preammble delim_0"
$dotfill$"$delim_1$"
$dotfill$"$delim_2$"
$dotfill$"$postamble$

Contents

X-XXXX-XXXX-X/XX/$X.XX+$X.XX

1

Multiagent Systems for Internet Applications

Michael N. Huhns and Larry M. Stephens, *University of South Carolina*

CONTENTS

Abstract The World-Wide Web is evolving from an environment for *people* to obtain information to an environment for *computers* to accomplish tasks on behalf of people. The resultant Semantic Web will be computer-friendly through the introduction of standardized Web services. This chapter describes how Web services will become more agent-like, and how the envisioned capabilities and uses for the "Semantic Web" will require implementations in the form of multiagent systems. It also describes how to construct multiagent systems that implement Web-based software applications.

1.1 Introduction

Web services are the most important Internet technology since the browser. They embody com-

X-XXXX-XXXX-X/XX/$X.XX+$X.XX

putational functionality that corporations and organizations are making available to clients over the Internet. Web services have many of the same characteristics and requirements as simple software agents, and because of the kinds of demands and expectations that people have for their future uses, it seems apparent that Web services will soon have to be more like complete software agents. Hence, the best way to construct Web services will be in terms of multiagent systems. In this chapter, we describe the essential characteristics and features of agents and multiagent systems, how to build them, and then how to apply them to Web services.

The environment for Web services, and computing in general, is fast becoming ubiquitous and pervasive. It is ubiquitous because computing power and access to the Internet is being made available everywhere; it is pervasive because computing is being embedded in the very fabric of our environment. Xerox Corp. has coined the phrase "smart matter" to capture the idea of computations occurring within formerly passive objects and substances. For example, our houses, our furniture, and our clothes will contain computers that will enable our surroundings to adapt to our preferences and needs. New visions of interactivity portend that scientific, commercial, educational, and industrial enterprises will be linked, and human spheres previously untouched by computing and information technology, such as our personal, recreational, and community life, will be affected.

This chapter suggests a multiagent-based architecture for all of the different devices, components, and computers to understand each other, so that they will be able to work together effectively and efficiently. The architecture that we describe is becoming canonical. Agents are used to represent users, resources, middleware, security, execution engines, ontologies, and brokering, as depicted in Figure 1.1. As the technology advances, we can expect such specialized agents to be used as standardized building blocks for information systems and Web services.

Figure 1.1: The agents in a multiagent Internet application first determine a users request (the responsibility of the user agent) and then satisfy it by managing its processing. Under the control of the execution agent, the request might be sent to one or more databases or Web sites, which are managed by resource agents

Multiagent systems are applicable not only to the diverse information soon to be available locally over household, automobile, and environment networks, but also to the huge amount of information available globally over the World-Wide Web being made available as Web services.

Organizations are beginning to represent their attributes, capabilities, and products on the Internet as services that can be invoked by potential clients. By invoking each other's functionalities, the Web services from different organizations can be combined in novel and unplanned ways to yield larger, more comprehensive functionalities with much greater value than the individual component services could provide.

Web services are XML-based, work through firewalls, are lightweight, and are supported by

all software companies. They are a key component of Microsoft's .NET initiative, and are deemed essential to the business directions being taken by IBM, Sun, and SAP. Web services are also central to the envisioned *Semantic Web* [Berners-Lee et al., 2001], which is what the World Wide Web is evolving into. But the Semantic Web is also seen as a friendly environment for software agents, which will add capabilities and functionality to the Web. What will be the relationship between multiagent systems and Web services?

1.1.1 Benefits of an Approach Based on Multiagent Systems

Multiagent systems can form the fundamental building blocks for not only Web services, but also software systems in general, even if the software systems do not themselves require any agent-like behaviors [Jennings, 2000]. When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics:

- Agent-based modules, because they are active, more closely represent real-world things, which are the subjects of many applications.
- Modules can hold beliefs about the world, especially about themselves and others; if their behavior is consistent with their beliefs, then their behavior will be more predictable and reliable.
- Modules can negotiate with each other, enter into social commitments to collaborate, and can change their mind about their results.

The benefits of building software out of agents are [Coelho et al., 1994; Huhns, 2001]

1. Agents enable dynamic composibility, where the components of a system can be unknown until runtime
2. Agents allow interaction abstractions, where interactions can be unknown until runtime
3. Because agents can be added to a system one-at-a-time, software can continue to be customized over its lifetime, even potentially by end-users
4. Because agents can represent multiple viewpoints and can use different decision procedures, they can produce more robust systems. The essence of multiple viewpoints and multiple decision procedures is redundancy, which is the basis for error detection and correction.

An agent-based system can cope with a growing application domain by increasing the number of agents, each agent's capability, the computational resources available to each agent, or the infrastructure services needed by the agents to make them more productive. That is, either the agents or their interactions can be enhanced.

As described in section 1.2.3, agents share many functional characteristics with Web services. We show how to build agents that implement Web services and achieve the benefits listed above. In addition, we describe how personal agents can aid users in finding information on the Web, keeping data current (such as trends in stocks and bonds), and alerts to problems and opportunities (such as bargains on e-Bay). We next survey how agent technology has progressed since its inception, and indicate where it is heading.

1.1.2 Brief History of Multiagent Systems

Agents and agency have been the object of study for centuries. They were first considered in the philosophy of action and ethics. In this century, with the rise of psychology as a discipline, human agency has been studied intensively.

Within the five decades of artificial intelligence (AI), computational agents have been an active topic of exploration. The AI work in its earliest stages investigated agents explicitly, albeit with simple models. Motivated by results from psychology, advances in mathematical logic, and concepts such as the Turing Test, early researchers in AI concentrated on building individual intelligent systems or one of their components, such as reasoning mechanisms or learning techniques. This characterized the first 25 years of AI. However, the fact that some problems, such as sensing a domain, are inherently distributed coupled with advances in distributed computing, led several researchers to investigate distributed problem solving and distributed artificial intelligence (DAI). Progress and directions in these areas became informed more by sociology and economics than by psychology.

From the late seventies onward, the resultant DAI research community [Huhns, 1987; Bond and Gasser, 1988; Gasser and Huhns, 1989] concerned itself with agents as computational entities that interacted with each other to solve various kinds of distributed problems. To this end, whereas AI at large borrowed abstractions, such as beliefs and intentions, from psychology, the DAI community borrowed abstractions and insights from sociology, organizational theory, economics, and the philosophies of language and linguistics. These abstractions complement rather than oppose the psychological abstractions, but—being about groups of agents—are fundamentally better suited to large distributed applications.

With the expansion of the Internet and the web in the nineties, we witnessed the emergence of software agents, geared to open information environments. These agents perform tasks on behalf of a user, or serve as nodes—brokers or information sources—in the global information system. Although software agents of this variety do not involve specially innovative techniques, it is their synthesis of existing techniques and their suitability to their application that makes them powerful and popular. Thus, much of the attention they have received is well-deserved.

1.2 Infrastructure and Context for Web-Based Agents

1.2.1 The Semantic Web

The World-Wide Web was designed for humans. It is based on a simple concept: information consists of pages of text and graphics that contain links, and each link leads to another page of information, with all of the pages meant to be viewed by a person. The constructs used to describe and encode a page, the Hypertext Markup Language (html), describe the appearance of the page, but not its contents. Software agents do not care about appearance, but rather the contents. The Semantic Web will add Web services that are envisioned to be

- Understandable to computers
- Adaptable and personalized to clients
- Dynamically composable by clients
- Suitable for robust transaction processing by virtual enterprises.

There are, however, some agents that make use of the Web as it is now. A typical kind of such agent is a *shopbot*, an agent that visits the on-line catalogs of retailers and returns the prices being charged for an item that a user might want to buy. The shopbots operate by a form of "screen-scraping," in which they download catalog pages and search for the name of the item of interest, and then the nearest set of characters that has a dollar-sign, which presumably is the item's price. The shopbots also might submit the same forms that a human might submit and then parse the returned pages that merchants expect are being viewed by humans. The Semantic Web will make the Web

more accessible to agents by making use of semantic constructs, such as ontologies represented in OWL, RDF, and XML, so that agents can *understand* what is on a page.

1.2.2 Standards and Protocols for Web Services

A Web service is functionality that can be engaged over the Web. Web services are currently based on the triad of functionalities depicted in Figure 1.2. The architecture for Web services is founded on principles and standards for connection, communication, description, and discovery. For providers and requestors of services to be connected and exchange information, there must be a common language. This is provided by the eXtensible Modeling Language (XML). Short descriptions of the current protocols for Web service connection, description, and discovery are in the following paragraphs; more complete descriptions are found elsewhere in this book.

Figure 1.2: The general architectural model for Web services. Web services rely on the functionalities of publish, find, and bind. The equivalent agent-based functionalities are shown in parentheses, where all interactions among the agents are via an agent-communication language (ACL). Any agent might serve as a broker. Also, the service provider's capabilities might be found without using a broker

A common protocol is required for systems to communicate with each other, so that they can request services, such as to schedule appointments, order parts, and deliver information. This is provided by the Simple Object Access Protocol (SOAP) [Box et al., 2000].

The services must be described in a machine-readable form, where the names of functions, their required parameters, and their results can be specified. This is provided by the Web Services Description Language (WSDL).

Finally, clients—users and businesses—need a way to find the services they need. This is provided by Universal Description, Discovery, and Integration (UDDI), which specifies a registry or "yellow pages" of services.

Besides standards for XML, SOAP, WSDL, and UDDI, there is a need for broad agreement on the semantics of specific domains. This is provided by the Resource Description Framework (RDF) [Decker et al., 2000a,b], the OWL Web Ontology Language [Smith et al., 2003], and, more generally, ontologies [Heflin and Hendler, 2000].

1.2.3 Directory Services

The purpose of a directory service is for components and participants to be able to locate each other, where the components and participants might be applications, agents, Web service providers, Web service requestors, people, objects, and procedures. There are two general types of directories, determined by how entries are found in the directory: (1) name servers or *white pages*, where entries are found by their name, and (2) *yellow pages*, where entries are found by their characteristics and capabilities.

The implementation of a basic directory is a simple database-like mechanism that allows participants to insert descriptions of the services they offer and query for services offered by other participants. A more advanced directory might be more active than others, in that it might provide not only a search service, but also a brokering or facilitating service. For example, a participant might request a brokerage service to recruit one or more agents that can answer a query. The brokerage service would use knowledge about the requirements and capabilities of registered service providers to determine the appropriate providers to which to forward a query. It would then send the query to those providers, relay their answers back to the original requestor, and learn about the properties of the responses it passes on (e.g., the brokerage service might determine that advertised results from provider X are incomplete, and so seek out a substitute for provider X).

UDDI is itself a Web service that is based on XML and SOAP. It provides both a white-pages and a yellow-pages service, but not a brokering or facilitating service.

The DARPA DAML effort has also specified a syntax and semantics for describing services, known as DAML-S (now migrating to OWL-S http://www.daml.org/services). This service description provides

- Declarative ads for properties and capabilities, used for discovery
- Declarative APIs, used for execution
- Declarative prerequisites and consequences, used for composition and interoperation.

1.3 Agent Implementations of Web Services

Typical agent architectures have many of the same features as Web services. Agent architectures provide yellow-page and white-page directories, where agents advertise their distinct functionalities and where other agents search to locate the agents in order to request those functionalities. However, agents extend Web services in several important ways:

- A Web service knows only about itself, but not about its users/clients/customers. Agents are often self-aware at a metalevel, and through learning and model building gain awareness of other agents and their capabilities as interactions among the agents occur. This is important, because without such awareness a Web service would be unable to take advantage of new capabilities in its environment, and could not customize its service to a client, such as by providing improved services to repeat customers.
- Web services, unlike agents, are not designed to use and reconcile ontologies. If the client and provider of the service happen to use different ontologies, then the result of invoking the Web service would be incomprehensible to the client.
- Agents are inherently communicative, whereas Web services are passive until invoked. Agents can provide alerts and updates when new information becomes available. Current standards and protocols make no provision for even subscribing to a service to receive periodic updates.

- A Web service, as currently defined and used, is not autonomous. Autonomy is a characteristic of agents, and it is also a characteristic of many envisioned Internet-based applications. Among agents, autonomy generally refers to social autonomy, where an agent is aware of its colleagues and is sociable, but nevertheless exercises its independence in certain circumstances. Autonomy is in natural tension with coordination or with the higher-level notion of a commitment. To be coordinated with other agents or to keep its commitments, an agent must relinquish some of its autonomy. However, an agent that is sociable and responsible can still be autonomous. It would attempt to coordinate with others where appropriate and to keep its commitments as much as possible, but it would exercise its autonomy in entering into those commitments in the first place.
- Agents are cooperative, and by forming teams and coalitions can provide higher-level and more comprehensive services. Current standards for Web services do not provide for composing functionalities.

1.4 Building Web-Service Agents

1.4.1 Agent Types

To better communicate some of the most popular agent architectures, this chapter uses UML diagrams to guide an implementer's design. However, before we describe these diagrams, we need to review some of the basic features of agents. Consider the architecture in Figure 1.3 for a simple agent interacting with an information environment, which might be the Internet, an intranet, or a virtual private network (VPN). The agent senses its environment, uses what it senses to decide upon an action, and then performs the action through its effectors. Sensory input can include received messages, and the action can be the sending of messages.

Figure 1.3: A simple interaction between an agent and its information environment (adapted from [Russell and Norvig, 2003]). "What action I should do now" depends on the agent's goals and perhaps ethical considerations, as noted in Figure 1.6

To construct an agent, we need a more detailed understanding of how it functions. In particular, if we are to construct one using conventional object-oriented analysis and design techniques, we should know in what ways an agent is more than just a simple object. Agent features relevant

to implementation are unique identity, proactivity, persistence, autonomy, and sociability [Weiß, 1999].

An agent inherits its *unique identity* simply by being an object. To be *proactive*, an agent must be an object with an internal event loop, such as any object in a derivation of the Java thread class would have. Here is simple pseudocode for a typical event loop, where events result from sensing an environment:

```
Environment e;
RuleSet r;
while (true) {
   state = senseEnvironment(e);
   a = chooseAction(state, r);
   e.applyAction(a);
}
```

This is an infinite loop, which also provides the agent with *persistence*. Ephemeral agents would find it difficult to converse, making them, by necessity, asocial. Additionally, persistence makes it worthwhile for agents to learn about and model each other. To benefit from such modeling, they must be able to distinguish one agent from another, hence the need for agents to have unique identities.

Agent *autonomy* is akin to human free will and enables an agent to choose its own actions. For an agent constructed as an object with methods, autonomy can be implemented by declaring all of the methods private. With this restriction, only the agent can invoke its own methods, under its own control, and no external object can force the agent to do anything it does not intend to do. Other objects can communicate with the agent by creating events or artifacts, especially messages, in the environment that the agent can perceive and react to.

Enabling an agent to converse with other agents achieves *sociability*. The conversations, normally conducted by sending and receiving messages, provide opportunities for agents to coordinate their activities and cooperate, if so inclined. Further sociability can be achieved by generalizing the input class of objects an agent might perceive to include a piece of sensory information and an event defined by the agent. Events serving as inputs are simply "reminders" that the agent sets for itself. For example, an agent that wants to wait five minutes for a reply would set an event to fire after five minutes. If the reply arrives before the event, the agent can disable the event. If it receives the event, then it knows it did not receive the reply in time and can proceed accordingly.

The UML diagrams in Figures 1.4 and 1.5 can help in understanding or constructing a software agent. These diagrams do not address every functional aspect of an agent's architecture. Instead they provide a general framework for implementing traditional agent architectures [Weiß, 1999].

Reactive Agents

A reactive agent is the simplest kind to build, since it does not maintain information about the state of its environment but simply reacts to current perceptions. Our design for such an agent, shown in Figure 1.4, is fairly intuitive, encapsulating a collection of behaviors, sometimes known as plans, and the means for selecting an appropriate one. A collection of objects, in the object-oriented sense, lets a developer add and remove behaviors without having to modify the action selection code, since an iterator can be used to traverse the list of behaviors. Each behavior fires when it matches the environment, and each can `inhibit` other behaviors. Our action-selection loop is not as efficient as it could be, since `getAction` operates in $O(n)$ time (where n is the number of behaviors). A better implementation could lower the computation time to $O(logn)$ using decision trees, or $O(1)$ using hardware or parallel processing. The developer is responsible for ensuring that at least one behavior will match for every environment. This can be achieved by defining a default behavior that matches all inputs but is inhibited by all other behaviors that match.

BDI Agents

A belief-desire-intention (BDI) architecture includes and uses an explicit representation for an agent's beliefs (state), desires (goals), and intentions (plans). The beliefs include self-understanding (I believe I can perform Task-A), beliefs about the capabilities of other agents (I believe Agent-B can perform Task-B), and beliefs about the environment (based on my sensors, I believe I am three

Figure 1.4: Diagram of a simple reactive architecture for an agent. The agent's `run()` method executes the action specified by the current behavior and state

feet from the wall). The intentions persist until accomplished or are determined to be unachievable. Representative BDI systems—the Procedural Reasoning System (PRS) and JAM—all define a new programming language and implement an interpreter for it. The advantage of this approach is that the interpreter can stop the program at any time, save its state, and execute some other intention if it needs to. The disadvantage is that the interpreter—not an intention—runs the system; the current intention may no longer be applicable if the environment changes.

The BDI architecture shown in Figure 1.5 eliminates this problem. It uses a voluntary multitasking method instead, whereby the environment thread constantly checks to make sure the current intention is applicable. If not, the agent executes `stopCurrentIntention()`, which will call the intention's `stopExecuting()` method. Thus, the intention is responsible for stopping itself and cleaning up. By giving each intention this capability, we eliminate the possibility of a deadlock resulting from the intentions having some resource reserved when it was stopped. The following pseudocode illustrates the two main loops, one for each thread, of the BDI architecture. The variables `a`, `B`, `D`, and `I` represent the agent and its beliefs, desires, and intentions.

The agents run method consists of finding the best applicable intention and executing it to completion. If executing the intention returns true, the meaning is that the desire was achieved, so the desire is removed from the desire set. If the environment thread finds that an executing plan is no longer applicable and calls for a stop, the intention will promptly return from its `execute()` call with a false. Notice that the environment thread modifies the agents set of beliefs. The belief set

Figure 1.5: Diagram of a belief-desire-intention architecture for an agent

needs to synchronize these changes with any changes that the intentions make to the set of beliefs.

```
Agent::run() {
  Environment e;
  e.run(); //start environment in its own thread
  while (true) {
    I = a.getBestIntention();
    if (I.execute(a)) // true if intention was achieved
      a.D.remove(I.goal); // I.goal is a desire
  }
}

Environment::run(){
  while (true) {
    a.B.incorporateNewObservations(e.getInput(a));
    if (! a.currentIntentionIsOK())
      a.stopCurrentIntention();
    sleep(someShortTime);
  }
}
```

Finally, the environment threads sleep time can be modified, depending on the systems real-time requirements. If we do not need the agent to change intentions rapidly when the environment changes, the thread can sleep longer. Otherwise, a short sleep will make the agent check the environment more frequently, using more computational resources. A more efficient call-back mechanism could easily replace the current run method if the agents input mechanism supported it.

Layered Architectures

Other common architectures for software agents consist of layers of capabilities, where the higher layers perform higher levels of reasoning. For example, Figure 1.6 shows the architecture of an agent that has a philosophical and ethical basis for choosing its actions. The layers typically interact in one of three ways: (case 1) inputs are given to all of the layers at the same time, (case 2) inputs are given to the highest layer first for deliberation, and then its guidance is propagated downward through each of the lower layers until the lowest layer performs the ultimate action, and (case 3) inputs are given to the lowest layer first, which provides a list of possible actions that are successively filtered by each of the higher layers until a final action remains.

The lowest level of the architecture enables an agent to react to immediate events [Müller et al., 1994]. The middle layers are concerned with an agent's interactions with others [Castelfranchi, 1998; Castelfranchi et al., 2000; Rao and Georgeff, 1991; Cohen and Levesque, 1990], while the highest level enables the agent to consider the long-term effects of its behavior on the rest of its society [Mohamed and Huhns, 2001]. Agents are typically constructed starting at the bottom of this architecture, with increasingly more abstract reasoning abilities layered on top.

Awareness of other agents and of one's own role in a society, which are implicit at the social commitment level and above, can enable agents to behave coherently [Gasser, 1991]. Tambe et al. [Tambe et al., 2000] have shown how a team of agents flying helicopters will continue to function as a coherent team after their leader has crashed, because another agent will assume the leadership role. More precisely, the agents will adjust their individual intentions in order to fulfill the commitments made by the team.

Figure 1.6: Architecture for a philosophical agent. The architecture defines layers of deliberation for enabling an agent to behave appropriately in a society of agents

Behaviors and Activity Management

Most popular agent architectures, including the two we diagrammed, include a set of behaviors and a method for scheduling them. A behavior is distinguished from an action in that an action is an atomic event, while a behavior can span a longer period of time. In multiagent systems, we can also distinguish between physical behaviors that generate actions, and conversations between agents. We can consider behaviors and conversations to be classes inheriting from an abstract activity class. We can then define an activity manager responsible for scheduling activities.

This general activity manager design lends itself to the implementation of many popular agent architectures while maintaining the proper encapsulation and decomposability required in good object-oriented programming. Specifically, activity is an abstract class that defines the interface to be implemented by all behaviors and conversations. The behavior class can implement any helper functions needed in the particular domain (for example, subroutines for triangulating the agents position). The conversation class can implement a finite-state machine for use by the particular conversations. For example, by simply filling in the appropriate states and adding functions to handle the transitions, an agent can define a contracting protocol as a class that inherits from conversation. Details of how this is done depend on how the conversation class implements a finite-state machine, which varies depending on the systems real-time requirements.

Defining each activity as its own independent object and implementing a separate activity manager has several advantages. The most important is the separation between domain and control knowledge, a feature first popularized by blackboard systems. The activities will embody all the knowledge about the particular domain the agent inhabits, while the activity manager embodies knowledge about the deadlines and other scheduling constraints the agent faces. By implementing each activity as a separate class, we compel the programmer to separate the agents abilities into encapsulated objects that other activities can then reuse. The activity hierarchy forces all activities to implement a minimal interface, which also facilitates reuse. Finally, placing the activities within the hierarchy provides many opportunities for reuse through inheritance. For example, the conversation class can implement a general lost-message error-handling procedure that all the conversations can use.

Architectural Support

Figures 1.4 and 1.5 provide general guidelines for implementing agent architectures using an object-oriented language. As agents become more complex, developers will likely have to expand upon our techniques. We believe these guidelines are general enough that it will not be necessary to rewrite the entire agent from scratch when adding new functionality.

Of course, a complete agent-based system requires an infrastructure to provide for message transport, directory services, and event notification and delivery. These are usually provided as operating system services or, increasingly, in an agent-friendly form by higher level distributed protocols such as Jini (http://www.sun.com/jini/), Bluetooth (http://www.bluetooth.com), and FIPA's (the Foundation of Intelligent Physical Agents at http://www.fipa.org/) emerging standards. See http://www.multiagent.com/, a site maintained by José Vidal, for additional information about agent tools and architectures.

The canonical multiagent architecture, shown in Figure 1.1, is suitable for many applications. The architecture incorporates a variety of resource agents that represent databases, Web sites, sensors, and file systems. Specifications for the behaviors of two types of resource agents are shown in Figures 1.7 and 1.8. Figure 1.7 contains a procedural specification for the behavior of a resource (wrapper) agent that makes a database system active and accessible to other agents. In a supply-chain management scenario, the database agent might represent a supplier's order-processing system; customers could send orders to the agent (inform) and then issue queries for status and billing information. A procedural specification for the behavior of an Internet agent that actively monitors a Web site for new or updated information is shown in Figure 1.8. An agent that implements this procedure could be used by customers looking for updated pricing or product information.

1.4.2 Agent Communication Languages

Agents representing different users might collaborate in finding and fusing information, but compete for goods and resources. Similarly, service agents may collaborate or compete with user, resource, and other service agents. Whether they are collaborators or competitors, the agents must interact purposefully with each other. Most purposeful interactions—whether to inform, query, or deceive—require the agents to talk to one another, and talking intelligibly requires a mutually understood language.

Speech Acts

Speech acts have to do with communication—they have nothing to do with speaking as such, except that human communication often involves speech. Speech act theory was invented in the fifties and sixties to help understand human language [Austin, 1962]. The idea was that with language you not only make statements, but also perform actions. For example, when you request something, you do not just report on a request; you actually cause the request. When a justice of the peace declares a couple man and wife, she is not reporting on their marital status, but changing it.

The stylized syntactic form for speech acts that begins "I hereby request ..." or "I hereby declare ..." is called a *performative*. With a performative, literally, saying it makes it so. Verbs that cannot be put in this form are not speech acts. For example, "solve" is not a performative, because "I hereby solve this problem" is not sufficient.

Several thousand verbs in English correspond to performatives. Many classifications have been suggested for these, but the following are sufficient for most computing purposes:

- assertives (informing),
- directives (requesting or querying),
- commissives (promising),

Figure 1.7: Activity diagram showing the procedural behavior of a resource agent that makes a database system active and accessible to other agents

Figure 1.8: Activity diagram showing the procedural behavior of an Internet agent that actively monitors a Web site for new or updated information

- prohibitives,
- declaratives (causing events in themselves as, for example, the justice of the peace does in a marriage ceremony), and
- expressives (expressing emotions).

In natural language, it is not easy to determine what speech act is being performed. In artificial languages, we do not have this problem. However, the meanings of speech acts depend on what the agents believe, intend, and know how to perform and on the society in which they reside. It is difficult to characterize meaning because all of these things are themselves difficult.

Common Language

Agent projects investigated languages for many years. Early on, agents were local to each project, and their languages were mostly idiosyncratic. The challenge now is to have any agent talk to any other agent, shich suggests a common language; ideally, all the agents that implement the (same) language will be mutually intelligible.

Such a common language needs an unambiguous syntax, so the agents can all parse sentences the same way. It should have a well-defined semantics or meaning, so the agents can all understand sentences the same way. It should be well known, so different designers can implement it and so it has a chance of encountering another agent who knows the same language. And it should have the expressive power to communicate the kinds of things agents may need to say to one another.

So what language should you give or teach your agent so that it will understand and be understood? The current popular choice is being administered by the Foundation for Intelligent Agents (FIPA), at http://www.fipa.org. The FIPA ACL separates the domain-dependent part of a communication—the content—from the domain-independent part—the packaging—and then provides a standard for the domain-independent part.

FIPA specifies just six performatives, but they can be composed to enable agents to express more complex beliefs and expectations. For example, an agent can request to be informed about one of several alternatives. The performatives deal explicitly with actions, so requests are for communicative actions to be done by the message recipient.

The FIPA specification comes with a formal semantics, and it guarantees that there is only one way to interpret an agent's communications. Without this guarantee, agents (and their designers) would have to choose among several alternatives, leading to potential misunderstandings and unnecessary work.

1.4.3 Knowledge and Ontologies for Agents

An ontology is a computational model of some portion of the world. It is often captured in some form of a semantic network—a graph whose nodes are concepts or individual objects and whose arcs represent relationships or associations among the concepts. This network is augmented by properties and attributes, constraints, functions, and rules that govern the behavior of the concepts.

Formally, an ontology is an agreement about a shared conceptualization, which includes frameworks for modeling domain knowledge and agreements about the representation of particular domain theories. Definitions associate the names of entities in a universe of discourse (for example, classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these names.

For information systems, or for the Internet, ontologies can be used to organize keywords and database concepts by capturing the semantic relationships among the keywords or among the tables and fields in a database. The semantic relationships give users an abstract view of an information space for their domain of interest.

Figure 1.9: Communication between agents sharing a travel ontology

A Shared Virtual World

How can such an ontology help our software agents? It can provide a shared virtual world in which each agent can ground its beliefs and actions. When we talk with our travel agent, we rely on the fact that we all live in the same physical world containing planes, trains, and automobiles. We know, for example, that a 777 is a type of airliner that can carry us to our destination.

When our agents talk, the only world they share is one consisting of bits and bytes—which does not allow for a very interesting discussion! An ontology gives the agents a richer and more useful domain of discourse.

The previous section described FIPA, which specifies the syntax but not the semantics of the messages that agents can exchange. It also allows the agents to state which ontology they are presuming as the basis for their messages.

Suppose two agents have access to an ontology for travel, with concepts such as airplanes and destinations, and suppose the first agent tells the second about a flight on a 777. Suppose further that the concept "777" is not part of the travel ontology. How could the second agent understand? The first agent could explain that a 777 is a kind of airplane, which is a concept in the travel ontology. The second agent would then know the general characteristics of a 777. This communication is illustrated in Figure 1.4.3.

Relationships Represented

Most ontologies represent and support relationships among classes of meaning. Among the most important of these relationships are:

- *Generalization and inheritance*, which are abstractions for sharing similarities among classes while preserving their differences. Generalization is the relationship between a class and one or more refined versions of it. Each subclass inherits the features of its superclass, adding other features of its own. Generalization and inheritance are transitive across an arbitrary number of levels. They are also antisymmetric.

- *Aggregation*, the part-whole or part-of relationship, in which classes representing the components of something are associated with the class representing the entire assembly. Aggregation is also transitive, as well as antisymmetric. Some of the properties of the assembly class propagate to the component classes.
- *Instantiation*, which is the relationship between a class and each of the individuals that constitute it.

Some of the other relationships that occur frequently in ontologies are *owns*, *causes*, and *contains*. Causes and contains are transitive and antisymmetric; owns propagates over aggregation, because when you own something, you also own all of its parts.

1.4.4 Reasoning Systems

A simple and convenient means to incorporate a reasoning capability into an Internet software agent is via a rule-execution engine, such as JESS [Friedman-Hill, 2003]. With JESS, knowledge is supplied in the form of declarative rules. There can be many or only a few rules, and Jess will continually apply them to data in the form of a knowledge base. Typically the rules represent the heuristic knowledge of a human expert in some domain, and the knowledge base represents the state of an evolving situation.

An example rule in JESS is

```
(defrule recognize-airliner
  "If an object ?X is a plane and carries passengers,
   then assert that ?X is an airliner."
  (isA ?X plane)
  (carries ?X passengers)
  =>
  (assert (isA ?X airliner)))
```

The associated knowledge base might contain facts about an airline company concerning their equipment and its characteriztics, such as

```
(assert (isA 777-N9682 plane))
(assert (carries 777-N9682 passengers))
```

Many of the common agent development environments, such as JADE [Bellifemine and Trucco, 2003], ZEUS [Nwana et al., 1999], and FIPA-OS [Nortel Networks, 2003], include facilities for incorporating JESS into the agents a developer is constructing.

1.4.5 Cooperation

The most widely used means by which agents arrange to cooperate is the contract-net protocol. This interaction protocol allows an initiating agent to solicit proposals from other agents by sending a *Call for Proposals*, evaluating their proposals, and then accepting the preferred one (or even rejecting all of them). Any agent can initiate the protocol, so it can be applied recursively.

The initiator sends a message with a `CFP` speech act that specifies the action to be performed and, if needed, conditions upon its execution. The responders can reply by sending a `PROPOSE` message that includes any preconditions for their action, such as their cost or schedule. Alternatively, responders may send a `REFUSE` message to indicate their disinterest or a `NOT-UNDERSTOOD` message to indicate a communication problem. The initiator then evaluates the received proposals and sends an `ACCEPT-PROPOSAL` message to the agents whose proposal will be accepted and a `REJECT-PROPOSAL` message to the others. Once the chosen responders have completed their

task, they respond with an `INFORM` of the result of the action or with a `FAILURE` if anything went wrong.

1.5 Composing Cooperative Web Services

Imagine that a merchant would like to enable a customer to be able to track the shipping of a sold item. Currently, the best the merchant can do is to point the customer to the shipper's Web site, and the customer can then go there to check on delivery status. If the merchant could compose its own production notification system with the shipper's Web services, the result would be a customized delivery notification service by which the customer—or the customer's agents—could find the status of a purchase in real time.

As Web uses (and thus Web interactions) become more complex, it will be increasingly difficult for one server to provide a total solution and increasingly difficult for one client to integrate solutions from many servers. Web services currently involve a single client accessing a single server, but soon applications will demand federated servers with multiple clients sharing results. Cooperative peer-to-peer solutions will have to be managed, and this is an area where agents have excelled. In doing so, agents can balance cooperation with the interests of their owner.

1.6 Conclusion

Web services are extremely flexible, and a major advantage is that a developer of Web services does not have to know who or what will be using the services being provided. They can be used to tie together the internal information systems of a single company or the interoperational systems of virtual enterprises. But how Web services tie the systems together will be based on technologies being developed for multiagent systems. The result will be a Semantic Web that enables work to get done and better decisions to be made.

Acknowledgement. The US National Science Foundation supported this work under grant number IIS-0083362.

References

John L. Austin. *How to Do Things with Words*. Clarendon Press, Oxford, 1962.

Fabio Bellifemine and Tiziana Trucco. Java agent development framework, 2003. http://sharon.cselt.it/projects/jade/.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5): 34–43, 2001.

Alan Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000. www.w3.org/TR/SOAP.

Cristiano Castelfranchi. Modelling social action for AI agents. *Artificial Intelligence*, 103:157–182, 1998.

Cristiano Castelfranchi, Frank Dignum, Catholyn M. Jonker, and Jan Treur. Deliberate normative agents: Principles and architecture. In Nicholas R. Jennings and Yves Lesperance, editors, *Intelligent Agents VI: Agent Theories, Architectures, and Languages (ATAL-99)*, volume 1757, pages 364–378, Berlin, 2000. Springer-Verlag.

Helder Coelho, Luis Antunes, and Luis Moniz. On agent design rationale. In *Proceedings of the XI Simposio Brasileiro de Inteligencia Artificial (SBIA)*, pages 43–58, Fortaleza, Brazil, 1994.

Philip R. Cohen and Hector J. Levesque. Persistence, intention, and commitment. In Philip Cohen, Jerry Morgan, and Martha Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, 1990.

Stefan Decker, Sergey Melnik, Frank van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of XML and RDF. *IEEE Internet Computing*, 4(5):63–74, September 2000a.

Stefan Decker, Prasenjit Mitra, and Sergey Melnik. Framework for the semantic web: An RDF tutorial. *IEEE Internet Computing*, 4(6):68–73, November 2000b.

Ernest J. Friedman-Hill. Jess, the Java expert system shell, 2003. http://herzberg.ca.sandia.gov/jess.

Les Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.

Les Gasser and Michael N. Huhns, editors. *Distributed Artificial Intelligence, Volume II*. Morgan Kaufmann, London, 1989.

Jeff Heflin and James A. Hendler. Dynamic ontologies on the Web. In *Proceedings of American Association for Artificial Intelligence Conference (AAAI)*, pages 443–449, Menlo Park, CA, 2000. AAAI Press.

Michael N. Huhns, editor. *Distributed Artificial Intelligence*. Morgan Kaufmann, London, 1987.

Michael N. Huhns. Interaction-oriented programming. In Paulo Ciancarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Artificial Intelligence*, pages 29–44, Berlin, 2001. Springer-Verlag.

Michael N. Huhns and Munindar P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, 1998.

Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

Abdulla M. Mohamed and Michael N. Huhns. Multiagent benevolence as a societal norm. In Rosaria Conte and Chrysanthos Dellarocas, editors, *Social Order in Multiagent Systems*, pages 65–84. Kluwer, Boston, 2001.

Jörg P. Müller, Markus Pischel, and Michael Thiel. Modeling reactive behavior in vertically layered agent architectures. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents*, volume 890 of *Lecture Notes in Artificial Intelligence*, pages 261–276, Berlin, 1994. Springer-Verlag.

Nortel Networks. FIPA-OS, 2003. http://fipa-os.sourceforge.net/.

Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence*, 13(1), 1999.

Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, 1991. Reprinted in Huhns and Singh [1998].

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice-Hall, Upper Saddle River, NJ, 2003.

Michael K. Smith, Chris Welty, and Deborah McGuiness. Web ontology language (OWL) guide version 1.0, 2003. http://www.w3.org/TR/2003/WD-owl-guide-20030210/.

Milind Tambe, David V. Pynadath, and Nicolas Chauvat. Building dynamic agent organizations in cyberspace. *IEEE Internet Computing*, 4(2):65–73, February 2000.

Gerhard Weiß, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.