. This remains the basic way to poll tables in SNMP agents since the new SNMP versions that support the bulk retrieval are not yet deployed except in a limited number of newly-acquired devices.

Most of the management platforms existing today retrieve tables by performing successive `GetNextRequests` starting from the table root. This is inefficient since it requires as many polling requests as elements in the table. In addition, the management applications rarely require all the data in a table. Instead, an application is in general

interested only in a few number of columns. For example, an application monitoring the `interfaces` of a certain host may not need to poll the `ifDescr`, `ifSpeed` and `ifMtu` columns since they have constant values.

The way tables are handled in the Polling Layer is through columns. The application declares what columns are needed for the instrumentation of its objects so that only these columns are polled. The poller performs the polling of multiple columns in parallel by using the same `GetNextRequest` packets. If the columns belong to the same table, then each `GetNextRequest` allows us to retrieve a row of this table. Therefore, the maximum number of requests sent to poll a table is equal to the number of table rows. Columns from different tables can be polled using the same `GetNextRequest`. As soon as the rows of a table are completely retrieved, its columns are removed from the `GetNextRequest` packets. The process continues until the columns of the table with the greatest number of rows is completely polled.

Therefore, the knowledge of the exact information required by the management applications allows us to sensibly reduce the number of requests required to poll tables. For example, the usual interfaces table of a desktop computer having two interfaces is retrieved in only two packet exchanges regardless of the number of columns required by the management application.

A.4 Performance Results

In order to evaluate the performance of our polling layer, we randomly submitted images requests to a particular SNMP agent and measured the number of polling packets sent during five-minute periods. We measured the same number of packets that would have been sent if the polling layer had not been used, i.e. without optimization. Submitted images had random values for the different usage parameters and were cancelled after a random time interval. The freshness precision ε is chosen randomly between 5% and 15% of the freshness period Δt.

At a first stage, we show in Figure A.3 the results when freshness periods are random multiples of ten seconds. We considered both cases, under a normal low monitoring demand (on the left side), and under a heavy monitoring demand resulting from an intensive use of the management application (on the right side). Under a low polling demand, we notice that the polling layer performs at most the same number of transmitted polling packets as without optimization. In fact, during the time interval between $t = 1000$ and $t = 1300$ (five hours), we had 74 polling packets without optimization, and 66 packets when using the polling layer, which implies a gain of 10% nevertheless.

The right side of Figure A.3 shows how the polling layer behaves compared to a non-

optimized polling. During the time interval between $150mn$ and $200mn$, an intensive monitoring activity is started. We can easily notice how the number of sent polling packets were cut nearly by half when the polling layer is used. Precisely, between $t = 150$ and $t = 200$ we had 620 packets without optimization, and 278 packets with optimization. This implies a gain of more than 55%.

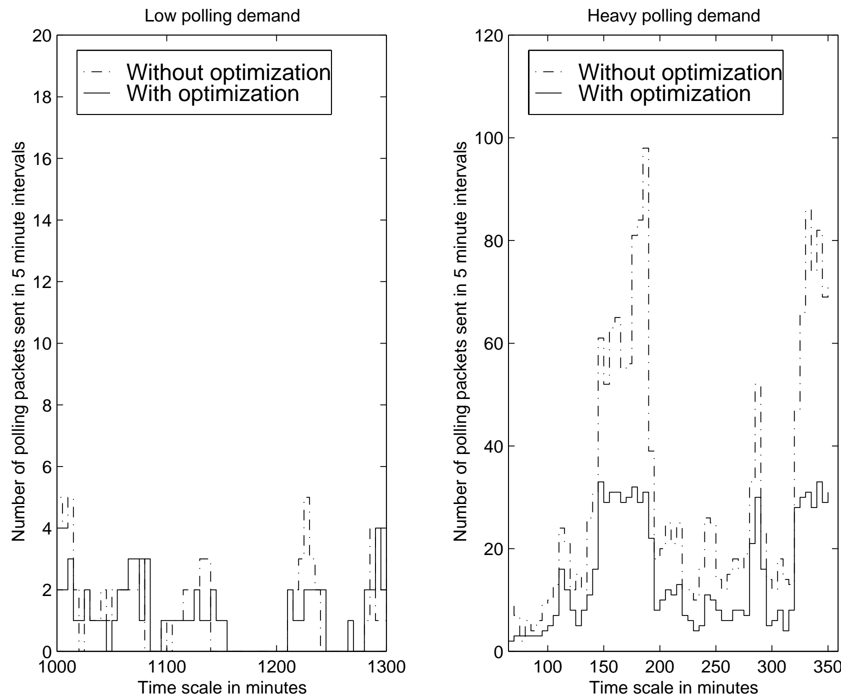


Figure A.3: Polling optimization with freshness periods multiple of 10 seconds

However, choosing Δt as a multiple of 10 seconds is rather pessimistic. In practice, the polling freshness periods are expressed as multiples of higher time units. Unless a very detailed sampling is required, the polling periods take regular values such as 1mn, 2mn, 5mn, 15mn, 30mn, 1 hour, etc. In order to study the impact of the freshness granularity, we performed similar measurements as above, but with Δt chosen as a multiple of $30sec$ instead of $10sec$. We obtained the results depicted in Figure A.4.

The left side shows the difference in the numbers of transmitted polling packets under a medium monitoring demand. The figure shows a clear gain compared to the case of Figure A.3. At time $t = 300$, we registered 933 packets for a non-optimized polling versus 513 packets using the polling layer. This implies a gain of 45%.

The right side of Figure A.4 shows the case of a burst of a heavy monitoring demand. In this case, the polling layer offers a very important gain. Between $t = 60$ and $t = 100$, we

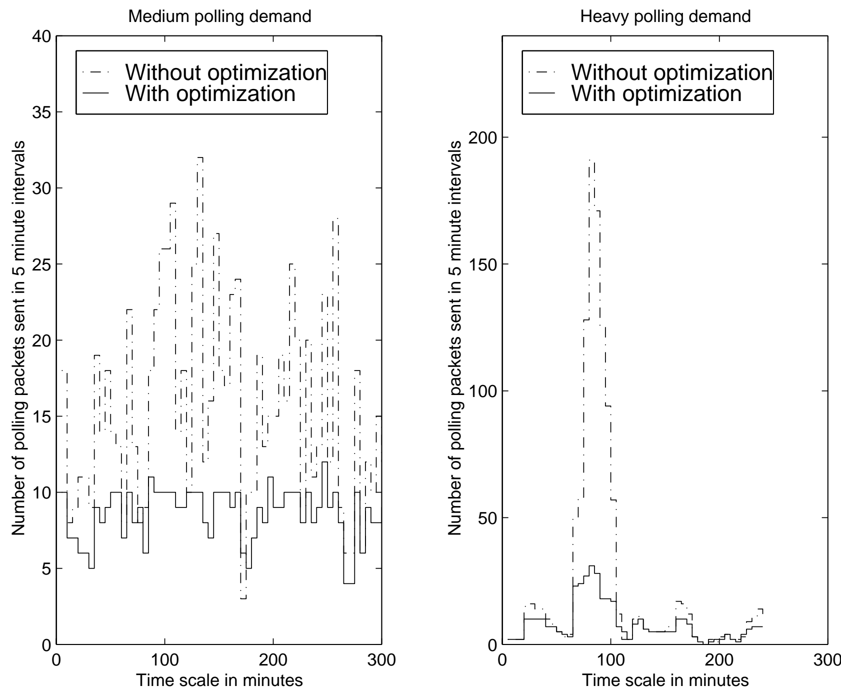


Figure A.4: Polling optimization with freshness periods multiple of 30 seconds

registered 821 packets without optimization and 172 packets with the use of the polling layer. This presents a gain of 79%.

These results show that the polling layer reduces the total number of required polling packets sent to the SNMP agent. In the worst cases, it would produce the same number of packets as without optimization. These worst cases occur when the monitoring demand is very low. As soon as the polling demand grows, the polling layer provides a considerable gain. What is important from a network administrator viewpoint is that in the case of an intensive monitoring activity such as a network failure diagnosis or a precise performance measurement, the network will not be overloaded with management traffic which could worsen the network failure or misguide the performances results.

A.5 Discussion

Some of the optimization mechanisms we used in the polling layer are not new for SNMP-based management. For example, parsing MIB tables in columns was mentioned in [6] and [7]. There are also many other approaches to reducing the SNMP traffic. Most of these approaches are revolutionary and intrusive. For example, the works described in [8] and [9] make use of the Mobile Agent technology enabled by the Java program-

ming language. However, Mobile Agents require an additional distributed runtime environment to be installed on the hosts where the agents may migrate. Moreover, many network components cannot run the Java Virtual Machine yet. Other works suggest to improve SNMP with new capabilities or even to replace it in favor of other protocols such as HTTP. A leading work in this direction is described in [10] and [11]. However, such suggestions can be considered only for the long term since the SNMP agents which are already deployed in a large number of network devices cannot be replaced in the short term. Our approach is not intrusive and we do not require any modification to the existing SNMP-instrumented devices and software.

A comparable approach was described in [12] and then further elaborated in [13]. Basically, the polling is governed by a caching model that defines different update policies. Major defined update policies include a *periodical update* with a given interval, a *lazy update* for which no polling is generated unless the variable value is explicitly requested, and a *lazy update with latency* which is a lazy update that generates a polling only if the last update time exceeds a certain period of time. In [13], more refined update policies are introduced based on the deviation of the variable value. In our opinion, the optimization of polling traffic according to the evolution of the polled variable, and more generally, by predicting the future changes of its value (e.g., see [14]) is costly in terms of resources. The polling layer must be able to keep information on the past behavior of the polled variable and to infer its future evolution. This will lead to huge memory and processing usages when the number of polled variables increase.

The polling layer we developed is also based on the concept of usage-based polling in which the polling activity is adapted to the management applications' evolving requirements. These requirements are expressed using the usage parameters introduced in subsection A.3.2. The usage parameters allow for a finer control on the polling activity. For example, the polling precision ε and the offset δ have no equivalent in the above-mentioned work ([12] and [13]). In addition, our polling layer combines this usage-based polling with an optimization at the SNMP packet level which is performed in the poller component described in subsection A.3.3. The combination of these optimizations lead to an important reduction of the polling packets sent over the network.

There are however some remaining issues for our polling layer. An important issue is that an SNMP PDU has a limited size. If the number of polled variables is large, the reply from the SNMP agent may not fit into the PDU and the whole packet is therefore lost. It is difficult to estimate the number of variables that can be included within the same packet. This number depends on the size of the Object Identifier as well as on the variable type. However, during the performance measurements that were run over long periods, this problem never occurred. We successfully could include 20 string variables within the

same packet without causing a response overflow. A more rigorous approach will be to make the poller component predict a pessimistic size of the response PDU according to the types of the polled variables as described in the MIBs.

Another factor that may cause the polling packet to be lost is that in the case of an SNMP-level error in any variable in the `GetResponse` or `GetNextResponse` packet, a null value is returned for all the other variables by the SNMP agent. This is particular for SNMPv1 and was fixed for the later versions.

A.6 Conclusion

We proposed to introduce a polling layer that minimizes the use of network resources for polling operations. A key enabling factor is the definition of the usage parameters that define how an application requires the image of a polled variable to be, and how it is to be used. By declaring the usage parameters of each image, the polling layer is capable of generating only the minimum number of polling queries to the SNMP agents. The performed measurements show that in the worst cases, the polling layer provides at least a small gain compared to a non-optimized polling. Importantly however, in the critical periods, i.e. when the monitoring activity is very intensive, the polling layer provides a large gain. This is a great advantage because the SNMP agents still receive only an acceptable number of queries even when they are heavily polled.

The polling layer can be deployed in a very flexible way. Being completely written in Java, it can be easily integrated with existing management systems as an intermediate polling server. It could also be integrated in middle-level manager agents in a hierarchical management system.

In our team, the polling layer is to be used to ensure the instrumentation of high-level information models of several management applications. These include applications of Virtual Reality for Network Management ([15]) and applications of distributed intelligent agents.

Bibliography

[1] Raúl Oliveira, Dominique Sidou, and Jacques Labetoulle. Customized network management based on applications requirements. In *Proceedings of the First IEEE International Workshop on Enterprise Networking - ENW '96*, Dallas, Texas, USA, June 1996.

[2] Patricia G. S. Flofissi, Yechiam Yemini, and Danilo Florissi. QoSockets: a new extension to the sockets API for end-to-end application QoS management. In Sloman et al. [16].

[3] Peter Parnes, Kare Synnes, and Dick Schefström. Real-time control and management of distributed applications using IP-multicast. In Sloman et al. [16].

[4] CIM specification v2.0. http://www.dmtf.org/cim/index.html, March 1998.

[5] William Stallings. *SNMP, SNMPv2 and RMON, Practical Network Management*. Addison-Wesley, USA, 1996.

[6] Jean-Philippe Martin-Flatin and Ron Sprenkels. Bulk transfers of MIB data. *The Simple Times*, 7(1), March 1999.

[7] Nikos Anerousis. An information model for generating computed views of management information. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Newark, DE, October 1998.

[8] Michael Zapf, Klaus Herrmann, Kurt Geihs, and Johann Wolfgang. Decentralized SNMP management with mobile agents. In Sloman et al. [16].

[9] B. Pagurek, Y. Wang, and T. White. Integration of mobile agents with SNMP: Why and how. Submitted to NOMS'2000, 2000.

[10] Jean-Philipe Martin-Flatin. Push vs. pull in web-based network management. In Sloman et al. [16].

[11] Jean-Philippe Martin-Flatin, L. Bovet, and Jean-Pierre Hubaux. JAMAP: a web-based management platform for IP networks. In Rolf Stadler and Burkhard Stiller, editors, *Proceedings of the 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, number 1700 in Lecture Notes in Computer Science. Springer, October 1999. Available at http://ica2www.epfl.ch/~jpmf/papers/dsom99.pdf.

[12] F. Stamatelopoulos, N. Roussopoulos, and B. Maglaris. Using a DBMS for hierarchical network management. Engineer Conference, NETWORLD+INTEROP'95, March 1995. http://www.netmode.ece.ntua.gr/papers/interop_e16.eps.

[13] Fotis Stamatelopoulos and Basil Maglaris. A caching model for efficient distributed network and systems management. In *3rd International Symposium on Computers and Communications - ISCC'98*. IEEE, July 1998. http://www.netmode.ece.ntua.gr/papers/iscc98.ps.

[14] Jia Jiao, Shamin Naqui, Danny Raz, and Binay Sugla. Minimizing the monitoring cost in network management. In Sloman et al. [16].

[15] P. Abel, P. Gros, D. Loisel, and J. P. Paris. Virtual reality and network management. In *7ème journées du GT Réalité Virutelle*, June 1999.

[16] Morris Sloman, Subrata Mazumdar, and Emil Lupu, editors. *Integrated Network Management VI: Distributed Management for the Networked Millennium*, IFIP/IEEE International Symposium on Integrated Network Management, Boston, MA, USA, May 1999. IEEE Publishing.

Appendix B

Example of Skill Declaration

The Master Skill of the PVC Case Study

```
(:skill SkMasterPVC
                          // Skills necessary for the Master skill
                          // to operate correctly
      ((:prerequisiteSkill SkTopology SkSlavePVC)

                          // First capability: createPVC
       (:verb createPVC
                :attribute source :value *
                :attribute dest :value *
                :attribute upcNr :value *
                :attribute userName :value *)

                          // Second capability: deletePVC
       (:verb deletePVC
                :attribute pvcId :value *)

                          // Belief template attributes
      (:attribute nodeList :value *)
      (:attribute agent :value *)
      (:attribute switch :value *)
      (:attribute iport :value *)
      (:attribute oport :value *)
```

```
(:attribute resId :value *)
(:attribute result :value *)
(:attribute desc :value *)
(:attribute status :value *)
(:attribute reason :value *))

                    // Information about the createPVC capability
                       // 1. Complete syntax
(  (SkMasterPVC createPVC :source * :dest * :upcNr * :userName * :reqId *)

                       // 2. Prerequisite beliefs
((me :name * :host *))

                       // 3. Created beliefs
((SkMasterPVC :result * :pvcId * :reqId *)
(SkMasterPVC :pvcId * :source * :dest * :userName * :upcNr * :status *)
(SkMasterPVC :result * :reqId * :reason *))

                       // 4. Invoked capabilities
((SkTopology findSP :source * :dest *)
 (SkSlavePVC reservePVC :iport * oport * :upc * :resId *)
 (SkSlavePVC createPVC :resId * :pvcId *)
 (SkSlavePVC cancelRes :resId *)
 (sendTo * SkSlavePVC reservePVC :iport * oport * :upc * :resId *)
 (sendTo * SkSlavePVC createPVC :resId * :pvcId *)
 (sendTo * SkSlavePVC cancelRes :resId *)
)

                       // 5. Used beliefs
((SkTopology :nodeList *)
 (SkSlavePVC :desc 'reservationDone' :resId *)
 (SkSlavePVC :desc 'reservationError' :resId * :reason *)
 (SkSlavePVC :desc 'creationDone' :resId *)
 (SkSlavePVC :desc 'creationError' :resId * :reason *)
 (recvFrom * SkSlavePVC :desc 'reservationDone' :resId *)
 (recvFrom * SkSlavePVC :desc 'reservationError' :resId * :reason *)
 (recvFrom * SkSlavePVC :desc 'creationDone' :resId *)
```

```
 (recvFrom * SkSlavePVC :desc 'creationError' :resId * :reason *)
)  )

                     // Information about the deletePVC capability
                        // 1. Complete syntax
((SkMasterPVC deletePVC :pvcId *)

()                      // 2. empty prequisite beliefs
()                      // 3. empty created beliefs

                        // 4. invoked capabilites
((SkSlavePVC deletePVC :pvcId *))

()                        // 5. empty used beliefs

 )

)     // End of skill definition for SkMasterPVC
```