`Cahoots`: A Mobile Agent Bidirectional One-to-Many Communications Framework

Verók István
<`mailto:vi@shamir.ebizlab.hit.bme.hu`>
Thesis Advisor:
Dr. Vajda István
<`mailto:vajda@tsys.hit.bme.hu`>
Department of Telecommunication
Budapest University of Technology and Economics

2001.05.20.

Contents

List of Figures

List of Tables

Chapter 1

Preface

Agent technology is still in its infancy. Research has cleared up basic theoretical and practical questions, these are slowly being standardized. Most prominently, message format protocols are established. More complex ingredients, however, are lacking. My thesis presents one such ingredient, a communications framework that mobile agents may find useful in one-to-many scenarios.

Chapter 2 presents a brief historical overview, starting from the grass roots, and highlights those cornerstones that were instrumental in the formation of today's agent technology.

Chapter 3 elaborates on the current efforts at the standardization of intelligent agents. The foremost standards authority is the Foundation for Intelligent Physical Agents (FIPA). The complete set of current FIPA standards is reviewed to elucidate the strengths and uncertainties in agent technology as we understand it now. Stable areas of knowledge are identified, along with directions of current research.

Chapter 4 focuses on the framework's design. Its applicability and goals (the requirements) are discussed first, along with identification of the various user roles that could benefit from it. Activities (use cases) available to these roles follow from who they are and what they want to achieve. A full use case catalog is enumerated. To provide all this functionality, an architecture is designed, with an emphasis on lightweight footprint and interoperation. Where appropriate, the architecture's innards are also explained in terms of familiar design patterns.

Chapter 5 takes a look at the tools used for implementation and testing, touching upon each briefly. Sample test applications that were also implemented are described.

Chapter 6 examines the finished product with respect to its stated goals, performance, and the dynamic properties of the architecture's behavior. Advantages and shortcomings are evaluated to determine where this framework (and, more generally, the architecture it implements) can be used to the most advantage.

Chapter 7 draws several conclusions that emerged from this work, concerning both this framework and agent technology in general.

Quotation Acknowledgements

The glossary contains design pattern descriptions taken verbatim from the [GoF1994] catalog's Intent sections. The FIPA standards inspired some of the illustrations and a handful of glossary item wordings.

Note

The name of the framework, `Cahoots`, is just a fancy word that crossed my mind while thinking about collaborating agents.

Chapter 2

The Road to Mobile Agents

In the beginning, there were assembly languages only. They consisted of cryptic mnemonics that translated directly to machine instructions, operating with concepts that exposed the particular machine's internal architecture completely. Programmers were free to arrange every single detail just the way they wanted it. With this freedom also came the *obligation* to do all of it. Running the software instructed the executing processor to perform the listed operations step-by-step, and nothing more. This quickly became burdensome for all but the most basic needs. Programmers found themselves repeatedly telling their computers how to perform certain common tasks. Since computers are great at handling repetitive jobs, it seemed logical that computers should somehow lend a hand with these recurring details. For this, various languages were created. Most of these didn't feature very much. Aside from a bit more readable syntax, they represented the same jump-here-and-do-this mindset. The first real change came in the early 1970s with structured programming.

Its most revolutionary aspect — subroutines — allowed programmers to break their code into chunks, grouping related parts together and separating unrelated sections. Subroutines enabled a hierarchical organization of the code: any given task could be composed of smaller tasks, plus some glue code to round out what remained undone in between. Also, compound data structures were conceived, easing the management of many instances of similar data items. Arrays, lists, structures, hash tables — first-class citizens in some languages, built from first principles in others.

The second paradigm shift, object orientation, started in the 1980s. Structuring programs into discrete levels of abstraction, is often not straightforward and even more often just plain awkward. The dichotomy of code and data is not very natural to a non-expert. Thus, instead of forcing strict hierarchies, a free-form modeling approach emerged. Most problem domains can be described using participants that interact with each other. The idea was to let programmers model these participants and their interactions directly. (Participants were termed "objects" and interactions were termed "messages" in object-oriented (OO) parlance.) Code and data are no longer the two opposite sides of the coin they used to be: they are integrated ("encapsulated") into seamless bundles that are opaque from the outside. It doesn't matter how an object is actually implemented, its actual code and data structures are nearly irrelevant. What matters is its *interface*: the ways it can interact with the world outside.

Parallel to these developments, computer networks appeared. The large community of software developers was using isolated computers, that is, machines that stood on their own. Even though the first tentative experiments to link up multiple computers to communicate with each other were performed back in the late 1960s, first real network exposure came only in the mid-1980s. At that time, those links were used for little else that data transfer. E-mail quickly became the killer app, and various applications sent each other data over the wire.

Within a few years, programmers wanted to construct software that runs not only on a single machine, but also with various parts located on separate network nodes. This was relatively easy to do: after all, it was still data transfer between software components. Why should components be unable to use other components just because they were on different hosts? Sending subroutine input parameters and returning their result values over the wire ("remote procedure calls") was a huge development in the early 1990s. Distributed computing looked promising.

A dilemma remained, however. Object-oriented programming, after all, consisted of interactions — and participants, too! Objects themselves, since they included code, did not migrate as easily on a network as pure data did. In order for an object to be meaningful everywhere, all nodes involved needed to be able to run its code. This were possible if all nodes used the same hardware — an unrealistic expectation in real life. At the very least, then, code had to be interpreted and made to run with the same effects on diverse hardware. This proved to be a more workable approach.

The final piece of theoretical underpinning appeared in those years, too. Object migration is perfect for proactive *agent*s that work to achieve some goal on behalf of the person or company that runs the agent. Not tied to any particular location, an agent could act as independently as it saw fit. By migrating to various hosts, it may save network traffic and keep the communication local. Thus, mobile agent technology was born.

Chapter 3

FIPA Standards

Agent technology is under vigorous research worldwide. Many independent technical endeavors employ agents in solving their particular problems. As a result, many competing ideas have emerged to solve the most burning questions. In order to avoid needless duplication of efforts and promote product interoperability, many large industrial companies (such as Fujitsu, Hewlett-Packard, IBM, Intel, Lockheed Martin, Lucent, Mitsubishi, Nortel, and many others), with the support of the international agent research community, have formed the Foundation for Intelligent Physical Agents (FIPA) in 1996.

FIPA standards are based on industry "status quo" feedback in a bottom-up fashion, hence they reflect the most current trends in agent software. This implies that they are still in a state of flux. None has yet made it past Experimental status (some are not even past Preliminary). In past years, FIPA standards (FIPA '97 and FIPA '98, respectively) were contiguous documents. Beginning in 2000, however, the suite of standards has been broken into smaller fragments (each identified by a five-digit number), to be evolved separately. Out of the 90 standards fragments FIPA has produced so far, only 40 survive. The rest have been deprecated, their content incorporated into the surviving fragments or simply dropped.

What follows is a quick overview of current FIPA standards (hefty 700 pages in their entirety), grouped by logical functionality. It presents the conceptual view of intelligent agents as envisaged by FIPA, and shows how these ingredients combine to form the whole architecture.

00001	FIPA Abstract Architecture Specification
00089	FIPA Domains and Policies Specification

Table 3.1: Abstract Architecture Specifications

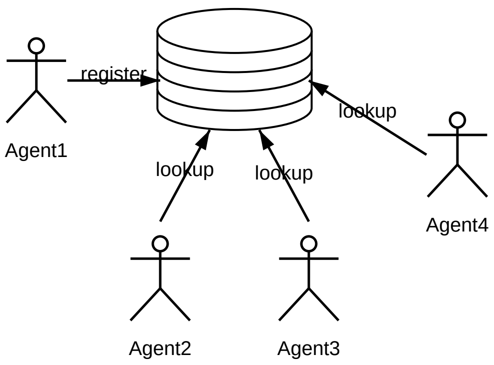


Figure 3.1: Agents Find Each Other through a Directory Service

3.1 Abstract Architecture

[FIPA00001] defines many terms and puts them into high-level context, a lot of them to be precisely defined and enumerated in other specifications. A quick rundown follows.

An AGENT is an autonomous piece of software that works in order to further a well-defined goal on behalf of the principal (person, company) that started it. For this, they may model relevant aspects of the world, and reason about it. Agents that reason at an advanced level are called INTELLIGENT AGENTS. FIPA standards codify the interfaces these agents present to the outside for various functionalities, not internal implementation details.

Agents offer various services to the outside world, and they use services presented to them. A SERVICE is defined in terms of a set of actions that it supports (very object-oriented thinking). The most basic service is a DIRECTORY SERVICE: agents may register with directory services in order to be queried, found and used by other agents (see figure 3.1). A DIRECTORY ENTRY for an agent contains at least the agent's NAME (unique) and its LOCATOR (contact address), and may also contain ATTRIBUTES (key-value tuples) that further describe the entry.

Agents communicate using MESSAGEs. These messages are built in a logical fashion (that is, a sending agent expects that the result of sending a message will help it in achieving a goal), and their CONTENTs are expressed in the AGENT COMMUNICATION LANGUAGE (ACL). The most prominent feature of the ACL is that it consists of speech acts (also called communicative acts, CAs). SPEECH ACTS are utterances that reflect the speaker's intents, with the property that by simply speaking them, the speaker commits the given act. Examples may be given in the stylized form "I hereby inform..." or "I hereby request..." or "I hereby disconfirm...". Utterances that cannot be put in this form are not speech acts. For example, saying "I hereby inform you that your train is 5 minutes late" does the informing as well, but saying "I hereby solve this equation" does not actually solve the equation. The former is a speech act, the latter is not. (For a full list of speech acts, see section 3.4.1.)

Speech acts invariably contain references that qualify the context in terms of the application problem domain. These references are expressed using a CONTENT LANGUAGE. FIPA defines several content languages, each with different strengths and weaknesses in expressive power. First order predicate logic may be best expressed in SL, while Constraint Satisfaction Problems are candidates for CCL. (For a full list of content languages, see section 3.4.2) These content languages, in turn, refer to various concrete concepts of a problem domain.

These concepts are part of an ONTOLOGY, a set of symbols together with an associated interpretation that may be shared by a community of agents or software. An ontology includes a vocabulary of symbols referring to objects in the subject domain, as well as symbols referring to relationships that may be evident in the domain. For most agents, the ontologies they can handle are implicitly given by their implementations. By sharing a common ontology, agents ascribe the same meaning to the same symbols, thus communication can take place. The parties involved will need to understand the common ACL, the content language used (these are standardized by FIPA), and they need to share a common ontology in order to meaningfully communicate (this should preferably come from widespread industrial adoption in the given problem domain).

Having formed a content to be sent, the agent transforms it into a PAYLOAD (a particular ENCODING REPRESENTATION of the content, possibly with some security and confidentiality features), and attaches an ENVELOPE, which contains at least the sender and the recipient of the message. The payload and the envelope together form a TRANSPORT MESSAGE (see figure 3.2), which is then sent to the recipient's address via an appropriate TRANSPORT (provided by various message transport services). A recipient may have more than one TRANSPORT DESCRIPTIONS (physical addresses), thus it may be possibly reachable via more than one transports.

In this way, the abstract architecture provides for two kinds of interoperability: trans-

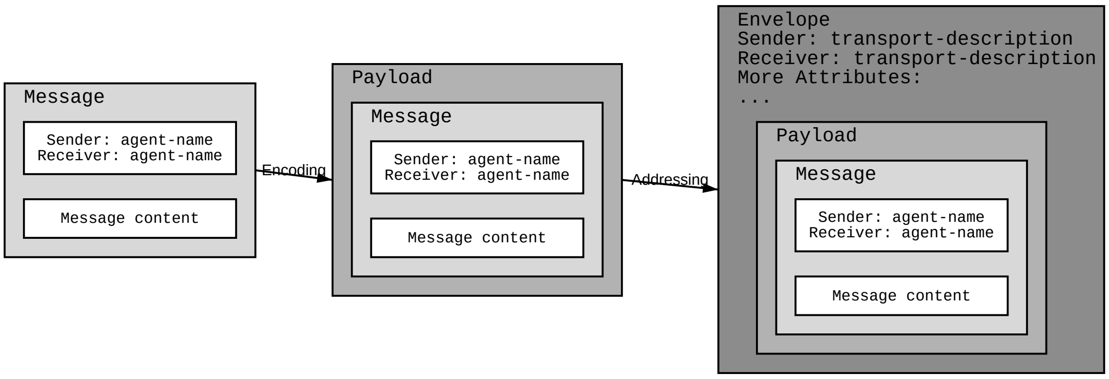


Figure 3.2: Composition of a Transport Message

port interoperability and message representation interoperability. Transport interoperability means a physical way of getting a packet of bytes from one agent to another, while message representation interoperability is enabling all agents to interpret those bytes as a message with a well-defined content, so that it carries the same meaning for everyone.

[FIPA00089] talks about high-level constraints (PERMISSIONs and OBLIGATIONs) that agent platforms may wish to enforce over the behavior of agents they serve (i.e. never use more than an allotted slice of processor time, no access to files outside the required working data set, truthful reporting on internal state). Such constraints could make up a POLICY, collected into one place so that it may be uniformly enforced over multiple agents. As a policy enforcement mechanism, DOMAINs may be set up so that belonging agents obey the domain policy. Enforcement may take the form of guards, or access points, that grant or deny permission to access a resource on a given AP. Resources could only be dereferenced through a permissions-checking guard. Permissions are inherently more easily enforced than obligations, since violation of an obligation can only be detected in hindsight. A tentative obligation enforcement mechanism could be a reputation service: occurrences of misbehavior are reported and widely publicized, so that future customers can quickly avoid untrustworthy service providers. This could be a powerful deterrent in a stable services market.

3.2 Agent Management

[FIPA00023] instantiates many concepts that were only conceptually defined in the abstract architecture. A new concept, essential to implementation, is the Agent Platform (AP). AGENT PLATFORMs provide execution time and space to agents. APs are not lim-

00023	FIPA Agent Management Specification
00087	FIPA Agent Management Support for Mobility Specification

Table 3.2: Agent Management Specifications

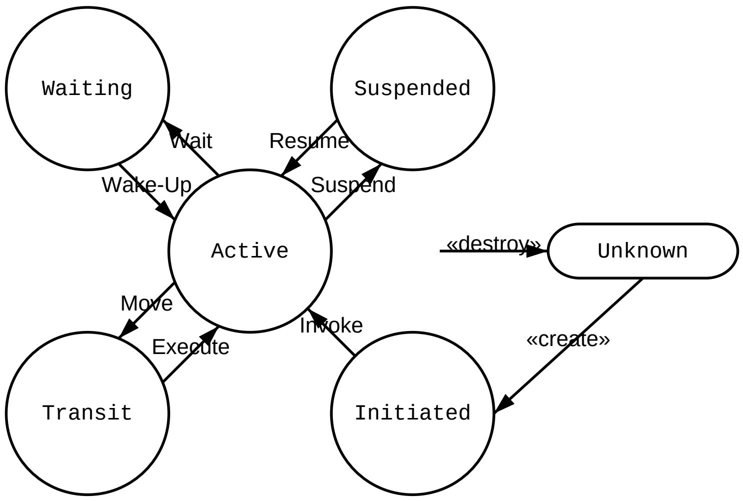


Figure 3.3: Agent Life Cycle

ited to be hosts connected to a network. Processes running on the same host could be separate APs. A cluster of load-balanced hosts may form one AP, as far as the outside world is concerned. Such implementation details do not cloud the fundamental difference between agent communications "local" to the AP and "remote" communications that traverse multiple APs.

Agents are addressed through identifiers. An agent identifier is composed of a logical name (permanent label from inception until termination), several transport addresses (usually pointing straight to the AP the agent is currently on, or to an address that forwards to the current AP), and resolvers. RESOLVERs are simply other agents that may be able to return a live transport address for the agent in question, used when the other addresses fail. Resolvers are denoted again by agent identifiers. What this means is that resolvers themselves may be mobile, just so long as some stationary agents or further resolvers exist to contact them. Indirection ad infinitum.

Agents in an AP may only be in one of five states: Initiated, Active, Suspended (forced inactivity), Waiting (active, but waiting for some external factor, such as long-running I/O operations), Transit (in the process of migrating to another AP). State change is only possible by becoming Active first, then changing to another state (see figure 3.3).

Agents must register their identifiers with the AGENT MANAGEMENT SYSTEM (AMS)

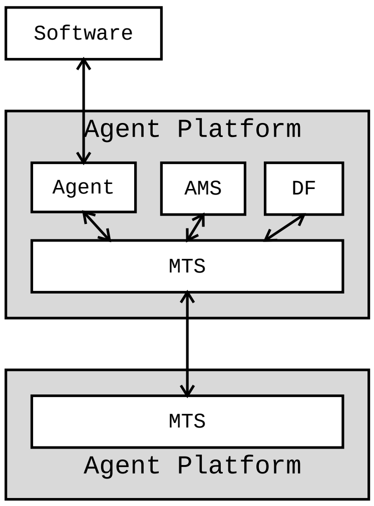


Figure 3.4: Agent Management Reference Model

component of the AP they are running on (WHITE PAGES SERVICE). The identifiers may later be deregistered, modified, and looked up. The AMS may stop and restart agents, or ask them to migrate, with the caveat that the agent may choose to ignore the request. The only request all FIPA-compliant agents must honor is `QUIT`.

Agents resident on an AP should enlist their services in the AP's DIRECTORY FACILITATOR. The DF gives agents the chance to look up other agents and be looked up in turn (YELLOW PAGES SERVICE). Agent platforms are not required to run DFs, in fact, multiple APs may use one DF (thus creating a federation). DFs are also not responsible for the data they carry. Their respective owners should refresh out-of-date information.

Communication between agents residing on separate agent platforms is achieved using a MESSAGE TRANSPORT SERVICE (MTS) (see figure 3.4).

[FIPA00087] acknowledges that mobile agents may migrate between agent platforms. It outlines the theoretical steps of simple and full mobility protocols, the difference being that with simple protocols, migration is handled entirely by the AP. Full mobility protocols let the agent handle mobility step-by-step, thus enabling customization — at the cost of increased implementation complexity. For example, cloning may be easily achieved using a full mobility protocol by simply omitting the steps that terminate the agent's local copy after a successful migration; and another agent may be invoked (created) on a remote AP out of nowhere by cloning but not sending it any local data.

Mobile agent descriptions (to be registered with the AMS) feature operating system labels, language labels, and agent platform labels. These may be needed to determine

00067	FIPA Agent Message Transport Service Specification

Table 3.3: Message Transport Service Specifications

whether migration to a given AP is possible.

A brief appendix considers integration with the CORBA Mobile Agent Facility standard. MAF (also called MASIF) is CORBA's attempt at mobile agents. It does not standardize, though, any code or data mobility capabilities, only a way to negotiate (through CORBA method calls) whether two endpoints both support the same runtime environment for object execution. This includes negotiating the operating system, language, and agent platform software on the endpoints. The standard's only concrete instantiation for these capabilities is, ironically, Java RMI. As such, MAF is not even an agent mobility negotiation standard, but an object mobility negotiation standard — one level of abstraction lower.[1]

3.3 Agent Message Transport

Message Transport Services (MTS), defined in [FIPA00067], are needed because agents carry out their respective goals in cooperation, enlisting the help of other agents where appropriate. For this, they need to communicate with other agents. Other agents are contacted via transports. A transport is a communications scheme, and a transport address points to the current location of an agent in that scheme. A concrete implementation of an MTS is called an Agent Communication Channel (ACC).

Messages consist of two separate parts: envelope and content. The headers that make up the envelope are accumulated in transit (similarly to reach headers accumulating on normal e-mail), with each AP adding headers composed of name-value pairs. The headers `:to`, `:from`, and `:acl-representation` are required initially. Multiple recipients may be listed, even multiple transport addresses for any given recipient. In this case, the transport addresses will be tried one after another until successful delivery.

The message content is simply a speech act of the ACL, with the intention that its being processed by recipients brings about some desired goal.

Prior to being sent over the wire, ACL content is SERIALIZED into either a binary format [FIPA00069], a string [FIPA00070], or XML [FIPA00071]. The envelope is serialized into

[1]MAF has been published live on the Web in early 2000, and its IDL interface still contains typos that even prevent it from compiling. The fact that all this time nobody bothered to correct the standard ought to say something about the number of people who actually use it.

00069	FIPA ACL Message Representation in Bit-Efficient Encoding Specification
00070	FIPA ACL Message Representation in String Specification
00071	FIPA ACL Message Representation in XML Specification

Table 3.4: Message Content Representation Specifications

00085	FIPA Agent Message Transport Envelope Representation in XML Specification
00088	FIPA Agent Message Transport Envelope Representation in Bit-Efficient Encoding Specification

Table 3.5: Message Envelope Representation Specifications

XML [FIPA00085] or a binary format [FIPA00088]. The old way (serializing the envelope into a LISP-like string) is explicitly deprecated, reason unknown.

Once the packet of bits to send is assembled, it may be sent using a variety of transports, depending on the available transport addresses.

[FIPA00075] is a refined version of past years' sole CORBA interface: it no longer requires the message to be a string, and the envelope is handled separately from the content. The packet is sent by a single method call: `message()`.

[FIPA00076] sends the packet over the Wireless Application Protocol (WAP) stack, which is used in handheld mobile phones and other devices.

[FIPA00084] sends it in an HTTP request, using the POST request method. This is a tried-and-true protocol of the Internet, readily handled by any web server.

3.4 Agent Communication

[FIPA00061] specifies that a message's contents include the sender and the receiver of the message, and, of course, one speech act. Optional fields `:language` and `:ontology` identify the content language and ontology used in this speech act — this is crucial for properly interpreting the contents. The `:protocol` field may denote an interaction protocol, if one is used. Any agent that responds to a message that has an identifier in its `:reply-with`

00075	FIPA Agent Message Transport Protocol for IIOP Specification
00076	FIPA Agent Message Transport Protocol for WAP Specification
00084	FIPA Agent Message Transport Protocol for HTTP Specification

Table 3.6: Message Transport Protocol Specifications

00061	FIPA ACL Message Structure Specification

Table 3.7: Message Structure Specifications

00037	FIPA Communicative Act Library Specification

Table 3.8: Communicative Acts Specifications

field must put the same identifier into the response's `:in-reply-to` field. This makes it easy for an agent to track individual conversations whose constituent messages may arrive interleaved with those of other conversations.

3.4.1 Communicative Acts

[FIPA00037] presents a catalog of communicative acts that a message may contain. Remember that these are utterances of intent that are performed at once by being uttered. They are `Accept-Proposal`, `Agree`, `Cancel`, `Call-for-Proposal`, `Confirm`, `Disconfirm`, `Failure`, `Inform`, `Inform-If`, `Inform-Ref`, `Not-Understood`, `Propagate`, `Propose`, `Proxy`, `Query-If`, `Query-Ref`, `Refuse`, `Reject-Proposal`, `Request`, `Request-When`, `Request-Whenever`, and `Subscribe`. They are presented with formal descriptions and explanations.

One caveat, though: footnote 3 in section 3.5 (`Confirm`) says: “Arguably there are situations where an agent might not want to be sincere, for example to protect confidential information. We consider these cases to be beyond the current scope of this specification.”

What this means is that the formal descriptions always assume sincerity of the sender, which belief may be unfounded in real-world applications. So take them with a healthy grain of salt.

3.4.2 Content Languages

[FIPA00007] is nothing but a statement of administrativia on FIPA's handling of new content language proposals. In order to qualify as a content language, expressive power

00007	FIPA Content Languages Specification
00008	FIPA SL Content Language Specification
00009	FIPA CCL Content Language Specification
00010	FIPA KIF Content Language Specification
00011	FIPA RDF Content Language Specification

Table 3.9: Content Languages Specifications

00025	FIPA Interaction Protocol Library Specification
00026	FIPA Request Interaction Protocol Specification
00027	FIPA Query Interaction Protocol Specification
00028	FIPA Request When Interaction Protocol Specification
00029	FIPA Contract Net Interaction Protocol Specification
00030	FIPA Iterated Contract Net Interaction Protocol Specification
00031	FIPA English Auction Interaction Protocol Specification
00032	FIPA Dutch Auction Interaction Protocol Specification
00033	FIPA Brokering Interaction Protocol Specification
00034	FIPA Recruiting Interaction Protocol Specification
00035	FIPA Subscribe Interaction Protocol Specification
00036	FIPA Propose Interaction Protocol Specification

Table 3.10: Interaction Protocols Specifications

must include at least one of: objects, propositions, and actions.

[FIPA00008] presents the single most widely used content language: SL (Semantic Language). It is capable of expressing first-order predicate logic with various quantifications. In order to ease practical implementation, three subsets are also defined: SL0 only allows representation of actions, determination of the result of a term representing a computation, the completion of an action, and simple binary propositions. SL1 adds Boolean connectives to represent propositional expressions. SL2 allows first order predicate and modal logic, but is restricted to ensure that it must be decidable. Well-known effective algorithms exist that can derive whether or not an FIPA SL2 well-formed formula is a logical consequence of a set of well-formed formulae (for instance KSAT and Monadic).

[FIPA00009] specifies the CCL (Constraint Choice Language), which can also be useful: it describes Constraint Satisfaction Problems and their solutions. CSP involves search over a multidimensional solution space that is constrained by excluding invalid combinations using posted constraint relations. This is a widely used technique.

[FIPA00010] adds the draft ANSI standard KIF (Knowledge Interchange Format).

[FIPA00011] describes another content language: RDF (the World Wide Web Consortium's Resource Description Framework), as adopted by FIPA.

3.4.3 Interaction Protocols

Interaction protocols are nothing more than templates for dialogs between multiple agents. Message exchange patterns, really. They are all 1 or 2 pages long. [FIPA00025] introduces a notation that helps to keep them that terse. Agent UML (AUML) is an extension to UML specifically geared towards messaging: agent lifelines with threads of interaction, nested and parameterized protocols — these are elegant and compact notations.

[FIPA00026]: Request: allows one agent to request another to perform some action and the receiving agent to perform the action or reply, in some way, that it cannot.

[FIPA00027]: Query: the receiving agent is requested to perform some kind of inform action. Requesting to inform is a query, and there are two query-acts: `query-if` and `query-ref` and either act may be used to initiate this protocol. In either case, an `inform` is used in response.

[FIPA00028]: Request When: provides a framework for the `request-when` communicative act. The initiator uses the `request-when` action to request that the participant do some action once a given precondition becomes true. If the requested agent understands the request and does not initially refuse, it will agree and wait until the precondition occurs. Then, it will attempt to perform the action and notify the requester accordingly. If after the initial agreement the participant is no longer able to perform the action, it will send a `refuse` action to the initiator.

[FIPA00029]: Contract Net: one manager agent wishes to have some task performed by one or more other agents and further wishes to optimize a function that characterizes the task (price, time to completion, fair distribution of tasks). The manager solicits proposals from other agents by issuing a call for proposals (`cfp`) act, which specifies the task and any conditions the manager is placing upon the execution of the task. Any proposal from a potential contractor includes preconditions that the contractor is setting out for the task, which may be the price, or the time when the task will be done. Alternatively, the contractor may refuse to propose. Once the deadline passes, the manager evaluates any received proposals and selects agents to perform the task; one, several or no agents may be chosen. The agents of the selected proposal(s) will be sent an `accept-proposal` act and the others will receive a `reject-proposal` act. The proposals are binding on the contractor, so that once the manager accepts the proposal, the contractor acquires an obligation to perform the task. Once the contractor has completed the task, it sends a completion message to the manager.

[FIPA00030]: Iterated Contract Net: an extension of the basic Contract Net IP, but it differs by allowing multi-round iterative bidding. As with the Contract Net IP, the manager issues the initial call for proposals with the act. The contractors then answer with their bids as propose acts and the manager may then accept one or more of the bids, rejecting the others, or may iterate the process by issuing a revised `cfp`. The intent is that the manager seeks to get better bids from the contractors by modifying the call and requesting new (equivalently, revised) bids. The process terminates when the manager refuses all proposals and does not issue a new `cfp`, accepts one or more of the bids or the contractors all refuse to bid.

[FIPA00031]: English Auction: the auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced (a `cfp` act multicast to all participants), the auctioneer waits to see if any buyers will signal their willingness to pay the proposed price. In the real world, as soon as one buyer indicates that it will accept the price, all bidders (by simply being in the same room) implicitly get to know about it. With agents, the auctioneer notifies all bidders of the new bid (with explicit accept-proposal and reject-proposal acts), then issues a new call for bids with an incremented price. The auction continues until no buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a buyer exceeds the auctioneer's (privately known) reservation price, the good is sold to that buyer for the agreed price (this may be a separate Request IP). If the last accepted price is less than the reservation price, the good is not sold.

[FIPA00032]: Dutch Auction: the auctioneer attempts to find the market price for a good by starting bidding at a price much higher than the expected market value, then progressively reducing the price. The auction terminates if one of the buyers accepts the price or the auction reduces the price to the reserve price with no buyers. It is quite possible that the auctioneer receives two or more bids for the same good, so multiple simultaneous competing bids are all rejected, except for the winning bid, which is up to the auctioneer to select.

[FIPA00033]: Brokering: a broker offers a set of communication facilitation services to other agents using some knowledge about the requirements and capabilities of those agents. A typical example of brokering is one in which an agent can request a broker to find one or more agents who can answer a query. The broker then determines a set of appropriate agents to which to forward the query, sends it to those agents and relays their answers back to the original requester.

[FIPA00034]: Recruiting: the same as Brokering, but the answers go directly to the requester, not through the broker.

[FIPA00035]: Subscribe: an agent requests to be notified whenever a condition specified in the subscription message becomes true.

00014	FIPA Nomadic Application Support Specification
00079	FIPA Agent Software Integration Specification
00086	FIPA Ontology Service Specification

Table 3.11: Advanced Services Specifications

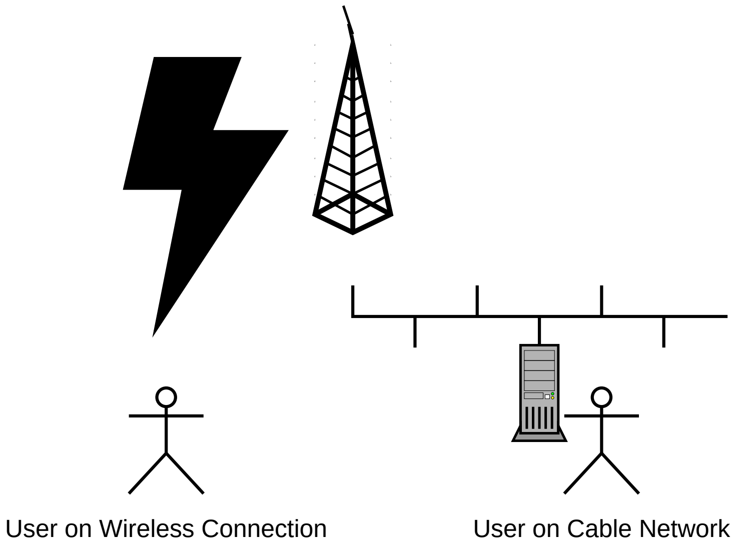


Figure 3.5: Nomadic Application

[FIPA00036]: Propose: an initiator agent proposes to the receiving agents that the initiator will do the actions described in the `propose` communicative act when the receiving agents accept this proposal. If they do (with an `accept-proposal` act), then the initiator performs the proposed actions and returns a status response.

3.5 Applications

Application-level specifications mostly describe specific solutions to specific problem domains. Three specifications in this group still command general interest:

Applications that may want to communicate over intermittent links are called NOMADIC APPLICATIONS. Examples for such behavior are business travelers and paramedic support systems (wireless connections to ambulance vehicles and wireline LAN in hospital campuses, the ambulance's computer can be docked onto the cable network when it is in the hospital — see figure 3.5). In such environments, message transport between APs may be quality-constrained, and applications to use such MTSes may need to adjust their demands accordingly (not to download images, for example).

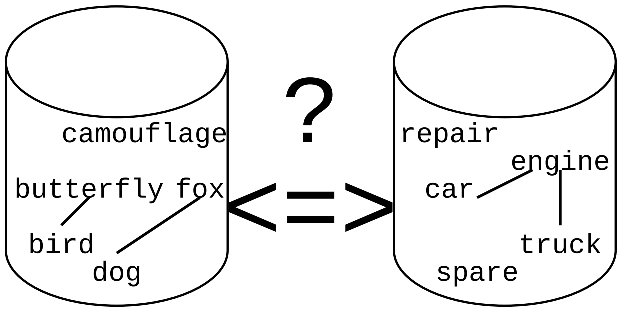


Figure 3.6: Ontology Equivalence Testing

For this purpose, [FIPA00014] describes a Monitoring Agent (MA) to gather quite detailed quality metrics of all quality-constrained message transport services. An enormous number of quality metrics is defined (from packet loss ratio to jitter, and just about everything else), and the MA may choose to track any number of them. A separate Control Agent (CA) is also specified to manage (establish, activate, close) connections on behalf of user agents, based on the measurements obtained from the MA. Other, nomadic agents may use this agent, for instance, to pool connections, or to automatically switch from one form of communication to another when the quality of the previously used MTS drops significantly.

[FIPA00079] clarifies the way agents should use legacy (non-agent) software. It recommends that agents should refrain from accessing legacy software directly. For reusability and conceptual purity, each legacy system should be wrapped in an appropriate wrapper agent, and agents should use them only through the wrappers. To help agents to access these wrappers in a uniform manner, an Agent Resource Broker (ARB) agent is described. Basically, it is a directory of operational parameters for various legacy systems (for example, a directory that lists network printers and their parameters: color, resolution, pages per minute, cost, initialization data for different printing modes). Legacy printers should store all their initialization data in an ARB, and provide a wrapper agent to properly initialize the printer to handle a given request. Any agents that want to print should first find a suitable printer by querying an ARB for printers with certain capabilities, then they should contact a wrapper agent to actually contact the printer and operate it. No direct contact exists between the client agent and the printer hardware at any time. Printer control data comes from an ARB, a general printer driver is looked up in a DF, and the driver itself manipulates the printer with the control data. This is a beautiful example of the State design pattern.

00080	FIPA Personal Travel Assistance Specification
00081	FIPA Audio-Visual Entertainment and Broadcasting Specification
00082	FIPA Network Management and Provisioning Specification
00083	FIPA Personal Assistant Specification

Table 3.12: Application Domain Specifications

[FIPA00086] is, without doubt, the most futuristic specification among the FIPA documents. It presents a (barely functioning) Ontology Agent (OA) to act as a public repository of explicit ontologies (as opposed to ones only implicitly present in the implementation of agents). This component should enable manipulation of and queries into a given ontology, equivalence testing of two ontologies (see figure 3.6), and translation from one to another, if possible. Clearly, this raises a lot of issues in knowledge representation, nearly all of them unsolved and hotly debated at present. Much future research is expected to concentrate on this area.

[FIPA00080], [FIPA00081], [FIPA00082] and [FIPA00083] are rather “hollow” specs, in the sense that they don’t contain much in the way of agent technology. They specify ontologies to describe parameters of several problem domains and enable interested parties to negotiate them (travel reservation, TV content selection using program ratings, resource allocation for virtual private network services, personal scheduling). No general applicability beyond the specific problem domains, though.

Chapter 4

Design

As seen above, FIPA standards account for a wide range of implementation diversity that occurs in various agent systems. Various interaction protocols, freely changeable ontologies, specialized content languages, different message encodings, multiple transports. But these standards are still basic building blocks. Complete working systems are built to solve concrete problem domain issues ("business logic"), and these standards are just means to that end.

FIPA message transport services (and the interaction protocols that use those services) are explicitly one-to-one transports. But what if an agent wishes to converse with multiple other agents? It has to handle every detail of communicating with many others: bookkeeping of where everyone else is, retries in case of failure are all distractions from writing good business logic. Also, once they are written for a particular application, they are not very reusable. Within one application, communication code is usually not easily replaced. And this tightly integrated existing code does not help another application: it will have to start almost from scratch.

Besides the reusability issue, the question of efficiency can also be raised. FIPA defined the message transport service agent to be a stateless, fundamentally unicast service. This is, of course, to preserve as much generality as possible. Statelessness is a virtue when messages are sent to a wide variety of recipients, relatively few at a time, but a different set each time. In this case, we don't have any advance knowledge to optimize message delivery, and the necessarily incurred channel establishment overhead must be paid for each transmission.

If, however, a restricted set of participants wish to communicate, significant optimizations can be made. Channels can be preestablished, incurring the overhead only once, and messages may subsequently be routed on the existent channels. This is even more appealing when the messages are routed not according to recipients' physical location, but their relations to the sender. To make this last point clear, consider the following scenario.

Cherie wants to buy a mobile phone. She just gained access to the Internet, now she wants to make full use of it. She grabs an agent, tells it to look for mobile phones under $100, and releases the agent into the network. The agent visits all nearby vendors, plus several individuals who want to trade their devices. But this visit is not a simple one-time occasion: a communication substructure is erected between the various sellers and Cherie. This structure connects all sellers to Cherie in a fashion so that the trade results at the various sellers gradually trickle back to Cherie. Maybe not directly, maybe through several intermediary sellers, but the results eventually get back to her.

For each seller visited, an individual copy of Cherie's agent is migrated to the seller's host, linked up with the structure, and the negotiation starts. Local negotiation is so fast, a sophisticated haggling can take place, with many thousands of offers and responses, if need be — it would leave a Syrian bazaar seller gasping for breath. When an agent instance feels that it has a pretty good position and should request further advice before proceeding, it may tell other instances about its results so far. Those instances are said to be "upstream" — that is, closer to Cherie — and may have even better results, since everybody propagates their results upstream. When a reply is received, some "downstream" nodes may want to cease their negotiation, since other sellers have negotiated with so much better results. Thus, gradually, the set of negotiation partners is pruned and haggling continues at the remaining nodes. As the results keep coming in, Cherie may decide to reject all or accept one of them, in which cases the process is finished.

For this thesis, I designed and built such a communications framework (called `Cahoots`). Below I list the requirements that shaped its initial design.

4.1 Requirements

The initial requirements are as follows, all of equal strength:

Integrated mobility support: This application scenario — one party communicating with multiple partners, directing messages to form a rooted bidirectional tree-like structure — is more efficiently handled by a separate, stateful messaging middleware than FIPA's simple MTS. What's more, integrating some form of mobility support can greatly enhance the uses for such a component. Remember, in Cherie's example, her agents migrated out to the particular locations to conduct negotiations locally.

Interoperability: Agents technology is a fragmented field with respect to implementation. Many commercial and open-source agent platforms are available. Since FIPA standardizes only the way they should exchange messages, some more restrictions need to be imposed to achieve true agent mobility between disparate agent platforms. These restrictions should be as few as possible.

Interchangeable topologies: A separate messaging component, simply by being loosely coupled, is easily lifted out of the program's design and replaced. This encourages change and evaluation of different components to do the same job in different ways. For example, Cherie could connect to the sellers in various topologies (not just in a single star topology, as most naive one-to-many communication happens), depending on the qualities she expects. The choice of the particular topology should not affect the rest of her business logic, it should function unaffected. In particular, the exact number and identity of direct neighbors in any topology at any time instance should not matter, it is an unnecessary detail.

Scalability: The implemented structures should help partition workload of the business logic, compartmentalizing results procession should come as a natural consequence of being distributed on many network nodes, and not as an afterthought.

Fault tolerance: The business logic should be as cushioned from network errors as possible. This includes retries and graceful degradation, where applicable.

In the future, other aspects of such a component may also be considered important.

4.2 Use Cases

This subsection presents a detailed view of `Cahoots`, as seen by the business logic. Various actors are identified, and all actions available to them are enumerated and explained.

The business logic of any complete system that utilizes `Cahoots` is cleanly divided into three actors: root, injectable, and site. The root and the injectables act on behalf of the entity that initiates the communication. The sites act on behalf of their respective maintainers. Note the distinction between a deployable and an injectable: an injectable is composed of pure business logic, while a deployable is an injectable plus whatever glue code is also deployed to connect the injectable to the rest of the structure (see figure 4.1).

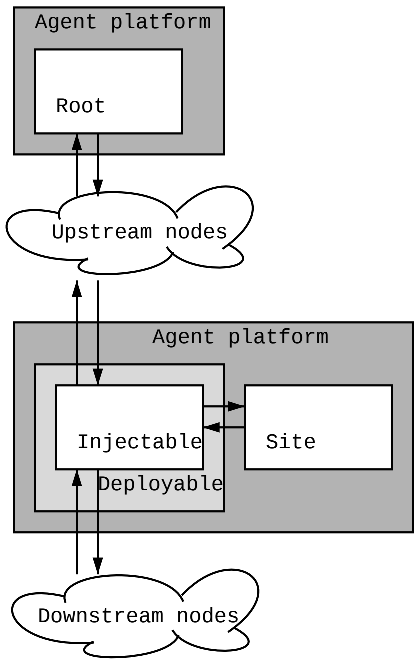


Figure 4.1: Cahoots Concepts

4.2.1 Root

The root actor is located at the root of the `Cahoots` structure. In the example, the root actor is Cherie's agent that looks up potential partners and initiates communication with them. The root receives messages that come to it from downstream (all other nodes are downstream, with respect to the root). The first thing a root does is create a new structure. All subsequent operations are then carried out using that structure. Below, use cases that require participation from the root are enumerated (see figure 4.2), and each is elaborated upon, complete with use case diagrams. Omitted are the diagrams for use cases that consist of a single operation or resemble another use case discussed earlier.

Create a new structure

A new structure is built using three pieces of data: the list of receptors (receptors receive the deployables and introduce them into their local agent platform), the topology to be used, and the injectables to be placed into the receptors (see figure 4.3).

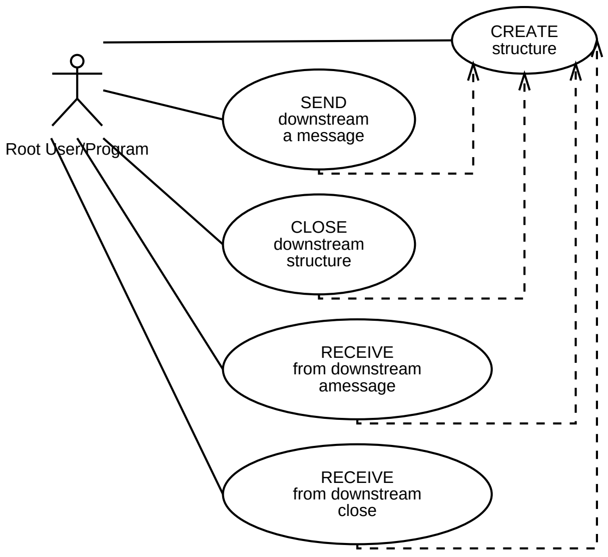

Root User/Program
CREATE structure
SEND downstream a message
CLOSE downstream structure
RECEIVE from downstream amessage
RECEIVE from downstream close

Figure 4.2: Use Cases Involving a Root

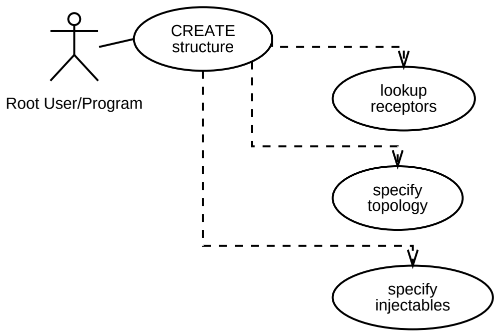


Figure 4.3: Use Case Diagram: Root Creates a Structure

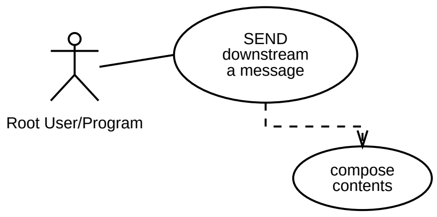


Figure 4.4: Use Case Diagram: Root Sends a Message Downstream

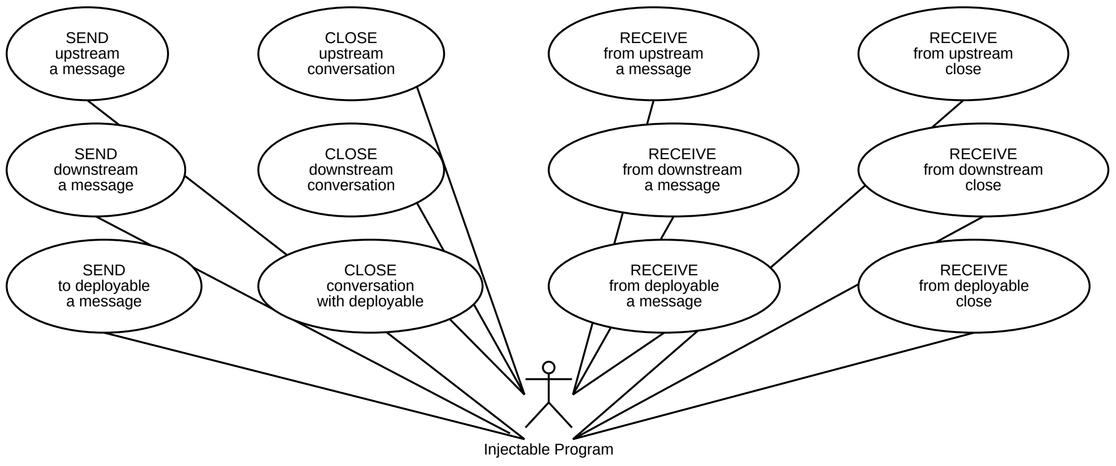

SEND upstream a message
CLOSE upstream conversation
RECEIVE from upstream a message
RECEIVE from upstream close
SEND downstream a message
CLOSE downstream conversation
RECEIVE from downstream a message
RECEIVE from downstream close
SEND to deployable a message
CLOSE conversation with deployable
RECEIVE from deployable a message
RECEIVE from deployable close
Injectable Program

Figure 4.5: Use Cases Involving an Injectable

Send a message downstream

Once the structure is in place, sending a message is easily done without further overhead: just compose its payload and send it (see figure 4.4).

Close the structure

Dismantles the structure, releasing all used resources on all participating agent platforms.

Receive a message from downstream

This is a passive event: a message reaches the root from a downstream node.

Receive notification that the structure has been closed

This is a passive event: all downstream nodes have abandoned the structure, there are no more parties left to communicate with.

4.2.2 Injectable

An injectable is a piece of business logic that has been migrated to a negotiation partner's agent platform to act on behalf of the root. Exchanges messages with other injectables and the root via the erected communication substructure. Also engages in local communication with the site whose agent platform it visited. Use cases that require participation from an injectable are shown in figure 4.5.

Send a message upstream / downstream / to the local site

After composing the contents, a message can be sent upstream / downstream / to the local site without any other overhead.

Terminate conversation with the upstream nodes / the local site (patch up the structure)

Once connection is totally lost to either all the upstream nodes or the local site, the injectable must terminate itself, since it is nothing more than an intermediary between upstream nodes and sites.

Terminate conversation with the downstream nodes

The conversation with the downstream nodes can be terminated without further administration: the injectable is not obliged to close down, neither are the downstream nodes if they have other live upstream nodes to communicate with.

Receive a message from upstream / from downstream / from the local site

This is a passive event: a message reaches the injectable from an upstream / downstream node / the local site.

Receive notification that all upstream nodes / the local site terminated the conversation

This is a passive event: the injectable must terminate itself, since there is no use for it. No more communication between root and sites can commence any more, hence all affected resources must be freed for reuse by other applications.

Receive notification that all downstream nodes terminated the conversation

This is a passive event: usually does not require any further administration.

4.2.3 Site

A site is a program that connects to the injectable and conducts a conversation with that injectable on behalf of the other party (e.g. a local seller). A site communicates solely with the injectable that arrived into its agent platform (see figure 4.6).

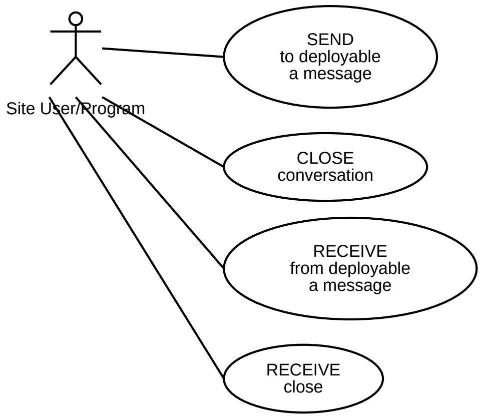

Site User/Program
SEND to deployable a message
CLOSE conversation
RECEIVE from deployable a message
RECEIVE close

Figure 4.6: Use Cases Involving a Site

Send a message to the local deployable

After composing the contents, a message can be sent to the local deployable without any other overhead.

Terminate conversation with the local deployable

Once connection is totally lost to the local deployable, the site must terminate itself, since no more messages will be forthcoming.

Receive a message from the local deployable

This is a passive event: a message reaches the site from the local deployable.

Receive notification that the local deployable terminated the conversation

This is a passive event: the site must also terminate itself.

4.3 Architecture

Having surveyed the set of capabilities `Cahoots` must present to the various actors of business logic, we now focus one level lower: how to actually implement those capabilities. See figure 4.7 for a visual overview of the architecture. To reduce clutter, operation signatures inherited from ancestor interfaces and classes have not been listed in descendants again (even if implementation is redefined in a subclass). The following discussion explains the package in detail, borrowing heavily from the design pattern terminology [GoF1994] [Hol2000].

At the "passive" side, operation begins with the creation of a `Receptor`. This is a listening proxy, a dormant hook waiting to be contacted. A `Receptor` is a low-resource substitute for `Site`s, registered at well-known directory services. Once a `Receptor` is looked up and contacted by a `Deployer` (with the intention of hooking it up into a structure), it manufactures a `Site` on behalf of the entity that runs the `Receptor` (and the `Site`), it receives a `Deployable` from the `Deployer`, and links the `Site` and the `Deployable` for local communication. Having finished these preparations, the `Deployer` is then free to continue its journey to the remaining `Receptors` it wishes to contact. The newly created `Site` and `Deployable` engage in local negotiations, and the `Receptor` sleeps on until another `Deployable` arrives to set up a separate dialogue. Note that both `Deployable` and `Site` implement `Administratable`, which means that they can be registered (by the particular `Receptor` that initiates their creation) in the local agent platform as individual agents, subject to the agent management capabilities offered by that AP.

`SiteFactory` is a prime example of the ABSTRACT FACTORY design pattern (implemented with a FACTORY METHOD). By substituting different factory objects at runtime, different products may be created instead of hardwiring them into the framework code. `Receptor` is a PROXY, of course. To accommodate each agent platform to the `Administratable` interface, the `Receptor` needs ADAPTER objects to adapt `Administratable` to the local agent platform's interface.

At the "active" side, as seen from the root agent, a `StructureFactory` erects a structure over a set of previously looked-up `Receptors` and using a given `InjectableFactory` to put `Injectables` into each `Receptor`. The actual structure is made by a `RelationsDeployer`, which visits each `Receptor` in turn, one after another, carrying a `Relations` subclass along. It deploys an `Injectable` from the factory at each, accompanied by a `Node` to link the `Injectable` to the rest of the structure through a single well-defined interface, independent of the particular structure topology. The actual topology is determined by the `Relations` subclass: every `Node` gets linked to those other `Nodes` that the `Relations` subclass determines for it. In this way, the separate `Relations` descendants `ChainRelations`, `HubRelations`, `BinaryTreeRelations`, `LinkedThroughBinaryTreeRelations` are respective algorithms that shape the overall topology.

`StructureFactory` is a FACADE that completely hides the usage details of `RelationsDeployer` and `Relations` subclasses, making them easier to use. `RelationsDeployer` is an inter-

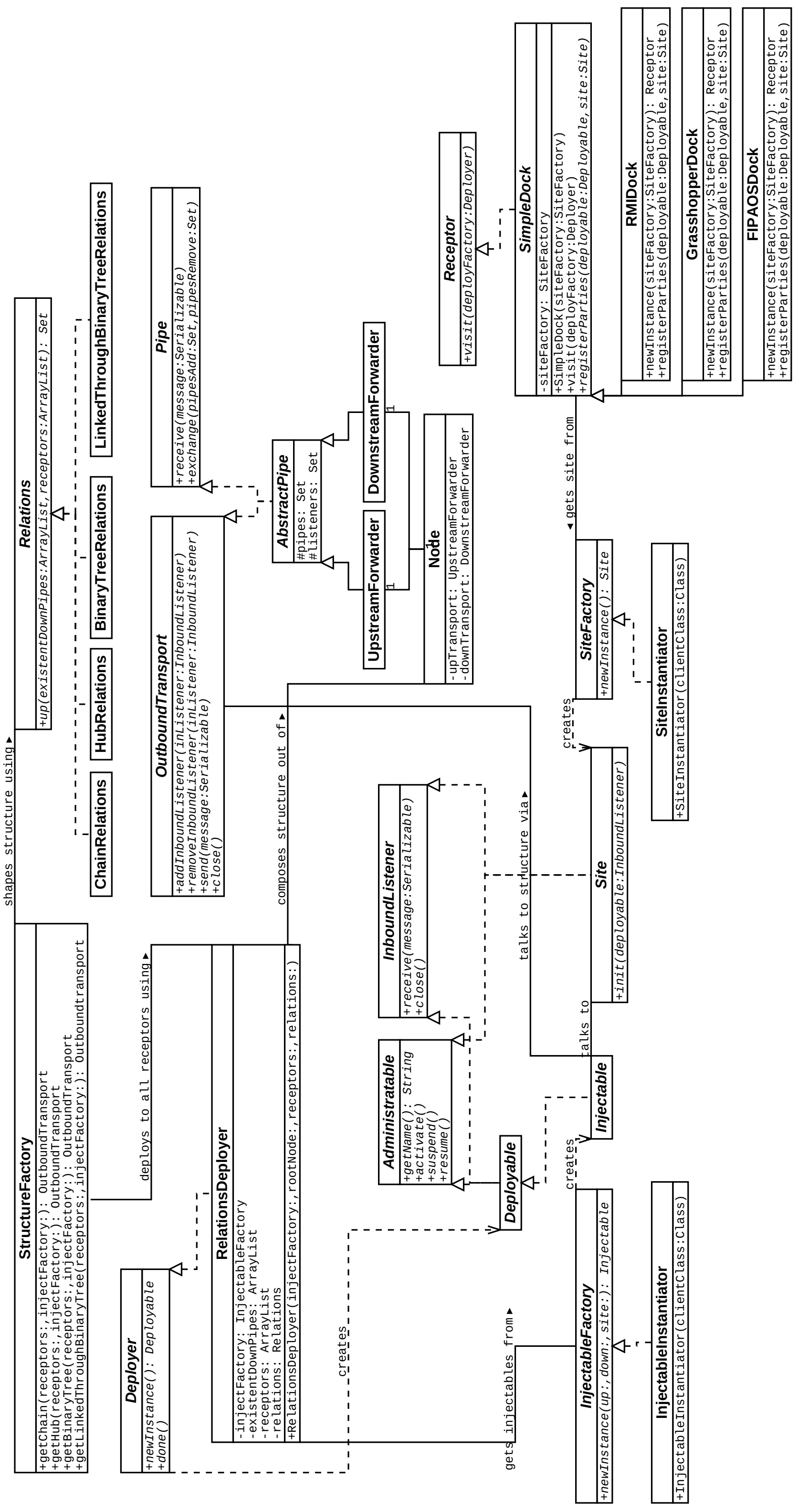

StructureFactory
+getChain(receptors:,injectFactory:): OutboundTransport
+getHub(receptors:,injectFactory:): OutboundTransport
+getBinaryTree(receptors:,injectFactory:): OutboundTransport
+getLinkedThroughBinaryTree(receptors:,injectFactory:): Outboundtransport
shapes structure using ▸
Relations
+up(existentDownPipes:ArrayList,receptors:ArrayList): Set
ChainRelations
HubRelations
BinaryTreeRelations
LinkedThroughBinaryTreeRelations
deploys to all receptors using ▸
Deployer
+newInstance(): Deployable
+done()
RelationsDeployer
-injectFactory: InjectableFactory
-existentDownPipes: ArrayList
-receptors: ArrayList
-relations: Relations
+RelationsDeployer(injectFactory:,rootNode:,receptors:,relations:)
OutboundTransport
+addInboundListener(inListener:InboundListener)
+removeInboundListener(inListener:InboundListener)
+send(message:Serializable)
+close()
Pipe
+receive(message:Serializable)
+exchange(pipesAdd:Set,pipesRemove:Set)
composes structure out of ▸
AbstractPipe
#pipes: Set
#listeners: Set
UpstreamForwarder
DownstreamForwarder
1
1
Node
-upTransport: UpstreamForwarder
-downTransport: DownstreamForwarder
creates
gets injectables from ▸
Administratable
+getName(): String
+activate()
+suspend()
+resume()
InboundListener
+receive(message:Serializable)
+close()
Deployable
talks to structure via ▸
talks to
Injectable
creates
Site
+init(deployable:InboundListener)
creates
InjectableFactory
+newInstance(up:,down:,site:): Injectable
InjectableInstantiator
+InjectableInstantiator(clientClass:Class)
SiteFactory
+newInstance(): Site
SiteInstantiator
+SiteInstantiator(clientClass:class)
◂ gets site from
Receptor
+visit(deployFactory:Deployer)
SimpleDock
-siteFactory: SiteFactory
+SimpleDock(siteFactory:SiteFactory)
+visit(deployFactory:Deployer)
+registerParties(deployable:Deployable,site:Site)
RMIDock
+newInstance(siteFactory:SiteFactory): Receptor
+registerParties(deployable:Deployable,site:Site)
GrasshopperDock
+newInstance(siteFactory:SiteFactory): Receptor
+registerParties(deployable:Deployable,site:Site)
FIPAOSDock
+newInstance(siteFactory:SiteFactory): Receptor
+registerParties(deployable:Deployable,site:Site)

Figure 4.7: Interface and Class Hierarchy

esting mix of BUILDER (the same build process can yield different structures, depending on the `Relations` used) and VISITOR (different `Deployers` could set up a structure differently when they have arrived). `RelationsDeployer` is implemented using TEMPLATE METHODs (primed with the `InjectableFactory` to fully determine its operation). It uses a `Relations` subclass as a STRATEGY. These Strategies encapsulate the topology linkup algorithms into separate objects, so simply plugging in another Strategy results in a different topology. The finished structure (embodied by `OutboundTransport` interfaces towards the root and injectables) is itself a MEDIATOR: a decoupling between multiple participators in an interaction, avoiding the need for them to refer to each other directly.

All structures are built out of `Nodes`. `Nodes` manage their relationships: each `Node` references its direct live upstream and downstream neighbors. This is done through components called `UpstreamForwarder` and `DownstreamForwarder`, descended from a common ancestor `AbstractPipe`. These forwarders implement both `OutboundTransport` (for `Injectable`-to-`Node` communication) and `Pipe` (for `Node`-to-`Node` communication). An `Injectable` can only send a message, close the transport, subscribe for incoming messages or cancel its subscription. Among `Nodes` (actually, `Pipes`, since that's how they see each other), only two atomic operations occur: message forwarding and relationship management. A `Node` may request another to add some `Pipes` to its neighboring `Pipe` sets and to remove some from the same set. For instance, a typical complex operation for a `Node` is exit the structure but patching up around it to preserve message flow for all its direct neighbors. This involves telling all upstream neighbor `Pipes` to add all the downstream neighbor `Pipes` and remove the exiting upstream `Pipe`; and telling all downstream neighbor `Pipes` to add all upstream neighbor `Pipes` and remove the exiting downstream `Pipe`.

The relationship between the `Pipe` concept and the forwarders is a BRIDGE. The forwarders are implemented using TEMPLATE METHODs. An `Injectable` gets notified of incoming messages via the OBSERVER pattern: it subscribes to the transport for notification. Internally, the forwarders use ITERATORs to go through every neighboring node in a given task (such as the patch-up mentioned above), independently of the concrete data storage structures that holds those references. And finally, the distributed nature of message passing along a single-rooted topology resembles a CHAIN OF RESPONSIBILITY: propagating messages towards places of greater authority to handle the message and make a decision whether to pass it further.

Chapter 5

Implementation

Once a stable design was reached, implementation proceeded with creating actual code and testing it.

5.1 Tools Used

Due to its novelty and possible benefits, agent technology commands great research interest worldwide. As a consequence, many ways are tried and many tools are applied to implement agents. Different tools have different uses, and the tools actually chosen reflect the given application goals at hand. Below I list the tools that I used, with a short description and a justification for the choice. Of course, these do not mean that only the given tool is capable of performing a certain capability, only that I used them successfully in this work.

5.1.1 Java

A prerequisite for code migration is that every node involved interpret all arriving executable code in the exact same way. This could be most trivially achieved by using the exact same hardware in all nodes. This is unrealistic, of course. Any hardware upgrade would immediately have to extend to the whole network, a most unfeasible aspect.

A better way is making all nodes behave as if they were of the same hardware. By defining an abstract virtual machine (VM), and providing conforming implementations on all nodes, the individual hardware differences are suppressed, as far as the migrating code is concerned. All code, programmed for the virtual machine, can then run unchanged on any host that presents a VM.

The virtual machine idea is not particularly new, it was around even in the 1970s, just think of "redcode", a game that survives to this day. (Its VM runs so-called "warriors", little competing code fragments that attempt to destroy all the others. The challenge lies in writing effective warriors, and contests are still held yearly.) Processor emulator software also belongs under the VM umbrella.

In light of this background, the fact that mobile agents also utilize VMs hardly comes as a surprise. All major agent platforms use interpreted languages. With scripting languages, such as Tcl, the source code interpreter implements the VM (in a loose sense), and code migration is simply done by putting the agent source code on the wire.

Java, on the other hand, uses a compact bytecode, easily and efficiently implemented on most processors today. This is advantageous for performance reasons, but Java offers more. It is a full-fledged high-level language. Contrary to popular belief, it is not a pioneer of brand-new concepts, it is actually rather conservative. Compared to C++, most of its positive aspects come from the omission of some "feature". No `goto` statement, no direct pointer arithmetics, no explicit memory management (that is taken care of by a garbage collector, an old idea itself). No multiple inheritance is also a bonus (nevertheless, objects can still be polymorphic using the `interface` construct). One feature is really missing, though: the Java type system does not support parametrized types (also called "generics" — C++ attempted that with its `template` construct, with dubious results).

The Java class libraries include Object Serialization and Remote Method Invocation. These are the most important for mobile agents. Serialization writes the data state of arbitrary object structures into a byte stream (and vice versa). Starting with an object, all member variables and other objects reachable from that object are also recursively serialized. The net result of a serialization is a snapshot of the transitive closure, captured in a byte stream. RMI builds on this by downloading the necessary bytecode through HTTP, and introducing them into the JVM that executes the code. In this fashion, code and data migrate seamlessly.

Of course, Microsoft heeded Java's success, and now they are working on a family of languages, most prominent of which is C# (pronounced C-sharp), targeted to rival Java. How successful C# will be is unpredictable, as only prebeta downloads are available currently (the product is due to appear in Windows XP), but even so, the similarities are striking. It also uses an interpreter (called .Net runtime, as opposed to the JVM), and also preprocesses the source (to Intermediate Language, as opposed to bytecode). With Microsoft's market muscle in mind, it could exert a heavy influence on future agent development.

5.1.2 Grasshopper

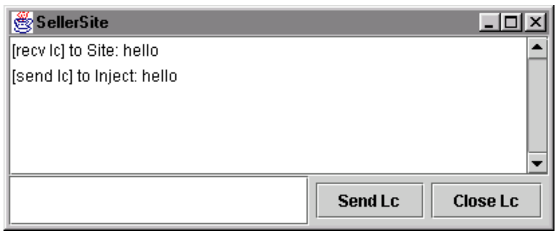
The German company IKV++ [IKV2001] has been developing this Java-based proprietary agent platform for years. Most European agent research programs use Grasshopper as the underlying platform [ACTS1999]. Grasshopper is both OMG MASIF and FIPA '98-compliant, thus usable by the existing installed base that uses those standards. Note, though, that it does not yet support FIPA 2000.

Grasshopper provides agent management primitive operations, built on Java RMI. It extends RMI by providing various default implementations of inter-platform (host-host) message transports (RMI calls them "socket factories"), adding new capabilities with these transports is a matter is plugging them right in. IKV++ itself recommends such a transport, called IAIK-SSL: it allows network traffic to be embedded in SSL encryption, with all of its security implications (confidentiality, mutual identification, nonrepudiation).

In addition to starting and stopping agents, moving and copying them from platform to platform, Grasshopper defines several stages of life cycle that all agents follow, this provides better control over agents. Within a host, it defines the concept of a place, which is simply a labeled location that serves as a meeting place for agents that look up places by their name. Each host runs an agency (the Grasshopper term for agent platform), and these can be federated to form a region (much like the shared DF case in [FIPA00023]). A Grasshopper region automatically publishes agent migration to all its agencies, thus any agent within the region can be looked up simply by name, the platform figures out the physical address automatically. This is usually needed for messaging between agents in separate agencies. Although such message exchange goes against the agent philosophy somewhat (since we use agents to cut network traffic in the first place), Grasshopper allows its use when deemed necessary. Grasshopper is also capable of asynchronous communication (method call results arrive back in the uncertain future), and sports a very coarse-grained multicast feature (which `Cahoots` does not use).

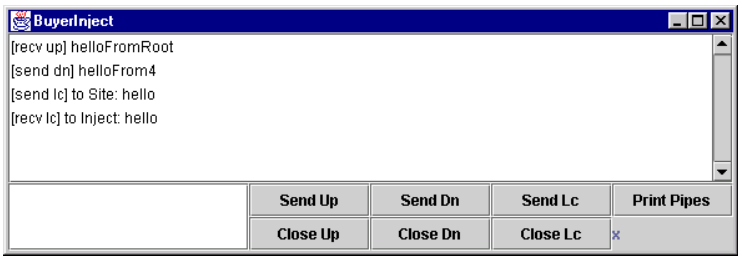
5.1.3 FIPA-OS

A Java-based FIPA 2000-compliant agent platform originated by Canadian telecom giant Nortel Networks [NN2001] and open-sourced in late 2000. It implements a growing set of FIPA 2000 standards.


SellerSite

[recv lc] to Site: hello
[send lc] to Inject: hello

Figure 5.1: Screenshot: Chat Site


BuyerInject

[recv up] helloFromRoot
[send dn] helloFrom4
[send lc] to Site: hello
[recv lc] to Inject: hello

Figure 5.2: Screenshot: Chat Injectable

FIPA-OS is specifically geared to help build intelligent agents that possibly handle more than one conversation. It automatically dispatches incoming messages to appropriate handler methods, according to the interaction protocols understood by the agent. The messages are automatically parsed from serialized representations into object data structures.

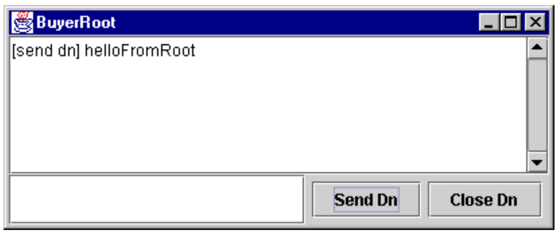
The agent platform provided by FIPA-OS provides the same basic management functions as Grasshopper. As the FIPA standards specify, it offers separate AMS and DF agents and an MTS for inter-platform communication. Agents also have the option to become persistent by using one of the persistence solutions of FIPA-OS (state snapshots by serialization or into a database).

5.2 Sample Applications

The following sample applications were used to debug and test Cahoots.

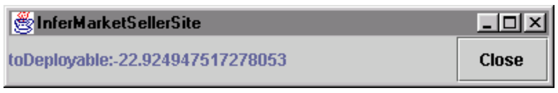
5.2.1 Chat

A fully interactive chat-like sample application to demonstrate how messages propagate from one node to the others. Every actor (root, injectable, site) displays a message window with its complete local history: message receptions and sends, and their origins and targets.


BuyerRoot

[send dn] helloFromRoot

Figure 5.3: Screenshot: Chat Root


InferMarketSellerSite

toDeployable:-22.924947517278053

Figure 5.4: Screenshot: InferMarket Site

The sites (figure 5.1) communicate locally with the deployed injectables. The injectables (figure 5.2) may send messages to all direct neighbors both upstream and downstream in the structure, and receive from all of them, too. The root (figure 5.3), of course, can only propagate to downstream neighbors.

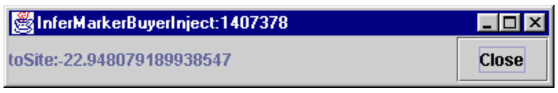
When entered in the text field, any string may be sent. Past messages can be recalled from the scrollable event log by double-clicking them.

5.2.2 InferMarket

A semi-interactive application that actually mimics distributed one-to-many price negotiation.

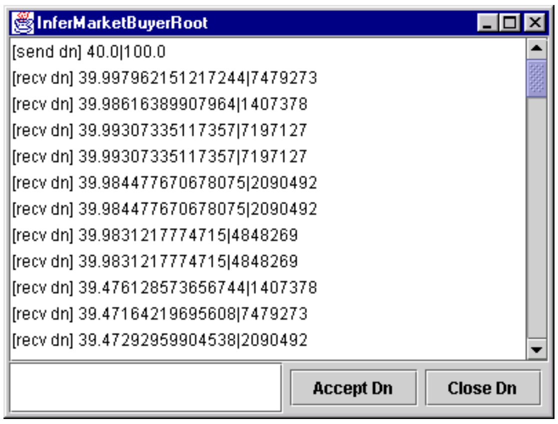
For this simple application, price negotiation between injectables and sites consists of exchanging numbers. Sites (figure 5.4) answer all offers with a number that differs by a small random amount.

Injectables (figure 5.5) are basically active objects [Hol1999], servicing requests serially (by queuing incoming requests for further processing, an application of the COMMAND design pattern). During the conversation, injectables maintain their individual copy of the lower and upper limits and direct their offers according to those. Should the site suggest prices too high (above the upper limit, twice in a row), the negotiation is judged pointless


InferMarkerBuyerInject:1407378

toSite:-22.948079189938547

Figure 5.5: Screenshot: InferMarket Injectable



```
[send dn] 40.0|100.0
[recv dn] 39.997962151217244|7479273
[recv dn] 39.98616389907964|1407378
[recv dn] 39.99307335117357|7197127
[recv dn] 39.99307335117357|7197127
[recv dn] 39.984477670678075|2090492
[recv dn] 39.984477670678075|2090492
[recv dn] 39.9831217774715|4848269
[recv dn] 39.9831217774715|4848269
[recv dn] 39.476128573656744|1407378
[recv dn] 39.47164219695608|7479273
[recv dn] 39.47292959904538|2090492
```

Figure 5.6: Screenshot: InferMarket Root

and the injectable exits the structure. If the site replies with prices between the two limits, the injectable continues the negotiation without any outside intervention, steering it toward the lower limit. If the conversation reaches the lower bound, the site's suggestion is reported up (tagged with a unique identifier for this node), and the conversation continues. Any better offers from the site are also sent up.

An inquiry from downstream is handled as follows: if its price offer is better than the injectable's lower price limit, we decrease the limit to that value. The offer is forwarded up regardless of this adjustment.

An upstream reply may contain either new price limits or an acceptance for a concrete injectable. In the former case, all nodes adjust their limits, pass the message downstream and continue negotiation. In the latter case, the chosen injectable finalizes the deal with the site through a two-way handshake. The other injectables simply abort their negotiation and quit.

The root (figure 5.6), having deployed the injectables, starts the negotiation by specifying a lower and an upper price limit to the injectables. Then it just listens to the incoming offers. The user has the option to choose an offer and accept it. The acceptance is propagated to all downstream injectables. After the process is concluded, the structure is dismantled.

This negotiation procedure is very rudimentary. Upstream go prices to be confirmed, downstream go new price limits and acceptances. Not very intelligent at negotiating prices, but sufficient for stress-testing `Cahoots` with actual network traffic.

Chapter 6

Evaluation

Testing of the framework was performed using the two sample applications referred to above. The hardware used was 5 PCs, located in the EBizLab, each using a 10Mbps thin Ethernet wire connected to a Cisco Catalyst 2900 XL switch. All machines were fully inside the lab's internal local network to avoid problems with NAT (IP masquerading, performed by the lab's gateway to the university backbone). The used local IP addresses were in the 10.105.1.x subnet range, with $x = 2, 3, 4, 6, 16$. Operating systems on the machines were Microsoft Windows NT 4.0 and Microsoft Windows 2000 Professional. The Java environment was in all cases version 1.3.0_02. During the testing runs, the injectables were introduced into a heterogeneous setup of normal RMI (no platform support), Grasshopper, and FIPA-OS agent platforms. The empirical results are summarized below.

6.1 Performance

The very first impression of the system was that the structure construction took a while. Factors external to `Cahoots` could greatly influence user perception, though — lookup of `Receptors` from nameservices, for instance, especially when the nameservice did not respond normally and the client application timed out before going on to the next nameservice. When these factors were suppressed, there was still an overhead. For the 5 machines, deploying a structure usually took about 25 seconds. The structure topology did not matter, since deployment was done sequentially, visiting one machine after another, to preserve full generality (the option to make any topology). Specialized setup strategies could, in theory, parallelize the topology establishment somewhat, but only for special topologies. Full generality demands that the deployer have references to all already-existing nodes at

Number of Nodes	1	2	3	4	5
Deployment Time (seconds)	11	14	17	20	25

Table 6.1: Measured Deployment Times

all times, and no other deployment method guarantees this.

6.1.1 Reliability

The communication structures proved quite resistant to network errors. Structure partitions that, on occasion, lost all upstream contact (be it from network errors or manual termination, just to see what happens) dismantled themselves quickly and reliably, freeing up used resources. The partition that included the root continued to function, it just collected results from the nodes still available. This was a powerful demonstration of graceful degradation.

6.1.2 Speed

As a consequence of the linear setup method, the structure construction overhead was a direct linear function of the number of nodes to be visited, each additional node incurring another 3 seconds (see table 6.1). Network errors could, of course, add to this best-case value. For a very large number of participants, structure construction may be a very large performance hit. Once constructed, the structure passed messages very rapidly. Java RMI only sent class files when necessary (when the remote party did not have the necessary code to execute a given method or make sense of a given data structure), this kept the number of HTTP connections at a minimum. Data flowed between network hosts with no perceptible delay between departure and arrival.

6.1.3 Scalability

Once the structure was in place, the vast majority of the communication could be commenced locally, with network traffic kept at a moderate amount. Most of the time, binary trees or linked-through binary trees were used (see figure 6.1).

An unoptimized, very naive negotiator program (InferMarket), using a linked-through binary tree, handled over 99% of its total traffic locally. For a detailed statistical breakdown of a sample 2-minute run, see table 6.2. Note that this table characterizes the InferMarket sample application rather than `Cahoots`. Any thoughtfully written negotiator application

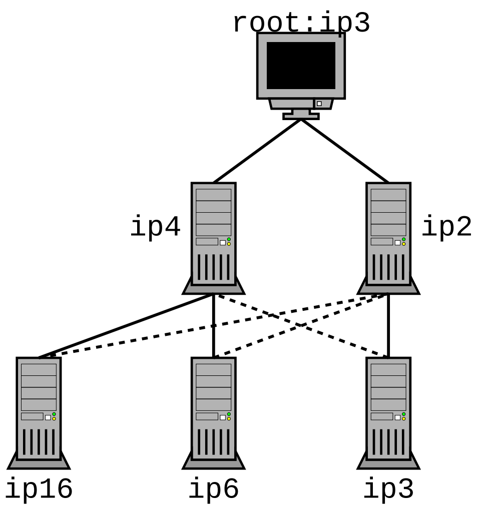


Figure 6.1: Logical Topology of a Typical Experiment

	ip4	ip2	ip16	ip6	ip3
Bytes sent upstream	345933	289439	83139	76981	78003
Bytes sent downstream	2026	2026	3037	3037	3037
Bytes sent locally	5255511	4441809	5636624	5208335	5406874
Bytes received from upstream	2026	2026	3037	3037	3037
Bytes received from downstream	238123	238123	0	0	0
Bytes received locally	5254515	4378609	5634549	5206396	5404921
Messages sent upstream	337	282	81	75	76
Messages sent downstream	2	2	3	3	3
Messages sent locally	5159	4361	5534	5114	5309
Messages received from upstream	2	2	3	3	3
Messages received from downstream	232	232	0	0	0
Messages received locally	5158	4299	5532	5112	5307
Avg bytes/msg sent upstream	1026	1026	1026	1026	1026
Avg bytes/msg sent downstream	1013	1013	1012	1012	1012
Avg bytes/msg sent locally	1018	1018	1018	1018	1018
Avg bytes/msg received from upstream	1013	1013	1012	1012	1012
Avg bytes/msg received from downstream	1026	1026	0	0	0
Avg bytes/msg received locally	1018	1018	1018	1018	1018
Lifetime (milliseconds)	144568	144327	138179	134774	130949

Table 6.2: Statistical Breakdown of a Sample InferMarket Run

can easily cut down several more orders of magnitude on the network/local communication ratio.

The distributed nature of the framework naturally encouraged writing distributed negotiation algorithms, spreading the required computational power over many network nodes, rather than concentrating them in the root. The root still had some more processing to do than the rest, since it had to make the final decisions (possibly — preferably? — with user assistance). Nodes closer to the root usually had to pass more messages than leaf nodes, but in general, all nodes in a well-balanced structure should handle the same average amount of messages. The better the negotiator, the more likely this will in fact happen.

6.2 Global Behavior

Using preestablished communication structures has several advantages and disadvantages. The following paragraphs discuss the remaining design goals and additional considerations.

6.2.1 Interoperability

The platform-independent nature of Java enables any program written on top of it to run unchanged on a wide range of hardware and operating systems. Agent platforms are no exception, and as all of them supported RMI as the common denominator between network hosts, adapting the incoming injectables and locally created sites to the particular agent platforms was easy. One drawback of using RMI as the wire format is that it provides no security features by default. As diverse as the Internet is, however, arbitrary agent platforms cannot be expected to support, say, SSL encryption on their endpoints. (Since a chain is just as strong as its weakest link, all participants would need to use a common secure communication method to achieve safety.) A fallback has to be used instead, and RMI is perfect for that fallback in the absence of further information.

Agent platforms that support some form of encryption could also publish this information in the directory services where they register their receptors. Theoretically, an agent could find an encryption supported by all the receptors, and erect a communication structure using that encryption on the wire. This is not currently supported by `Cahoots`, and it's far from clear if it should be.

6.2.2 Structure Distortion over Time

Structure distortion occurs when individual injectables, feeling that they can no longer contribute usefully to the ongoing negotiation, decide to quit. In this case, they patch up the structure around themselves, linking each upstream neighbor to all downstream neighbors and linking each downstream neighbor to all upstream neighbors and dropping out from the middle. A structure may also become disrupted when the network is partitioned, and a subset of the nodes is severed from the partition that includes the root. This latter possibility is, however, an exceptional event rather than the norm.

The chain and hub structures enjoy the advantage that they cannot become distorted over time. For the chain, there is exactly one upstream and one downstream connection going from each node, thus any number of patching up preserves this property. The hub structure consists of only one level, hence no patching up ever occurs. In either topology, partitioning may force a certain number of nodes to leave, but the structure is not deformed.

The binary tree and linked-through binary tree structures, however, show structure distortions. Each leaving node adds 1 pipe to upstream nodes, and 0 pipes to downstream nodes (these figures are 3 and 1 pipes for the linked-through tree). The net result is that as the structure gets thinner, it starts to accumulate relatively more interconnections among its nodes. This, of course, increases network traffic, but also increases reliability: redundancy ensures that messages have a greater chance reaching their destinations on a best-effort network.

If this link-accumulating behavior is undesired, the code for `Nodes` needs to be modified. Other structures could guarantee a bounded number of direct neighbors for all participating nodes. 2-3 trees are a simple example. This would occasionally incur the $O(\log n)$ cost of rearranging the tree as nodes drop out, generating additional network traffic.

Chapter 7

Postmortem

Fast-forward to 2030. We live in the age of calm computing [WB1996]. Computers have become an indispensable part of our lives, helping us in myriad small ways, performing many tasks unobtrusively, not drawing attention to themselves other than necessary. In fact, they have blended so seamlessly into the background that we become aware of them only when something is amiss. Except for power outages and software crashes, they constantly prefilter our mail and newsfeed, schedule new appointments and keep track of the food in the fridge.

Today, people purchase food in one of three types of stores: convenience stores, grocery stores and supermarkets. These stores balance the need to get a wide variety of food products against the need to get them in as short a time as possible. Basic items are bought quickly in a convenience store; the weekend purchases can take an hour or two and result in a trunkful of goods in the back of the family car.

How can computing help Cherie, our average user, with this activity? She wants food in her home, and she has a limited amount of time. Simply tracking down every needed item at the cheapest place to buy it is not really helpful. Going home on a weekday evening, she will most likely have time to stop at one or two shops; picking up the bread, ham, milk, eggs and ice cream at different stores is clearly out of the question for her. She needs help in choosing those shops she does have time for. Appropriate agents can do this also, no problem.

Cherie has other options, too. This being the 21st century, she might own cheap and fast network connectivity, letting her contact the stores not only for a few moments, but also for a prolonged time interval. She might even deploy her agents for days or weeks on end, constantly looking for discounts and price reductions, and taking advantage of them.

As long as being online costs less than the money she saves on the food prices plus the time she picks up her new possessions, she opts for constant net search.

So does everybody else. Once the technology becomes cheap enough, people swarm the net looking for ways to save money. The only reason they didn't browse every shop before they made a purchase is that it took too long, and it was too much of an effort. When technology helps them overcome that hurdle, they use it. This means an enormous amount of traffic for future networks to handle.

The shops also have an incentive to be accessible online. By reducing their prices a bit, they attract lots of customers. Of course, the other stores react to this by lowering their prices too, a very fast response. The net result: uniform prices, ensured by a dynamic equilibrium.

Is this setup feasible? At the extreme, a city may have millions of citizens and thousands of shops. Unless all stores house gigantic supercomputers to serve the millions of agents each of them receives, some form of access control is necessary, especially when the agents are sophisticated and perform quite a bit of thinking on their own, taking processor time. In this case, agent deployment could take a long time, maybe an hour. On the other hand, long-running agents do not have to be deployed often, so the long setup time could be acceptable. Events (such as price change notifications), however, could wake up every agent. The havoc of a million agents all clamoring for CPU time could only be handled by massive load balancing, compartmentalization, and other techniques.

Of course, most cities are not nearly this big. An average settlement in an average country amounts to several thousand people, resulting in much lighter loads. Even today's technology could serve a village or suburb. With 21st century technology...

Recognizing an opportunity, postal services experience a revival. Having lost a sizable portion of the messaging market to the Internet, they turn to small parcel delivery. They offer Cherie the possibility to receive, along with her morning newspaper, all the groceries she had bought at various stores. Cherie accepts gladly, because most of her purchases didn't really require her physical presence anyway. One box of ice cream is the same as another, she would just like to have one, any one. And, once a critical mass of market penetration is reached, this can be done profitably, because the marginal cost of carrying yet another customer's parcel is negligible compared to the cost of having a delivery truck at all. Cherie still goes to her favorite wine merchant to hand-pick that vintage Chateau Lafite, but she never again wastes time to buy bread and soap.

This vision led me to create `Cahoots`.

Chapter 8

Glossary

Abstract factory Creational design pattern: Provide an interface for creating families of dependent objects without specifying their concrete classes. (compare *Factory method*)

Adapter Structural design pattern: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Agent A computational process that implements the autonomous, communicating functionality of an application.

Agent attributes A set of properties associated with an agent by inclusion in its directory entry.

Agent communication language A language with a precisely defined syntax, semantics and pragmatics, which is the basis of communication between independently designed and developed agents. Communicative acts can be expressed in the ACL.

Agent management system The white pages service in an agent platform: manages the agents resident on that platform, enabling their copying, migration, and termination through outside intervention. (compare *Directory facilitator*, *Yellow pages service*)

Agent name An opaque, non-forgeable token that uniquely identifies an agent.

Agent platform A conceptual node in the agent universe. Inter-platform communication is done through a message transport service, while agents within the same platform may communicate directly.

Bridge Structural design pattern: Decouple an abstraction from its implementation so that the two can vary independently.

Builder Creational design pattern: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Chain of responsibility Behavioral design pattern: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command Behavioral design pattern: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Communicative act A special class of actions that correspond to the basic building blocks of dialogue between agents. A communicative act (also called "speech act") has a well-defined, declarative meaning independent of the content of any given act. (see also *Speech act theory*)

Content Content is that part of a communicative act that represents the domain-dependent component of the communication.

Content language A language used to express the content of a communication between agents.

Design pattern A description of communicating objects and classes that are customized to solve a general design problem in a particular situation. There are three kinds of design patterns: creational (see *Abstract factory*, *Builder*, *Factory method*), structural (see *Adapter*, *Bridge*, *Facade*, *Proxy*), and behavioral (see *Chain of responsibility*, *Command*, *Iterator*, *Mediator*, *Observer*, *State*, *Strategy*, *Template method*, *Visitor*).

Directory entry A composite entry containing the agent name, locator, and agent attributes of an agent.

Directory facilitator The yellow pages service in an agent platform: enables semantic registration and lookup of running agents. Entries may carry additional relevant information about the agents, such as the problem domain they work in, the interaction protocols they support, the auxiliary software components they require to successfully run. (compare *Agent management system*, *White pages service*)

Directory service A service providing a shared information repository in which directory entries may be stored and queried.

Encoding representation A way of representing an abstract syntax in a particular concrete syntax. Examples of possible representations are XML, FIPA strings, and serialized Java objects.

Envelope That part of a transport message containing information about how to send the message to the intended recipient(s). May also include additional information about the message encoding, encryption, and other quality parameters.

Facade Structural design pattern: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Factory method Creational design pattern: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (compare *Abstract factory*)

Interaction protocol A conceptual skeleton for message exchange between multiple agents. Describes the causal sequence of message transfers to achieve a particular information-exchanging transaction.

Iterator Behavioral design pattern: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Locator A set of transport descriptions used to communicate with an agent.

Mediator Behavioral design pattern: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Message A unit of communication between two agents. A message is expressed in an agent communication language, and encoded in an encoding representation.

Message transport service A service that supports the sending and receiving of transport messages between agents residing on separate agent platforms.

Middleware Housekeeping system component that interfaces two or more other components. Does not have a clear-cut independent function of its own, it always acts in conjunction with other components, but performs auxiliary "plumbing" to help the business logic's functioning. Messaging services, database drivers, persistence services are all examples of middleware.

Mobile agent An agent that may migrate between different agent platforms over the course of its lifetime. (oppose *Stationary agent*)

Nomadic application An application executing on a mobile device and communicating with other application components over intermittent links. Establishing links on demand and adapting to various link qualities are the prime characteristics.

Obligation A constraint over an agent's future behavior: the agent may be expected to send report updates every five minutes, or spend no more than half a minute at any one agent platform. Failure to do so can only be detected in hindsight, and offenders are therefore harder to contain and punish. (compare *Permission*)

Observer Behavioral design pattern: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Ontology A set of symbols together with an associated interpretation that may be shared by a community of agents or software. An ontology includes a vocabulary of symbols referring to objects in the subject domain, as well as symbols referring to relationships that may be evident in the domain.

Payload A message encoded in a manner suitable for inclusion in a transport message.

Permission A constraint over agent behavior: before committing certain sensitive operations, an agent must have permission for that operation. In the absence of that permission, the operation cannot be carried out. (compare *Obligation*)

Policy A uniform set of permissions and obligations that all agents belonging to a given domain must obey.

Policy domain The set of agents that obey a common policy of security restrictions. (compare *Problem domain*)

Problem domain The particular field of expertise in which a concrete software system is helping human activities. (compare *Policy domain*)

Proxy Structural design pattern: Provide a surrogate or placeholder for another object to control access to it.

Remote procedure call A request-response protocol that calls a procedure on a given network host with some input parameters, and returns the result to the calling host. (compare *Remote method invocation*)

Remote method invocation The object-oriented version of RPC: a request-response protocol for method invocation, but the input and output parameters may contain objects that migrate both their data state and their code. (compare *Remote procedure call*, see also *Serialization*)

Resolver An agent that may be able to return a live transport address to another agent, used when all addresses given previously fail.

Serialization The process of transforming an abstract data structure (such as a tree or a list) or a structure of objects into a bit-stream representation. The data structure can later be fully recovered (deserialized) from the stream. (see also *Remote method invocation*)

Service A service provided for agents and other services. A service is defined in terms of the operations that it supports.

Speech act theory A theory derived from the linguistic analysis of human communication. It is based on the idea that with language the speaker not only makes statements, but also performs actions. Speech acts can be put into the performative form, so called because saying the act makes it so (performs the act). “I hereby cancel the meeting we had scheduled for 3 PM” is a speech act: saying it cancels the meeting. “I hereby water the plants” is not: saying it does not water the plants. (see also *Communicative act*)

State Behavioral design pattern: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Stationary agent An agent that stays on the same agent platform throughout its full lifetime. (oppose *Mobile agent*)

Strategy Behavioral design pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template method Behavioral design pattern: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Transport A transport is a particular data delivery service supported by a given message transport service.

Transport description The address of a particular agent or service within the address space accessible by a given transport.

Transport message The object conveyed from agent to agent. It contains the transport description for the sender and receiver or receivers, together with a payload containing the message.

Visitor Behavioral design pattern: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

White pages service A service listing the current status of agents running on an agent platform, enabling their management through primitive operations.

Yellow pages service A service carrying thematically categorized, semantically rich information about agents and their whereabouts.

Chapter 9

List of Acronyms

ACC Agent Communication Channel

ACL Agent Communication Language

AMS Agent Management System

ANSI American National Standards Institute

AP Agent Platform

API Application Programming Interface

ARB Agent Resource Broker

AUML Agent UML

ATM Asynchronous Transfer Mode

CA Communicative Act, or Control Agent (context-dependent)

CCL Constraint Content Language

CORBA Common Object Request Broker Architecture

CPU Central Processor Unit

DF Directory Facilitator

FIPA Foundation for Intelligent Physical Agents

GoF Gang of Four

HAP Home Agent Platform

HTTP Hypertext Transfer Protocol

IDL Interface Definition Language

IIOP Internet Inter-Orb Protocol

IP Interaction Protocol, or Internet Protocol (context-dependent)

JVM Java Virtual Machine

KIF Knowledge Interchange Format

LAN Local Area Network

MA Monitoring Agent

MAF Mobile Agent Facility

MASIF Mobile Agent System Interoperability Facility

MTS Message Transport Service

NAT Network Address Translation

OA Ontology Agent

OO Object Orientation, Object Oriented

OOD Object Oriented Design

OOP Object Oriented Programming

RDF Resource Description Framework

RPC Remote Procedure Call

RMI Remote Method Invocation

SA Speech Act

SL Semantic Language

SSL	Secure Sockets Layer
Tcl	Tool Command Language
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
VM	Virtual Machine
WAP	Wireless Application Protocol
XML	Extensible Markup Language

Bibliography

[ACTS1999] ACTS (Advanced Communications Technologies and Services) Project InfoWin: *Agents Technology in Europe: ACTS Activities*, 1999. `http://www.infowin.org/`

[BB1997] Bigus, Joseph P. and Bigus, Jennifer: *Constructing Intelligent Agents with Java: A Programmer's Guide to Smarter Applications.* John Wiley & Sons, Inc., 1998.

[FIPA00001] Foundation for Intelligent Physical Agents: *FIPA Abstract Architecture Specification*, 2000. `http://www.fipa.org/specs/fipa00001/`

[FIPA00007] Foundation for Intelligent Physical Agents: *FIPA Content Languages Specification*, 2000. `http://www.fipa.org/specs/fipa00007/`

[FIPA00008] Foundation for Intelligent Physical Agents: *FIPA SL Content Language Specification*, 2000. `http://www.fipa.org/specs/fipa00008/`

[FIPA00009] Foundation for Intelligent Physical Agents: *FIPA CCL Content Language Specification*, 2000. `http://www.fipa.org/specs/fipa00009/`

[FIPA00010] Foundation for Intelligent Physical Agents: *FIPA KIF Content Language Specification*, 2000. `http://www.fipa.org/specs/fipa00010/`

[FIPA00011] Foundation for Intelligent Physical Agents: *FIPA RDF Content Language Specification*, 2000. `http://www.fipa.org/specs/fipa00011/`

[FIPA00014] Foundation for Intelligent Physical Agents: *FIPA Nomadic Application Support Specification*, 2000. `http://www.fipa.org/specs/fipa00014/`

[FIPA00023] Foundation for Intelligent Physical Agents: *FIPA Agent Management Specification*, 2000. `http://www.fipa.org/specs/fipa00023/`

[FIPA00025] Foundation for Intelligent Physical Agents: *FIPA Interaction Protocol Library Specification*, 2000. `http://www.fipa.org/specs/fipa00025/`

[FIPA00026] Foundation for Intelligent Physical Agents: *FIPA Request Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00026/`

[FIPA00027] Foundation for Intelligent Physical Agents: *FIPA Query Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00027/`

[FIPA00028] Foundation for Intelligent Physical Agents: *FIPA Request When Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00028/`

[FIPA00029] Foundation for Intelligent Physical Agents: *FIPA Contract Net Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00029/`

[FIPA00030] Foundation for Intelligent Physical Agents: *FIPA Iterated Contract Net Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00030/`

[FIPA00031] Foundation for Intelligent Physical Agents: *FIPA English Auction Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00031/`

[FIPA00032] Foundation for Intelligent Physical Agents: *FIPA Dutch Auction Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00032/`

[FIPA00033] Foundation for Intelligent Physical Agents: *FIPA Brokering Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00033/`

[FIPA00034] Foundation for Intelligent Physical Agents: *FIPA Recruiting Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00034/`

[FIPA00035] Foundation for Intelligent Physical Agents: *FIPA Subscribe Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00035/`

[FIPA00036] Foundation for Intelligent Physical Agents: *FIPA Propose Interaction Protocol Specification*, 2000. `http://www.fipa.org/specs/fipa00036/`

[FIPA00037] Foundation for Intelligent Physical Agents: *FIPA Communicative Act Library Specification*, 2000. `http://www.fipa.org/specs/fipa00037/`

[FIPA00061] Foundation for Intelligent Physical Agents: *FIPA ACL Message Structure Specification*, 2000. `http://www.fipa.org/specs/fipa00061/`

[FIPA00067] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Service Specification*, 2000. `http://www.fipa.org/specs/fipa00067/`

[FIPA00069] Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in Bit-Efficient Encoding Specification*, 2000. `http://www.fipa.org/specs/fipa00069/`

[FIPA00070] Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in String Specification*, 2000. `http://www.fipa.org/specs/fipa00070/`

[FIPA00071] Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in XML Specification*, 2000. `http://www.fipa.org/specs/fipa00071/`

[FIPA00075] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for IIOP Specification*, 2000. `http://www.fipa.org/specs/fipa00075/`

[FIPA00076] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for WAP Specification*, 2000. `http://www.fipa.org/specs/fipa00076/`

[FIPA00079] Foundation for Intelligent Physical Agents: *FIPA Agent Software Integration Specification*, 2000. `http://www.fipa.org/specs/fipa00079/`

[FIPA00080] Foundation for Intelligent Physical Agents: *FIPA Personal Travel Assistance Specification*, 2000. `http://www.fipa.org/specs/fipa00080/`

[FIPA00081] Foundation for Intelligent Physical Agents: *FIPA Audio-Visual Entertainment and Broadcasting Specification*, 2000. `http://www.fipa.org/specs/fipa00081/`

[FIPA00082] Foundation for Intelligent Physical Agents: *FIPA Network Management and Provisioning Specification*, 2000. `http://www.fipa.org/specs/fipa00082/`

[FIPA00083] Foundation for Intelligent Physical Agents: *FIPA Personal Assistant Specification*, 2000. `http://www.fipa.org/specs/fipa00083/`

[FIPA00084] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for HTTP Specification*, 2000. http://www.fipa.org/specs/fipa00084/

[FIPA00085] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Envelope Representation in XML Specification*, 2000. http://www.fipa.org/specs/fipa00085/

[FIPA00086] Foundation for Intelligent Physical Agents: *FIPA Ontology Service Specification*, 2000. http://www.fipa.org/specs/fipa00086/

[FIPA00087] Foundation for Intelligent Physical Agents: *FIPA Agent Management Support for Mobility Specification*, 2000. http://www.fipa.org/specs/fipa00087/

[FIPA00088] Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Envelope Representation in Bit-Efficient Encoding Specification*, 2000. http://www.fipa.org/specs/fipa00088/

[FIPA00089] Foundation for Intelligent Physical Agents: *FIPA Domains and Policies Specification*, 2000. http://www.fipa.org/specs/fipa00089/

[GoF1994] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Hol1999] Holub, Allen I.: *Programming Java Threads in the Real World*, Parts 1-9. JavaWorld, 1999.
http://www.javaworld.com/jw-09-1998/jw-09-threads_p.html,
http://www.javaworld.com/jw-10-1998/jw-10-toolbox_p.html,
http://www.javaworld.com/jw-11-1998/jw-11-toolbox_p.html,
http://www.javaworld.com/jw-12-1998/jw-12-toolbox_p.html,
http://www.javaworld.com/jw-02-1999/jw-02-toolbox_p.html,
http://www.javaworld.com/jw-03-1999/jw-03-toolbox_p.html,
http://www.javaworld.com/jw-04-1999/jw-04-toolbox_p.html,
http://www.javaworld.com/jw-05-1999/jw-05-toolbox_p.html,
http://www.javaworld.com/jw-06-1999/jw-06-toolbox_p.html.

[Hol2000] Holub, Allen I.: *Building User Interfaces for Object-Oriented Systems*, Parts 1-6. JavaWorld, 2000.
http://www.javaworld.com/jw-07-1999/jw-07-toolbox_p.html,
http://www.javaworld.com/jw-09-1999/jw-09-toolbox_p.html,
http://www.javaworld.com/jw-10-1999/jw-10-toolbox_p.html,
http://www.javaworld.com/jw-12-1999/jw-12-toolbox_p.html,
http://www.javaworld.com/jw-01-2000/jw-01-toolbox_p.html,
http://www.javaworld.com/jw-03-2000/jw-03-toolbox_p.html.

[IKV2001] IKV++: Grasshopper homepage, 2001. http://www.grasshopper.de/

[NN2001] Nortel Networks: FIPA-OS homepage, 2001. http://fipa-os.sourceforge.net/

[OMG2000] Object Management Group: *Mobile Agent Facility Specification*, 2000.
http://cgi.omg.org/cgi-bin/doc?formal/00-01-02.pdf,
http://cgi.omg.org/cgi-bin/doc?formal/00-06-40.txt

[W3C1998] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 Specification*, 1998. http://www.w3.org/TR/REC-xml

[WB1996] Weiser, Mark and Brown, John Seely: *Designing Calm Computing*, 1996.
http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm