Hive: Distributed Agents for Networking Things

Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes
MIT Media Lab E15–305 20 Ames Street Cambridge, MA 02139 USA
`<nelson@media.mit.edu>` `http://hive.media.mit.edu/`

August 3, 1999 Appearing in ASA/MA '99

Abstract

Hive is a distributed agents platform, a decentralized system for building applications by networking local system resources. This paper presents the architecture of Hive, concentrating on the idea of an "ecology of distributed agents" and its implementation in a practical Java based system. Hive provides ad-hoc agent interaction, ontologies of agent capabilities, mobile agents, and a graphical interface to the distributed system. We are applying Hive to the problems of networking "Things That Think," putting computation and communication in everyday places such as your shoes, your kitchen, or your own body. TTT shares the challenges and potentials of ubiquitous computing and embedded network applications. We have found that the flexibility of a distributed agents architecture is well suited for this application domain, enabling us to easily build applications and to reconfigure our systems on the fly. Hive enables us to make our environment and network more alive.

This paper is dedicated to the memory of Mark Weiser, a visionary and a guide.

1 Computation: ubiquitous, distributed

Computation is changing. Computers are no longer isolated number factories; they are on our desks, on our wrists, in our pockets, and embedded in devices all over our homes and offices. Computers are constantly communicating with each other via wireless networks, LANs, and the Internet. This new reality of computation demands new paradigms for building computer systems.

We believe that software agents are an important abstraction for building distributed systems. Software agents — small, autonomous, self-describing programs — are an excellent building block for complex open-ended networked applications. We have created Hive, a software system implementing an ecology of distributed agents. We are applying Hive to creating applications for our new networked computer reality, focusing on connecting embedded computers, or Things That Think (TTT).

The design of Hive has been strongly influenced by the World Wide Web. We take two lessons from the success of the Web: the importance of decentralized systems and the value of simple abstractions. The Web is deeply decentralized in that any web page can be linked to any other without any central administration; similarly, Hive systems are built entirely out of peer-to-peer relationships between agents. The key Web abstraction, the web page, is mirrored in Hive by the software agent. But while the Web is a distributed system for static data, Hive is for dynamic computation. We wish to make the Internet alive.

1.1 Ecologies of distributed agents

Our metaphor for building networked systems is the *ecology of distributed agents* [21]. In an ecology of agents, an application is created out of the interaction of multiple agents across a network. Each agent is located in a particular place (in Hive, called a *cell*), and uses various local resources (*shadows*). Agents communicate with each other to share information and access to resources. An application is made from the communications and actions of agents.

Why do we use agents? From a programmer's perspective, a Hive agent is just a distributed object with an execution thread. But in Hive agents are more than just objects, they are the building block of an active distributed system. Agents provide a conceptual wrapper for several useful ideas:

- Agents are autonomous: they can be sent into a system and entrusted to carry out goals.
- Agents are proactive: they encapsulate computational activity.

- Agents are self-describing: an ontology of agent capabilities can be used to describe and discover available services.
- Agents can interact: they can work together to complete a task.
- Agents can be mobile: mobile agents provide a simple abstraction for complex, dynamically distributed systems.

The ecology of distributed agents is a decentralized system. We believe decentralization is essential to allow a system to grow. Decentralization comes at a cost: agents are responsible for locating the resources they need, finding each other, and negotiating their relationships. Hosts are responsible for protecting themselves from unwanted agents and keeping their own consistency. And there is no single place one can point to and say "this is the center of the system, this is where it comes together, this is how we know that it is working correctly." Such challenges are the reality of distributed systems today. The details of how we manage these problems are described throughout this paper.

Hive is an embodiment of an ecology of distributed agents. This paper describes specifics of how Hive is implemented, the applications we have built with Hive, and the lessons we have learned from testing Hive in real systems. But first, we introduce our primary application domain.

1.2 Things That Think

The goal of the Things That Think project at the Media Lab is to take the power of computers and networks and put them into our everyday objects. TTT enhances the physical world with computation and communication. TTT makes our computer networks more complex and capable as every room, appliance, and even light socket begins to "think." A full explication of Things That Think is beyond the scope of this paper; a detailed description is available in [12].

This agenda is shared by many researchers. Mark Weiser details related ideas under the name "ubiquitous computing" [32]. In the consumer world, embedded computers are quite common and technologies such as Bluetooth are bringing networking to consumer devices. Toys are a leader of this trend, with networked toys like Microsoft Actimates or Furby and industry initiatives such as Intel Play [35]. And the computer industry is tackling the problems of networked embedded systems applications with technologies such as Inferno [9] and Jini [2].

Ubiquitous embedded networking presents a variety of challenges for application infrastructures. Devices are scattered all over the network, so the system must be fully distributed. Administrative overhead has to be low, meaning the system cannot be too centralized. New devices need to easily be added, requiring an open architecture. Finally, people will want to add new devices and create new applications, doing things with the system that cannot be foreseen by its designers. The system must be *flexible*.

An ecology of distributed agents is a good match for these requirements. Hive's use of agent ontologies and mobility makes it flexible. Agent descriptions allow new capabilities to be announced, discovered, and used. Mobile code allows for parts of the system to be reprogrammed dynamically, for the network itself to evolve as necessary.

Hive is a good infrastructure for TTT, and TTT is a good problem for Hive. It is not enough to build an agent system in the abstract; the system has to be tested, used, understood. Building TTT applications with Hive has helped us validate its design principles and taught us many lessons about the design of agent-based distributed systems.

2 The Hive architecture

Hive consists of three components: cells, shadows, and agents. The Hive network is a decentralized collection of *cells*. A Hive cell is the analog to a web server, a program running on a specific computer with a published network address. Each cell contains a set of local resources called *shadows* that encapsulate capabilities such as a screen display or a digital camera. Each cell is also host to many *agents* that use local resources and communicate with each other. By analogy to a conventional operating system, a cell is like a kernel, shadows are like device drivers, and agents are like processes.

A schematic diagram of the Hive architecture is presented in figure 1. Hive is a set of Java libraries; currently, the system consists of roughly 280 classes in 24,000 lines of code, half of which are generic infrastructure and half specific code for about 30 devices and 60 agents. In addition to the basic Java libraries, Hive makes extensive use of Java Remote Method Invocation (RMI) and serialization for agent communication and mobility. For ontology support Hive uses the SiRPAC RDF library [25].

2.1 Cells: nodes in the decentralized network

The Hive cell is a program that participates in the network. Cells perform two primary tasks: hosting software agents and managing access to local resources through shadows. The ideal model would place a Hive cell on every device. However, Hive cells require a fair amount of CPU power and memory, and so they currently run on desktop-sized computers that proxy for several devices. A typical cell has several devices providing input resources (such as a motion sensor, camera, or digital tag reader) and output resources (a computer display, a small robot, or a speaker).

Hive has a *location dependent* model of a distributed system. Hive cells are not all the same: each cell has a specific

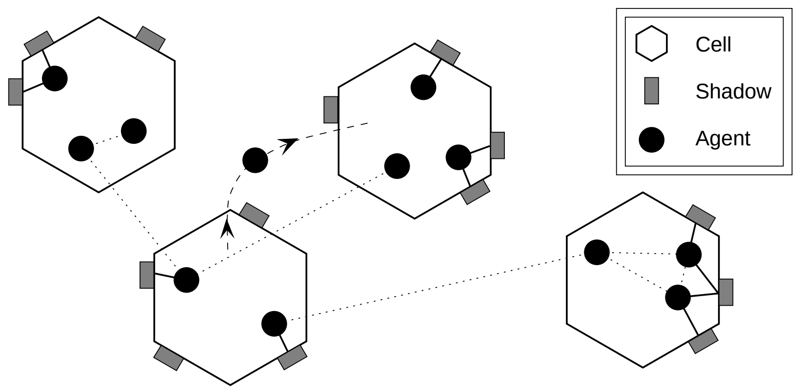

Figure 1. Hive Architecture

set of shadows and a specific population of agents. If one wants access to a particular device, one has to contact an agent on the cell that has access to that device's shadow. This model is in contrast to many distributed systems that try to abstract away the concept of location. Hive explicitly uses the locations of shadows and agents to help organize and partition the system. Things have places.

Hive cells are created as necessary to represent devices, and are intended to be long-lived processes. Each Hive cell is an equal peer in the network. Agents on one cell can communicate with agents on any other for a completely free-form, decentralized system. Federations of Hive cells are formed ad-hoc. Hive cells typically run a "server list agent" that contacts a registry to maintain membership in the global Hive network, but cells are free to arrange for their own federations as needed.

2.2 Shadows: local resources

A Hive cell by itself is simply a shell; a cell is interesting because of the shadows it provides. The term 'shadow' suggests its role of encapsulating a local resource. Physical devices are 'shadowed' into the Hive cell. For example, a Hive cell may have a digital camera plugged into it. The camera's shadow provides a software interface to the hardware, with methods such as `takePicture()` and `setBrightness()`. If an agent wants to directly use a camera, it moves to the camera's cell, asks the cell for the camera shadow, and then invokes the shadow with standard Java method calls.

Shadows are simple, they are nothing but an API to access a specific resource. However, the shadow abstraction is very useful for structuring a system. Shadows are the static part of a Hive cell; a system designer writes a shadow once, encapsulating all of a device's behavior, and then freezes that code in place. The flexibility of the system comes from the mobile agents; the shadows provide the static layer which the agents access.

Shadows also provide the place for a Hive cell to enforce a security or resource control policy. Shadows are trusted code. Shadows are not mobile, and they do not communicate directly over the network. To export a device's functionality off the local cell, an agent must mediate between the shadow and the network.

2.3 Agents: active computation

If shadows are the fixed, static, local component of the system, then agents are the active, dynamic, networked aspect. Hive agent sembody the network interface and policy for resources. Technically, a Hive agent is a combination of a Java object, an execution thread, a remote interface for network communication, and a self-description. These four simple pieces together create an agent, a full-fledged autonomous process in the distributed system.

Agents live on specific cells, accessing shadows for the resources they need. Agents export selected functionality to the network and communicate with each other to share those functions. For example, a camera agent can export the picture taking functionality of the camera shadow to remote agents. An image displayer agent can then invoke this method over the network, implementing a simple remote picture taking application. Hive applications are built out of a collection of interacting agents. Agent interactions, descriptions, and mobility are discussed in section 3.

2.4 User interface

Hive provides a graphical user interface to the distributed system. In Hive, the computer display is simply another shadow; any agent can draw on the screen by sending a Java graphical component to the appropriate shadow. The Hive user interface is implemented as an agent. The `AWTUIAgent` provides a graph view of the state of the system and a control interface by communicating with system agents such as the ServerControlAgent and ServerInfoAgent. By using agents for system tasks, Hive itself benefits from the distributed agents approach.

A screenshot of the user interface is shown in figure 2. The user interface displays two major aspects of the system: agents and the connections between agents. Each agent is represented as an icon. The agents at left are a simple camera application (described below), the agents at right are system management agents. Agent communication paths are shown as arrows between icons. The visualization is dynamic — as agents message each other, messages are shown as animations along the connection arrows. The interface can display multiple cells; this example shows two.

The user interface is not for information, it is also for control. New agents can be created from the drop-down menus. Agents can be connected to each other by drawing lines between them. And agents can be moved or killed by popping up their menus. The Hive user interface makes it easy for users to experiment with agent populations and create new applications with the stroke of a mouse.

2.5 A simple example

The image shown in section 2 is from a basic application we use to demonstrate the Hive system. The core of this example consists of three agents: an on-screen button (upper left), a digital camera (middle left), and an on-screen image display (lower left). The application is simple: the user clicks the button, the camera takes a picture, and the picture is displayed on screen.

Three agents work together to build this application: the button, the camera, and the display. The button and display both send graphical components to the cell's computer screen. The camera shadow uses a local resource, a digital camera. When the camera agent is created it asks the cell for access to the camera shadow; if all is well, access is granted and the agent can take pictures and send them over the network.

Once all three agents are established, all that is left is to connect them. The image display agent connects to the camera agent, and the camera agent connects to the button agent. When a user clicks on the button, the button agent sends an event to the camera agent. The camera agent interprets this as a command to take a picture and sends an event to the display agent with the image embedded as data. In this example, the camera agent is on a second computer; the agents are communicating over the network to create the application.

This distributed application is not itself new. What makes Hive interesting is the simplicity and flexibility with which applications like this can be built. New applications can easily be built by connecting existing agents. An agent that detects the motion of a door opening can connect to the camera to get pictures of everyone who comes into the room. A face recognition agent can use the camera to discover the names of visitors. Or an agent could scour the Hive network looking for camera agents, to build a photo gallery of an entire building. Finally, mobile agents can be dropped anywhere into the network, dynamically upgrading and changing pieces of the application. All of this flexibility can be used any pre-planning and without altering any other agent's code. This is the power of an ecology of distributed agents.

3 Hive agent implementation

Hive agents live out their lives on cells, possibly traveling the network. Agent interaction is ad-hoc, based on simple distributed object techniques. Hive agents have a rich ontological basis, using both the Java type system and the Resource Description Framework [19] as description languages. While creating a practical system to network Things That Think we have tried to balance working on deep agent research questions against creating simple, useful software for everyday programmers. Hive draws techniques from many parts of agent research including agent autonomy, mobility, multi-agent interaction, and agent ontologies. For each of these fields, we have tried to extract the most essential pieces, bypassing complexities and unsolved problems.

3.1 Agent anatomy

All Hive agents are rooted in a common agent base class. We have tried to adopt a minimal approach to designing Hive, only adding fields and methods when we are absolutely sure we need them. We have drawn inspiration from other mobile agent systems such as Aglets [17], particularly for the agent lifecycle, but have tried to simplify where possible.

Every agent stores two fields: the cell the agent is living on and a pointer to a "description" object that is used for semantic description. In addition, application agent subclasses typically have additional internal state for their own computation.

The agent class is a subclass of UnicastRemoteObject, the basis of remote objects in RMI. This class allows for the methods of an agent to be executed remotely. Choosing RMI as the agent communication mechanism simplifies the Hive architecture: agents are simply remote objects, we do not require any extra agent communication system. The cost for this simplicity is control; all management of agent communication has to be done within RMI. Hive does not have explicit control over agent messaging, placing

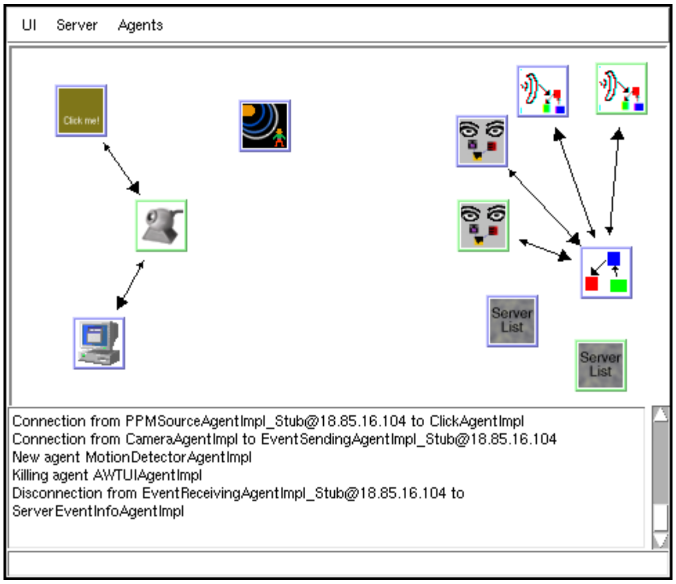

Figure 2. Hive Camera Application

the responsibility for understanding messages and deciding whether to act on them with the agents themselves.

3.2 Agent interaction

Agent interaction in Hive is completely ad-hoc. We make no requirements for agent communication, it is up to individual agents to decide how to talk to each other. Agent communication occurs via RMI's remote method invocation, allowing for agents to synchronously call methods on each other. We have added asynchronous calls to RMI so that agents can decouple their interactions. Java's strong typing syntactically structures agent communication, but agents are free to define their own semantics.

Each agent has two types: the true type of the agent's object, a subclass of `AgentImpl`, and the agent's remote interface, a subtype of `Agent`. This duality is common in RPC-like systems. While initially it might seem a nuisance to write two types for every agent, the split between an agent's remote interface and its local implementation is useful. In particular, agents have bookkeeping and administrative methods that would be inappropriate other agents to call. Because agents can only access each other's remote interfaces, sensitive methods are protected by Java's type system. And in practice many agents can simply inherit their remote interface, eliminating the need to define one.

The base agent interface has several methods that can be called remotely. Some methods are informational, such as `getServer()` and `getDescription()`. Other remote methods are administrative, such as `diePlease()`. This may seem a strange method to make remotely accessible, but it allows agents to manage other agents. And while any agent can ask another to die, the remote agent does not necessarily have to comply.

Finally, agents have remote methods for managing abstract "connections." An agent can ask another to `connectTo()` or `disconnectFrom()` an agent, or ask it to `listAllConnections()`. As with all aspects of an agent's interface, these methods are purely advisory. Agents are free to define their own meaning of "connection" and to ignore requests. We have found that several simple communication patterns, particularly event publish/subscribe, are broadly useful. For example, `EventListeningAgents` connect to `EventSendingAgents`, following the Jini Distributed Event specification [2] to pass event objects around the system.

In addition to these standard agent methods, application agents can define extra methods in their interfaces. For example, Hive uses agents to manage system functions. A cell's `ServerControlAgent` has remote methods such as `moveAgent()` or `shutDownServer()`. Other agents (such as the user interface agent) communicate with this agent to control remote cells. The `ServerControlAgent` is responsible for deciding what to do when these methods are invoked; agents implement their own security policy.

3.3 Syntactic ontology

Hive relies on the ability of agents to describe themselves to make ad-hoc agent interaction coherent. Every agent in Hive is described in terms of two orthogonal ontologies: syntactic and semantic. Hive cells provide a query service based on these descriptions so that agents can find each other. For example, a remote agent can ask for a list of all agents that are "`EventSendingAgents` that can provide me with motion data in room E15–305."

The syntactic ontology of agents in Hive comes for free from Java. An agent's syntactic description is simply its Java type. While this may seem trivial, its usefulness should not be overlooked. The type of an object says much about its capabilities with reference to a well-established and well-understood ontology, the class hierarchy of the system. Given a reference to an agent, it is easy to learn which types that it supports and, therefore, the messages it understands. The `queryAgents()` interface on a Hive cell allows the requester to list the types it is looking for.

There are several hard problems we have not solved with the syntactic ontology. Two classes might have the same name but in fact be different versions or different types entirely. Hive can not guarantee consistency. Mobile code means that the syntactic ontology of Hive is open-ended; if a new agent class comes into the Hive network, other agents might not know about that type to query for it or message to it. We do not see any way to solve these problem without answering fundamental epistemological questions. In keeping with the practical bent of the Hive system we have chosen to defer working on these issues. So far, they have not hampered our ability to create useful systems. We believe that in building practical applications people can engineer their agents to work around these problems, by either creating an ontology for a project or following consensus practice.

3.4 Semantic ontology

A syntactic ontology is not enough to describe agents, there are many bits of information about an agent that do not fit into a rigid class hierarchy. For instance, the fact that an agent is representing a device "in room E15–305" is not easy to codify syntactically. One could add a `getRoom()` method to the agent as part of a `RoomIdentifying` interface, but then that method would make no sense on many agents that do not have a physical location.

The limitations of Java class hierarchies for semantic description are compounded by the fact that an agent might want to assume different capabilities at runtime. For example, an agent that filters event streams might need to assume many different identities depending on what event source it is connected to. But in Java one cannot add interfaces to an object at runtime, the type of an object is static.

To circumvent these limitations, Hive uses a second orthogonal ontology to describe "semantic" information about agents. This ontology utilizes the Resource Description Framework (RDF) [19], itself based on XML [5]. RDF provides a structured way to attach nouns and verbs to agents. For example, an agent's semantic description might state its physical location, a human readable nickname, the owner of the device it is using, and a description of the meaning of its data. Hive agents carry an RDF description, available by calling `getDescription()`. Other agents can inspect this description to learn about the agent, and the cell query interface supports matching across sets of RDF descriptions. Agents are free to change their descriptions at runtime.

Our semantic ontology does not solve any deep philosophical problems, but it allows application designers to express their own solutions. We expect that application designers will develop their own schemas to make agent communication semantically consistent. These schemas may be completely ad-hoc or may be strictly defined by an SGML DTD and a DDML schema [4]. We expect users to collaboratively *define* schema by common consensus while building Hive systems. As the Hive network grows, ontologies should emerge from common practice.

3.5 Mobility

A final aspect of Hive agents is that they are mobile. Hive mobility has two portions: mobile code and mobile agents. Mobile code enables the software on each Hive cell to be updated dynamically. Mobile agents build on mobile code, allowing agents to move themselves around the Hive network. Hive agents have weak mobility [3]: the agent must make its own arrangements for preserving its execution state on transport. We believe strong mobility is preferable but is currently too difficult to implement in Java. There are a couple of experimental systems that achieve strong mobility by modifying the JavaVM [14] [22], but a standard mechanism has yet to be defined.

Some of the advantages of mobile agents are well understood [7] [18]. If an agent is using lots of bandwidth or needs low-latency access to a resource, then it can be more efficient for the agent to move to the resource's cell than to communicate with it (through another agent) over the network. And if a Hive cell has unreliable network access, moving agents to a more stable Hive cell can make the system more robust.

We are most interested in the use of mobility to achieve flexibility. This property is especially important in heterogeneous networks of embedded systems. A Hive cell might be deployed as embedded software in a smart device, something small whose firmware cannot easily be upgraded. With Hive in its firmware, the device-specific software can easily be updated by simply sending a new mobile agent to it. And mobile agents allow the functionality of cells to be customized. For example, mobile agents can turn a camera into a motion detector. A camera agent has a simple `takePicture()` method. By comparing images, another agent can decide if something has moved. A motion detector agent can be created and moved to the host with the camera, saving the bandwidth of shipping images across the network. And the motion detector function can remain a separate component, keeping the camera agent simple. Hive's use of mobility enables efficiency and conceptual cleanness, useful attributes for building applications.

Finally, Hive provides mobile agents for future expandability. We believe that mobility will ultimately enable a new form of distributed system, one where agents travel around the network freely performing their tasks and the network as a whole comes alive with computation.

4 Hive applications and experiments

The design of Hive has been motivated by the specific requirements of our application domain, Things That Think. The iterative experience of building applications with Hive has greatly influenced its design. The ease with which applications can be built in Hive demonstrates that an ecology of agents is an effective way to create a distributed system.

4.1 Honey, I Shrunk the CDs, Part II

A simple system we have built with Hive is a jukebox with a physical interface. The application appears straightforward: a user selects a poker chip with the name of a song written on it. She drops it on a table, and the song she has selected start to play. The poker chip is a stand-in for the CD, a physical icon.

The activity behind the scenes is more complex. The poker chip has an embedded RF ID tag. The table contains a tag reader that senses tags and passes their IDs along a serial port. A tag reader agent watches the serial port shadow and broadcasts a Hive event when a new tag is sensed. This event is picked up by a database agent that maps a tag ID to a song name, broadcasting it to a third DJ agent that plays the MP3 file. The three agents working together over two machines make the jukebox work. The system is self-repairing; agents on one Hive cell watch for failure of the other agents, restarting them as necessary.

An earlier version of this system was implemented as one monolithic application. The Hive version took roughly half the effort to implement and is more robust and more flexible. By factoring the pieces of the system into three agents, it is easy to add new interfaces to the jukebox such as a standard remote control, a web page, or an "intelligent agent DJ" that could pick music based on people's preferences.

4.2 Tangible interfaces and ambient displays

Honey, I Shrunk the CDs is an example of a *tangible interface*, a new form of human computer interface using physical objects instead the standard keyboard, mouse, CRT [15]. The Media Lab has an active tangible interfaces research agenda, building a variety of creative things that think. Many of these are "ambient displays," devices such as an animated pinwheel or soft light display that subtly convey information. In the past it has been difficult for others to use these displays in their own projects: the hardware interface for each device is different and often undocumented. But with Hive all that is necessary is to create a Hive cell with a shadow for each device. The device can then be used by any Hive agent anywhere on the Internet.

A Hive interface has been created for Craig Wisneski's Personal Ambient Displays [33]. These things are small objects with different input and output capabilities. Some objects grow warm or vibrate on command, others sense when they are touched or shaken. These objects are controllable through Hive. They can be connected so that when one is shaken the other warms up. Or they can be hooked to external information sources so that one starts to vibrate when your stock portfolio's value drops precipitously. Hive provides a flexible, simple way to experiment with connecting these devices to each other and to other Hive-enabled devices.

4.3 Wearable computing

Another Hive-related research project in the Media Lab is wearable computing, making people themselves be "things that think" by putting computation in their clothing [28] [24]. The Media Lab is outfitted with "locusts," beacons that broadcast the room they are in. Brad Rhodes wearable computer uses these to figure out where he is in the building. And because Brad's wearable computer is a Hive cell, his Hive agents can choose to make this information available to other agents so that people can easily find him. An agent in his office can use this information to play his "theme music" when he walks in by telling a DJ agent from Honey, I Shrunk the CDs to play the appropriate music. Wearables offer new opportunities for personalized, context-sensitive applications. The decentralization of Hive makes it possible to preserve privacy as well, as your own agents control your personal data.

4.4 Counter Intelligence

The largest application built with Hive to date is "Counter Intelligence," a project to make kitchens smarter [13]. Counter Intelligence outfits a kitchen with embedded networking: the pantry knows what ingredients you have, the counter knows what you are currently using, a scale knows how much you have added to the bowl, and the oven is automatically set to bake at the right temperature. A recipe planning agent (incorporating STRIPs [10] and procedural nets [26]) runs behind the scenes, helping you bake your cake. Counter Intelligence is a working system, currently implemented with eighteen agents running on two computers.

Hive provides the infrastructure for these agents, simplifying the process of managing their interactions. The ontology system is key for allowing the agents to coordinate and dynamically adapt to newly available hardware; planning agents inspect what information is available in the kitchen and use that data to help effect a recipe. The system is open-ended; new devices can be added to the kitchen, or you could even network your kitchen to your mother's across the country to get her help while baking. Counter Intelligence has been a good testbed for Hive, placing strong demands on the agent infrastructure.

4.5 Summary of applications

The applications described above give the flavor of things we are networking with Hive. Hive has proven to be a useful application infrastructure for connecting things. But the real power of Hive is not just that one can network a few things in Hive, it is that the Hive network itself is open and flexible enough that new connections can be made with little cost. For example, it is trivial to use Hive to network wearable computing to tangible interfaces, or use both sets of devices to control your kitchen. Once an agent is written to represent a capability, Hive allows anyone to connect those agents together and build new distributed applications. From the experience of the Web, we know the power of such synergy.

5 Lessons from Hive

The experience of building real systems with distributed agents has been educational from the perspectives of software engineering, distributed systems, and agents. As we continue to experiment with Hive we have uncovered several practical lessons as well as pointers for future work.

5.1 Java, RMI, asynchronous messaging

Java is a wonderful language for building distributed agent systems. Many of the things about Java that make it pleasant in general are particularly useful for distributed systems. Strong typing gives Hive a syntactic ontology for free. The simple object oriented model, particularly the split between interface and implementation, makes it easy to build class hierarchies of agents. AWT makes it easy to write portable graphical agents. And the Java security model makes it thinkable to build a system that executes untrusted code.

While Java is fairly advanced for Internet programming, we have found that the options available for distributed Java are a bit lacking. Hive started out as a Voyager 1.0 based system. Voyager provides many nice capabilities for distributed agents, including messaging and mobility [37]. Voyager was a good toolkit for building our system, but in order to give ourselves more flexibility in messaging and mobility and to give the option of moving towards Jini, last year we reworked Hive to use Java RMI. This retooling was an interesting experience in itself; porting the codebase over was surprisingly easy as most of the agents are isolated from the details of transport.

A major advantage of the change to RMI was the robustness that comes from being forced to handle communication errors. This argument is made well in "A Note on Distributed Computing" [31]. Voyager tries to provide a transparent model of networking, where messaging failures between agents do not have to explicitly be caught. The danger of this approach is that network failures *do* happen, and it is better if agents explicitly know this and have to deal with the potential for error at every communication. RMI's requirement to catch `RemoteException` makes it easier to build robust distributed systems.

RMI has limited us in several ways. RMI in Java 1.1 has several implementation limitations that make it difficult to scale to systems with many concurrent agents. And RMI's design is entirely synchronous calls, which do not work well for agent communication. Synchronous communication means that agents are unduly dependent on each other, having to wait for each other to respond to messages. Synchronous messaging is particularly inappropriate for distributed event systems, where the sender typically does not care if the event was even delivered. In our view, asynchronous messaging is a fundamental requirement for any distributed agent system. We have added asynchronous messaging on top of RMI, but a fairly complex implementation is required to avoid scalability problems.

5.2 Decentralization

We have also learned that decentralization is a useful strategy at many levels. We believe the ultimate advantage of decentralization is scalability. Centralized systems break when the central manager is overwhelmed, but decentralized systems can spread the load. However, the Hive network is still small enough that we have not seen many technical scalability advantages.

We have found organizational advantages to decentralization. New Hive users need no setup, they can simply start a cell and join the network. This lowers the barrier for use of Hive. Individuals are free to develop their own agents without any central coordination. If someone makes a new kind of device that sends out device data, they can deploy an agent to represent it and have other agents use it without explicitly teaching the system about the new agent type. And the free software community has found that decentralized software development is an effective way to quickly build large systems [23].

Another surprising advantage is the robustness of the decentralization of server bookkeeping tasks. Hive agents take care of maintaining a federation of servers, broadcasting server state, controlling the server, and displaying the user interface. Sometimes these agents crash. But just because one agent fails does not mean the whole system dies. For example, the user interface might stop working, but the rest of the application agents merrily continue. The user can even create a new user interface agent and send it over to the cell, fixing the problem! This sort of robustness is quite appealing.

Decentralization has drawbacks. Some desirable functions require global state, such as maintaining a listing of Hive cells running in the network. Current implementations of these functions in Hive are cumbersome, naively centralized, and do not scale well. Making decentralized systems consistent is difficult; one cannot rely on a central architect to make things right, each portion of the system is responsible for itself. For a small distributed system this might be a weakness, but this decentralization is essential if the system is to grow beyond the management of a single administrator.

5.3 Mobility

Mobility has proven to be a challenging problem, both in the details of implementing mobile code as well as the general picture of using mobile agents. When moving from Voyager to RMI we implemented our own small mobile agent system (based on standard serialization and network class loading), but getting the implementation details of Java class loaders right has been tricky. Furthermore, our experience has pointed out two deep problems in mobile code that are unsolved. One issue is versioning: when code is mobile, it is common to have multiple versions of classes in the network at the same time. Java's tools for managing versioning are incomplete. It is also quite subtle to load all of an agent's code. Because of Java's late class loading, an agent might decide it needs a class file long after the host it came from has gone away. Current solutions are awkward, generally requiring specifying a static code base or trying to precalculate the closure of all required classes.

An honest appraisal reveals our second problem with mobility in Hive: while we do benefit from mobile code, we do not use mobile *agents* very often. Our problem is conceptual; most of our current applications are fixed, agents do not need to wander the network. We have had a few practical examples of using mobile agents — sending an agent over to a computer because it was easier than walking over to it, or demonstrating the system to a new user by sending agents to their cell. But we still mostly statically and locally manage the agent population. All the classic arguments for mobility such as improving bandwidth, limiting latency, and supporting disconnected operation apply to Hive applications as well. And we expect as the system grows the system flexibility benefits of mobile agents will increase; some of our ideas for how this may happen are in section 6.2.

5.4 Ad-hoc interaction

Finally, we have found that the ad-hoc agent interaction of Hive has suited us quite well. The open nature of agent interaction might make the Hive system seem perilously unpredictable and inconsistent. Theoretically, it probably is: any agent could lie, any agent could ignore messages, any agent can interpret any method however it wants to. Abstractly, we can say nothing about a Hive application's correctness. But this uncertainty is the reality of the Internet, and we have learned to work with it. In general, people are cooperative. Agent designers can work together to build compatible software.

At this stage, restricting ourselves to a paradigm that is formally correct would hamper the growth of Hive. Formal agent communication mechanisms tend to be understood only by specialists and often require centralization of development and restricted agent communication. Hive takes the view that it is more important to enable system designers to build the right thing than to prevent them from building wrong things. Hive treats the most seriously wrong things as security problems and the rest as bugs.

The agent interaction paradigm was chosen to make it simple to build multi-agent systems. Our users write Hive programs much like they would write conventional programs. This familiarity is a strong advantage in gaining users. We do not know how well this ad-hoc interaction will scale. As long as the Hive community remains small, we can communicate in person and work up consistent class hierarchies and semantic ontologies. But if Hive grows to thousands of users, that friendly coordination will no longer work. To this concern we can only remark that we would love to have the problem of too many users, and that when this day comes to pass we will understand the shape of the problems well enough to choose solutions.

6 Future work

Our ultimate goal for Hive is to make the Internet come alive with agents running everywhere, interacting via discovery and ad-hoc communication, moving from host to host. To reach this goal we still need to address several large topics and add new capabilities. Foremost, Hive must be stable enough for cells to run persistently. We are mostly there: cells run for weeks at a time without any problems. The main difficulty is that because Hive is under rapid development, old versions of the system become incompatible very quickly. We need to find some way to handle the issues of versioning. We also need the experience of more applications and more users. The more users of Hive, the more interesting the network becomes. We are making an open source release of Hive at the end of the summer in 1999, so that others in the research community can use it to build their own distributed applications.

6.1 Security plans

One major missing component for Hive is a full security system. Mobile agent security breaks down into three problems: protecting the host from the agents, protecting the agents from each other, and protecting the agents from the hosts [30]. The first problem is largely solvable in Java, through a combination of the Java 1.2 sandbox model and the Hive shadow abstraction. The main limitation is that Java has no model of resource accounting, there is no way restrict how much CPU time or memory an agent uses. There are various efforts underway to address this problem [8] [36]. Ultimately, we see many exciting research opportunities for applying economic models to resource control [20] [6].

Protecting agents from each other is harder, but we believe it can be addressed through a combination of Java strong typing and an authenticated credential system. The problem of protecting agents from malicious hosts is very difficult: solutions are partial at best [27]. The Hive architecture explicitly states that agents are only allowed to execute on a cell at the cell owner's whim. We believe many useful applications need no more than this guarantee, assuming that users have an incentive to cooperate.

6.2 Mobility

While mobile code has proven useful in Hive, mobile agents have not played as big a role as we had hoped. This fact is largely a symptom of the maturity of the system. As the Hive network grows larger, mobility will become more attractive as a solution for managing the agent population. In an ideal decentralized system, any user should be able to add new functionality dynamically, without consulting any central authority. Mobile agents are the simplest, most straightforward way to enable this, and we expect that having mobile agents will make Hive better prepared to handle larger distributed systems.

We are anticipating several near-term applications of mobile agents within Hive. We envision "census-taker" agents that wander the Hive network, exploring to see what cells and agents are available in the world. We also expect to create information gathering agents that travel the globe, moving around to take advantage of spare CPU cycles or following daybreak from cell to cell to capture pictures of the dawn. Overall, we see mobility as an essential part of making Hive flexible enough to grow to encompass the possibilities of open distributed systems.

7 Related work: Jini, agents

Hive's model of an ecology of distributed agents draws from the experience of many other agent systems, in general emphasizing practicality and simplicity of programming over formality and provability. Our core method of building applications from distributed agents is inspired in part by Actors [1], albeit without the strong formalism of Actor theory. Hive's mobility is in many ways a reimplementation of systems such as D'Agents [14] [16], Aglets [17], and Mole [29]. We have tried to implement a simple version of mobility, ultimately intending to create our own solutions to the hard problems such as versioning and shipping complete code. By contrast to many multi-agent systems, Hive does not have any formal model of agent communication or negotiation [34]. We believe that useful distributed systems can be built without this conceptual overhead and that theory will be most applicable after we understand the problems that occur in practice. Higher level communication mechanisms such as KQML [11] can be added on top of the basic RMI communication layer as necessary. We have made the deliberate choice to design Hive to be palatable to systems builders, sometimes at the expense of theory.

Many distributed systems are being built today to address problems of embedded network applications. Sun's Jini system [2] is a leading architecture in this domain. A comparison between Hive and Jini is useful for highlighting the details of Hive's ecology of distributed agents. In some cases we are trying to make Hive more like Jini. In other cases, we believe Hive has advantages.

Both Hive and Jini are distributed application infrastructures, both are based on Java, and both rely on RMI distributed objects and mobile code. Both systems have discovery and lookup implementations. Hive and Jini both make extensive use of events for communication; indeed, Hive uses the Jini distributed event specification. And both systems represent devices and capabilities on the network, proxying if necessary.

Jini services are roughly analogous to the combination of Hive's shadows and agents. But Jini does not have anything like the conceptual split between the two. The distinction between shadows and agents gives Hive a useful abstraction between local, trusted code and networked, untrusted code. The autonomy of Hive agents gives a clear place to place computational activity in the system.

Another important difference is Hive's location-dependent model. In Hive, an agent's cell is an important fact; it tells you where the agent is on the network, (potentially) where it is physically, what resources it has access to, etc. Jini focuses mostly on services; the actual place a service is hosted on is not a major part of the Jini model. We believe that the location dependence of Hive contributes to scalability, both technically and conceptually.

Both Jini and Hive rely on mobile code for flexibility, and the argument for the usefulness of mobility is the same. A difference is that Jini only has single hop mobility: a service can upload a smart proxy to the user as an interface, but that proxy does not then migrate around the network. Currently, we do not make much use of Hive's multi-hop mobile agents, but we believe that it will become more important as the system grows.

For description, both Jini and Hive use the Java type system for a syntactic ontology. But where Hive uses RDF for semantic descriptions of agent capabilities, Jini uses more Java types. The Jini Lookup Attribute system is more closely like our use of RDF, but we believe the RDF model is more flexible due to the ability to perform deeply structured queries. Jini's Lookup Attribute system does not support queries on subattributes of attributes.

Hive does not currently support Jini's leasing or transactions, but probably should. Transactions will be useful to allow agents to enter into multi-message communications with a guarantee of consistency. Leasing will be a useful hook for allowing Hive agents to explicitly negotiate their relationships; an agent can express the decision to work with another agent for a limited period of time as a lease.

As a practical matter, Hive and Jini could be integrated by encapsulating a Jini service as a Hive shadow or making a Hive agent present itself as a Jini service.

Finally, Hive has so far stayed inside the Media Lab network. But in every decision we have designed Hive to expand beyond that, to work across the Internet. The abstractions inherit in the ecology of distributed agents gives us a conceptual model for organizing a worldwide network of interacting processes.

8 Conclusions

We have presented Hive, an implementation of an ecology of distributed agents. We have taken many ideas from agents research and put them together into a coherent system, an application infrastructure for Things That Think. We have proven that agents are a good abstraction for building distributed systems. And we have found that ad-hoc agent communication along with an ontology mechanism is sufficient to build useful systems. Finally, we believe mobile agents are a useful abstraction, but that they are more applicable to larger distributed systems. Agents are a fundamental building block for coherent distributed systems; ecologies of distributed agents can grow to inhabit our global network of millions of computers and Things That Think.

Acknowledgements

We thank Todd Papaioannou, Marc Hedlund, Brad Rhodes, Manor Askenazi, and the ASA/MA reviewers for their helpful comments.

References

[1] Gul Agha. Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems. In E. Najm and J. B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*. Chapman & Hall, 1997. http://osl.cs.uiuc.edu/Papers/fmoods.ps

[2] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini Specification*. Addison-Wesley, 1999. http://www.sun.com/jini/

[3] Joachim Baumann. Mobility in the Mobile Agent System Mole. In *CaberNet: 3rd Plenary Workshop*, 1997. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1997-baumann-05-paper.html

[4] Ronald Bourret, John Cowan, Ingo Macherius, and Simon St. Laurent. Document Definition Markup Language. Technical report, W3C, 1999. http://www.w3.org/TR/NOTE-ddml

[5] 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

[6] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based Resource Control for Mobile Agents. In *Proceedings of the 1998 International Conference on Autonomous Agents*, 1998. ftp://ftp.cs.dartmouth.edu/TR/TR97-326.ps.Z

[7] 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

[8] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of 1998 ACM OOPSLA Conference*, 1998. http://www.cs.cornell.edu/slk/papers/oopsla98.ps

[9] Sean M. Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom. The Inferno Operating System. *Bell Labs Technical Journal*, Winter 1997. http://www.lucent.com/ideas2/perspectives/bltj/winter_97/paper01/index.html

[10] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Readings in Planning*, pages 189–208. 1971.

[11] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, 1997. http://www.cs.umbc.edu/agents/introduction/kqmlacl.ps

[12] Neil Gershenfeld. *When Things Start to Think*. Henry Holt & Company, 1999. ISBN: 0805058745. http://www.media.mit.edu/physics/publications/books/ba/

[13] Matthew K. Gray. Infrastructure for an Intelligent Kitchen. Master's thesis, MIT Media Lab, 1999.

[14] Robert S. Gray. *Agent TCL: A Flexible and Secure Mobile-agent System*. PhD thesis, Dartmouth College, 1997. http://actcomm.dartmouth.edu/~rgray/thesis/thesis.ps.Z

[15] Hiroshi Ishii and Brygg Ullmer. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *Proceedings of CHI 97*, pages 234–241. ACM Press, March 1997.

[16] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997. http://computer.org/internet/ic1997/w4058abs.htm

[17] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents With Aglets*. Addison-Wesley, 1998. ISBN: 0201325829.

[18] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, March 1999. http://www.acm.org/pubs/citations/journals/cacm/1999-42-3/p88-lange/

[19] Ora Lassila and Ralph Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, W3 Consortium, 1998. http://www.w3.org/TR/WD-rdf-syntax/

[20] 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

[21] Nelson Minar. Designing an Ecology of Distributed Agents. Master's thesis, Massachusetts Institute of Technology, September 1998. http://www.media.mit.edu/~nelson/research/masters-thesis/

[22] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the 1997 USENIX Technical Conference*, pages 91–104, 1997. http://www.cs.umd.edu/~acha/papers/usenix97-submitted.html

[23] Eric S. Raymond. The Cathedral and the Bazaar, 1997. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

[24] Bradley J. Rhodes, Nelson Minar, and Josh Weaver. Wearable Computing Meets Ubiquitous Computing: Reaping the Best of Both Worlds. In *Proceedings of the International Symposium on Wearable Computers (ISWC '99)*, October 1999. http://www.media.mit.edu/~rhodes/Papers/wearhive.html

[25] Jaane Saarela. SiRPAC — Simple RDF Parser and Compiler, 1999. http://web1.w3.org/RDF/Implementations/SiRPAC/

[26] Earl D. Sacerdoti. The Nonlinear Nature of Plans. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 1975.

[27] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, chapter 4, pages 44–61. Springer-Verlag, 1997. ISBN: 3540647929. http://www.icsi.berkeley.edu/~tschudin/

[28] Thad Starner, Steve Mann, Bradley Rhodes, Jeffrey Levine, Jennifer Healey, Dana Kirsch, Rosalind W. Picard, and Alex Pentland. Augmented Reality Through Wearable Computing. *Presence (Special Issue on Augmented Reality)*, 6(4), 1997. http://wearables.www.media.mit.edu/projects/wearables/

[29] Markus Straer, Joachim Baumann, and Fritz Hohl. Mole – A Java Based Mobile Agent System. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/ECOOP96.ps.gz

[30] Giovanni Vigna. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. ISBN: 3540647929.

[31] 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

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

[33] Craig Wisneski. The Design of Personal Ambient Displays. Master's thesis, MIT Media Laboratory, 1999.

[34] M. J. Wooldridge and N. R. Jennings. Agent Theories, Architectures and Languages: A Survey. In *Intelligent Agents*, pages 1–39. Springer-Verlag, 1994.

[35] Intel Play. http://www.intelplay.com

[36] Javares mailing list. http://gee.cs.oswego.edu/dl/javares/

[37] Voyager Core Technology. http://www.objectspace.com/products/voyager/core/index.html