DESIGNING AN ECOLOGY OF DISTRIBUTED AGENTS

by
Nelson Minar

B.A. Mathematics (1994)
Reed College

`<nelson@media.mit.edu>`
`http://www.media.mit.edu/~nelson/`

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology

September 1998

DESIGNING AN ECOLOGY OF DISTRIBUTED AGENTS

Nelson Minar

<nelson@media.mit.edu>
http://www.media.mit.edu/˜nelson/

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, on August 7th 1998, in partial fulfillment of the requirements for the degree of Master of Science in Media Arts and Sciences

Abstract

The Internet is a rich environment for computation. There is a need for design principles to organize distributed computational activity on the Internet, something analogous to the way the World Wide Web is an organizing principle for documents. This thesis introduces the idea of an *ecology of distributed agents* as a paradigm for building distributed software. Computers run servers that are local environments of computation. Applications are built out of agents that live in these servers. Mobile agents move to servers to use local resources and servers support agent query services to allow agents to discover each other and communicate information over the network. A system that creates an ecology of distributed agents, Straum, is presented with a technical discussion of its implementation. Two applications, communicating user presence and monitoring server activity, are presented along with sketches of other possible applications. Straum and the underlying design paradigm are evaluated with respect to other distributed systems research. Finally, ideas for future work are presented, plans towards making the Internet a natural environment for computational activity.

Acknowledgments

I am very grateful for the advice and support of my advisor, Pattie Maes, for her feedback and for keeping me focussed in my research. The comments and time given by my readers Mitchel Resnick and Judith Donath have greatly improved and clarified this work. Thank you to MIT UROP Kyle Jamieson for his contributions to the Straum software and to UROP Maggie Oh for designing agent icons. The `dist-obj` community has been a wonderful forum for discussing ideas about distributed systems; I am particularly thankful for the continued input of Ron Resnick, Joe Kiniry, and Mark Baker. Finally, I owe a lot to my friend and colleague Kwin Kramer. Our conversations and work together have greatly influenced this thesis.

DESIGNING AN ECOLOGY OF DISTRIBUTED AGENTS

Thesis Readers

Advisor ____

Pattie Maes
Associate Professor of Media Arts and Sciences
MIT Media Lab

Reader ____

Mitchel Resnick
Associate Professor of Media Arts and Sciences
MIT Media Lab

Reader ____

Judith Donath
Assistant Professor of Media Arts and Sciences
MIT Media Lab

Contents

Chapter 1

Computation in the Network

1.1 The Rise of the Network

With the advent of the Internet age, the nature and structure of computation has fundamentally changed. A computer is no longer a box standing in the corner that operates only when the high priest is standing in front of it mumbling the right magic words. Computers are on our desks, part of our daily lives both at work and at home. And desktop computers are no longer isolated; they are on the Internet, often 24 hours a day, a mere packet away. The Internet is a global network swarming with activity. Millions of computers are sending messages back and forth to each other all the time for purposes both mundane and arcane. What can we do with all of the power of this network?

As the Internet grows larger the capability of the system increases. The most obvious increase is in CPU power: thousands of computers crunching numbers together can perform impressive tasks such as breaking cryptosystems [3] [22] [59] solving number theory problems [68], or processing huge volumes of radio signals [72]. Similarly, networked computers can also provide raw storage capacity, allowing documents to be published literally forever [1].

Desktop computers are currently the most ubiquitous networked devices, with the familiar interface of a bitmapped screen, a keyboard, and a mouse. But all sorts of other computers are also part of the digital æther, devices with interesting ways of interacting with people. Wrist watches, automobiles, coffee machines [70], even people's bodies [46] [39] are all becoming networked. As networked devices infiltrate our physical world, we will see more and more applications of computer networks in our daily lives. Smart objects can bring in data from our physical environments, such as biosensors carrying in information about a runner's heartrate [41]. They can also express data, communicating information in more intimate and immediate ways [47]. The question is, how do we make all of these devices work together?

The growth of the Internet means that the network itself will become more *active*, with more programs and processes operating and interacting on the network. Key enablers of this transformation are low-cost computers and wireless networks. But these technologies only make physical networking possible, they do not say much about how devices will interoperate, about what they will say when they are networked.

There are many hard questions in creating systems out of a myriad of networked components. What will the devices do? How will they work together, coordinate, cooperate? A particularly difficult problem is the open Internet environment, where computers come and go without any central approval. How do we build distributed systems that work with such diverse components?

These issues can be summed up with one essential question; how do we build computational activity *in* the Internet? This thesis proposes a particular methodology to address the design question, an "ecology of distributed agents." In pursuing this idea, suggestions for specific applications will be developed and used as a means to understand the future of the Internet as an environment for interacting processes.

1.2 The Lesson of the World Wide Web

The central question in the design of large-scale distributed systems is "how do we manage complexity?" System design is particularly difficult on the Internet because the Internet is an open network: anyone can add or remove a computer to it at any time. While this openness is the major strength of the Internet, it is also one of its greatest weaknesses. There is no way to trust another computer on the Internet, except in very limited cases. Our daily experience on the Net is plagued by random outages, faulty software, and security problems. The Internet is also largely decentralized, with very limited administration. System designers cannot rely on a centrally administered reliable network of trusted computers.[1] How do we get something useful done on this difficult substrate?

The World Wide Web is the most powerful example we have today of a large-scale Internet system. The Web was started only in 1990, but in just eight years it has grown to an enormous size, encompassing some 320 million documents [28]. The Web has literally transformed our notion of distributing information. Not only can anyone publish documents (and anyone does!), those documents are available instantaneously anywhere in the world at an incredibly low cost, and exist in a rich interactive document context.

As a distributed computer system, the Web is fairly simple: the components are (mostly) inert documents, not running programs like those that most distributed systems researchers work with. The technology is quite basic: the HTTP protocol enables a document to be downloaded, and the URL gives a document an address. The interactions between components are very limited: no complicated interlocking transactions, just the humble hyperlink to express "this document is related to that document." But despite the relatively simple and static nature of the Web, it is valuable to look at what makes it work as a lesson on how to build Internet systems.

The Web is not remarkable only because of the volume of information: it is the *connections* between documents, their interaction, that makes the Web truly useful. A simple device, the hyperlink, enables a giant web of interrelated documents all referencing each other. The author of a Web page can make a link to any other document on the Web trivially, just by pointing at the referrent's URL address. It is easy to overlook the significance of this architectural feature of the Web — one can make a web page interact with any another web page without any central authority approving of the link, without any cost to the second document's owner, with nothing more complicated than the address of the document.

Search engines are the second part of the story of the power of the Web. Anyone can visit one of the many available search engines, type in a simple query, and instantly be presented with a list of documents that are related to the query term. The remarkable thing is that search engines work without any central coordination. An owner of a web site does not have to do anything in order to have his documents indexed in search engines. No agreements

[1]Paradoxically, the chaos of the Internet is the very thing that makes it grow so effectively. We should love the complications, not be afraid of them.

have to be reached, no special transaction protocol has to be followed, no service request has to be filed. A search engine's spiders just go out and fetch pages just like any other reader of the document would do. From a systems design point of view this is horribly inefficient, but the fact is it *works*.

In looking at the Web for lessons on how to build distributed applications, the crucial lesson is to make systems that are *simple* and *decentralized*. Overdesigned protocols are too unfriendly; a simple approach like the Web can be incredibly powerful. And central coordination is a bottleneck; if authors had to register with a central authority every time a new link to a page was made, a new web page created, or even a new web site added, the Web's growth would have been very limited.[2]

The Web is a wonderfully effective application working on the global network. However, it only uses a fraction of the Internet's capability; Web servers seldom do anything proactive, they merely wait until something queries them for a page. And Web browsing computers are largely quiescent, sitting there waiting for a user to command it. A lot of potential in the Internet is going to waste. We can learn lessons from the Web in how to build scalable systems, but the time has come to add more activity, more process.

1.3 Active Internet Processes

Our current personal information spheres consist largely of an email address and a few static web pages. This thesis work concentrates on allowing people to augment their infospheres with active services and dynamic capabilities. Much like web servers allow users to have their own personal web pages available to the world, this research aims to make it possible for users to have their own personal processes living in the Internet.

There are a huge number of possible applications for active distributed computation ranging from cluster computing [22], to interactive shared environments, to science-fiction-sounding visions of intelligence emerging in the network [25]. This thesis focuses on the possibilities implied by the presence of computational devices in our everyday environment. Computers are not just generic resources — they are particularly personal devices, with

[2]The eventual breakdown of the "What's New with NCSA Mosaic" site tracking all new web pages is an example of the futility of centrally tracking Web growth.

specific interface capabilities. Each computer is also physically located; machines are in a particular place, a home or office or outdoor space, and they have a particular relationship to the people around them. And while desktops are somewhat generic, other networked devices that are being built (such as wearable computers, room sensors, ambient displays, etc.) are even more idiosyncratic and site-specific.

Computers are input devices, bringing in data from all over the world. They are also output devices, providing ways to display information in particular locations. The Internet is the way to transport all this scattered data, to collect information, process it, and then ship it out for presentation. The physical network provides the way to get the bits from one place to another, but those bits still need to be given meaning: They need to be routed, manipulated, and interpreted. And that interpretation happens every step along the way as the data is moved. Managing the data that comes from the networked world is not as simple as browsing the Web: the data needs to be found, processed, and delivered. And the programs that do this processing create a buzz of activity on the Internet, a swarm of process.

And so the network is naturally becoming more active, more alive, as programs run to manipulate data. The Internet is becoming one large distributed computer system. The question then becomes, how do we create a system that allows as much power and capability as the World Wide Web, but for active programs, not just static data? How do we design systems that use the latent power in the Internet, while also managing the trust and reliability issues? How can we give users the tools to understand and control the interactions of all of these running programs? And what will these systems do when we do build them? What will their capabilities be?

1.4 Straum

This thesis describes Straum,[3] an infrastructure developed for building distributed applications on the Internet. In summary, Straum is a way of building a system of interacting processes on a network. These processes are encapsulated as software agents; the natural environment of these agents is the network of Straum systems in the Internet. Every computer

[3]The name is taken from Vernor Vinge's novel *Fire Upon the Deep* [49]. Straum is the place where a particularly virulent form of network activity started.

system participating in the Straum system runs a server, a program that has two main tasks: running agents and providing access to information resources in the local environment. Applications are built out of software agents that coordinate to communicate and display data.

As an example, consider a simple distributed application where someone in Boston wanted to know the temperature in Budapest. In Straum, this would be accomplished with two servers and two agents. The thermometer in Budapest and the display in Boston would be resources managed by two Straum servers. When the user in Boston wants to start up the application, he or she would create an agent and send it to the server in Budapest to locate the local resource of temperature information. Meanwhile, a second agent would be created on the Boston machine, a display agent to graph the temperature. This agent would communicate with the server in Budapest, find the information agent with the temperature, and arrange recurring communication to ask for data.

A more traditional Internet solution to this problem would be to simply create a web page in Budapest that listed the temperature. The Straum approach is both more flexible and powerful. The explicit nature of agent creation and communication allows users to easily understand what is going on in the system and gives clear places to put in access control policies. And unlike a web-based hack, Straum is scalable to large numbers of interacting processes. Adding new types of data gathered in Budapest, creating new types of presentation in Boston, or gathering data from multiple sites is as simple as creating a few new agents, not redesigning the whole system. Complicated analysis is also possible: for example, the site in Boston could process the data looking for patterns and send it on to a meteorology facility that used the processed data to try to predict the weather. An ecology of agents is a useful paradigm for building distributed applications; the rest of this document examines these arguments in detail.

1.5 Overview of the Thesis Document

Chapter 2 describes the paradigm used to design the Straum system. An *ecology of distributed agents* is a way of managing the complexity of distributed applications by providing a simple natural metaphor for the interaction of the computational components. Servers in Straum are like islands, local environments with specific natural resources. Agents in Straum are the creatures living on these islands, each an autonomous part of a larger ecosystem.

Two applications have been developed on Straum so far. One application helps to create communities on the Internet by revealing information about users' *presence*, enabling social interaction by making people more aware of other people who are in front of their computer on the Internet. The second application is for monitoring the status of a computer's *activity*, allowing server administrators to watch the behavior of their systems. These applications will be described in detail in chapter 3 as a way of explaining how the Straum architecture functions. In addition, several other planned applications will be sketched, including more sophisticated activity monitoring systems, processing of data from the physical world, and enabling "smart room" applications.

The technical design of Straum is presented in chapter 4 and appendices A and B are useful references to the agents that have been built in Straum and the class hierarchy for the system. Design decisions are explained and evaluated, both to reflect on the power of the ecological metaphor and in particular to comment on the technical challenges of building Internet-wide distributed systems. Specific lessons learned in the construction of Straum include insights on current research issues in distributed systems such as code mobility, ontologies for describing capabilities, and the problems with transparency in networked systems.

We are living now in the cusp of a transformation in our world, one in which computer networks and networked devices are becoming a pervasive and essential part of our daily existence. Integrating these individual devices into large-scale, coherent applications is a challenging problem. Straum today is a demonstration and first implementation of the idea of ecologies of distributed agents. In chapter 5, conclusions drawn from the experience of this project are presented, along with a roadmap of specific future research and implementation directions. The problems discussed here are vital to creating the distributed systems that will shape our networked world in the near future.

Chapter 2

Ecologies of Distributed Agents

Given that we want computation on the network, how do we build it? What paradigm do we use for building applications that involve programs running on multiple computers? How do we build systems that are practical in the Internet, in an environment of ad-hoc networking and untrusted computers?

In current practice there are three major methodologies for building distributed systems. In historical order, they are message passing systems, remote procedure call, and distributed object systems [5]. Each of these technologies builds on the last. (Discussion of a fourth emerging methodology, state-space systems such as Linda [18] or JavaSpaces [66], will be deferred to the chapter 5.)

Straum is built as an extension of distributed objects, distributed software agents. In this chapter, the idea of an *ecology* of distributed agents will be introduced: the notion of a population of agents existing together in an environment of server computers. This methodology will be shown to be a natural way of building distributed computation, and particularly well-suited to the open environment of the Internet.

2.1 Distributed Object Systems

The first model of distributed programming is the message passing model [17]. The idea is very simple and literal: if one program wishes to communicate with another, it does so by simply sending a message, a packet on the network, to the other program. Message passing systems form the core of all networked systems. The Internet itself is a large message passing system, with extensions such as long-lived connections and address resolution. Many Internet applications such as FTP, the Web, or email are based on simple message passing.

Remote procedure call (RPC) was the first successful abstraction on top of message passing architectures [32]. If message passing systems are analogous to assembly language programming, RPC is like procedural programming, adding the abstraction of the function call to distributed systems. In RPC, if a program wants to communicate with a program on another computer it just calls a function on that other computer's machine much like it would call one on its own machine. RPC adds "transparency" to distributed programming, the illusion that talking to a remote program is not any different than talking to a local program. NFS, the usual Unix networked file system, is the most commonly known RPC based application.

Distributed object systems work similarly to RPC, only with an object oriented abstraction on top of procedure calls. Instead of calling a fixed method name distributed object systems gives programs references to remote objects, allowing the program to manipulate, call methods, and store the remote object just as a local object. In essence, distributed objects add a concept of *thing* to the concept of function from RPC.

Distributed objects are the current state of the art in building distributed systems. The major standard is OMG CORBA, a language-neutral specification for communicating object systems [37]. Competitors to CORBA include Microsoft's DCOM architecture [45] and the various distributed object systems layered on top of Java [65] [74]. These architectures all share the basic underlying paradigm of distributed objects but handle specific subtleties such as mobility, typing, synchronicity, and language independence differently.

For completeness, it should be noted that there are many subtleties and variations in distributed programming systems. Particular implementations of message passing, RPC, and distributed objects all differ in the way they deal with particular issues such as asynchrony, group communication, discovery, typing, and security. The path Straum takes through these details will be discussed as they are relevant, particularly in chapters 4 and 5.

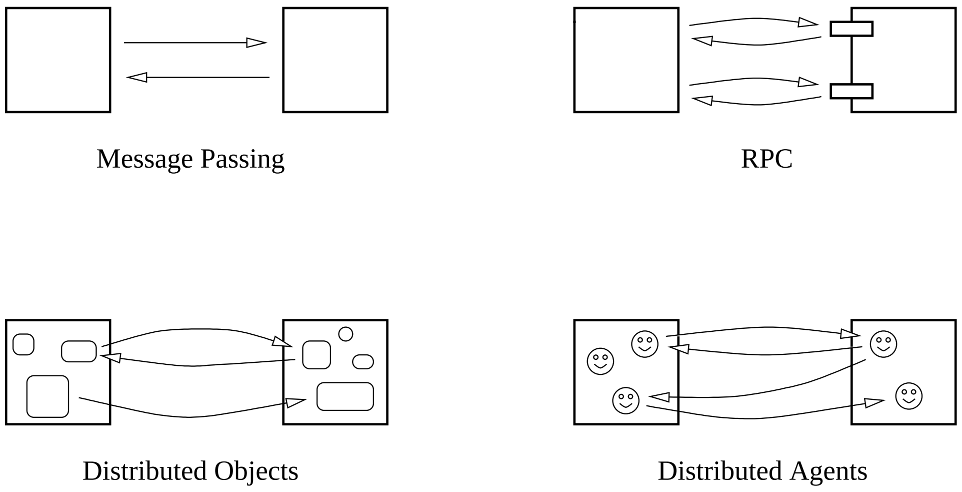


Figure 2.1: Distributed systems models.

2.2 Distributed Agents

Distributed objects are a good way to describe things on the network, resources and capabilities that are out there and available. However, distributed object systems are mostly static; they do not say much about activity on the network. To address this, a concept of process needs to be added to distributed objects. Distributed agents is about adding autonomy to objects, adding agency, to make them active participants on the network. Figure 2.1 characterizes message passing, RPC, distributed objects, and distributed agents.

Software agents research is very open and diverse. Almost anything that is a program has been called an “agent,” from web robots that filter information to software help systems to large artificial intelligence constructions. If there is a consensus in software agent research, it is that agents are *autonomous*, *proactive*, and *adaptive* [30].

Straum’s distributed agents make the most use of the first two characteristics of agents, autonomy and proactivity. Straum agents are autonomous: they are identifiable units of computation, executing independently, with their own set of goals partially independent of their larger environment. They are also proactive: every agent is a running program, an active process. In distributed object terms, then, a Straum distributed agent is simply an object with a thread of execution. This idea may seem simple (and it is!) but the implications of having many concurrently running, independent distributed objects are quite complex.

The third characteristic of agents, adaptable, has varied interpretations. Depending on the researcher, "adaptable" can mean anything from a thin veneer of explicit user configurability to a fully learning artificial intelligence. Straum makes no requirements that agents be in any way intelligent, although that ability is not disallowed, either. Straum remains agnostic on issues of artificial intelligence.

Straum does require agents to have a feature related to adaptability: self-description. An important aspect of a distributed agent is that it be able to describe the capabilities that it has. These descriptions allow agents to easily find other agents with the capabilities they need. Agents can then negotiate their communications during their lifetimes rather than requiring that a system engineer design the agent interactions ahead of time. Self-description enables adaptive interactions to emerge in the network: it is crucial to flexible distributed systems.

In Straum every agent carries with it an explicit list of the interfaces the agent understands and the meaning of the information it provides. In the simplest cases, an agent's description is similar to the type system of distributed objects: its capabilities are equivalent to the list of object methods that it understands. The type system of the distributed agent application defines an *ontology* of kinds of agents, one based on the syntactic description of the agent object's type. This ontology is open: the type system can be extended as the system is running with the addition of new types of agents. It is also ad hoc: any agent can add a new term into the implicit ontology without having to register with a central ontological database by just publishing it as part of its self-description. It is then the responsibility of other agents to understand what an agent's self-description means.

In addition to the basic type compatibility between interfaces, an agent's self-description can also include auxiliary information about the meaning of the interfaces that it possesses, entries in a semantic ontology. For example, a distributed object system typically has an ontology of data types such as `int` and `double` for integer and real numbers. Part of an agent's type is the fact that it has a method `int getCount()` which returns an integer, the count; this information is a syntactic marker. But in addition a distributed agent might also include the information that its `getCount()` method means the count of keystroke events it has seen in the last minute; this describes the semantics of the information. This extra information, the meaning of the method, is beyond the type system of basic distributed object systems. It must be defined with respect to another ontology, an ontology of meanings.

One final aspect of distributed agents that can be useful in building systems is the concept of "mobility." Mobile agents are agents that can move from one computer to another [55] [50]. More specifically, a mobile agent is capable of stopping its execution, transmitting its current state and its executable code to another computer, and then resuming its execution on the other computer.[1]

Mobility is a controversial technology for agent systems as it adds considerable complexity and security issues. But mobile code can be a helpful tool, allowing a distributed system to be more configurable while running. The current implementation of distributed agents in Straum allows for agent mobility and makes use of it in building up and dynamically configuring applications. Mobility is particularly useful in Straum in making it easy to upgrade an application by simply creating new agents, not redesigning the entire system. Ideas for the use of mobility in distributed agent systems are some of the most exciting future research directions. Sketches of applications that could use mobility more inventively are presented in section 3.4.

2.3 Ecologies of Agents

Distributed agents augment distributed objects with autonomy, proactivity, and adaptability. But how are the agents organized? Where do they execute, how do they coordinate their placement, how do they access information resources from the outside world? Distributed object systems designers have many different patterns for building applications out of objects, from simple client-server systems to three tier enterprise architectures to complicated federated systems. Distributed agent systems are new enough that a consensus on specific patterns have not emerged, every system has it's own design principles.

[1]It is actually more complicated for an agent to move than to replicate; the agent has to destroy itself on the host it is moving from. This fact about software agents is quite different from biological entities and partly explains why computer viruses are so prevalent.

2.3.1 Overview of Ecology

Straum introduces the idea of an *ecology of distributed agents* as an organizing principle for building distributed systems. The word “ecology” is borrowed as a metaphor to suggest a system where software agents in the Internet are like plants and animals in the natural world. There is a strong similarity between the complex interaction of organisms in a natural ecosystem and the complex interaction of software components in a networked system. The word “ecology” is also meant to specifically refer to the field of artificial life and other complex systems computer science work [27] [23].

In a computational ecology software agents have a specific lifecycle. Every agent is born in a particular environment, on a particular computer. The agent lives its computational life, possibly moving to new computers as the need strikes it, until it meets its end either by its own choice, by the hand of a user, or by an accidental mishap like a power failure. Agents interact with their local computational environment, consuming CPU, memory, and raw information resources from the local world, possibly using effectors on the computer such as displays or actuators. During the agent’s life it also meets and interact with other agents, consuming information resources from another agent and/or producing information to be consumed. The interactions between these agents creates a complex ecosystem of computational activity.

2.3.2 Servers

The servers that an agent lives on dictate the computational environment that the agent operates in. The server provides a home for the agent, giving it memory and CPU. In addition, servers provide a series of natural resources in the form of input and output capabilities. Servers implement a set of *monitors*, giving information about things local to that computer such as current keyboard activity or system load. Servers also provide effectors to agents such as bit-mapped displays to render information or a speaker to play sounds.

Straum is like an island ecosystem. Every Straum server is a different island with its own particular resources, its own owner, and its own unique mix of local resident agents. Islands are not completely isolated: two agents living on different servers can talk to each other across the network and agents can choose to move to a new server entirely.

An important aspect of Straum's model of distributed agents is the significance of *location* in a software agent's life. Computers on the Internet are not identical. Systems have different capabilities, they are owned by different people, and most importantly each machine has different local information and presentation resources. This is reflected in Straum by the fact that the location of any particular agent dictates which computer's core resources they have access to: what information they have, what kind of effector capabilities exist.

An agent must be on a specific computer to make use of its resources. To perform a distributed task, agents on different computers have to coordinate. For example, if an agent's goal is to communicate to Alice how fast Bob is typing, then that agent needs to be living on Alice's computer to draw the display and it needs to get in contact with an agent living on Bob's computer to acquire the typing rate rate information there.

This choice might seem odd from the point of view of conventional distributed systems design. It is possible to allow an agent to make use of resources on a remote host without having to communicate to a second agent. But this requires a specific, fixed interface be defined to export the remotely-usable capabilities. If all remote communication is at the agent level, then systems can be upgraded by simply changing the local agent population, not by reengineering the interfaces of the entire installed base.

Another unusual aspect of the ecology of distributed agents is that an agent only lives on a server at the largesse of that server's owner. The agent is dependent on the server for CPU and memory. If someone chooses to kill an agent on his or her server there is nothing the agent can do about it, it will die. This aspect of the ecology makes it more difficult to design reliable systems, but it is an absolute and essential problem in contemporary distributed computing. On the Internet, people often make use of computers they do not own. Most of the time, this use of a remote computer is dependent on the generosity of its owner. Many distributed systems try to pretend that reliability and trustworthiness are not problems, that the computer on the other end of the connection will always work. Straum faces up to the reality of the Internet by explicitly considering agent death as a possibility, exposing the design challenges directly to the systems architect.

2.3.3 Agents

Straum servers provide the sunlight, water, and raw nutrients. Agents are the inhabitants of this Straum world, the plants and animals that consume the natural resources and produce something processed, useful. Some agents never travel anywhere; they simply stay on their local computer and consume the data that is available there. Other agents look out on the network just a bit, find an agent that has some processed information that it needs and engages it in a lifelong relationship of exchange. But because agents are mobile, an agent could decide at some point in its lifetime it needs to move closer to a data source and migrate to a more prosperous locale. Agents do not even need to be fixed to one location: an agent could roam the network endlessly, traveling around in pursuit of its own goals, occasionally interacting with other agents and sampling the best that the local islands have to offer.

New agents can be created at any time, for any purpose. In the applications developed to date agents are largely created at the request of the user, but the architecture fully allows for agents to spawn other agents. Agents are not limited to a fixed set of types: a new species of agent can be created at any time by simply uploading an instance of the new type into a server. This flexibility allows for much power in creating a distributed system, but also a certain amount of risk. Ideas for managing the complexity of these systems are presented in chapter 5.

2.3.4 Existing Software Ecologies

This vision of flexible, mobile distributed computing is similar to other systems we see in the Internet. Tom Ray's NetTierra project [40] is one of the main precursors to the ideas in Straum. NetTierra aims to create a "Digital Reserve," a networked home for small programs to evolve in, to "set off a digital analog to the Cambrian explosion of diversity, in which multi-cellular digital organisms (parallel processes) will spontaneously increase in diversity and complexity." Ray's research is largely motivated by naturalist artificial life research; he intends to grow a complex ecosystem and then study it, as a way to reflect on natural evolution. Straum aims to create the same kind of complex ecosystem, but with a focus on applications to practical problems.

In addition to controlled experiments, there are many examples in the wild of this sort of distributed ecosystem interaction. The most common experience people have with these phenomena is computer viruses. Viruses are the closest things we have to artificial life in the Internet, small programs that spread from host to host without the permission of their owners. Unfortunately, viruses are usually damaging and never desired. One goal of Straum is to take the effectiveness of computer viruses and try to tame it into a safe, trustable, and useful system.

Another (more useful) example of the complicated interactions between distributed software are the various "bots" that inhabit chat systems, MUDs, Usenet, and the Internet at large. In his book "Bots: The Origin of New Species" [29], Andrew Leonard gives case studies of the complex and often surprising interactions between software agents in a variety of distributed systems. The main lesson of the book is that while the Internet allows for unprecedented capability for software agents interaction, it is difficult to predict or control the results of those interactions. Again, Straum aims to tame these phenomena by providing a more structured environment.

2.4 Conclusion

In summary, an ecology of distributed agents consists of several key points:

- Servers with local information and computation resources.
- An open-ended population of autonomous agents.
- Agent mobility so that agents can move to servers with resources they need.
- Agent interaction so that agents can communicate over the network.

The long-term goal of this research is to create a more open-ended distributed system, one in which surprises could emerge, capabilities that we had not expected to occur on our own. The key ideas enabling this goal are open, decentralized Internet systems and the structuring principle of an ecology of distributed agents. The Internet is the reality of distributed systems today; a powerful, unruly network of millions of computers. The design space of all distributed agents is vast, but the ecological metaphor helps to constrain it, localizing agent computation to particular places with specific restrictions to try to contain the chaos.

These are lofty goals. This thesis reports on the first implementation of these ideas. The implemented applications described in the next chapter apply the ideas of an ecology of distributed agents to develop two simple, relatively closed-ended systems. These applications fall short of the grand vision of a vast, seething network of distributed agents, but they do point to how a distributed system could grow to become more complex and powerful. The larger goal of open-ended distributed systems is addressed by the presentation of several scenarios that suggest how ecologies could be used to create new kinds of dynamic applications.

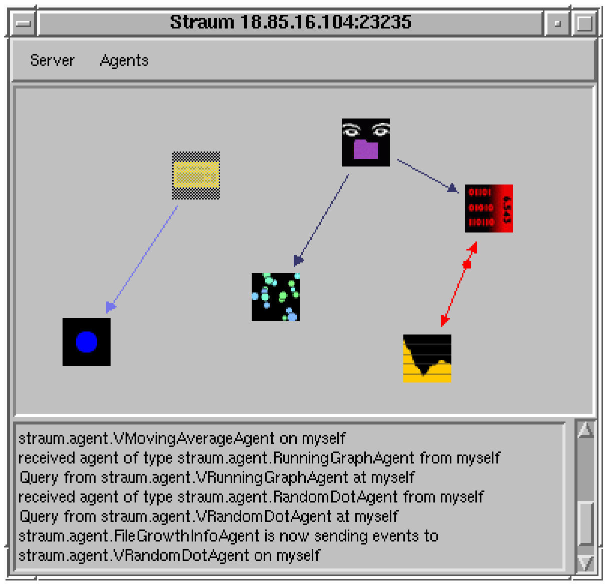
Chapter 3

Presence, Activity, and other Applications

The major motivation for creating an ecology of distributed agents is the flexibility it gives in designing applications. The hope is that with an open-ended distributed architecture suitable for use on the Internet, many new kinds of distributed applications can be created.

Straum is a first implementation of an ecology of distributed agents. Creation and development of the methodology and infrastructure has gone hand-in-hand with the design and development of several sample applications. This chapter describes the basic Straum system, presents two simple applications in detail, and then sketches out several other possible application scenarios. All of these applications are built out of the combinations of simple agents; the reader may find it useful to refer to appendix A for a listing of all of the agents referred to here.

These applications are based on a simple methodology for building programs: dataflow. The applications here are largely built as systems where data originates in one location, flows through a population of agents that process the data in some fashion, and is finally delivered to an agent that presents that data. Straum is not limited to only dataflow systems, but this type of application is relatively simple to work with and is easy to implement with an ecology of distributed agents.



```
straum.agent.VMovingAverageAgent on myself
received agent of type straum.agent.RunningGraphAgent from myself
Query from straum.agent.VRunningGraphAgent at myself
received agent of type straum.agent.RandomDotAgent from myself
Query from straum.agent.VRandomDotAgent at myself
straum.agent.FileGrowthInfoAgent is now sending events to
straum.agent.VRandomDotAgent on myself
```

Figure 3.1: Screenshot of Straum.

3.1 The Straum Application Design

Straum is implemented as a server program with particular populations of agents loaded into it to implement a specific application. The intention is that users of Straum would run their server all the time, providing a permanent home for their agents. When a user wants to perform a particular task, he or she creates a few task-specific agents on their server. The Straum server is an extension of a person in the Internet, much like an active version of his or her personal web pages.

Straum starts up with a server console window (see figure 3.1). The window has three parts: a menu bar for controlling Straum, a text console for notification messages, and a graphical view of the current agent population. The menu bar is straightforward: it provides user commands for creating agents as well as debugging facilities and the option to close down Straum. The text console reports status messages such as when a query for an agent service is received or a new agent comes onto the server.

The middle panel, the graphical view of the agents, is where most of the action is. Every agent that the local Straum knows about is drawn in this panel, represented by an icon that the agent itself provides. Agents that are living on the local server are drawn normally. Agents that this server has had some connection with but are not local (typically, agents communicating with some agent on the local server) are rendered at 50% transparency. In figure 3.1, there are five agents living on this user's server and a sixth (the keystroke agent in the upper left) that is on another server but is talking to a local agent. The display of agent icons allows a user to easily understand what agents are connected to his or her server.

The agent display is also useful for understanding the communication pathways between agents. Every time one agent sends a message to another Straum notifies the user of that communication by drawing an arrow between the two agents. Arrows start out red, then turn blue and slowly fade away over time if there is no further communication. Individual messages are drawn as small dots traveling along the arrows. Arrows are directional, reflecting whether the agents' communication is bidirectional. The graph view allows the user to quickly understand the interrelationship between agents.

When the user selects a menu item to create an agent a small form is popped up on the screen to allow the user to configure the agent. Information agents typically need no configuration. Display agents need to be told what host to look for information agents on and several agents have options to tune their behavior. Once created, display agents also create a separate window to do their actual display in. This window is a resource that the Straum server gives to the agent.

The Straum interface is minimal; many more user interface features would be desirable in a commercial application. But the implemented interface is sufficient to explore the appropriateness of an ecology of distributed agents for building a variety of applications.

3.2 User Presence

The Internet is not merely a way of transferring files. It is a social environment, a home, a place where millions of people are online and present at any moment. Traditionally the Internet has had few tools to reflect the presence of other people in real-time. The usual experience of the Internet is lonely, a person sitting by himself or herself in an empty room.

There is a need to have more social presence expressed online. For example, if I am online in Cambridge the same time my mother is online in Houston, I would like to know that we are both “there” at the same time. At a minimum, just knowing a person is out there is comforting and pleasant. It also creates opportunities for more communication — if I see my mother is online, maybe I will think to send her an email or call her on the telephone. Presence information is also useful in many other contexts. For example, a workgroup of people in different cities could use an Internet presence application to aid teamwork or a distributed community might use presence information to become more tightly knit.

3.2.1 Presence Applications

Many tools have been written to communicate some sort of social presence online. The old Berkeley Unix tools `finger` and `rwho`, for example, provide simple information such as whether a person is logged on to a particular computer, how many minutes it has been since he or she has typed, and how long ago he or she read email. These tools are limited — they require that you know what computer a user might be logged in,[1] the information they provide is not well standardized across implementations, and current practice is to not even run these services as they are potential security problems.

The contemporary equivalents to these services, tools such as ICQ [62], Ding! [58], and AOL Instant Messenger [56] are more reliable and better designed for the distributed nature of the Internet. They all share a common architecture — every user runs a client program that notifies a central server when they are online. The server then manages putting individual users into contact with each other. Once two clients have found each other, they typically use peer-to-peer messaging to continue communication. This semi-centralized approach seems to scale reasonably well, although the more popular services (ICQ has fifteen million users) have reliability problems.

However, these systems are all very limited in that they provide only a small amount of information about what a user is doing. ICQ, for example, will correctly tell other people whether I am currently connected to the Internet. However, it cannot automatically tell others whether I am currently at the keyboard, or how busy I am, or what I am doing at the moment

[1]This worked better in the old days when everyone on campus was logged into the same computer. With workstation networks, conventional finger is nearly useless.

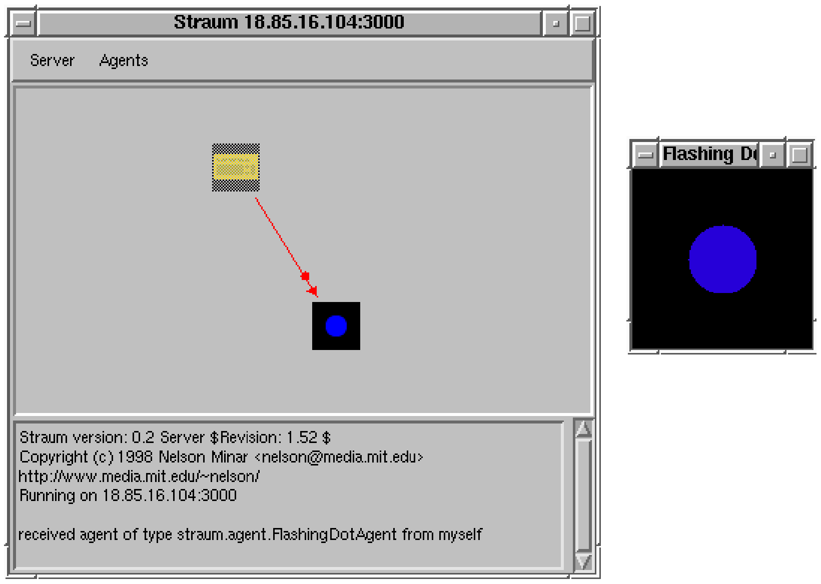
or whether I can be disturbed. There is a facility for the user to set his or her status to one of several selections (such as “away,” “extended away,” and “do not disturb”), but as this usually requires a manual setting it tends not to be used, and so even this limited information is seldom accurate. What is needed is a system that conveys more subtle information about a user’s presence.

For example, when I am working I have an awareness of my officemate’s presence, even though his back is to me and I cannot see him. The main channel of information I get about him is the sound of his typing; a rapid staccato indicates he is here and working hard, occasional keystrokes indicate he is reading email or a web page, and silence means he is either not working actively on the computer or is not in the office. This channel is not perfect, but it conveys a surprising amount of subtle information very simply. Internet presence applications should be augmented to include this kind of data.

User presence is a very personal thing — the kind of information I may want to know about someone I work with might be different than what I want to know about my mother. But programs such as ICQ are limited in an unfortunate way: they are closed-ended, fixed protocols with no extensibility. Adding a new presence capability to ICQ would require changing the protocol and upgrading the servers and all of the ICQ clients to the new system. A good application should be able to have new capabilities added to it without such impact. An application built out of an ecology of distributed agents allows for new types of agents to be added into the system without any redesign. This capability means that it is simple to create new ways to communicate a user’s presence as well as test and deliver agents that implement the new methods.

3.2.2 Straum’s Presence Application

Straum’s implementation of presence monitoring starts with the idea of listening to someone at the keyboard, replicating the experience over the Internet. In Straum, agents take the information available about one user’s use of a desktop computer and send it over the Internet to another user. For example, if Alice wanted to know how busy her friend Bob is, she simply creates agents to observe how fast he is typing. This basic presence application consists of two agents: one on Alice’s computer and one on Bob’s. Every time Bob presses a key, the keystroke information agent on Bob’s computer sends an event to an event display agent



```
Straum version: 0.2 Server $Revision: 1.52 $
Copyright (c) 1998 Nelson Minar <nelson@media.mit.edu>
http://www.media.mit.edu/~nelson/
Running on 18.85.16.104:3000

received agent of type straum.agent.FlashingDotAgent from myself
```

Figure 3.2: Alice's view of Bob with Flashing Dot Agent.

on Alice's computer which renders the keystroke in some specific way. Alice's view of the network along with her display agent are in figure 3.2.

3.2.3 Implementation Description

The implementation of the two agents is quite simple. Alice's display agent is a basic event client: it looks for a keystroke information agent on Bob's computer and then listens for keystroke events to display. The agent on Bob's machine is a simple "signal passthrough" agent; it listens to a keyboard monitoring facility on Bob's server and sends an event to Alice every time Bob presses a key. The two agents work together over the network to communicate an aspect of Bob's presence: the rate he is typing at. Bob could also watch Alice's keystroke by creating the two symmetric agents.

The information agent only sends one bit, the fact that an event happened, which in this case is interpreted as "a key was pressed." Naturally, the agent does not send *which* key was pressed; the goal is to communicate a general pattern of activity, not totally invade users' privacy! Alice's display agent could do many things with this bit of information: it could store it, process it, combine it with other information, or display it. To be literal, the agent could play the sound of a key click every time it received an event. In Straum's implemented presence application, the display agent is a "flashing dot agent" that draws a blue circle on the screen. Every time an event is received, the circle pulses. If many events are received in a short time then the circle warms up and turns red, and will cool down again after thirty seconds of inactivity. The timings in this particular agent are tuned specifically for typical keystroke rates; in practice, it gives the observer a fairly good measure of keyboard activity. But the flashing dot agent itself can take input from any event stream, it does not care what the events mean.

To actually build this application, all the users have to do is run Straum and the necessary agents. Alice and Bob, as regular Straum users, would each be always running their Straum server. If Alice decides to watch Bob at his keyboard one day, she creates a display agent and instructs it to ask Bob's server if it has any appropriate information agents. If the agent she needs is not running on Bob's computer, then she can create the right kind of information agent herself and send it over to Bob's computer.

Naturally, Bob has control over who is watching him. The key idea is that for an agent to listen to his keystrokes, it has to talk to the keystroke monitoring facility on his Straum server. And the communication between the information agent and the monitor can only happen on the same computer, not over the network, so the information agent has to be living on Bob's computer. Bob can choose not to allow any keystroke information agents to live on his computer at all, or to only allow a few friends (like Alice) to send him those agents, or to only allow them to live on his computer at certain times. By managing the agent population on his computer, Bob can create his own security policy. This is one way the ecosystem metaphor simplifies understanding the system: Bob is the ruler of his island, and he has final say over what agents live there.

3.2.4 Reflection on Presence Application

The keystroke monitor is only one possible application to communicate a user's presence. Straum has several other information monitors to communicate a user's activity at a computer. For example, Straum's system load average monitor can be useful to convey presence:

if my friend's computer is very busy, it may be an indicator that he is working hard as well. Other system monitors are easy to implement in Straum: watching mouse movement, or the particular applications someone is running, or the network traffic and disk usage the user is generating. Anytime a new system monitor facility is implemented on a user's machine existing Straum agents can immediately take advantage of that capability to communicate and interpret information about the user's presence.

There is no one single best way to monitor a user's presence. Instead, there are many different types of information one could use: a person's activity at the keyboard, or usage of the computer's resources, or what task he or she is currently performing. Straum's distributed agents architecture makes it simple to implement many different types of presence monitors, by providing a clean abstraction of how information is communicated between machines. And the particular pattern of data flowing through the system is immediately apparent simply by looking at the local ecology of agents.

3.3 Web Server Status

A second distributed application that has been implemented in Straum is a program to monitor the status of a web server, to see how much traffic it is getting and whether the server is overloaded. This application could be useful to server administrators and other people interested in their web server's usage patterns.

3.3.1 System Status Applications

There are many tools out there to monitor system behavior. Most of the available applications are general system load monitors; programs such as xosview or Norton System Doctor show the user a variety of graphs, charts, and flashing lights to indicate how much CPU load there is, how much free memory exists, how busy the network is, etc.

These programs can be quite configurable; for example, Norton System Doctor has some eighty different sensors that can each send their data to several different display and alarm capabilities. Most of these applications are not distributed, so that the monitor has to run on the computer it is monitoring. A notable exception are tools based on SNMP, the Simple

Network Management Protocol; these are explicitly distributed and somewhat more flexible than typical system monitoring tools [43].

Like user presence applications, system monitoring applications also tend towards the monolithic. Adding a new type of instrument is often tantamount to upgrading the entire program. But installing a new version of a program is expensive. For a desktop user it might only be an inconvenience, but for an administrator managing a cluster of hundreds of critical servers it is not acceptable to have to spend valuable time upgrading the monitoring application on every system. Furthermore, the problem is not always more features; these system monitors often incur overhead even when not being used. It should be easy to *remove* features from applications as well as to add them.

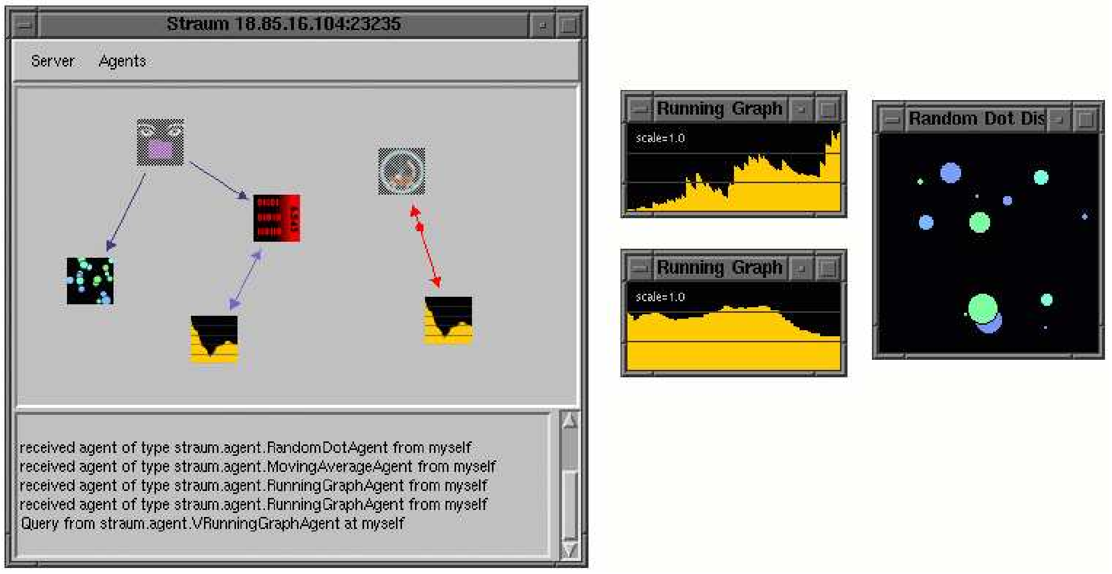
An ecology of distributed agents can be used to easily provide extensibility of features. Each computer in the network has a set of resources, the raw system information that is available on that computer. Information agents live on each machine, consuming the raw materials and producing useful, processed information. Display agents can then live anywhere on the network, connecting to the information agents on the relevant hosts to acquire the data the user needs. The population of agents provides a simple metaphor for understanding what system monitoring facilities are currently being used and the distributed nature of the communication makes it possible to run these applications over the Internet.

3.3.2 Straum's System Status Application

Straum implements a web server watcher by creating several agents to monitor access to the web server and the server's computational load. This is obviously a simplistic version of a full-featured server monitor, but it serves to demonstrate several features of Straum. In particular, this server monitor highlights the the power of composing agents as well as the convenience of using mobile agents to move a facility to a remote host.

3.3.3 Implementation Description

Figure 3.3 shows the web server monitor application in action. The example implements three displays of the activity of a web server. The monitoring host has a graph of the server's load average, a graph of the average number of web hits, and an event-oriented display of



```
received agent of type straum.agent.RandomDotAgent from myself
received agent of type straum.agent.MovingAverageAgent from myself
received agent of type straum.agent.RunningGraphAgent from myself
received agent of type straum.agent.RunningGraphAgent from myself
Query from straum.agent.VRunningGraphAgent at myself
```

Figure 3.3: Web server monitoring application.

dots, one per web hit. This application makes use of several different kinds of agent species. There are two information agents running on the remote host, one data processing agent on the local host, and three display agents on the local host rendering information into three different windows.

Data starts flowing from the web server. Two species of information agents live there — a load average agent and a file growth agent. The load average agent does not send any data out on its own; instead, whenever it is asked it will return a number, the current system load. The file growth agent is proactive, much like the keystroke agent in the presence application. It watches a file and sends an event every time the file gets bigger. In this case, it is watching the web server log. Every time a web page is hit the log file gets bigger, and so the file growth agent is actually sending out an event for every web page access.

On the monitoring client side, four agents are running. The event-oriented random dot agent works exactly like the flashing dot agent in the user presence application. It listens for events; every time it receives an event (indicating a web hit) the agent draws a circle with random color and placement. Over time the circles shrink and disappear, reflecting the age of the data (a device borrowed from the ChatCircles project [15]). Straum's user interface displays a one-way arrow from the file growth agent on the remote host to the random dot agent on the localhost to indicate that information only flows in one direction; from information agent to display agent. Note that a flashing dot agent could be used instead of a random dot agent. The two agents are interchangeable, they only differ in how they display the data.

A second agent displays the system load average as a graph. This agent is more proactive: it queries the information agent on the server every five seconds for the current load average. It takes the number that the information agent gives it (the load average) and plots it on a graph. The user interface draws a bidirectional arrow because the communication between these agents is two-way; the display agent makes a request and the information agent replies.

The third display, the average web hit graph, is built out of two cooperating agents. No display agent listens directly to the information agent on the web server; instead, a moving average agent is interposed. This data processing agent listens for events from the server's file growth agent and accumulates a moving average of the number of events every three seconds. A second running graph agent (exactly the same type as the one used to graph load average) then asks this moving average agent for a number every five seconds. The combined result is a graph of the number of web hits.

3.3.4 Reflection on Implementation

This example application demonstrates several advantages of the ecology of distributed agents. A key power is the composability of the agents. Each agent has a simple requirement, the information resources it produces and/or consumes. For example, a file growth agent produces one resource: events. Any agent that consumes events can communicate with that agent: the random dot display, the flashing dot display from the user presence application, or the moving average agent. Similarly, several agents produce or consume a different kind of resource: numbers. The running graph agent can consume information from any agent that produces numbers, such as the load average agent. The moving average agent is a particularly noteworthy number producer since it also is an event consumer. It serves a role like herbivores in an ecosystem, converting plants into meat or in this case, events into numbers. This sort of simple data conversion is not exactly a new idea, but Straum allows agents to take an active role in interpreting data.

Straum extends the power of dataflow by basing the system on distributed agents. Just as described in the user presence application, the displaying client is free to create agents and send them to the web server (with its permission, of course). In addition to the two information agents which must live on the web server in order to access the data, the user can also choose to place the data processing agent on any machine that it has access to. In the example described above, the moving average lives on the display client side. But this choice would be inefficient if there are large number of web hits because the raw event traffic would be sent over the network. Instead, the user could decide to create the agent and move it over to the web server, to do the number crunching on the web server and only send over a small amount of data to update the graph. Because the agents are truly mobile, the moving average agent could even decide itself that it would be more efficient to do the computation on another computer and migrate during its lifetime to another machine, thereby doing dynamic load balancing. With distributed agents it is easy to make choices about where to locate the computation without having to predetermine anything.

Furthermore, the agents are not just distributed in any random fashion; they are in particular locations, part of a particular population, and under the control of the owner of the server they inhabit. This design paradigm makes it easy to understand what is going on in the distributed system, to understand the effects of location and interaction. The only agents that can get data about the web server are the ones living directly on its Straum server. Similarly, the display agent has to be living on the computer that owns the screen. The data processing agent can live anywhere, but it has a specific interaction with the other agents. The ecosystem is an organizing principle, making the interactions between agents and their locations explicit and direct.

3.3.5 Possible Improvements

Several things could be done to flesh out this example application. The most obvious option is to add more monitors: network saturation, memory usage, error reporting. Agents to report exceptional conditions would be particularly useful; a watchdog agent could be created that regularly tests that the web server is running. If the server goes down, the watchdog agent would send an event which could then be caught by an error display agent that pages the web site administrator.

The application could also be improved by taking advantage of the composability of agents. For example, the existing random dot display can be used to convey more information than just the fact that an event was fired. A web log analysis agent could be interposed that interprets the data that the file growth agent sends. This analysis agent could then direct the dot display agent to, for example, show web hits from the US in one color and the rest of the world in another. Composing agents to change the interpretation of data is simple.

The system is configurable after the core Straum servers are deployed with no reengineering required. If a third party comes up with a fancy new kind of web log analysis algorithm, an agent can be created with the new code and put into the system without any sort of change to the rest of the installed base. And new computers can easily be brought in; if an administrator decides she wants to watch the web server performance on her palmtop computer when she leaves for the weekend, all she does is ask her agents to migrate into her palmtop and carry their computation with them. Because the communication between agents is dynamically negotiated, there is a lot of flexibility in how the application is built.

3.4 Other Application Scenarios

Straum is intended to be a platform for many types of applications. The examples implemented so far have been fairly simple systems, communicating data from one computer to another computer in a largely predetermined fashion. But distributed agents are capable of more complex interactions than simple dataflow and the ecological paradigm allows for much more flexible applications than the simple cases described above. This section describes several other applications that have been envisioned, sketching their design and highlighting how Straum simplifies their implementation.

3.4.1 Monitoring Multiple Servers

An interesting capability of an ecosystem of distributed agents is that it is very simple to aggregate data from several different information sources. For example, a web site is often run off of many different servers; a web site administrator would like to watch the load on

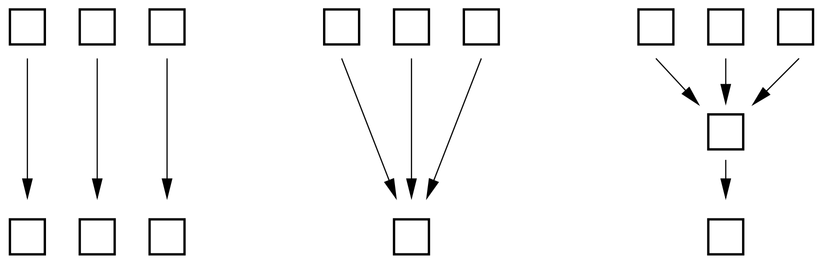


Figure 3.4: Three arrangements for many-to-one monitoring.

all of these sites to monitor for any possible bottlenecks or outages.

It is trivial to extend the web watcher application discussed above to watch the hits from several different web servers. The user could create several display agents pointing at information agents on different hosts or just create one display agent that knows to register itself as a listener to web hits on several different servers. Alternately, the user could create an event aggregating data processing agent. This agent acts as a funnel, taking in events from multiple streams and joining them out on one output stream. Adding an extra agent into the pipeline improves the system; the display agent is simpler as it only needs to listen to one agent. And the extra agent allows flexibility in choosing on which host the actual aggregation occurs as well as the option to redirect the processed event stream later. A schematic of these three different implementations of many-to-one monitoring is presented in figure 3.4.

However, with many sources of information coming in at once it might not be appropriate to naively join them together. Thousands of web hits on a busy network of servers would overwhelm the simple data presentation system. With Straum, it is easy to add extra agents to do higher level event processing, extracting higher level patterns of data. Related web hits on different servers (a page loaded from one server, images loaded from another) could be collapsed into one single event by an agent that knows about the relation between the two servers. This sort of analysis can be positioned anywhere in the Straum network: close to the data sources to reduce event-reporting network traffic or close to the observer to reduce computational load on the servers. These patterns of analysis can easily be modified and updated in the ecology at runtime.

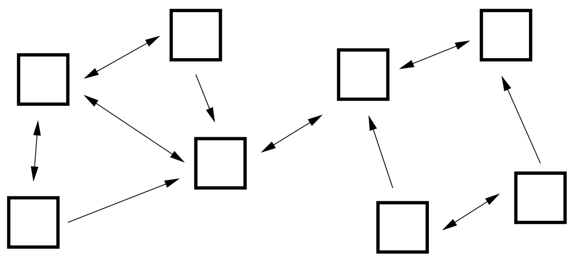


Figure 3.5: Monitoring pattern for a community of developers.

3.4.2 Workgroup Community

The original Straum presence application was described as a one to one application: one observer, one observed person. Naturally, it would be simple to extend this application to a many to one situation so that one user is watching several different people. The construction is just like the many to one server application discussed above.

The Straum network becomes more interesting if you consider a whole workgroup of users all running Straum together. For example, consider a group of programmers working together over the Internet on a collaborative software project. They are scattered all over the world and coordinating through the network.[2] Each user in the system runs Straum with a few agents to export information about their presence and a population of other agents observing that information from their peers.

To each individual user, his or her use of Straum appears to simply be many to one. The data that the Straum agents carry about a user's presence serves to knit the community together. However, the aggregate Straum network has more information embedded in it than any individual user's view of it. The actual pattern of agent interaction — which people are paying attention to which people — also contains interesting social information. For example, if a team of programmers is split into two teams working on separate parts of the system then the clustering of agent interactions should follow the team organization (figure 3.5). A person watching these agent interactions could inspect the clusters and derive the

[2]Widely distributed development is a fairly common situation with open source projects, for example in the Linux community.

team structure simply from looking at the dataflow in the Straum network.

For a small workgroup this sort of analysis might seem complicated; the members of the group probably know their interactions already. But for large projects with hundreds of developers the members of the community might not be aware of their relationships. An agent that watches the interactions of *other* agents can gather information about the flow of information in the Straum network, thereby revealing social patterns in the network. This kind of information could be used by a manager who wants to improve efficiency or by the participants themselves as a way to become self-aware about the social dynamics of their group. Sometimes the patterns of communication between users is as interesting as what they are actually communicating [14].

A group of programmers working closely together might also form the nexus of an interesting community of Straum developers. For example, suppose one of the programmers gets bored one day with the project and hacks up a nifty new kind of display agent. Thanks to the ease of Straum's mobile code architecture, he or she could simply send the agent off to a couple of friends to let them play with it. One of those people might also like the agent and forward it off to others, and fairly soon this new agent would populate the local Straum network with minimal effort. It would even be possible to build an agent whose job it was to look for new agents and propagate interesting ones to friends automatically. Straum makes it possible to make software distribution more fluid by using the openness of an ecology of distributed agents.

3.4.3 Ambient Displays and Smart Rooms

The application scenarios described above have all been in the context of conventional desktop and server computers. These scenarios are limited by the capabilities of the input/output devices, the sources of information about people and the screen to display data. In practice, a desktop computer is actually a fairly strange device for measuring a person's presence. Even something as basic as monitoring typing rates is surprisingly difficult in contemporary operating systems[3] and it can only tell you about a person's presence if they happen to be

[3]The Win32 version has to install a DLL in every running program, the Unix version has to hook events in every open window.

typing on the box on their desk. Similarly, computer screens are also fairly rigid ways to display information, limiting the presentation to graphical displays in little windows.

The potential applications of Straum become much more exciting when we consider integrating it with objects in people's everyday environments. The various research programs in ubiquitous computing [54], smart rooms [38], and Things That Think [73] all point to a host of ways of bringing computational devices into our everyday world. But even if the physical engineering problem of embedding small networked computers in our environment is solved, there is still the application-layer integration question; how will these devices work together and coordinate? One possibility is to use a population of distributed agents living in the various devices, giving each object just enough smarts to send or receive the data that it needs.

As a simple example, consider the work of Hiroshi Ishii and Brygg Ullmer to represent accesses to a web server as the sound of rain falling [24]. The simplest way to implement this would be to run a long speaker wire from the web server to the speaker. But wires are ugly and slow, and a specific installation would limit what that speaker could be used for. Instead, a nicer solution would be to make the speaker a true peer on the network, running Straum. The speaker environment would provide one resource to agents: the ability to play an arbitrary sound. It is then simple for the raindrop web-server application to be built; merely create a raindrop-sound agent, tell it to listen to an agent web server for accesses, and send the agent to the appropriate speaker in the room. Later, if you want the sound to play in another room, just move the agent to a new speaker. If you need to use a Straum-enabled speaker for another application as well, simply load another agent into it. The speaker can implement its own policy for handling resource contention (perhaps mixing two simultaneous sounds), or the agents themselves could negotiate sharing.

Similarly, Straum provides an easy way to hook sensors in our physical environment to other kinds of ambient displays. For example, consider a building that is outfitted with many different motion sensors. Each sensor hosts one Straum agent, an event notifier that tells listeners when a person walks by that particular spot in the building. This specific information is sensitive; the building manager would not want to give it out to protect people's privacy. So the manager puts access restrictions on those servers and creates several processing agents on a public server that 'wash' the data, perhaps time averaging it a bit or combining input from several sensors so a person's motions are not easily trackable. This less sensitive data would then be available to the occupants of the building to access, interpret, and display.

Each of these particular applications could be implemented in an ad-hoc fashion by hard-wiring network connections and using proprietary protocols. The point is that Straum provides a generic platform for this sort of system. All the manufacturer of a particular device has to do is arrange for Straum to run on it.[4] Once that is done the particular device becomes a generic resource, an open capability that can easily be used for a variety of applications.

3.4.4 Automobiles

Automobiles are a good domain for Straum's combination of mobile agents and the location-based model of computation. Cars are interesting platforms for distributed agent computation: they are personal spaces, they move, and their location is an important property for their owners. With a Straum server in every car, mobile agents would have an interesting environment in which to live and a host of problems to solve.[5]

One potential application is location-dependent advertising. For example, suppose McDonald's wanted to advertise to you when you drove by their stores. The local McDonald's would run a Straum server stocked full of agents that are primed to bring an advertisement into your car. As you drive by, the agent would hop from the store into your car, taking advantage of the local display resources (the car stereo, perhaps a heads-up display on the windshield) to suggest tempting Happy Meals. Naturally, you could configure your car's Straum server to refuse these advertisements or restrict them to only play a quiet advertisement on the rear speakers. But if the advertising offers an incentive (perhaps a discount coupon) then you might want to allow the advertising agent in. Straum provides the infrastructure for these agents and leaves the control over their capabilities up to the car's owner.

Straum could also be used to help monitor car traffic. If every automobile on the highway runs a Straum server, then a network of cars creates a perfect medium for agents to coordinate data about traffic patterns. Agents would live in the network of automobiles, traveling

[4]Admittedly, this might not be trivial. But embedded Java is quickly becoming a reality and the Straum system itself has very little overhead.

[5]Thank you to Judith Donath for suggesting these car-related applications.

up and down the highways by hopping from car to car. These agents would carry information about local traffic conditions and display relevant data to interested drivers. Some agents could be generic agents launched by the state police: they would simply alert drivers to traffic conditions on the road ahead. Users could also create and launch personalized, specially programmed agents that would scout out the driver's particular path and return with information, possibly suggest alternative routes. Working out the details of these algorithms is not trivial, but the Trafficopter project points to the possibilities of distributed data collection for traffic patterns [35].

3.4.5 Automated Trading

One final example application that highlights the applicability of Straum in an open network is the possibility of agent-mediated trading of commodities. As a simple example, consider a suite of four agents; a stock quote agent, a data analysis agent, a user interface agent, and a trading agent. Data flows from the beginning to the end. The quote agent reports the current price of a commodity. An analysis agent takes this data in and analyzes it for signals that indicate that this is a particular time to make a transaction. If the data analysis agent finds something, it fires off a message to a user interface agent to ask the user if he or she wants to act on the signal. If the user clicks "OK," the interface agent then fires off a message to the trading agent to place the trade. Brave people who trust software could just let another agent make this decision, thereby fully automating their trading strategy.

What makes this example interesting is that all four of these agents will not be on one person's computer. Naturally, the user interface agent would live close to the user: the agent has a close communication with the user. But stock quote services are an information commodity, consumers purchase the data stream. So the stock quote agent would be living on a computer close to the trading room floor, providing data to all subscribers. Similarly, trading services are also a commodity. The actual trading agent would live at the brokerage, accepting orders from customers' agents.

The piece in the middle, the data analysis agent, is the most interesting. Trading systems are sold by vendors; there is a brisk business in selling software packages that look for trading signals. Straum would streamline the process of selling these packages. A user user would buy a system from the author, who would then spawn off one copy of the agent and send it over to its new master's computer. Or maybe the system is very secret and the author does not want to give away the code. In that case he or she would merely rent the services of the analysis agent, giving the user the right to connect to the analysis agent and read its signals. The ecology of agents is a flexible way of composing applications out of smaller pieces, allowing choices not only for distribution but for ownership.

3.5 Summary

Straum implements a methodology for designing software, the ecology of distributed agents. This paradigm is quite powerful for building applications on the Internet, allowing much flexibility and open-endedness in structuring systems. The core technology of agent-mediated peer-to-peer communications makes composing pieces of software together easy. The examples of communicating a user's presence and monitoring the status of a web server show how it is simple to construct a distributed application out of simple agents.

These implemented programs demonstrate the usefulness of Straum today in desktop applications. But they also point to other interesting application domains. The simplicity of the distributed agent architecture makes it natural for being the 'glue' that knits together smart objects in our environments. Agents can easily be inserted to mediate information from many different sources. And the interaction of the agents can itself be interesting data, reflecting patterns of use in a community of users.

Finally, the strong notions of locality and mobility, the structure of the ecosystem, points to a new conception of distributed systems. Software should not just be brittle installations of code that can only run in one place; programs can live *in* the network. New agents can be sent far and wide across the network, populating users' machines with new functionality. Proprietary agents can remain at their author's homes, keeping their code safe from prying eyes. An ecology of distributed agents provides great flexibilty for building applications in the Internet.

Chapter 4

Straum Implementation and Evaluation

Straum's implementation of an ecology of distributed agents is a fairly straightforward extension of existing distributed object software. Straum derives most of its functionality, particularly its portability and clean design, from Sun's Java language and runtime environment. Support for distributed object programming comes from Objectspace's Voyager library [74]. Everything else — the system monitors, the graphical interface, and the server and agents themselves — is implemented in the Straum libraries..

Straum's Java classes are broadly split into two categories: server code and agent code. The server code provides the home for the agents, the various system monitor facilities, and the majority of the user interface. The agent code implements a few base classes for common types of agents but is largely taken up by a suite of application specific agents. The server code is 1400 lines plus another 500 lines for system monitors. The agent code is 1200 lines; individual agents vary from 20 to 100 lines each. In this chapter details of the implementation of servers and agents will be presented followed by a discussion of the lessons learned in building and testing the system. The class hierarchy of the server and agents are diagrammed in appendix B.

4.1 Servers

Servers provide the execution context and resources for agents. The server also provides the user interface; it manages the agent graph interface and gives the user controls to create new agents. Underneath, the server is responsible for managing resources and access, setting security policy, and generally keeping track of what is going on. In many ways the Straum server is analogous to an operating system for agents, an OS written for the Java virtual machine.

4.1.1 Technology Base

Most of Straum's capability derives from its base in Java [2]. Java is more than another computer language, it is an entire environment, a virtual machine architecture. The language aspects of Java make it quite suitable for designing a project such as Straum: the object oriented model is quite clean and the language has good support for Internet programming. But the Java virtual machine (VM) is what makes Java particularly powerful for building an environment for distributed agents.

Java's most hyped VM characteristic is the "write once run anywhere" capability. Java programs are in a bytecode that can be interpreted by any Java VM. This portability is useful — it means that a Straum agent can trivially run anywhere that Java runs. But beyond that, the Java VM also provides an entire layer of abstraction for the Straum execution environment. It gives the capacity to hook all sorts of agent actions, such as accounting for their CPU and memory usage or restricting their network access. Because most Java VM implementations are closed, in practice it is difficult to take advantage of all of this potential. Sun's Java is fairly good at giving access for security restriction, the well known "sandbox model," but currently provides no hooks for CPU accounting and the like. As open Java VM implementations such as Kaffe [67] and Japhar [63] mature it will be possible to have more fine-grained control over agents behavior in the servers. Specific ideas for this are discussed in chapter 5.

Java has excellent support for Internet programming. The low-level socket libraries make message passing easy and the RMI and Serialization libraries [65] make it relatively straightforward to do distributed object computation. However for this project a different distributed object system, ObjectSpace's Voyager 1.0 [74] was selected. Voyager was chosen because it seemed to provide a simpler, more powerful, more transparent interface to distributed objects and mobile agents.

Voyager provides a very direct model of distributed objects. In addition to a normal Java

`Object` reference to an object, every object in Voyager has a `VObject` reference. This `VObject` reference points to a proxy object representing the underlying object. The proxy reference can be messaged just like the regular object reference; messages pass through the proxy to the real object. But the proxy is more powerful: it can be a reference to a object on another computer, another Java VM. In this case, the proxy forwards the message to the underlying object over the network. Proxies provide transparent access to distributed objects: the programmer does not even need to know if the object is local or remote.

The key feature of Voyager is that any object can be "remote enabled" by simply creating a `VObject` reference to it. There are several messaging models for remote access: synchronous method calls that work just like normal methods in a nondistributed context, "one-way" calls that return immediately and do not guarantee delivery, and "future" calls that are asynchronous like one-ways but allow return values to be collected later. In addition, Voyager provides many other useful services. These include distributed garbage collection, a simple name database, and a group communication facility. Finally, Voyager also provides object and agent mobility. Every `VObject` implements a `moveTo` method that causes an object to be moved to another Java VM. The underlying support for this capability are mostly present in Java 1.1, but Voyager encapsulates object mobility, making it a simple primitive for programmers to use.

Straum uses the core Voyager features, particularly the distributed messaging and mobility. In general, Voyager worked well for Straum. However, many of the other capabilities in Voyager either seem superfluous or inappropriate in the context of an ecology of distributed agents. Reflection on using Voyager for implementing Straum are evaluated in section 4.3.

4.1.2 Resource Management

The main task of the server is to manage the resources of the local machine. The primary resource is the CPU; the server must create threads to allow agents to run. The server also manages access to local input and output resources, in particular access to monitors and the capability to create display windows.

When a Straum server starts up, it initializes Voyager to listen for messages and mobile agents on a well known port. It also registers itself with the alias "server" so that incoming agents can find the server object itself. The server then creates monitor objects and the user interface window. Finally, the server's own thread exits; it has nothing active left to do. Instead, the server waits for the user to ask for a service via the interface or for an agent to ask for a service through a method call.

From an agent's point of view the most important method the server implements is `newAgent`. This method must be called by an agent as soon as it arrives on a server.[1] When the server receives this message, it stores a reference to the new agent in its list of the local agents and spawns a thread for the agent to run in. The `newAgent` method is the way that an agent joins the local population and gets the CPU it needs to live.

Once an agent has moved to a server it is free to access the local monitor resources on that system. The suite of possible monitors is hard-coded into the server object: Straum servers simply have methods such as `getKeystrokeMonitor` and `getMemoryUsageMonitor`. Different servers can implement different monitors; each server can have a different set of natural resources. For example, the Unix version of Straum does not have a window creation monitor, only the Windows version does. It is up to the agents to handle the failure to find a particular resource.

The monitors themselves are application specific and their implementation is not particularly interesting to the design of the ecology. Briefly, the sysinfo library provides JavaBeans style [64] access to low-level system information such as keystroke events and system load average. The base sysinfo package provides abstract interfaces to system monitors, and specific implementations are provided in subpackages for Windows and Unix.[2] This library is independent of Straum and could be reused in any Java application; its functionality is merely imported by Straum servers to provide data access.

In addition to access to input in the form of monitors, Straum servers also provide one output resource, the ability to create a window on the screen. Agents are free to create graphical components with Java's graphics libraries; the agent then calls `acceptComponentFrom` on the server to have the component drawn on the screen. Straum servers therefore also give agents the capability to affect the environment the server is running in. Other output re-

[1]Currently, Straum doesn't enforce this — a malicious agent could disobey the protocol and avoid being tracked by Straum. The problem is that Voyager has already created and run the object; possibilities for enforcing this protocol are deferred to the security discussion in chapter 5.

[2]Actually implementing these monitors was surprisingly time-consuming: it turns out to be quite difficult to, for example, capture keystrokes in Windows.

sources could easily be added such as the ability to play a sound or control an ambient display device.

4.1.3 Agent Query Services

Servers provide a home to agents and access to input and output resources. The server also provides the capability for agents to find each other. Discovery is implemented by the server's `queryAgents` method. If an agent is looking for an agent on a remote server to provide information, it calls `queryAgents` on that remote server with a specification of what it is looking for and gets back a list of matching agents. This method is the only one that agents are allowed to call on a server remotely, all other server methods can only be called by an agent living in that server.

The query specification itself is simply a list of Java types. For example, a flashing dot agent might be looking for agents that implement the `EventSending` interface: it makes a call to `otherServer.queryAgents(this, "EventSending");` to get a list of event sending agents on the remote server. The reply is a list of agents that match the given criteria. The flashing dot agent then picks one of these agents to talk to and calls the `addListener` method on the remote agent to subscribe itself as an event listener. If there are multiple agents that match the query, it is up to the querying agent to choose the right agent from the list.

The agent query facility implicitly uses a particular ontology of possible agents, the Java type system. There is a one to one correspondence between Java types and entries in the ontology of queryable agent services. This ontology is *syntactic*, it is rooted in Java syntax. It is the bare minimum necessary to make distributed agents work: if the remote agent is not of the correct type, then the necessary methods would not be present. For example, the query for `EventSending` guarantees that only agents that implement an `addListener` method are found in the query, so that remote agents then know it is safe to call `addListener`. This query system enforces a sort of "plug compatibility" between agents; the query is designed to find all the remote agents that could possibly talk with the querying agent.

However, plug compatibility is not the only criterion an agent might want to use when searching for other agents. For example, there can be several agents on a remote server that all implement `EventSending`; a keystroke agent, a file growth agent, and a window creation agent all send events. How does the querying agent distinguish between them? One way is to be more specific in the type query, to ask explicitly for a `KeystrokeInfoAgent` or the like. However, this approach is limited because of Java's rigid type system. The other approach is to also use a second ontology, a *semantic* ontology about the meaning of the information an agent provides. This second ontology has not been implemented yet; discussion of its design is in section 5.2.2.

4.1.4 User Interface

In addition to providing services to agents, the server is also responsible for providing an interface for the user to see what is happening on his or her Straum island. The implemented interface is relatively simple, providing a set of menus to manage agents, a text console for message reporting, and a graphical view of all of the known agents. The other part of the interface in Straum applications, what the agents themselves display, is under the control of the agents (subject to `acceptComponentFrom` being called by the agent).

The menu options are straightforward. The server menu allows the user to turn on debugging options and to quit Straum. The agents menu provides easy access to commands to create all the different kinds of agents. Specific agents have different parameters that can be set at create time. For example, display agents can be told which remote server to look for an information agent on.

The graphical view of the agent world is the most interesting part of the interface, providing a dynamic view of the state of the entire Straum system. The code is contained in the `AgentGraphCanvas` class, one of the most complicated pieces in Straum. The agent graph tracks two things: icons for each agent that the server knows about, and arrows for the communication between agents.

The agent graph is event driven. Every time an agent sends or receives a message and every time a new agent comes to a server the graph is sent a message telling about the event. When a new agent is referenced, the graph asks that agent for an image and then uses it to draw the agent icon. If the agent is remote, the icon is drawn with 50% transparency. When a message is reported, the graph updates the arrow between the two communicating agents to reflect the recent message activity. Arrows are drawn red initially (with a small animated dot to convey the message activity), then turn blue and fade away until the next message is sent on the link.

The concept behind the `AgentGraphCanvas` is that distributed computation should be *observable* — it should be easy for a user to see what is going on in the system. Unfortunately, this property is not common in the design of computational systems. For example, in Voyager it is impossible to trap all agent messages at the server level.[3] Instead, Straum relies on agents to send messages to the server notifying it when a message has been sent. This requirement is unacceptable in a production system. It should be the server's responsibility to monitor communications, not the agents'. Distributed object infrastructures should be augmented to make observing messaging easier.

4.2 Agents

The major goal for Straum's implementation is to provide a simple, clean interface for creating new agents. The server is the operating environment for agents: because it is written once by the central architect its code can be as complicated as necessary to get the job done. But agents might be written by any user of Straum. Creating new agents needs to be simple. An important corollary is that server code is trusted, but agent code may not be. Agents can potentially come into a server from anywhere on the open Internet. The server needs to protect itself from malicious or buggy agents.

4.2.1 Agent Requirements

The major requirement of a Straum agent is that it implements a `doBehavior` method. This method is called by Straum after an agent has arrived at a new home and has had a thread created for it. The code to define the agent's actual behavior goes here: the agent can look for local monitors, create windows, query other servers for agents, or do whatever else the agent wishes to do. Most agent classes only override this one method. In addition, the agent can also provide a `getIconPixels` method to provide an image for the server's agent graph interface.

[3]Voyager does try to provide this service with its `ObjectListener` and `SystemListener` interfaces, but the information is incomplete.

```
protected void doBehavior() {
  displayCanvas = new FlashingDotCanvas();
  myServer.acceptComponentFrom(this, displayCanvas,
                               "Flashing Dot");

  this.findRemoteEventAgent();

  // Now main loop
  while (!timeToDie) {
    displayCanvas.updateDisplay();
    try {
      Thread.sleep(updateInterval);
    } catch (InterruptedException e) {}
  }
}

// callback method. Manage display
public synchronized void event(VBaseAgent sender,
                               Object event) {
  super.event(sender, event);
  displayCanvas.growAndWarmDot();
}
```

Figure 4.1: Flashing Dot Agent Code

As an example, figure 4.1 is the significant parts of the code for a flashing dot agent: the `doBehavior` method and the `event` method that is called by the remote agent for each event. Lines 2–4 create a display object for the flashing dot agent and asks the server to display it. Line 6 calls a method to find a remote event agent; this method is inherited from the agent support libraries and calls `queryAgents` on the remote server looking for agents that implement `EventSending`. Lines 8–14 implement the main loop of the agent: the agent simply updates its display every 100 milliseconds. Finally, lines 17–22 implement the event receiving interface for the agent. The remote event agent found in line 6 calls this method over the network: when the flashing dot agent receives this message, it changes its display to reflect the received event.

This code is not the entire software for the flashing dot agent, but it does give the flavor of implementing a Straum agent. A few bookkeeping methods have been omitted: the `getIconPixels` to give the agent a face, the `main` method for launching agents on the command line, and two accessor methods to manage the update interval parameter on the agent. In addition, a fair amount of graphical code (65 lines) is contained in `FlashingDotCanvas`, but this code is graphic design, not related to Straum or distributed computation. The 22 lines of code presented above are the essential parts of the agent, the code that is required for an agent to be part of Straum.

4.2.2 Agent Support

An agent could be written for Straum from scratch using only core Java and the basic Voyager support for mobile objects. It is important that Straum can work this way — because agent code can come in from anywhere on the network, there is no guarantee that an agent will inherit code from the "right" classes. However, normal application developers will use the various agent support classes provided with Straum.

All agents implemented in Straum today inherit code from `BaseAgent`, a core class that provides basic agent support. This class provides three variables: `myServer`, which contains a pointer to the local server the agent is living on; `vthis`, a `VObject` reference to the agent itself (the distributed equivalent to the `this` pointer); and a `timeToDie` flag that is used to politely notify an agent that it should die. `BaseAgent` also provides three methods to wrap up some of the complexity of the Voyager interfaces. The `launch` method is called to send an agent to another host. It takes one argument, the name of the host to move to, and moves the object to the new server. The `land` method is called on the other end after the move; it finds the server object by looking up its Voyager alias, sets up the `myServer` and `vthis` pointers, and notifies the server that the agent has arrived. Finally, the `run` method is the actual method invoked in the thread that the server creates for the agent; this simply turns around and calls the application agent's `doBehavior` method.

Several agents are simple enough that this is all the support they need. They just directly inherit from `BaseAgent`. These include the load average information agent, the memory usage agent, and the running graph agent. However, there are several information agents that all provide the same basic service, a simple passthrough to export event-oriented monitor data out to other agents on the network. These include the file growth information agent, the keystroke information agent, and the window creation information agent. These agents all inherit from `SignalPassthrough`, a class that simplifies the process of exporting events from a monitor.

The `SignalPassthrough` class maintains a list of listeners, other agents who are interested in the agent's event stream. Agents add themselves with `SignalPassthrough`'s `addListener` method: the information agent checks that the requesting agent really is capable of receiving events, adds it to the list, and reports the new listener to the user. The `signal` method is the other half of the `SignalPassthrough`: it takes a signal received from a monitor and re-propagates it as an event to the listening agents. The actual method call is a `OneWay` method: the information agent sends a message to each listener but does not check for either a return value or that the message was delivered correctly. This allows the information agent to be robust in the case where one of the listeners is slow or fails entirely, but does make event propagation potentially unreliable.

The final agent support class is `EventReceivingAgent`, a base class for agents that receive events from one remote source. This class implements a stub for the `event` method that simply notifies the server of a message being sent; actual agents override this method to provide application-specific behaviors when an event is received (such as the flashing dot's `growAndWarmDot` behavior). In addition, `EventReceivingAgent` implements a `findRemoteEventAgent` method: this method makes contact with the remote server, calls `queryAgents` there to look for agents that implement `EventSending`, and then adds itself as a listener to the remote event sending agent. This simple code is used by the flashing dot, random dot, and moving average agents to simplify the process of finding the right remote agent.

The rest of the code in the Straum agents package is all application specific agents and graphics code to implement agent displays. Writing new agents is very simple, requiring only a single `doBehavior` method to be implemented that is the agent's actual code. Now that the server is implemented, a major avenue of future work is to create new types of agents to implement new applications.

4.3 Evaluation and Comparison to Other Systems

The experience of writing Straum has been largely positive. The hardest part was getting the design correct: the metaphor of an ecology of agents, the split between agent and server functionality, and the carefully limited use of distributed messaging. Much needs to be done to take the existing Straum code base and turn it into a production environment; specific plans for that are discussed in chapter 5. However, as a whole, Straum has been an effective educational experience in building distributed applications.

The biggest weakness in the work to date is that the existing Straum applications do not reflect the larger goals of an ecology of distributed agents. The project started out as an attempt at building computation in the network, at making the Internet come alive with millions of interacting agents. Obviously, this is too ambitious a project for the scope of a master's thesis. The existing applications are very simplistic, too static and dataflow driven to really show the power and flexibility of this design approach. The application scenarios discussed in section 3.4 give several possible future avenues for advancing the research, for exploring the possibilities of more complex distributed systems.

The work done to date has helped develop the vision of open-ended distributed systems. There is much value in having a working system: the code base is developed enough to allow for those future applications to be built. In addition, in the course of this research much has been learned both about other related systems and what sort of infrastructure is necessary to build distributed agent applications on the Internet.

4.3.1 Voyager's Distributed Object Model

Distributed object programming is just now coming into common practice. There is agreement about the basic model but many differences about the particulars of implementation. One interesting difference is in the notion of "transparency," the attempt to provide a model so that talking to something on a remote computer looks exactly like talking to something on the local computer. On the face of it, transparency is ideal in distributed systems; if you can just hide all that network mess underneath the hood then programmers can write distributed code without having to work very hard.

However, in an influential paper [52] Jim Waldo argues that transparency in distributed systems is often dangerous and misleading. Waldo points out that the semantics of messaging remote computers is often quite different than local ones: there are additional failure modes and issues of concurrency, asynchrony, and consistency. Straum, with its location-oriented computing, has a model of agents that precludes full transparency. The locality of an agent is important as it dictates the kind of resources the agent has available to it. Locality also affects issues of trust and security.

The underlying distributed object package that was chosen, Voyager, is a mostly transparent system. If an agent has a `VObject` pointer to another agent the default behavior is entirely network transparent: messaging takes place the same way whether the object is local or remote. The transparent distributed object model of Voyager has several far-reaching implications. One of them is that it is very difficult to restrict which interfaces are published for remote agents. For example, the only method on a server that should be called from remote is `queryAgents`. But Voyager blindly exports all of the methods, thereby exposing inappropriate methods like `killAgent`. What Straum needs is a model where the programmer has to explicitly flag a method as exportable, much like programmers flag methods as `public` or `private`.

Transparency in Voyager also causes other problems. The default remote method call semantics are synchronous, to best duplicate conventional local programming. But many times this is exactly the wrong thing. For instance, an event notifying agent should not be hung because the listener has died. Voyager does provide alternate options for message semantics, but using them is a bit awkward. Moreover, because of the transparency of `VObject` it sometimes is not even clear that a method call *might* go out over the network, making it harder to restrict network exposure.

In summary, Voyager provides a completely open model of distributed object programming: anyone can talk to anyone else. But Straum has the exact opposite model, where agents are very careful as to whom they talk to and what interfaces they expose. It is possible to implement a more restricted communication model in Voyager but doing so breaks the fundamental design principles they chose. It may be possible that Voyager's transparent distribution is useful in some contexts, but in the location-oriented computing of an ecology of distributed agents a more restricted model is required. Other distributed object systems provide a better match.

4.3.2 Distributed Object Systems

The grand dame of distributed object systems is CORBA [37]. CORBA is quite large and complicated, providing a huge variety of different distributed object services. However, CORBA tends to be very static: CORBA does not allow mobile agents or easy runtime composition of objects.[4] The main strength of CORBA is that it is language independent, but for research projects such as Straum this capability is not a requirement. It is not clear how language independence would even operate with true mobile code.

There are many other Java distributed object systems available. One attractive alternative to Voyager is Java's own remote method invocation, RMI [65]. On its face RMI provides much the same basic functionality as Voyager, the ability to call a method on an object on another computer. But there is an important difference: classes that wish to be called remotely have to explicitly tag the individual methods they wish to be exported. While this makes RMI seem clumsy at first, it can be useful in requiring the programmer to carefully think about the methods that are exposed to remote calls. This discipline is desirable for Straum.

Unfortunately RMI does not provide some of the other capabilities that are in Voyager. The main weakness of RMI for Straum is that RMI has very little support for mobile objects. However, the mobile agent facilities needed by Straum can quickly be built out of Java's serialization and class loading capabilities. This approach would also allow experimentation

[4]There is an OMG mobile agents facility proposal underway.

with different implementations of mobile code and might help lead the way towards solving problems such as class versioning.

Caltech's Infospheres project [10] [11] also starts with a Java distributed object model. Out of the base of distributed objects, Infospheres adds composable component, allowing a user to publish active objects as part of their "information sphere" on the Internet. Straum is similar to the Caltech research in studying issues of scalability and reconfigurability by designing systems that allow composability through object capability specification and discovery services. Straum differs from Infospheres in that it is primarily agent based, with emphasis on loose confederations of mobile agents. The Infospheres project is notable for their guarantees of reliability and verifiability.

4.3.3 Mobile Agent Systems

Mobile agents are a new and fertile research field. There are a host of mobile agent systems out there, all providing similar services. Like Voyager, most of these systems are focussed largely on getting the mobile agents plumbing working, to make it possible to send a program from point A to B. But most systems develop some sort of model of computation that particularly addresses location and security. In the past mobile agent systems were developed in any language that has a runtime environment (most notably Dartmouth's Agent TCL, now a multi-language system called D'Agents [20]) but current research is mostly focussed on Java. Active mobile agent systems include the University of Stuttgart's Mole [69], IBM's Aglets [26], Mitsubishi's Concordia [57], General Magic's Odyssey [71], and ObjectSpace's Voyager [74].

All of the mobile agent systems implemented have some sort of notion of location of computation. This model seems implicit in mobile agent research: if an agent is going to move from one place to another, the system needs to have a concept of place. However, different projects address the concept of what distinguishes places differently. Voyager, for example, has no built-in support for distinguishing one place from another: Straum adds more locality to Voyager by adding the concepts of local resources and the local agent population. In contrast, Odyssey's Place abstraction or Mole's mobile agent model both account for location-dependent computation, albeit in a less explicit form than Straum's server-maintained set of resources.

One minor difference between Straum and many other mobile agents platforms is how an agent's movement is managed. Many systems such as Concordia have an explicit notion of an agent itinerary that lists the places the agent is to travel to. Straum does not support such a facility. Instead, agents themselves simply choose to where to travel to. In practice this does not make much of a difference in the power of the system, although perhaps in a future version of Straum itineraries could be added as an agent support class.

Another distinction between mobile agent systems is how they handle security, in particular protecting the host from the agents that move to them and restricting access to particular resources. Most systems have some basic protection for the host. For example, IBM Aglets uses the standard Java security manager interface to control agent operations, thereby preventing agents from accessing parts of the host that are not allowed. Concordia provides a very rich security model including protection for the host, access control for resources on the host, and protection for the agents themselves with a combination of code signing, and resource permissions. Straum currently has no security implemented at all, although the intended design closely resembles Concordia (section 5.2.4).

Finally, Straum seems to largely be alone in using an ecological model to describe a group of interacting agents. This idea was partly inspired by the NetTierra system [40], but that system is well outside the mainstream of mobile agents research. Instead, most mobile agent system examples work with a predefined set of agent interactions in the context of particular applications. As researchers become more comfortable with the basics of mobile agents, we can expect to see a flowering of different models for how to manage their interactions.

4.3.4 Group Communication

Straum's communication model is entirely peer-to-peer. Agents communicate entirely one to one and any sort of communication from one agent to a group is done by sending unicast messages one by one. This architecture is suitable for many of the applications discussed in chapter 3 where there are localized sources of information and relatively few receivers. But in applications where there may be a large number of agents all listening to one information source this simplistic approach will not be efficient or robust. Instead, some sort of multicast or tuplespace approach will be necessary.

One possible technology that could address this need is SoftWired's iBus system [61].

iBus provides an alternative to peer to peer communication, the *information bus*. This device is a channel that an object can send messages to. All objects subscribed to that bus receive the message. The core of iBus is IP Multicast, but the protocol stacks in iBus are nicely configurable to provide varying options for quality of service. Straum could easily add iBus as an extra capability on the server so that agents that needed iBus services could simply message out an iBus channel instead of a standard remote method call.

Another group communication paradigm that could be useful in Straum is the notion of “tuplespaces,” developed initially by Gelernter in his Linda system [18] and implemented recently by Sun as JavaSpaces [66]. A JavaSpace is like an abstract distributed database, a place where arbitrary objects can be placed and associated with each other. JavaSpaces provide a very elegant abstraction for relationships between objects. Straum could simply add JavaSpaces as another service, maintaining a JavaSpace on each service for objects to share. But a more interesting avenue might be to try to marry the Straum abstraction of a population of agents with the JavaSpace abstraction of groupings of objects. The models are different, but there is enough overlap that a combination might be fruitful.

4.3.5 Jini

In the last month this thesis was being written, Sun made a big announcement about Jini, a new distributed systems architecture being built for Java [51]. Jini is not yet released and information is still a bit sketchy, but what they have explained so far looks very promising as a way of building flexible distributed systems. The available white papers and specifications give enough information for a preliminary comparison between Jini and Straum.

Jini builds on the existing Java core, making particular use of serialization and RMI for a distributed object infrastructure and class loading for mobile code. Jini adds the notion of a “service,” a way to publish specifications on what a distributed object can do. For example, a hard drive might be a service, an object with a published interface for storing and retrieving files. Jini provides a discovery meta-service to enable remote objects to find relevant services.

Jini and Straum are very similar in supporting discovery. However, there are important differences between the two systems. Straum uses Voyager instead of RMI, although now the RMI model seems better suited for Straum (see section 4.3.2). Instead of services, Straum implements a two layer approach: monitors that are the raw resources only available on the local machine, and agents to mediate higher level access to the monitors. The discovery service of Jini is much like Straum's `queryAgents` interface, although the details of Jini's ontology are not yet available for comparison.

Jini is quite different from Straum in other important ways. First, Jini currently has no implementation of software agents at all: the metaphor of autonomy and adaptability is not there. Jini is more like a distributed objects system, not a distributed agents system. One implication is that Jini cannot have the split between monitors and agents that Straum uses to simplify access to local resources; this seems to be a weakness in Jini.

Closely related is that from what has been published so far, Jini has very little support for *location*. Jini's model is more like one computer system with many services registered in it, while Straum's model is to have many different computer systems in different locations, each with their own services, communicating together. There are hints in Jini's design documents that they intend Jini to be used across the entire Internet, but it is not clear yet how that will work.

In many ways Jini will be superior to what has been implemented in Straum. Jini adds a host of other useful distributed systems features including time leases on services, two phase commit transaction support for consistency, access control based security, JavaSpaces (section 4.3.4), and a more robust distributed event system. These capabilities are all presumably integrated into a simple, well-designed Jini API.

If Jini delivers on its promises it will be a very nice toolkit for creating distributed systems. The core distinction between Straum and Jini is that Jini has no notion of agents nor any way of localizing and coordinating active processes in the system. Therefore, Jini has no model of an ecology of agents. An important future research project will be to combine Jini and Straum, to add an ecology of distributed agents to Jini's rich distributed objects model.

Chapter 5

Conclusions and Future Work

This thesis has laid out a new way to build distributed software on the Internet. The core paradigm, the ecology of distributed agents, is a simple and effective way of organizing distributed systems. Straum, the first implemented version of an ecology of agents, allows distributed agent applications to easily be built and deployed. This work has uncovered many interesting issues in distributed systems, and also points to a program of future research towards building software that lives in the Internet.

5.1 Lessons for Distributed Systems

The process of designing and using Straum has revealed many object lessons in practical distributed systems. Among them are issues about transparency, the power of designing systems with clear compartmentalization of functionality, and the use of mobile agents to create open and flexible systems.

5.1.1 Transparency and Location

As discussed in section 4.3.1, transparency in a distributed system is not necessarily a desirable quality. Building a system so that programmers and users do not have to know about the location of computational objects is appealing on the surface: it would seem to simplify the task of understanding the system. But in practice it is too difficult to truly hide all of the complexities of a distributed system. And even if we develop technology someday powerful enough to smooth over all possible errors from distribution, it is still not clear that transparency is desirable.

The location of a computational process, in particular its local environment and input and output capabilities, is a fundamentally important property in a distributed system. Distributed systems should not wash over the distinction of place. The physical location of an agent dictates the resources it has access to. The logical location of an agent dictates the virtual environment it has access to, in particular its relationship to other agents. Two agents on the same computer are fundamentally different than two agents on different computers: they have faster and more reliable communication, they share a common frame of reference, and their collocation often indicates that there is some deeper relationship between the two agents. Location is an interesting property of computers and agents. Systems should not be designed so transparently that locality disappears.

5.1.2 Compartmentalization of Function

Straum divides the distributed software world up into two clear pieces: servers that manage local resources and agents that implement applications. This split, based on the split between the kernel and the applications in conventional operating systems, greatly simplifies the design of the system.

Servers are relatively heavyweight, stable code. In a deployed application the servers will be the legacy code, the things that cannot easily be changed once the system is installed. It is important that the servers be designed correctly early on. That means that the server's functionality needs to be simple. Straum servers are as simple as possible: a server simply accepts agents, tracks the agent population, and manages access to the local monitor and display resources. In the future Things That Think world one could imagine Straum servers being installed in every smart object. When every light fixture, appliance, and piece of furniture has a server in it, it will not be acceptable to require users to upgrade all of their servers to implement a new application. The Straum server design, in having a rigidly defined boundary on its function, makes it possible to imagine implementing the server correctly and installing it only once.

With the servers fixed, the agents are then free to change on the fly. Agents can be lightweight, ephemeral processes, living in a server for as long as they need and disposed of when the application is finished or it is time for an upgrade.

An additional compartmentalization in Straum is the relatively rigid boundaries between individual agents. The interaction between agents is tightly controlled, limited to interfaces that are negotiated through the query service. Agents are protected from the vagaries of the implementations of other agents, meaning that an agent's code can be changed without breaking every agent that knows about it. The boundary between agents and the server and the boundary between agents themselves simplifies the system design.

5.1.3 Mobile Agents

Straum uses a relatively new and controversial technology, mobile agents. The application of mobile agents in Straum so far is fairly limited. An agent can be elsewhere during its lifetime, but typically this is only done at an agent's birth. Even this simpleuse of mobility gives Straum flexibility: it is easy to send new functionality into a remote server. Combined with the careful design of servers, mobile code can potentially solve the problem of legacy software on the installed base. Install the server once on your computer, leave it running, and when you need to change how the computer operates simply load a new agent into it.

5.2 Future Implementation Work

The experience implementing Straum has focussed the vision of an ecology of distributed agents from a vague idea to a specific application platform. The most important components to enable this vision have been implemented. But there are many more things that need to be added to Straum to make it simpler to use, more powerful, and more robust for serious usage. This section discusses parts of the system that have been designed already but have not yet been implemented.

5.2.1 Technology upgrade: RMI and Jini

The next bit of work planned is to replace the Voyager distributed object system with RMI. As discussed in section 4.3.2, RMI's model of distributed objects seems to be better suited for

Straum than Voyager. Reimplementing Straum with RMI will help clean up the distributed messaging in Straum by requiring agent programmers to carefully think every time a message is sent over the network. It will also open up a pathway towards integrating some of Jini's functionality into Straum, and should also simplify licensing issues.

The main work in moving to RMI will be creating a new mobile object facility. This should not be too difficult as it mostly requires stitching together existing class loaders and serialization and providing a clean interface. Once that first piece of work is done, then the rest of the work is simply removing the Voyager-specific communication and replacing it with RMI-specific communication. Because remote communication is so limited in Straum, this task is not expected to be very difficult.

RMI cleans up the distributed messaging in Straum. It will also facilitate using Jini, which should provide nicer versions of several Straum features including resource discovery and event distribution. The support for leases, transaction support, and security will also make Straum a more robust application.

5.2.2 Semantic Ontology

Straum so far has only implemented a syntactic ontology, the type system. A semantic ontology system is also necessary so that agents can easily distinguish between, for example, two event sending agents. Such a system is relatively simple: the idea is to simply augment every agent with a `getMeaning` call that returns a list of strings. The strings will be arbitrary nouns. For example, a keystroke information agent might return "keystrokes." The querying agent can then interpret this extra information however they want to. This simple approach will be useful, but it does not solve larger questions of coordinating ontologies. That issue is discussed in section 5.3.5.

5.2.3 Agent Persistence

Currently, if a user shuts down their Straum server all of the agents disappear. This restriction is fine for limited use, but it would be nice if the agent population were persistent so that they could be restored when the server starts up. In theory this is simple; just serialize the agents and write them to disk. One possible issue is class versioning; one has to insure the right version of code is loaded for the right version of agents. Java now has some limited support for versioning that will probably need to be extended.

5.2.4 Security

Straum so far has no security at all. It is already approaching the point where this is unacceptable because it makes it dangerous to have long-running Straum servers running on the Internet. The first part of Straum security will be a simple host-based system, restricting who is allowed to connect to the server. It should be possible for a user to restrict access to the server so that, for example, only people inside the Media Lab can connect. This simplistic security mechanism is quite helpful and very easy to implement.

The second level of security to implement is a custom Java security manager that creates a sandbox for remote agents [36]. This security system will restrict what any agent coming into the system can do. For example, it would allow communication from the agent to the Server object but prevent an agent from directly accessing the hard drive. Basic sandboxes are a well supported capability in Java although it is difficult to correctly define the policy so as to allow agents to be useful and yet secured.

The problem with this sort of sandbox is that it is not very specific; in general, all agents are treated the same. What one would like is a policy where particular agents have particular capabilities. For example, agents from my friends would have more powers than agents from a stranger. Java 1.1 allows for code signing so that one can determine who is responsible for the agent's program and grant permissions based on how much you trust the software. The Concordia project notes that code signing is not enough and that one needs user authentication, since often one wants to know who the agent belongs to, not who wrote the code [53]. Using pieces of code signing and user authentication, combined with the upcoming Java 1.2 policy architecture [19], it will be possible to give individual agents specific capabilities depending on what user they represent.

Sandbox techniques will allow the Straum server to limit what information resources an agent has access to: what monitors, what display capabilities. But they do not currently allow Straum to control how much CPU time an agent takes or how much memory it uses. This limitation is fundamental in the design of Java; it is easy to control what methods an object can call but not what raw resources are used. Straum will eventually need to implement these sorts of restrictions to prevent denial of service attacks on servers. It might even enable a marketplace for spare CPU cycles such as the Enterprise system [31]; when a computer is idle, its Straum server could advertise to allow paying agents to run CPU intensive operations. Unfortunately, doing any sort of CPU or memory usage accounting will require augmenting the Java VM.

5.3 Future Design Work

In addition to these various capabilities that are fairly well understood and mostly need implementation, there are several features that would be good to have in the system that are not yet well understood. The design of these functions has not been worked out, nor have the implications of adding them. This work represents the most exciting future research for Straum as it means expanding the capabilities of the system itself.

5.3.1 Dynamic Monitors

The current set of monitors built into a Straum server is hard coded: each server process loads its own monitors and builds interfaces such as `getKeystrokeMonitor` to access them. This approach is not ideal because it means that special servers have to be compiled for every possible set of monitors. Straum needs a way for servers to load monitors at runtime and an interface so that local agents can find the monitor they need. Loading a monitor and allowing agents to find it is quite similar to loading an agent and allowing other agents to find it; some of the same code could be used to support both functions. However, there is a danger that this approach might confuse the boundary between agents and monitors. This is not appropriate as monitors should not have all the capabilities and complexities of agents. As more experience with deploying Straum is gathered, the current ad hoc system can be replaced with one designed to meet observed needs.

5.3.2 Agent Parameters

Many of the agents implemented in Straum have obvious parameter settings to tune their behavior. For example, display agents typically have update rates, the moving average agent has parameters for the size of data windows, and the file growth agent has the actual file it is watching. Straum currently hard codes these parameters, but that is clearly not a proper solution.

There are several options for parameterizing agents. One solution is to provide an interface for the user to set parameters — this is clearly useful for the display agents. Another option is to have agents have a defined interface for setting parameters, so for example a remote agent could call `setWindowSize` to configure a moving average agent for its purposes. That solution raises issues on who has permission to reconfigure an agent. A third option is to not parameterize agents at all, and instead create new agents with new settings on the fly and send that in. So if a user wanted to watch a particular file, instead of configuring a remote file watching agent he or she would simply create a new file watching agent with the right filename and send it to the remote host. This solution has nice properties of agent identity but could quickly become inefficient. In practice, some mixture of these solutions will probably be necessary.

5.3.3 Server Directories

Straum servers have a location, a place in the network. Currently the name for a server's location is simply its IP address. This naming is functional but a bit awkward to work with; Straum would benefit from a naming system that is more abstract, so a user can specify "Nelson" instead of `18.85.16.104`. DNS names alone are not enough: an interesting service might have a dynamic IP address. Some sort of directory of servers would be useful. A simple implementation would be a centralized server, but a distributed directory like DNS or a group model like JavaSpaces would be scalable.

5.3.4 Server Neighborhoods

Straum has a concept of "place," the individual server. However, it has no concept of geography, of topology, of interconnections between servers. As with the Internet itself, every Straum server is right next door to every other server, merely one message away. There is a real convenience to this lack of order in computer relationships, it makes any possible communication easy. However, it also makes for a completely unstructured environment.

One possible way to structure the environment is to introduce an idea of “neighboring” servers, Straum islands that are “close” in virtual space. For example, I might define my Straum server to be next-door to those of my personal friends and my workgroup. Agents would then be restricted to only being able to communicate with neighboring servers. Naturally, these neighborhoods should be reconfigurable so that servers could choose to become neighbors in order to facilitate communication.

There are many reasons why a spatial restriction on Straum might be useful. First, it will simplify the interface of the system; when a user creates a new agent and tells it to monitor someone, he or she could simply chose from the list of neighbors. A server topology would also help structure the space; much like the locality of agents on one computer is a useful property for organizing and understanding a system, the concept of neighborhood might help organize more complex interactions between groups of Straum agents. Finally, a “neighborhood” is a simple hook for defining security policy: agents from one’s neighbors might be more trusted. A special neighborhood guard server could even protect a group of users by screening agents that want to come in through the neighborhood gate. Adding more structure to the Straum network could allow more sophisticated organization of the global agent population.

5.3.5 Ontology Coordination

Straum takes an ad hoc approach to ontologies — new types of things can simply be defined and used without any central coordinator. This choice scales nicely and avoids the difficult question of defining an a priori ontology of possible agent types. However, it also shifts the hard work of figuring out what an entry means to the querying agent. Ie: if a new agent publishes that it has information of type `SpecialObject`, or that the information means “widgets”, how will the remote agent understand what these nouns mean?

The belief underlying Straum’s ad hoc ontology is that it is not necessary to solve this problem, at least in the short run. Instead, people will just go on defining entries in the ontology just like they define new words in a language; other agent designers will then need to actively keep abreast of current developments in Straum ontologies to make sure their agents interoperate. This approach is not entirely unprecedented. For example, XML has an equivalent ontological problem in the proliferation of document type definitions (DTDs) [7].

It remains to be seen whether XML will result in a Tower of Babel of millions of different kinds of documents, or whether developers will converge on a common, ad-hoc ontology. If Straum proves popular enough that these concerns are palpable, then it will be successful indeed!

However, some sort of system should be built in Straum to simplify ontology coordination. What is needed is a way for agents to easily tell the world when they have defined a new entry so that other agents can know about the new term. This registry will probably need to be a centralized resource. For the syntactic ontology this is relatively simple: agents can just report their Java types to the ontology registry and have the information recorded; the structure of Java's type system will help coordinate entries. The semantic ontology is much more free-form however, and it is possible that conflicts will arise where two agents register the same noun with a different meaning or two different nouns for the same meaning. Guidance for solving this problem can be taken from work done towards using XML for enabling active applications [6] as well as from KQML research, in particular the Ontolingua system [16].

5.3.6 Distributed Agent Construction Kits

Straum's visual interface to agents strongly suggests graphical programming environments where programmers can build applications by simply drawing wires between components. However, Straum's view is currently read-only. The links between agents are created deep in the agents' own code, the user cannot link two agents up visually. One avenue of future design work is to build a visual programming system into Straum so that users could easily connect agents on different machines simply by drawing links between them. This interface would replace the currently clumsy way of telling an agent explicitly which host to look at and what kind of information to look for, thereby enabling a simpler construction of applications out of agents.

A more ambitious project is to open up the construction of the agents themselves, to provide a high level interface to allow users to specify an agent's behavior. This approach would greatly improve the diversity of types of Straum agents as users could work with a simpler abstraction for building agents rather than writing low-level Java code for the `doBehavior` method. Designing this properly will take more research in programming paradigms, perhaps borrowing component assembly like JavaBeans [64] or workflow templates for assembling agent plans [34].

5.3.7 Economic Coordination

Straum creates a system that enables agents to find each other and enter into relationships. But Straum does not say anything about *why* agents would want to communicate, what the motivations and agreements are that prompted users to create communicating agents in the first place. In the applications envisioned so far there has typically been a user or group of users who have a preexisting need, a framework that they build agents into. But Straum is also intended to enable more fluid kinds of applications, ones in which agents themselves step forward and take over some work without a human planning the interaction.

One way to coordinate and motivate this type of behavior is to put economics into the system. For example, an agent might charge for its services; if an agent wants access to some sort of information source on another server, it would have to pay an agent on the remote system to provide that data. An economic system would give an incentive for agents to provide services: they can make money for their owners. Furthermore, economics can be a way of coordinating a distributed system. For example, if there is a cost associated with communication then an agent has an incentive to communicate more efficiently to save money. A new agent that could perform a service more efficiently could step in and offer its services. A marketplace of agents would be quite a dynamic system. Existing research in market based control [13], in particular conceptions such as Agoric computing [33], point to how economics can help coordinate a complex computational system. Straum could be an excellent platform on which to test these ideas.

5.3.8 Evolving Agents

Straum uses mobile agents to aid configurability of the system. But mobile agents could potentially make the system much more than just an application that people can easily reconfigure. If Straum servers become widely deployed the network as a whole will become a medium for agents to live in, a natural environment for software.

One possibility is that this new environment would have software evolving inside of it.

We already have a certain kind of software evolution; new versions of software are released every day, and users upgrade them or not as the mood strikes them. Eventually people stop running the old version and it dies out. Similarly, in Straum as people write new versions of agents they will be sent out into the system. If these new versions are better (more robust, more efficient, more featureful), then they will naturally replace the older versions. And so there would be a kind of human-aided evolution of the agent population.

This sort of evolution is not automatic; it requires agent authors to write new versions by hand and send them out. But what if the agent code itself is produced algorithmically? It is possible to have an agent's code be under the control of a genetic algorithm so that the agent's implementation changes dynamically. Combine this with a selection mechanism for "fit" agents, and Straum would have an autonomously evolving agent population. Naturally it will be quite difficult to actually make this evolution process effective but because the agent interfaces are so constrained it is not impossible. The implications of this are huge: on one hand you could have a system where more efficient programs just magically appear in the execution of the system. On the other hand, agent evolution might quickly get out of hand, producing hordes of useless software that somehow exploited unforeseen niches in the ecosystem. Online evolution is obviously not going to be a solution in the near term, but it is an exciting topic worth pursuing. And Straum's design, with its explicit ecology of agents, is a natural testbed for evolutionary computation.

5.4 The Future of Internet Systems

The purpose of Straum is not just to enable distributed dataflow applications. Straum is a work towards creating a new form of computation, a new paradigm for how we use the Internet. The goal is to make software more fluid, to put active processes into the Internet and make more complex and powerful systems.

The best way to advance this agenda with Straum is to build applications with it, to create practical things on top of the distributed agents infrastructure. We need more experience with agent based applications on the Internet; Straum is good enough to be a starting place. Section 3.4 discusses some plans for applications to build on top of the Straum system today. These applications, in particular the Things That Think related ones, are underway now.

However, it is also important to think beyond those relatively static applications, to building larger and more fluid applications where agent interactions are not predetermined, where the software could possibly surprise us. The shape of these applications, what they do, is not completely apparent. Straum has helped to develop a vision of the future of Internet systems, to explore this paradigm of an ecology of distributed agents. Through refining Straum and building more applications, this vision will become more complete.

Appendix A

Agent Bestiary

A.1 Information Agents

`MemoryUsageInfoAgent`

INPUT: None.

OUTPUT: Floating point number representing the fraction of memory used.

BEHAVIOR: Finds the memory usage monitor and quits its thread. Responds to `getFloat`.

`LoadAverageInfoAgent`

INPUT: None.

OUTPUT: Floating point number representing the current number of jobs waiting to run on the system.

BEHAVIOR: Finds the load average monitor and quits its thread. Responds to `getFloat`.

`FileGrowthInfoAgent`

INPUT: None.

OUTPUT: An event every time a watched file grows.

BEHAVIOR: Installs itself as a signal listener with the file growth monitor and quits its thread. Accepts listening agents and sends them events.

`KeystrokeInfoAgent`

INPUT: None.

OUTPUT: An event every time a key is pressed.

BEHAVIOR: Installs itself as a signal listener with the keystroke monitor and quits its thread. Accepts listening agents and sends them events.

`WindowInfoAgent`

INPUT: None.

OUTPUT: An event every time a window is created.

BEHAVIOR: Installs itself as a signal listener with the window creation monitor and quits its thread. Accepts listening agents and sends them events.

A.2 Data Analysis Agents

`MovingAverageAgent`

INPUT: Events.

OUTPUT: Floating point number representing the moving average of the number of events received in its history.

BEHAVIOR: Adds itself as a listener to an event generating agent on the remote host. Updates a count every time it receives the event, and runs a thread to update the history. Returns the current moving average in response to `getFloat`.

A.3 Display Agents

`FlashingDotAgent`

Input: Events.

Output: None.

Behavior: Adds itself as a listener to an event generating agent on the remote host. Creates a canvas with a dot in it. When an event is received, the agent makes the dot pulse. Also keeps a record of how many events have come recently: if there is a lot of activity, the agent warms the dot from blue to red.

`RandomDotAgent`

Input: Events.

Output: None.

Behavior: Adds itself as a listener to an event generating agent on the remote host. Creates a blank canvas. When an event is received, the agent draws a dot with random placement and colour. The agent slowly fades dots over time until they disappear. This display is closely modeled after the display in ChatCircles [15].

`RunningGraphAgent`

Input: Events.

Output: None.

Behavior: Finds a source of floating point numbers on the remote host. Creates a canvas for a graph. Periodically asks the remote agent for a floating point number and plots it on the graph.

Appendix B

Class Hierarchies

Figure B.1 is a diagram of the class hierarchy for the Straum server and system information packages. Figure B.2 is a diagram of the class hierarchy for Straum agents.

The diagrams follow the convention of the Java Class Libraries documentation [9]. A square box represents a class, a rounded box represents an interface. A small circle on the right side of a class box indicates the class is abstract. Solid lines show the inheritance hierarchy, dotted lines represent implementation of an interface.

For convenience, the package names have been shortened in these diagrams. The actual package names are

- `edu.mit.media.nelson.straum`
- `edu.mit.media.nelson.straum.agent`
- `edu.mit.media.nelson.straum.sysinfo`
- `edu.mit.media.nelson.straum.sysinfo.unix`
- `edu.mit.media.nelson.straum.sysinfo.win`

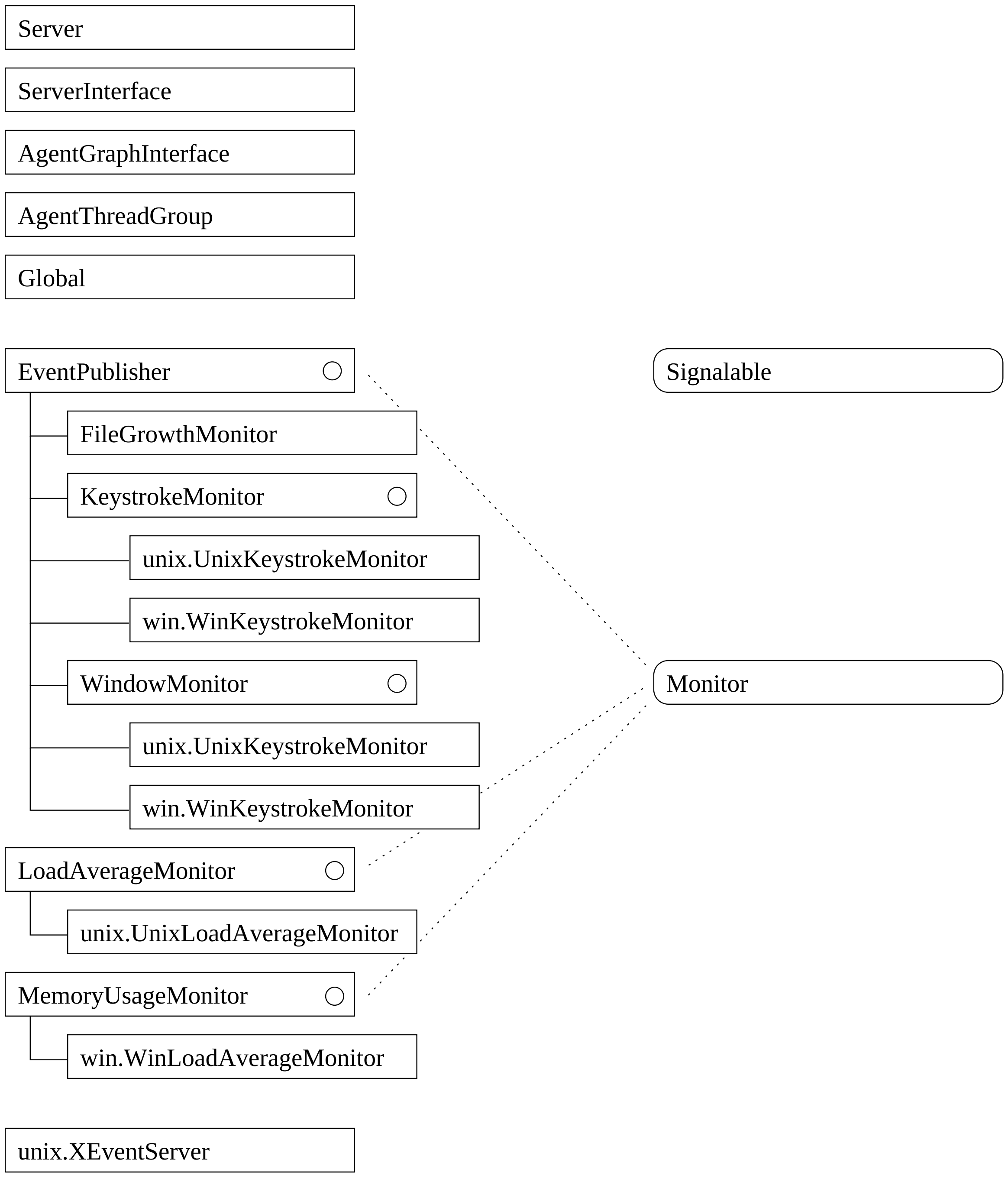


Figure B.1: Class hierarchy for Straum server and system information

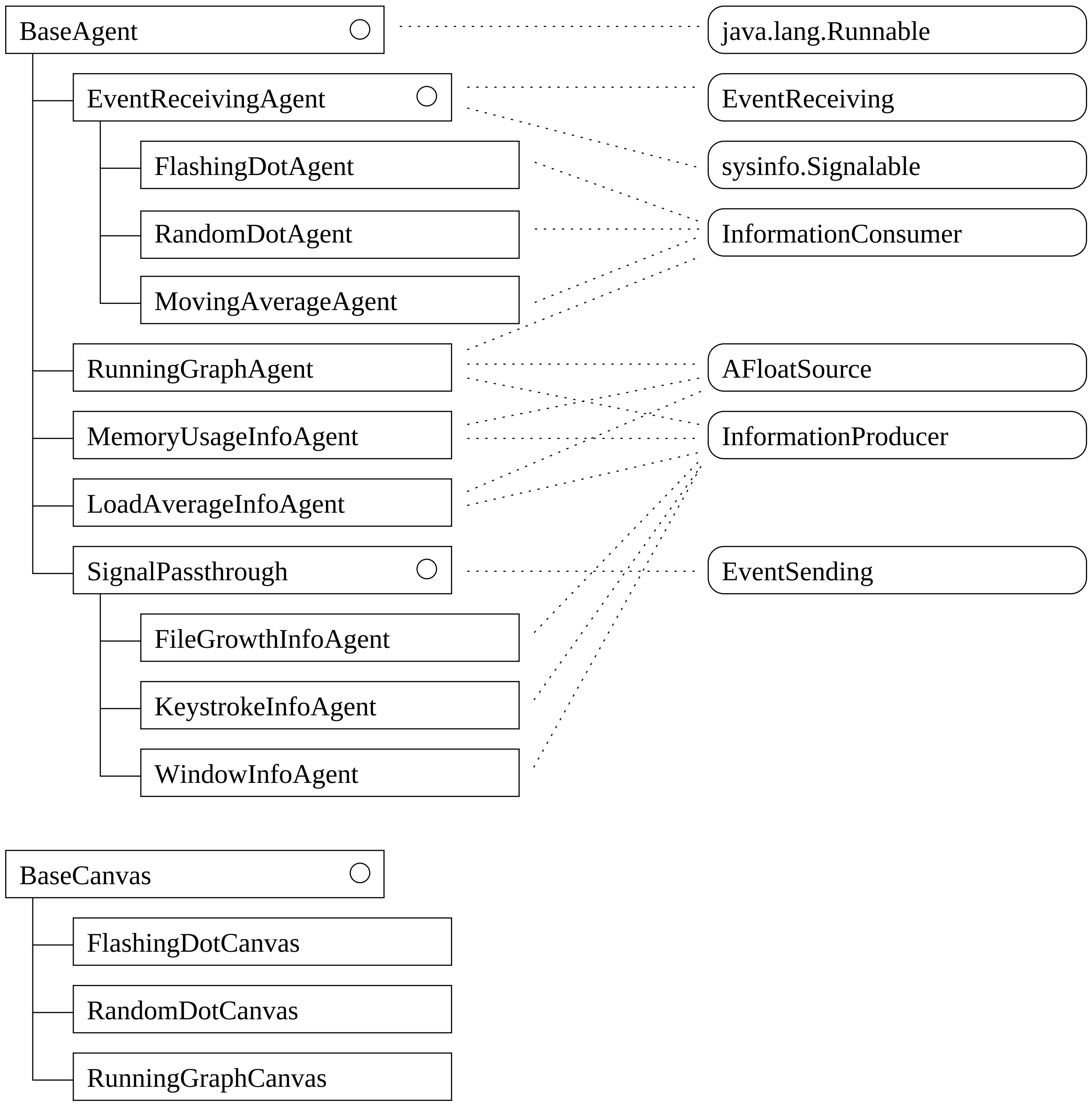


Figure B.2: Class hierarchy for Straum agents

Bibliography

[1] Ross J. Anderson. The Eternity Service. In *Proceedings of PragoCrypt '96*. Czech Technical University Publishing House, 1996. http://www.cl.cam.ac.uk/users/rja14/eternity/eternity.html

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. ISBN: 0–201–63455–4.

[3] Derek Atkins, Michael Graff, Arjen Lenstra, and Paul Leyland. The Magic Words are Squeamish Ossifrage. In *AsiaCrypt '94*, 1994.

[4] Mario Baldi, Silvano Gai, and Gian Pietro Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Proceedings of the First International Workshop on Mobile Agents*, Berlin, April 1997. http://www.polito.it/˜picco/papers/ma97.ps.gz

[5] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company, 1996. ISBN: 0137195842. http://www.browsebooks.com/Birman/index.html

[6] Jon Bosak. XML, Java, and the Future of the Web, March 1997. http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html

[7] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML). Technical Report PR-xml-971208, W3C, December 1997. http://www.w3.org/TR/PR-xml-971208

[8] G. Di Caro and M. Dorigo. Mobile Agents for Adaptive Routing. In *Proceedings of the 31st Hawaii International Conference on Systems*, January 1998. ftp://iridia.ulb.ac.be/pub/dorigo/conferences/IC.22-HICSS31.ps.gz

[9] Patrick Chan, Rosanna Less, and Douglas Kramer. *The Java Class Libraries*, volume 1. Addison Wesley, 2nd edition, 1998.

[10] K. Mani Chandy, Joseph Kiniry, Adam Rifkin, and Daniel Zimmerman. Framework for Structured Distributed Object Computing. 1997. http://www.infospheres.caltech.edu/papers/framework/framework.html

[11] K. Mani Chandy, Adam Rifkin, Paolo A.G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman. A World-Wide Distributed System Using Java and the Internet. Technical report, Caltech Computer Science, 1996. Caltech CS Technical Report CS-TR-96-0. http://www.infospheres.caltech.edu/papers/chandy_etal/hpdc.html

[12] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are they a Good Idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. http://www.research.ibm.com/massive/mobag.ps

[13] Scott H. Clearwater. *Market-Based Control, A Paradigm for Distributed Resource Allocation*. World Scientific Publishing Co., 1996. ISBN: 9810222548.

[14] Judith Donath. *Inhabiting the Virtual City: The Design of Social Environments for Electronic Communities*. PhD thesis, MIT Media Arts and Sciences, 1997. http://judith.www.media.mit.edu/Thesis/

[15] Judith Donath, Karrie Karahalios, and Fernanda Viegas. Visualizing Conversations. In *Proceedings of the 32nd Hawaii International Conference on Systems*, January 1999. http://www.media.mit.edu/˜fviegas/circles/new/index.html

[16] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua Server: A Tool for Collaborative Ontology Construction. Technical Report KSL-96-26, Knowledge Systems Laboratory, Stanford University, September 1996. ftp://ksl.stanford.edu/pub/KSL_Reports/KSL-96-26.ps

[17] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 1995. http://www.mpi-forum.org/docs/docs.html

[18] David Gelernter. *Mirror Worlds*. Oxford University Press, 1991.

[19] Li Gong. Java Security Architecture (JDK 1.2), June 1998. http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html

[20] Robert Gray. *Agent Tcl: A Flexible and Secure Mobile-Agent System*. PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Available as Dartmouth Computer Science Technical Report TR98-327. http://www.cs.dartmouth.edu/˜agent

[21] David Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, Computer Laboratory, University of Cambridge, June 1997. http://www.cl.cam.ac.uk/users/dah28/

[22] Brian Hayes. Computing Science: Collective Wisdom. *American Scientist*, 1998. http://www.amsci.org/amsci/issues/Comsci98/compsci1998-03.html

[23] B. A. Huberman, editor. *The Ecology of Computation*. Elsevier Science Publishers, 1988.

[24] Hiroshi Ishii and Brygg Ullmer. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '97)*, pages 234–241. ACM Press, March 1997. http://tangible.media.mit.edu/˜ullmer/papers/tangible-bits.pdf

[25] Kevin Kelley. *Out of Control*. Addison-Wesley, 1994.

[26] Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench: Programming Mobile Agents in Java, September 1996. http://www.trl.ibm.co.jp/aglets/whitepaper.htm

[27] Chris Langton, editor. *Artificial Life (Proceedings of the First International Conference*. Addison-Wesley, 1987. ISBN: 0–201–09356–1.

[28] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, April 1998. http://www.neci.nj.nec.com/homepages/lawrence/papers/search-science98/index.html

[29] Andrew Leonard. *Bots: The Origin of New Species*. Hardwired, 1997.

[30] Pattie Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):31–40, July 1994. http://pattie.www.media.mit.edu/people/pattie/CACM-94/CACM-94.p1.html

[31] Thomas W. Malone, Richard E. Fikes, Kenneth R. Grant, and Michael T. Howard. Enterprise: A Market-like Task Scheduler for Distributed Computing Environments. In B. A. Huberman, editor, *The Ecology of Computation*, pages 177–206. Elsevier Science Publishers, 1988.

[32] Network Working Group Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. RFC 1057, June 1988. http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1057.txt

[33] Mark S. Miller and K. Eric Drexler. Markets and Computation: Agoric Open Systems. In B. A. Huberman, editor, *The Ecology of Computation*, pages 133–176. Elsevier Science Publishers, 1988. http://www.webcom.com/˜agorics/agorpapers.html

[34] Kazuhiro Minami and Toshihiro Suzuki. Java-Based Moderator Templates for Multi-Agent Planning, October 1997. Presented at OOPSLA '97 Workshop on Java-based Paradigms for Agent Facilities. http://www.trl.ibm.co.jp/aglets/jmt/oopsla97/jmt-oopsla97.html

[35] Alexandros Moukas, Kostas Chandrinos, and Pattie Maes. Trafficopter: A Distributed Collection System for Traffic Information. In *Proceedings of Cooperative Information Agents '98*, volume 1435 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1998. http://lcs.www.media.mit.edu/projects/trafficopter/

[36] Scott Oaks. *Java Security*. O'Reilly & Associates, Inc., 1998. ISBN: 1–56592–403–7.

[37] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0*. Object Management Group (OMG), 2.0 edition.

[38] Alex P. Pentland. Smart Rooms. *Scientific American*, April 1996. http://www.sciam.com/0496issue/0496pentland.html

[39] Rosalind W. Picard and Jennifer Healye. Affective Wearables. *Personal Technologies*, 1(4):231–240, 1997. MIT Media Lab Vismod tech report 467. ftp://whitechapel.media.mit.edu/pub/tech-reports/TR-467.ps.Z

[40] Tom S. Ray. A Proposal to Create a Network-wide Biodiversity Reserve for Digital Organisms. Technical report, ATR, 1995. http://www.hip.atr.co.jp/˜ray/pubs/reserves/reserves.html

[41] Maria Redin. Marathon Man. Master's thesis, MIT Department of Electrical Engineering, 1998. http://ttt.www.media.mit.edu/SF/

[42] Mitchel Resnick. *Turtles, Turmites, and Traffic Jams*. MIT Press, 1994.

[43] Marshall T. Rose. *The Simple Book: An Introduction to Networking Management*. Prentice Hall, 1996. ISBN: 0134516591.

[44] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of Encounter*. MIT Press, 1994.

[45] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997. ISBN: 0–417–19381–X.

[46] Thad Starner, Steve Mann, Bradley Rhodes, Jeffrey Levine, Jennifer Healey, Dana Kirsch, Rosalind W. Picard, and Alex Pentland. Augmented Reality Through Wearable Computing,. *Presence*, 1997. MIT Media Lab Vismod tech report 397. http://wearables.www.media.mit.edu/projects/wearables/

[47] Hong Z. Tan and Alex Pentland. Tactual Displays for Wearable Computing. In *The First International Symposium on Wearable Computers*, pages 84–89. IEEE Computer Society, 1997. ISBN: 0–8186–8192–6.

[48] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997. http://www.tns.lcs.mit.edu/publications/ieeecomms97.html

[49] Vernor Vinge. *A Fire Upon the Deep*. Tor Books, 1993. ISBN: 0812515285.

[50] Jan Vitek and Christian Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222. http://cuiwww.unige.ch/˜ecoopws/tpi

[51] Jim Waldo. Jini Architecture Overview. Technical report, Sun Microsystems, Inc., 1998. http://java.sun.com/products/jini/

[52] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, Heidelberg, April 1997. http://www.sunlabs.com/techrep/1994/abstract-29.html

[53] Tom Walsh, Noemi Paciorek, and David Wong. Security and Reliability in Concordia. In *Proceedings of the 31st Hawaii International Conference on Systems*, January 1998. http://www.meitca.com/HSL/Projects/Concordia/HICSS98_Final.htm

[54] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, pages 94–101, September 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[55] James E. White. Telescript Technology: Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996. http://www.genmagic.com/agents/Whitepaper/whitepaper.html

[56] AOL Instant Messenger. http://www.aol.com/aim/home.html

[57] Concordia — Java Mobile Agent Technology. http://www.meitca.com/HSL/Projects/Concordia/

[58] Ding! http://www.activerse.com/ding/dingintro.html

[59] distributed.net: The Fastest Computer on Earth. http://www.distributed.net/

[60] dist-obj FAQ and mailing list archives. http://www.infospheres.caltech.edu/mailing_lists/dist-obj/distobjgroup.html

[61] iBus: The Java Multicast Object Bus. http://www.softwired.ch/ibus/

[62] ICQ. http://www.icq.com/

[63] Japhar — The Hungry Java Runtime. http://www.hungry.com/products/japhar/

[64] Java Beans home page. http://java.sun.com/beans/

[65] RMI: Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html

[66] JavaSpaces White Paper. http://java.sun.com/products/javaspaces/

[67] Kaffe OpenVM. http://www.transvirtual.com/kaffe.html

[68] The Great Internet Mersenne Prime Search. http://www.mersenne.org/prime.htm

[69] Project Mole — Mobile Agents, March 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html

[70] Mr. Java. http://mrjava.media.mit.edu/

[71] General Magic's Odyssey. http://www.genmagic.com/technology/odyssey.html

[72] SETI@Home. http://setiathome.ssl.berkeley.edu/

[73] Things That Think — MIT Media Lab. http://ttt.www.media.mit.edu/

[74] Objectspace Voyager Core Package Version 1.0 Technical Overview, 1997. http://www.objectspace.com/voyager/whitepapers/VoyagerTechOview.pdf