Hive: A Software Infrastructure for Things That Think

by

Oliver R. Roup

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering
In Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Massachusetts Institute of Technology 1999. All rights reserved.

Author ....
Department of
Electrical Engineering and Computer Science
May 21, 1999

Certified by ....
Michael Hawley
Assistant Professor of Media Arts and Sciences
Alex W. Dreyfoos ('54), Jr. Career Development Professor
of Media Arts and Sciences

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Hive: A Software Infrastructure for Things That Think

by

Oliver R. Roup

Submitted to the Department of
Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering
In Electrical Engineering and Computer Science

Abstract

Hive is a software toolkit to facilitate the creation of distributed systems of Things That Think. Hive contains a number of features that distinguish it from comparable toolkits including an unusual organizational metaphor, a reliance upon mobile code, and a powerful description facility to allow components of the system to find each other and communicate meaningfully.

As its name implies, Hive is based upon a biological metaphor. A Hive is composed of a collection of *Cells* which act as host to a population of *Agents*: mobile computational objects with some sort of agenda. Cells also play host to a number of distinct local resources called *Shadows*. A rich lookup scheme allows Agents to discover Shadows and each other and to facilitate meaningful ad-hoc connections.

The split between Shadows and Agents and the ability of Agents to move from Cell to Cell creates a great deal of flexibility. Code stability and wide-scale distribution of systems is encouraged.

Applications are implemented as the emergent behavior of an *ecology of Agents*: a population of Agents that move from Cell to Cell, discover resources, and interact with each other and the local Shadows.

Hive has been successfully deployed within the MIT Media Lab as the technical foundation for a number of widely varying projects and is scheduled for an Open Source release in the near future.

Acknowledgments

First and foremost, I would like to thank my co-conspirators and friends, Matthew Gray and Nelson Minar. Working with both of them has taught me immeasurably about moving from wild ideas to working code. Both have provided an enormous amount of personal support, without which this thesis would not have been completed.

Thank you to Professor Michael Hawley for his enduring support and guidance. Mike has a clear eye on the future and challenges his students to see beyond the status quo. In some ways, Hive is a response to Mike's challenge to, "Throw the computers away."

John Wroclawski, my academic advisor, has been a steady hand in sometimes stormy waters. In an environment where many view their advisors as hurdles to jump over, John realizes that which classes to take is not always the biggest thing on your mind. Thank you for being a true mentor.

To my family and friends, especially the KBL crew, whose hospitality I abused on many an evening, thank you for the support and belief.

Contents

1 Introduction **13**

1.1 The Future . . . 13

1.2 Summary and Organization . . . 14

2 Background **15**

2.1 From Punch Cards to Pointers and Beyond . . . 15

2.1.1 The Command-Line Interface . . . 15

2.1.2 Windows, Icons, Mice and Pointers . . . 16

2.1.3 The Grammar of Things . . . 16

2.2 Things That Think . . . 17

2.3 The Need for a Common Toolkit . . . 18

2.4 The Vision . . . 19

3 Design **21**

3.1 Design Criteria . . . 21

3.2 The World Wide Web . . . 21

3.3 Cells, Shadows and Agents . . . 22

3.3.1 Cells . . . 23

3.3.2 Shadows . . . 23

3.3.3 Agents . . . 25

3.4 Example Architecture . . . 27

3.5 Lookup . . . 30

3.5.1 Avoiding Unique Names . . . 31

3.5.2 Syntactic Lookup . . . 32

3.5.3 Semantic Lookup . . . 33

3.5.4 Object classification . . . 34

3.6 Security . . . 35

3.6.1 Protecting Cells from Agents . . . 36

3.6.2 Protecting Agents from Agents . . . 37
3.6.3 Protecting Agents from Cells . . . 37

4 Implementation **39**
4.1 Underlying Technologies . . . 39
4.1.1 Java . . . 39
4.1.2 RMI and Object Serialization . . . 40
4.1.3 Supporting Packages . . . 41
4.1.4 CellAddress . . . 43
4.2 Shadows . . . 43
4.2.1 Template Shadows . . . 43
4.2.2 Common Shadows . . . 44
4.3 Agents . . . 45
4.3.1 Mobility . . . 45
4.3.2 Template Agents . . . 46
4.3.3 Common Agents . . . 47
4.4 Graphical User Interface . . . 47
4.5 Applications . . . 48
4.5.1 Image Manipulation . . . 50
4.5.2 Jukebox Player . . . 51
4.5.3 Where's Brad? . . . 51
4.5.4 Personal Bits . . . 52
4.5.5 Counter Intelligence . . . 52
4.5.6 Net Weight . . . 52
4.5.7 The Future . . . 53
4.6 Related Work . . . 53
4.6.1 Jini . . . 54
4.6.2 Inferno . . . 55
4.6.3 Corba / DCOM / RPC . . . 55
4.6.4 Universal Plug and Play . . . 55

5 Conclusions **57**
5.1 Future Work . . . 58
5.1.1 Security . . . 58
5.1.2 Privacy . . . 58
5.1.3 Distributed Lookup . . . 59
5.1.4 Beyond a list of Cells . . . 59

5.1.5 Out of Band Channel Negotiation 60
5.1.6 Interface to the Web 60
5.1.7 Periodic Disconnection 60
5.1.8 Licensing Ramifications 60
5.1.9 Cell Description 61
5.2 Future Applications 61
5.2.1 You Are The Star 61
5.2.2 Air Dropped Infrastructure 62

A TTT History **63**
A.1 Mr. Java 63
A.2 Marathon Man 63
A.3 Everest Extreme Expedition 64

Bibliography **65**

Photo Credits **68**

List of Figures

3-1 The Hive Architecture 22
3-2 Camera: A Conventional Architecture 27
3-3 Camera: A Revised Device Driver 29
3-4 Camera: A Separate Resource 29
3-5 Camera: Polling Across the Network 30
3-6 Camera: Hive Implementation 31
3-7 Sample Semantic Label Data 33
3-8 Sample Semantic Description 34

4-1 Methods on a Hive Cell 42
4-2 Remote Methods on a Hive Cell 42
4-3 A Sample CellAddress 43
4-4 Methods for Shadow 43
4-5 Remote Methods for an Agent 45
4-6 Local Methods for an Agent 46
4-7 A Simple Hive Configuration 48
4-8 A More Complicated Hive Configuration 49
4-9 Hive Image Filters 50

Chapter 1

Introduction

If computers are getting cheaper and networks are getting more widespread, we find ourselves quickly coming to the question: What happens when the computers are free and the networks are everywhere?

1.1 The Future

Imagine it is several years from now and you are a busy city dweller who hasn't had time to get to the grocery store. Your refrigerator sees that you are running low on milk. You walk by the fridge, but rather than interrupting you when it knows you aren't in a position to do anything about it, it notices the cell phone in your pocket and quietly informs it that you are out of milk, just in case it is in a position to do anything about it. The next day, when you are driving home from work, your car realizes that given your current route, you will soon drive by a grocery store. It asks all the objects in the car if they need anything. Your cell phone responds that you are in need of milk, so your car turns down your radio and asks you if you would like to stop at the approaching grocery store, given that you are out of milk.

You recently purchased a new plush doll for your daughter. Like all modern toys, this one contains a computer and a network connection, but your daughter doesn't know that. What she does know is that her doll seems to know things about the world: How old she is, how tall she has gotten, things they have done together.

In the meantime, you also have one of the latest security systems in your home, and by virtue of it's job, it has a good deal of information: Who is home, how long

they have been there and so on.

Your daughter is in her room upstairs, playing with her doll when you pull into the driveway. Your daughter holds the doll close, so it can whisper in her ear: "Mommy's home."

The Hive Logo

1.2 Summary and Organization

Hive is a software toolkit created to facilitate the creation of distributed systems of Things That Think. Hive contains a number of features that distinguish it from other comparable toolkits including an unusual organizational metaphor, a reliance upon mobile code, and a powerful description facility to allow components of the system to find each other and communicate meaningfully. This thesis is divided into 5 sections: Section 2 provides background on the use of physical objects as an interface paradigm, and the motivation and vision behind the Things That Think consortium and the Hive project. Section 3 discusses the design of the Toolkit and describes the decisions and trade-offs that were made in it's construction. Section 4 describes some of the implementation details of the Toolkit itself, and section 5 describes some of the lessons learned and future directions of the project.

Chapter 2

Background

2.1 From Punch Cards to Pointers and Beyond

The amazing pace of growth in the capabilities of computers has resulted in a number of profound shifts in how we interact with our machines. The punched card machines of the 1960s and the common PC of today look and behave so differently that they are hardly recognizable as representatives of the same lineage. Each "era" in computer science can be distinguished by a few key ideas and metaphors that define the language that a user uses to communicate with a computer.

2.1.1 The Command-Line Interface

One of the early metaphors for interacting with a personal computer was that of the *command-line interface*. The computer would wait at a prompt for a user to input some commands. A command is generally of the form

```
command [argument] [argument] ...
```

where the command is the name of the program that the user wants to use, and argument is some optional statement that potentially modifies the outcome of the command. Many commands take input and produce output, so operating systems usually define *standard input* and *standard output* which are the default streams of communication for a command.

Pipes, which are denoted by the vertical bar (|) can be used to tie multiple commands together. A user types a string of the form

```
command-1 | command-2 [| command-3] ...
```

to pipe the output from command-1 into command-2 and the output from command-2 into command-3 and so on.

Understanding these three simple concepts allows a user to comprehend most of the complexity of the command line interface and interact powerfully with their machine.

2.1.2 Windows, Icons, Mice and Pointers

The availability of bit-mapped displays made it possible to create a user interface that contained richer graphical elements than just text. In 1984, Apple debuted the Macintosh, which introduced an entirely new metaphor for interacting with a personal computer. Rather than commands and pipes, the Macintosh uses a desktop metaphor. The screen is a desktop. Programs are displayed as windows that can be opened, closed, moved and resized. They can be manipulated by pushing buttons and entering text into labeled fields. Understanding this physically based metaphor has proven much easier than the command-line metaphor of the past, and it remains the dominant paradigm for interacting with a computer today.

2.1.3 The Grammar of Things

While the advent of the mouse and the bit-mapped display once felt like a liberation from the confines of the keyboard and the text terminal, today they seem constraining. When asked to describe a computer, most people describe a keyboard, a monitor, and perhaps a mouse, no matter what the computer is to be used for.[7]

When we decide to sit down and catch up on the news, we have different environmental requirements than when we do our taxes, and those are different still when we wish to play a game. but somehow we imagine that if we are using a computer one environment and one interface should be equally appropriate for all tasks. This one-size-fits-all approach has caused many other problems as well: Despite repeated effort at simplification, computers remain maddeningly complex, enough so

that most users stick to their most basic functions.

In contrast consider three objects: a toaster, a calculator and a telephone. Each of these objects comes with a minimum of instruction yet serves a single simple function which should be obvious to someone who sees them. There is no configuration, no "booting up," no operating system; all of these objects for the most part *just work* even though commonly, all three of them have computers inside.

One is left to wonder, has our mouse and desktop metaphor outlived it's usefulness? Is it possible that we can take our physical metaphor one step further: Rather than manipulating virtual objects like windows and trash cans, what if we could interact with our computers using real physical objects as our interface?[15][20]

2.2 Things That Think

The Things That Think Consortium of the MIT Media Lab was founded to pursue a particular vision: that the next wave in computing will be the one where they disappear.[43] One day soon, the most mundane of objects will be first class networked citizens. Everything from Coffee cups to kitchen appliances will be connected and available on the network. An individual object might have very limited capabilities. For example: A coffee cup might know whether it is clean or dirty, hot or cold, empty or full. A kitchen fridge might know it's contents and when they were all purchased. A light-bulb might know if it's on or off. As more devices become connected, the power of the aggregate system grows very quickly, creating some remarkable possibilities.

What if your name-tag could introduce you? What if your coffee cup knew what kind of coffee you liked? What if your kitchen knew you were trying to lose weight? What if your photos knew where they were taken? What if your shoes could teach you how to dance?

The vision of connected computers everywhere is certainly not unique. The Media Lab calls it "Things That Think." Xerox calls it "Ubiquitous Computing."[49] Apple calls it "The Third Wave of Computing." Sun calls it to mind by saying, "The Network is The Computer."[29] What all these visions share is that at some point in the near future, computation and connectivity, like power and heat, will become just another component of the technological fabric of our lives.

Attempts to automate and connect our world have been made in the past. They have all failed because they try to do too much. If you go to Walt Disney World, or any of a number of other places, you can see some sort of "Home of the Future" type display. In these mock-ups there are certain scenarios that are repeated again and again: The thermostat turns on before you get home. When you walk into your home, the lights and the stereo turn on. When your alarm clock goes off in the morning, the coffee machine starts. When the phone rings, the stereo turns off.

The problem is that all of these attempts have been pretty monolithic. The coffee machine that starts when your alarm goes off, actually needs it's own alarm because it has no way of knowing how yours is set. When you change your alarm clock, or even when the clocks change, you'd better remember to change the coffee machine's clock too or else you're going to wind up with burnt coffee.

Sometimes a company will release several appliances that make some attempt to intercommunicate. For example, if you buy a Sony VCR, you will find that the remote control has buttons for Sony Televisions and vice-versa. This approach is successful to some extent, and certainly helps Sony sell that second or third piece of consumer electronics into a home. The problem is that no single manufacturer is good at building all things. While Sony may make great televisions, they do not currently sell ovens, so having your television tell you when your turkey is finished is not currently possible. The problem is that our things have no way of talking to each other, and even if they could they don't speak the same language.

2.3 The Need for a Common Toolkit

For several years students and faculty at the Media Lab have created Things That Think of one sort of another, ranging from the useful to the exotic, the mundane to the bizarre. See Appendix A for a discussion of some of the early projects that helped inspire Hive.

After spending a year or two trying to build these devices in one way or another, a number of insights have emerged. First, while a great deal of effort had gone into the design of Things That Think from a physical sense, very little work had been done to systematize their design from a digital sense. Most of the time spent on TTT projects was devoted to creating an object that was compelling in a physical sense, but from a digital perspective the design was often an afterthought. The result was

that an enormous amount of work was being duplicated between projects. Many of these devices shared a basic requirement of being able to grab some data from the web, or communicate amongst themselves or provide some sort of interface. A lot of the hard work which went into building this functionality was being duplicated over and over between projects. Simple functions like user interface components were duplicated incompatibly several times between, and even within, different working groups.

The second insight that emerged was that while each of our projects were interesting on their own, networking them together into some sort of coherent system was hard enough that it was not being done. Although the devices built within the Media Lab have certainly been more unusual than those created by consumer electronics companies, the same problems have emerged. Each of our devices has been monolithic; a one-off. The barriers to interconnection posed by different platforms, languages, work environments and even mind-sets was hampering our ability to interconnect our devices forcing each of them into isolation.

The Hive Development Team
Clockwise from upper left: Raffi Krikorian, Nelson Minar, Matthew Gray, Oliver Roup, Michael Goertz

2.4 The Vision

The Hive project was started in August 1998 with the recognition that interconnecting our various devices in a robust and useful manner was hard, and that the right box of tools, and more importantly, *the right way of thinking about the problem* could bring our vision closer to reality. By systematizing the digital design of our devices, we could provide a common basis upon which all Things That Think could be built, as well as expose their capabilities to each other, so that finally, the whole really could be greater than the sum of it's parts.

Hive is intended as the universal glue that will tie together objects conceived by different groups with different agendas. Hive lets everyone stick to what they're good at. There is no reason that Sony Televisions shouldn't be able to talk to Frigidaire Fridges and Braun Bathroom scales.

The name Hive is a metaphor: It conjures the image of a living network, with thousands of independently acting entities, perhaps with competing agendas, but that from the system as a whole, complex and wonderful behavior will emerge.

Like the fax machine and the Internet, Hive is only interesting when there are many

connected devices. From the start Hive has been conceived, not as an exercise in theory, but as a practical toolkit to make building Things That Think easy. While using Hive within the Media Lab is certainly interesting, our vision for Hive is much bolder. The goal for the Hive project is to create a standard by which devices connect and intercommunicate across the Internet. What the World Wide Web has done for documents, Hive aims to do for process. We imagine people all across the Internet connecting their own devices, making their own systems, contributing to the growth of the Hive. We imagine a rich living network that stretches as far as humanity where devices connect and intercommunicate and share; not just data, but behavior. We want to make the Internet come alive.

Chapter 3

Design

3.1 Design Criteria

The real test of a Toolkit is in if anybody uses it. The software landscape is littered with projects that, for one reason or another, never get used. Sometimes they are too obscure and nobody knows about them. Sometimes they are too complicated and nobody can understand them. Sometimes they are poorly documented and they cannot be learned. Sometimes they are too simple, and they don't provide enough of a benefit to be useful. In order for us to consider it a success, Hive needs to negotiate all these pitfalls and actually get used. The World Wide Web, which is probably the world's most successful distributed system, provides a good deal of inspiration.

3.2 The World Wide Web

The World Wide Web is both *minimalist* and *decentralized*. It is composed of a few very simple pieces:

- A common address format for referring to documents. (The URL)
- A simple presentation language for documents. (HTML)
- A protocol for fetching documents from remote servers. (HTTP)
- The concept of the Hyper-link.

Each of these pieces is strikingly simple in design and absolutely necessary for the system to succeed. The World Wide Web without the hyper-link would be a vastly different entity. Also remarkable is what the web does not have. There is no centralized document authority or directory. To "publish" something on the web one merely needs to have a computer connected to the Internet. No outside authority or permission is needed.

Hive is designed with a similar sense of minimalism and decentralization. We set out first to specify only those parts necessary to create a useful and robust system, and second, wherever possible, to avoid centralization which would act as choke points for growth if Hive were ever to spread very widely. We specifically avoided trying to solve any of the big hard problems that have frustrated computer science for years like natural language understanding or machine learning.

3.3 Cells, Shadows and Agents

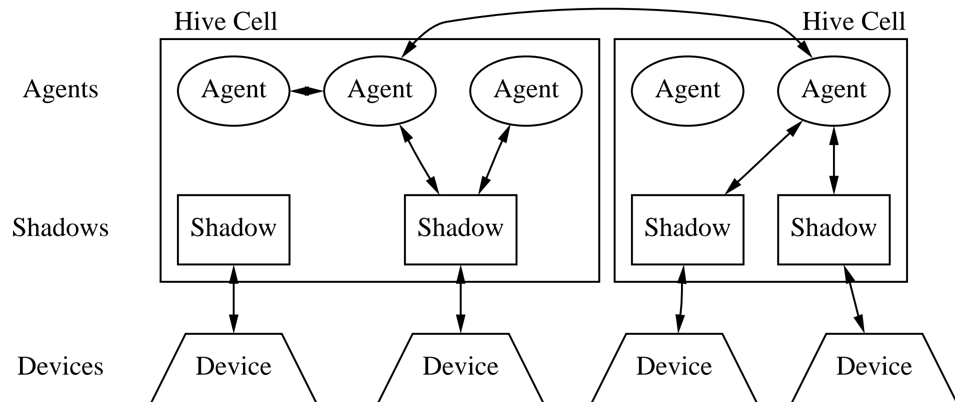


Figure 3-1: The Hive Architecture

As its name implies, Hive is based upon a biological metaphor. A Hive is composed of a collection of *Cells* which act as host to a population of *Agents*: mobile computational objects with some sort of agenda. Cells also play host to a number of distinct local resources called *Shadows*. A lookup scheme is provided to allow Agents to discover Shadows and each other and to facilitate ad-hoc connections. Applications are implemented as the emergent behavior of an *ecology of Agents*[28]: a population of Agents that move from Cell to Cell, discover resources, and interact with each other and the local Shadows. The notion of an ecology of Agents and many of

the other organizing principles of Hive originated in an Agent toolkit developed by Nelson Minar called *Straum.*[27] Figure 3-1 illustrates the Hive Architecture.

3.3.1 Cells

We call a computer running the Hive software a *Cell* [1] The Cell has five jobs:

1. A Cell provides an environment in which Agents can live. Whenever an Agent executes an instruction, allocates some memory or manipulates a Shadow, it is consuming resources on the Cell. The Cell may decide at any time to kill off the Agent. A polite Cell will not do so without warning, but this is not guaranteed.
2. Cells must provide facilities for the life-cycle and mobility of Agents. A Cell can create a new Agent, kill an existing Agent, move an Agent to another Cell or accept an Agent arriving from another Cell. Agent mobility is discussed in 4.3.1.
3. Cells host services which are local to that Cell as Shadows and provide a mechanism by which Agents can request access to them. See 3.3.2.
4. Cells provide a set of services that allow Agents and Shadows find each other and meaningfully interact with each other. This lookup scheme is discussed in 3.5.
5. Cells provide a security scheme to protect a host from it's Agents. Hive's security scheme is discussed in 3.6.

3.3.2 Shadows

Inside each Cell there exist a number of Shadows. A Shadow is a software representation of a device associated with a Cell. The choice of name reflects the idea that physical objects cast a Shadow into the virtual world. Shadows can be thought of as device drivers for things. Because the devices connected to a Cell would likely vary widely, different Shadows might be available on different Cells. They provide access to a device particular to a Cell.

A Shadow has 3 jobs:

[1] We often use the words *Cell* and *server* interchangibly. The word Cell was chosen to illustrate that the relationship between them is completely peer-to-peer, whereas servers and clients have a hierarchical relationship.

1. A Shadow provides an interface for the capabilities available on a particular device. For example, a motor might expose two methods:

   ```
   public float getSpeed();
   public void setSpeed(float speed);
   ```

   These allow manipulation and query of the state of the motor.

2. A Shadow must provide physical protection for the device. In the motor example given above, the `setSpeed()` method must prevent the motor from being burnt out.

3. A Shadow must provide concurrency protection for the device. Depending on what the motor was connected to, it might make sense to ensure that only one Agent could connect to the motor at a time.

Any device that is part of a Hive system must at some point be connected to a Hive Cell. Because Shadows provide support for specific capabilities of a Cell, they are not mobile and do not need to be written in platform independent code. Shadows may utilize any type of underlying facility to control the device they represent. Ideally a device would be "Hive Native" i.e., have a Hive Cell embedded directly into it, but for the time being this is too expensive in a great number of cases. Instead, a common scenario is that a single Cell will act as a proxy for a number of devices. These devices can be connected to the Cell through any convenient means. As the cost of computation continues to fall the Cell will get pushed closer and closer towards the device. One day, silicon will be so cheap that devices as simple and low cost as light-bulbs could ship with their own Hive Cell built right in.

Because Hive is designed to be deployed on a very wide scale we can expect that in various parts of the world, different connection technologies will be economically viable. For example, in the developed world at some point it might be viable to put Hive Cells into light bulbs, but we can expect that the same will not be true of the developing world. There, a user might be able to afford a single PC acting as proxy for all the light-bulbs in a building. Shadows should be designed such that from a programmatic perspective, the two situations are identical. An Agent will simply need to find a Cell that hosts a light-bulb Shadow, but whether that is the only resource available on that Cell or it is one of hundreds should be transparent.

There are a few common means by which devices are connected to computers. For example, numerous devices are connected to computers via a serial port. Rather

than force the user to re-invent the wheel every time they wish to connect a serial device to Hive, a number of *template Shadows* are provided. These Shadows do most of the hard work of utilizing a particular connection medium and only need be expanded by the user to address the specifics of the device they wish to connect. For a discussion of the Shadows that are currently available, see 4.2.

By design, Shadows are not accessible remotely from the network. It is not possible to connect to a Cell remotely and query or manipulate it's Shadows. Rather, an Agent must expose a Shadow to the network. This is for two reasons:

1. It relieves Shadows from security considerations. Shadows need not consider if a given user is permitted to perform a given function. This simplifies Shadow design greatly.

2. It promotes *code stability*. Because the interface to a Shadow will never be accessed directly, designers can focus on completeness rather than elegance. A Shadow must expose all the functions of the device it represents, but it needn't focus on getting it's abstraction right, or having clean network interfaces; that is deferred to the Agent.

Because Shadows provide access to a local device, they are not mobile and generally must be installed by the Cell administrator. In order to be able to manipulate a physical device, a Shadow must be *trusted code*: Before installation, an administrator must be confident that a Shadow will behave as advertised.

3.3.3 Agents

Agents are the actors within a Hive system.[3] When a user seeks to implement some new kind of application within Hive, they do so by implementing one or several new Agents. The very simplest Agents just sit on top of Shadows and expose them to the network. More sophisticated Agents can move around from Cell to Cell and interact with not just Shadows, but other Agents and even create new Agents of their own. All of this interaction is performed on an ad-hoc basis: there is no central list of resources or other Agents available on the network. Rather, Agents use a sophisticated lookup scheme to discover each other.

Agents have six characteristics:

1. An Agent has a *thread.* Agents have agendas and potentially perform computation of their own.

2. An Agent has a *host.* Every time it performs computation or consumes memory it is taking resources from the Cell on which it resides. A polite Agent will consume minimal resources from it's host because it may be terminated at any time.

3. An Agent has a *location.* Agents can only interact with Shadows located on the same Cell as them, so the location of an Agent is critical. Different resources like memory and processor time will be more or less abundant on different Cells, so Agent location is important even if an Agent does not access any Shadows. [2]

4. An Agent is potentially *mobile.* If an Agent wants to interact with resources available on a different Cell, the Agent can package itself up, travel out through the network and arrive at a new Cell with all of it's state intact. Agents are not *required* to be mobile, if they wish they can stay put, but a great deal of the infrastructure of Hive is dedicated to providing facility for Agent mobility.

5. Agents are *designed defensively.* In contrast to Shadows, Agents are remotely accessible from the network. An Agent can connect remotely to a Cell and ask for handles to all resident Agents, so when an Agent receives a command it must decide whether or not to accept it. This decision can be made on the basis of security considerations or any other concern.

6. Agents are *self-describing.* In order to facilitate the ad-hoc discovery and interaction of Agents and the resources available to them, every Agent carries around a powerful description of itself that it can modify at any time. This details of this scheme are discussed in 3.5.

By exposing a Shadow to the Network, an Agent embodies the *security policy* for that Shadow. If an Agent rejects a command that it receives, any Shadow underneath never sees that command. Thus, the Agent takes responsibility for protecting the Shadow from malicious access. This strategy creates significant flexibility. If a user wishes to revise the security policy for a Shadow, all they need to do is have

[2] In "A Note of Distributed Computing"[47] Jim Waldo effectively argues that distributed computation is inherently different from local and that because of failure and latency, attempting to mask the difference between them is ultimately a perilous endeavor. The consideration of location inherent in programming with Agents forces the user to recognize this distinction.

a different Agent expose it. Even if the new security policy is radically different, no changes are needed in the underlying Shadow, because Agents isolate Shadows from the network.

Because Agents completely shield a Shadow from the network, Agents can change the abstraction for the device they represent. This is critical because often there are several different metaphors that can be applied to a single device. Isolating Shadows from the network and using Agents to expose them allows a single device to be exposed with different metaphors without the need to revise the Shadow. This promotes code stability in the Shadow. This ability to change abstractions is one of the key benefits of Hive. An example is discussed in 3.4.

3.4 Example Architecture

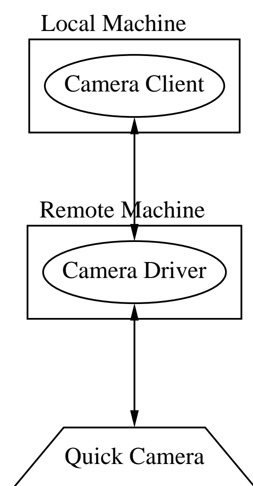


Figure 3-2: Camera: A Conventional Architecture

One of the early devices we connected to the Hive was a Connectix QuickCam. Figure 3-2 illustrates a conventional architecture for exposing a camera across the network. In Hive, the device is supported using a Shadow called `QuickCam`. The machine that hosts the QuickCam happens to be a machine running Linux, so we found it convenient for the Shadow to simply make calls to a command line program `cqcam` which is available for Linux. The Shadow has a method:

```
public PPM getPPM();
```

which returns an image when it is called.[3] Shadows are allowed to be platform specific and non portable, so this strategy is fine. When we install a QuickCam on another machine another strategy might be appropriate.

The most obvious way to think of a camera is as a frame-grabber. A user can request an image from the camera and it will return one. So, the default camera Agent called `CameraAgentImpl` simply re-exports the Shadow:

```
public PPM fetchImage();
```

This arrangement worked well, but after some time, it occurred to us that we could also think of a camera as a motion detector. In other words, if we request a sequence of images from a camera, we should be able to compare subsequent frames, and if there is an appreciable difference, flag that as motion within the camera's field of view.

If we had been using a traditional object scheme, we would have had two choices:

1. We could revise the code for the camera. This would be the most common approach, but it would be painful for a number of reasons: The new code would need to be distributed to all the machines running cameras. This would be a large burden, especially if the cameras existed in different administrative domains. Essentially this would mean publishing a revised driver and hoping users would download and install it.

   Another problem is this approach results in a *merged metaphor*. Combining the frame-grabber and the motion detector into a single resource muddies the metaphor behind both of them. While this might be fine for two devices it would grow quickly unwieldy with more. Imagine a modern video camera. It can be simultaneously imagined as video camera, frame-grabber, motion detector and microphone. Trying to adequately expose all of those functions from one resource in a coherent way would be extremely difficult. Figure 3-3 illustrates this approach.

2. Alternatively, we could create a separate motion detector resource which polled the camera for frames and then compared them for differences. Although this approach avoids the problem of merged metaphors, the new code

[3] Java's design treats the Image class as a platform specific object optimized for an individual architecture, and thus are not transportable over the network. Hive defines a new image class, the PPM which is not platform dependent, so it can be moved over the network, albeit at the cost of lower performance.

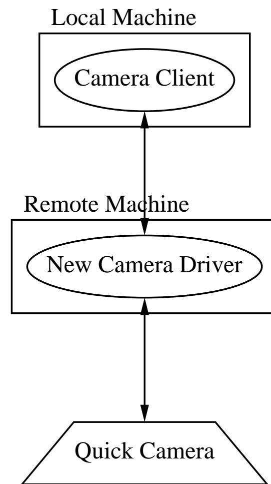


Figure 3-3: Camera: A Revised Device Driver

would still need to be distributed to all the machines running cameras. There would be no guarantee that a camera a particular user wanted to access would have upgraded to the new software. Figure 3-4 shows this approach.

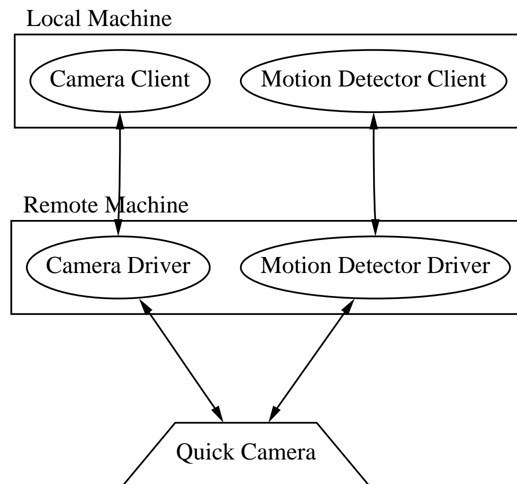


Figure 3-4: Camera: A Separate Resource

3. A final option would be for a user to create a motion detector resource on their own local machine and have it connect remotely to any available camera, polling for frames. This alleviates the burden of code distribution, because any user who wants to use a motion detector can host their own. Also, there is no problem of merged metaphors because the camera and motion detector are clearly separate objects.

The downside is that pulling all those frames over the network introduces a lot of network traffic, a lot of latency, and most importantly, a lot of conceptual awkwardness because the camera and the motion detector which represent the same physical device are hosted on two different computers, and for no reason other than historical precedent: Their development took place at different times. If this kind of split were multiplied across many iterations and many different devices, the system would very quickly become difficult to comprehend. This approach is shown in figure 3-5.

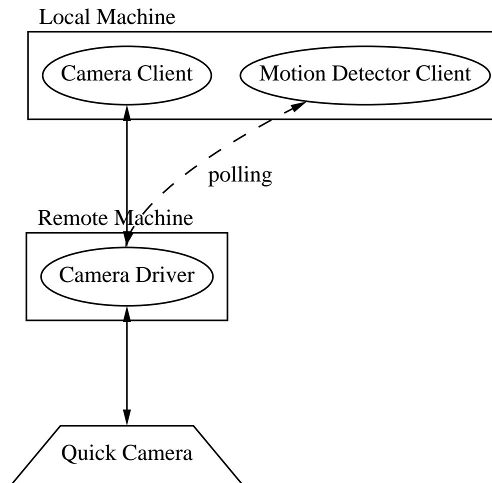


Figure 3-5: Camera: Polling Across the Network

Hive provides an alternative. In order to consider a camera as a motion detector as well, a user can write a new Agent that encapsulates the motion detector, and then the Agent can *migrate* to the Cell hosting the camera Shadow. This is very powerful because there is none of the latency, none of the network traffic, and none of the conceptual awkwardness associated with the more traditional scheme. No code need be hand installed by hand because the Shadow handles all of the code for manipulating the device itself; The new Agent merely manipulates the Shadow that is already there. The Hive implementation is shown in figure 3-6.

3.5 Lookup

By encapsulating the resources and agendas of a system into separate Shadows and Agents and providing for the mobility of the Agents, Hive creates an enormous

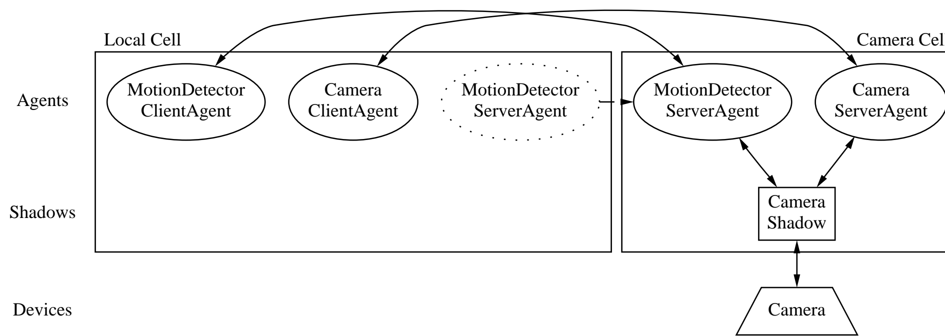


Figure 3-6: Camera: Hive Implementation

amount of flexibility in it's design. Agents can move from Cell to Cell and interact with whatever resources are available without necessarily knowing in advance what those resources will be. With this flexibility comes a challenge: If there are many resources and many Agents, finding the one that one wants becomes increasingly difficult. Agents need a way to find the Shadows and other Agents that they are interested in communicating with.

This is made more difficult by the decentralized nature of Hive. If it is a widespread system then there will be many resources and Agents implemented in parallel without knowledge of each other and there will be no central authority to distribute information. To be useful, Hive Agents need to be able to traverse a very large system and discover appropriate resources, even if those resources were designed well after the Agent. Hive attempts to address this problem by use of a rich lookup scheme.

In Hive, all Agents and Shadows are *self-describing*. The basic mechanism for Agent lookup is simple: Through a Cell, an Agent can ask for any Agents that match a particular description. The Cell compares this request to the description of the Agents it is hosting, and returns a list of all Agents matching that description. This list can then be further narrowed by additional queries.

3.5.1 Avoiding Unique Names

Hive does not provide any sort of centralized or authoritative directory structure. Agents are inherently transient. They are born, they move around, they die. They can modify their own descriptions. Attempting to collect all that data and keep it updated raises a host of problems and represents an administrative burden that

doesn't seem worthwhile. It raises other problems as well. In order to be able to list resources, a directory structure forces one to introduce the notion of a *unique name* for a resource.

Trying to create a consistent scheme of unique names actually turns out to be quite hard because it hits on a number of thorny philosophical problems. If an Agent makes an exact copy of itself, should the new Agent be considered the "same" as the original? If a Cell crashes and then restarts, recreating all of it's default Agents, are those Agents the "same" as those that were there before the crash? Imagine we have several identical cameras placed throughout a building exposed as Hive resources. It seems like a fair assumption that those cameras should have different names, because if one wants the camera on the fifth floor, then the camera in the lobby is not appropriate. However, what if the resources are not multiple cameras but rather multiple copies of the Boston phone-book? In this case, any up-to-date copy of the phone-book is interchangeable with any other and could be considered in some senses, "the same".

Hive avoids these issues entirely: Neither Hive Shadows nor Agents have names that are guaranteed to be unique. Instead, whenever an Agent gets a handle to a Shadow or another Agent, it is always as the result of a query. Agents are included in the results of a query based upon the description that they publish about themselves.

3.5.2 Syntactic Lookup

The first question an Agent asks about a resource is "What type of thing is it?" We call this the *Syntactic Description* of an object. This is really analogous to the question, "Does my plug fit into the resource's socket?" The answer to this question describes compatibility at the most most basic level. If a particular resource is of a type that an Agent has never heard of then there is little chance that meaningful communication can take place.

In Hive, syntactic descriptions are defined by the Java type-hierarchy. If an object implements a particular interface, then it can be considered a representative of that type. Because Java is a strongly typed language, the Syntactic description of an object is defined at design time and cannot be changed subsequently. This system works well because object oriented programming was designed using objects as the metaphor, so trying to map this to real physical objects is not too difficult.

An example of an interface that might be used within Hive is the `Toggleable` interface. We might define `Toggleable` as follows:

```
public interface Toggleable {
    public void setState(boolean state);
    public boolean getState();
}
```

An Agent that understands what `Toggleable` means can manipulate any object that implements that interface. This understanding provides the plug level test of compatibility that is necessary for communication to begin.

3.5.3 Semantic Lookup

The second question an Agent asks is "What does this thing mean?" We call this the *Semantic Description* of an object. The answer to this question provides context about the object that defines it's place in the world.

While there may be millions of Toggleable devices in the world, they are not interchangeable. One toggle may control power to a desk lamp; another may represent a circuit breaker to a building. Understanding the difference between these two devices is critical.

Hive uses a system where a resource can declare arbitrary key-value pairs to describe itself. For example, a `Toggleable` might wish to declare the following information about itself:

```
role = light-switch
owner = oroup@mit.edu
location = MIT Media Lab Room 468
install-date = 1999.05.21
```

Figure 3-7: Sample Semantic Label Data

Objects can inherit semantic descriptions from their parent or parents. For example, all light-switches in the Media Lab might inherit their owner from a common prototype light-switch. Thus, by changing the owner of the prototype light-switch, the new owner would propagate down to all the light-switches in the building.

In addition to simple strings, the value in a key-value pair can be a more complex

object. In the above example, rather than simply using an email address for owner, the resource could use a more complex object for it's owner. An owner, might be an object that itself had a name, an email address, and an employee number.

Semantic descriptions are really just arbitrary directed graphs. When an Agent wishes to find a resource that matches a description, it constructs one with the relevant fields filled in and then the Cell returns a list of the resources with descriptions that match. Figure 3-8 is a sample semantic description.

Semantic descriptions are much more thoroughly described in Matthew Gray's Masters Thesis, "Infrastructure for an Intelligent Kitchen".[9]

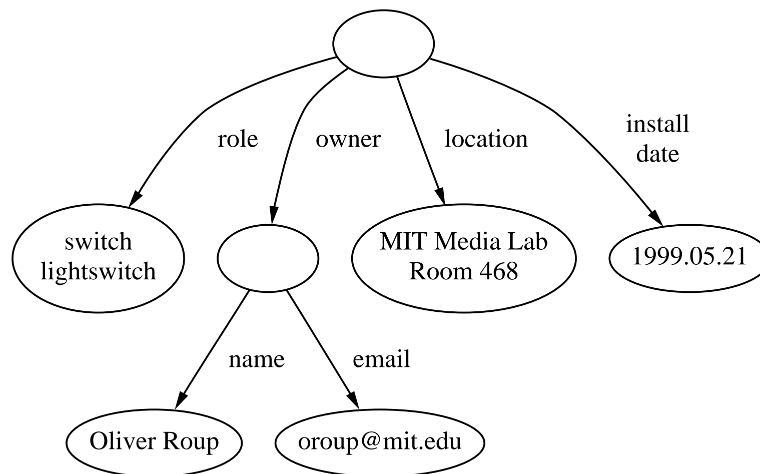


Figure 3-8: Sample Semantic Description

3.5.4 Object classification

An important consideration is how these semantic descriptions should be organized. A description will be of no use if nobody else uses the same terms of reference. Even with something as simple as a light-switch, one can imagine that a different user might have defined a light-switch as having a "purpose" instead of a "role" or a "caretaker" instead of an "owner". Humans can rely on their understanding of natural language to realize that the terms are similar, but computers have no such facility, so our classification of objects must be carefully designed.

Classes of objects are amenable to organization into hierarchies. *Taxonomy* is the orderly classification of plants and animals according to their presumed natural relationships. Taxonomy classifies organisms by Taxon. For example, everything is

assigned a Kingdom, Phylum, Class, Order, Family, Genus, and Species. Organisms belonging to the same kingdom are not necessarily very similar; organisms belonging to the same species are very similar and can produce offspring.

Generic objects can be similarly arranged into hierarchies. The sentence, "I have a Sony Trinitron" is much more specific than "I have a Television." A Sony Trinitron, is a relatively specific type of television, while "a television", is really more of a placeholder concept for which many different classes of objects could be appropriate.

In the case of living organisms, the Linnaean System is more or less universally accepted as the de-facto classification. Many other fields have significantly less agreement on classification of their contents. It may be tempting to try and fit all objects into the world into a single hierarchy, but if one starts to consider the wealth of resources in the whole world things start to break down quickly. Is a black and white television also a descendent of `television` or is it a different thing? Televisions can be used to listen to audio, so perhaps they should also inherit from `loudspeaker`. Is a video projector a descendent of `television` or is it something different? Attempting to classify everything in the world into a single hierarchy is a losing proposition, especially given that new things are being invented all the time. Even within Taxonomy, the seven terms provided by the Linnaean System have proved inadequate to classify all living things, forcing the application of terms like Cohort, Infraorder, and Tribe.

Hive skirts this issue by avoiding a single hierarchy for all objects. Instead, objects can be classified in multiple unrelated hierarchies. We use the word *ontology* to refer to an unsorted collection of hierarchies. Hive does not attempt to actually specify any of these ontologies. Rather, we imagine that consensus will emerge within certain domains, much the way that consensus emerged among the Life Sciences community on the Linnaean System.

3.6 Security

If we are to consider embedding computers into our everyday objects and then connecting them to the Internet, security will be of primary concern. While one might want to know precisely when they have run out of milk, they probably don't want telemarketers to know. There are countless pieces of science fiction that clearly document our fears of what happens when our machines "get too smart". If Hive

is to be successful, it needs to address human fears about the ubiquitous spread of technology that it represents. These can be divided into two separate concerns:

1. The components of Hive must only be able to do things for which they have been given permission. This concern is *security.* Although much study has been done on security matters, the introduction of mobile code into Hive raises all sorts of security concerns that need to be carefully addressed.

2. Hive must have a good strategy for deciding which things will be permitted and which will not. This concern generally falls under the category of *privacy*, and is more poorly understood. It is discussed in 5.1.2.

Security concerns can be divided into three separate categories[40]:

3.6.1 Protecting Cells from Agents

Nobody will be prepared to install a Hive Cell if there is a risk that an Agent can come onto their system and erase their hard-drive or steal their private data. This is the most urgent of our security concerns, but also the one that we are most equipped to deal with. There are two separate strategies for protecting Cells from Agents:

1. Every Hive Cell has a *security perimeter.* An Agent cannot get into a Cell unless it is allowed to do so. This decision could be made on a number of criteria. The safest strategy would be for a Cell to never let any new Agents inside. The existing Agents could still communicate into and out of the Cell, but no new Agents could gain entry to the Cell. A second, more liberal strategy would be for the Agent to produce some kind of credential when it arrived, and the Cell would decide whether to let the Agent in based on that. The credential could be a password or a public key signature of the Agent's code. Finally, a Cell could do some sort of heuristic analysis of the Agent and attempt to determine if it is malicious. This is not necessarily possible and at best would only be useful in a limited number of cases.

   Hive currently implements a rudimentary password based credential system for it's security perimeter, but this system will require much improvement if Hive is to see widespread deployment and popularity.

2. In addition, every Hive Cell has a *permissions policy.* Once the Agent is inside the Cell, the Agent should be restricted to only do those things that it has been given permission to do. In most programming languages enforcing that kind of security is not possible, but the Java Platform has extensive support for a *sand-box* which can protect a machine from malicious code.[5]

   Presently, Hive does not implement a permissions policy, but see 5.1.1 for discussion of future implementation plans.

3.6.2 Protecting Agents from Agents

Protecting Agents from each other is a much tougher problem. The general model is that when an Agent receives a particular command it should decide whether it wants to act on it or disregard it. This decision should be made based upon the nature of the request, the identity of the requestor, what else is going on in the Cell and potentially other factors.

In practice, this is tough to implement. Any Agent that has particular security concerns is free to implement it's own security apparatus, but there is not currently any structural support for inter-Agent security policies. See 5.1.1 for potential future implementation strategies.

3.6.3 Protecting Agents from Cells

If an Agent is empowered to conduct some bidding on a user's behalf or possesses some sort of confidential information, it would be nice to be able to protect Agents from their hosts. For example if an Agent contains some sort of auction bidding algorithm there is nothing to stop a Cell from freezing the clock, analyzing the Agent's internal code to extract the algorithm, and then using that knowledge against it to obtain the most favorable possible price. Unfortunately it is not clear that solving this problem technologically is even possible.

A potential solution to this problem is the use of *reputation incentives.* If a particular Cell is a marketplace for Agents to conduct business in, it's reputation is important to it's livelihood. If that Cell becomes known as a rigged marketplace, business will move elsewhere. If a group of Agents always have several Cells to choose from in which to conduct their business, Cells will have an incentive to maintain an honest reputation. Still, this scheme is inadequate for extremely valued transactions, where a Cell might consider destroying it's reputation to be a

worthwhile cost if the payoff is large enough. In this case, the only strategy is to prevent confidential information from moving to an insecure Cell. An Agent in the marketplace could essentially "phone home" to determine it's bidding strategy. The bidding algorithm could thus be protected on a trusted Cell.

Chapter 4

Implementation

4.1 Underlying Technologies

Hive is a software toolkit written in version 1.1 of the Java programming language.[4][22] It comprises roughly 22,000 lines of code of which approximately 10,000 lines are core to the Hive system itself, while the remainder is Shadows and Agents for various sample applications.

4.1.1 Java

In order to meet our design goals, the implementation platform for Hive needed to meet several criteria:

1. It needed to support *mobile code.* The design of Hive is absolutely reliant on the idea that running code should be able to move from one host to another.

2. It needed to be *cross-platform.* Specifying a particular processor architecture or operating system as vital to the design of Hive could only limit the success of Hive.

3. It needed to be *secure.* Hive facilitates the movement of pieces of executing code between hosts on an untrusted network. A mechanism needed to exist to assure users that Agents that arrived via the network would not destroy their new host. One alternative would be to enable code-signing so that the origin of code could be verified; Much preferable is a sand-box, whereby specifically

untrusted code can still be run with confidence.

4. It needed to be *available*. Given our desire to get real results in a short period of time, implementing our own language and environment from scratch was not an option we were prepared to consider.

Given the requirements, it is a fairly straightforward decision that Hive should be written in *Java*. While other interpretive environments can potentially be both cross-platform and support mobile code, no other environment offers a sand-box to protect hosts from potentially malicious Agents. Java has the added advantage that it is reasonably popular and easy to learn, so there is a greater likelihood that Hive would get used than if it were implemented in unfamiliar tools.

A disadvantage of Java is that it has relatively heavyweight resource requirements. In order to be able to support a Java Virtual Machine, a computer generally requires megabytes of memory and a reasonably powerful microprocessor. Although there are several efforts to create a lightweight version of Java suitable for embedded applications none of them appear stable and usable at the current time. We remain hopeful that one of these efforts will come to fruition. In the meantime, Hive offers strategies for the sharing of a single Cell between multiple physical devices. Ultimately, Moore's Law may render the problem meaningless. What today is uneconomical to embed in a toaster, tomorrow may well be.

Recently, JavaSoft has released version 1.2 of the Java Specification, also known as the Java 2 Platform.[23] Version 1.2 offers a wealth of improved features, most notably for Hive, a much improved security model.[17]

Support for Java 1.2 is still somewhat immature. Reference implementations exist for Solaris and Windows, but numerous bugs still exist, and support for Linux is still somewhat minimal. It is intended that Hive will be moved to Java 1.2 as soon as the system reaches maturity.

4.1.2 RMI and Object Serialization

To facilitate the creation and manipulation of distributed objects, Hive makes extensive use of *Remote Method Invocation* [37] and *Object Serialization*.[30] Although RMI is included with every Java implementation, it is not the only distributed toolkit available for Java. An early version of Hive was written using the Voyager toolkit from Objectspace.[31] Working with Voyager turned out to be problematic

because there was little recourse to fix problems that were encountered. Furthermore, Voyager's license was incompatible with our eventual intention to release Hive as Open Source.

RMI allow an object to present a different interface to callers that are remote, compared to the default interface available to local objects. This is done through the implementation of a *Remote Interface*. The type safe nature of Java prevents an object from being cast into types that are not specifically implemented. This design provides security for objects. Thus, remote objects cannot call methods that are not exposed through a remote interface. Voyager does not directly implement this feature, and instead relies upon the object itself to implement some sort of security check when it's methods are called.

4.1.3 Supporting Packages

Semantic Descriptions in Hive are stored as *XML*[6] encoded *RDF*[24] files. To facilitate the manipulation of RDF and XML, Hive uses three third-party packages: SiRPAC[39], Aelfred[2] and SAX.[41]

Command-line parsing in Hive is implemented using the GNU GetOpt package.[16] Serial port handling is implemented through use of the javax.comm package available from JavaSoft.[21] The underlying support for Serial communication in Linux is provided by the RXTX package.[38]

Cells provide a different interface to locally resident Agents than they do to Agents performing queries over the network, because local Agents are permitted greater freedom than remote ones. For example, local Agents are permitted to request and manipulate Shadows. Remote Agents are not.

All Cells are instances of the class `Server`. Cells provide methods for the managing the life-cycle; Agents can be created, moved and killed. Also, Cells provide facilities for Agents to find and manipulate Shadows.

Remotely, Cells expose the interface `RemoteServer`. A Cell exposes very few functions externally. An Agent can request the address of a Cell, query for Agents resident on the Cell and arrive at a Cell in hopes of being allowed inside. Any other functions that need to be remotely exposed by a Cell can be implemented by an Agent of some kind.

```
public class Server {
  public CellAddress getAddress();

  public AgentImpl createAgent(Class agentClass);
  public synchronized AgentImpl handleNewAgent(final AgentImpl agent);
  public boolean moveAgent(AgentImpl agent, CellAddress address);
  public boolean acceptAgent(byte[] agentBytes, Object token);
  public void killAgent(Agent agent);

  public ShadowDB getShadowDB();
  public DescSet getShadowDescriptions();

  public DescSet queryAgents(Object sender,
                             String[] syntactic,
                             String[] semantic);
}
```

Figure 4-1: Methods on a Hive Cell

```
public interface RemoteServer extends Remote {
  public CellAddress getAddress()
    throws RemoteException;
  public boolean acceptAgent(byte [] agentBytes, Object token)
    throws RemoteException;
  public DescSet queryAgents(Object sender,
                         String[] syntactic,
                         String[] semantic)
                            throws RemoteException,
                                   ClassNotFoundException;
}
```

Figure 4-2: Remote Methods on a Hive Cell

4.1.4 CellAddress

Every Cell is uniquely identified by a `CellAddress`. A CellAddress is a Common Internet Scheme Syntax compliant URL[45] that uses the new protocol identifier `hive` to denote a Hive Cell. By default, Hive Cells listen on TCP port *23231*, but an alternative may be specified in a Cell Address.

```
hive://hive.media.mit.edu:23231/
```

Figure 4-3: A Sample CellAddress

4.2 Shadows

In Hive, all Shadows are subclasses of the base class `Shadow`. Because Shadows can represent a wide variety of resources, the base Shadow class does very little. Methods are provided to initialize a Shadow, manipulate a description, and to allow loading a description from a file. Finally, a Shadow can choose to specify which Agent will expose it to the network by default. A Cell is free to ignore that recommendation if it chooses.

```
public abstract class Shadow implements Describable {
     public Shadow();

     public void init();
     public AgentImpl getDefaultAgent();

     public String getRDFFile();
     public void setRDFFile(String);

     public void loadDescription(String);
     public Description getDescription();
     public void setDescription(Description);
}
```

Figure 4-4: Methods for Shadow

4.2.1 Template Shadows

Many Shadows represent devices that are connected to the Cell through some sort of stream interface. Any device connected through a serial port, a TCP port or even many Shadows supported by underlying native libraries have a single stream of data

traveling to the device, and a single stream of data arriving from the device. These can be classified as *StreamShadows*. Often it is irrelevant what particular connection technology is being used to communicate with a device. The Hive toolkit provides a template interface, `StreamShadow` which does much of the work of communicating through streams with a Shadow, and abstracts away what particular hardware is being used for that communication. A `StreamShadow` has an underlying channel which simply provides an input and output stream to communicate with the device. Connecting a new device to communicate through a supported channel like a serial port is relatively trivial. A user need only create a Shadow which subclasses `StreamShadow` and parses the particular protocol of the new device. All the overhead of opening and managing a serial port is handled by the StreamShadow interface.

Another template Shadow is the `EventPublisherShadow`. An EventPublisherShadow is one whose device creates data of interest to multiple parties on a periodic basis. The EventPublisherShadow provides a mechanism by which Agents can subscribe for notification when an event occurs. All the work of handling subscriptions, unsubscriptions and broadcasting of events is handled by the template. A user need only decide what kind of event the Shadow should send out.

4.2.2 Common Shadows

A Cell is not required to host any particular Shadows. However, certain Shadows are used very commonly and worth mentioning.

`AgentComponentManagerShadow`: This Shadow allows Hive Agents to expose a graphical user interface. Hive Agents can supply an AWT[18] Component to this Shadow and the Shadow will put it on the local display. If the Agent goes away, the Shadow ensures that the component is disposed of. If a Cell does not host this Shadow, then Agents may not display graphical user elements. This is quite commonly the case for Cells that wish to run “headless” as servers.

`ServerControlShadow`: This Shadow allows a Cell to be administered by it’s Agents. Agents can be created, connected, killed and moved through use of this Shadow. If this Shadow is missing, then Agents will be unable to administer the Cell.

`SessionManagementShadow`: This Shadow allows the state of a Cell to be serialized and stored to disk. If the Cell crashes, the state of the Cell can be loaded off disk

so that the state will persist. If this Shadow is not present, then persistence of Cell State will not occur.

4.3 Agents

Hive uses a standard naming convention to distinguish between interfaces from local Agents and those from remotely visible Agents. A local Agent is a subclass of `AgentImpl` and all end with the suffix "AgentImpl". A remotely visible Agent is a subclass of `Agent` and has the suffix "Agent" as well.

```
public interface Agent extends Remote, Describable {
  public boolean connectTo(Agent otherAgent) throws RemoteException;
  public void disconnectFromAll() throws RemoteException;
  public void disconnectFrom(Agent otherAgent) throws RemoteException;
  public Description getDescription() throws RemoteException;
  public PPM getIconPPM() throws RemoteException;
  public String getName() throws RemoteException;
  public RemoteServer getServer() throws RemoteException;
  public void diePlease() throws RemoteException;
  public Vector listAllConnections() throws RemoteException;
}
```

Figure 4-5: Remote Methods for an Agent

4.3.1 Mobility

Although Java provides support for moving code between virtual machines, supporting Agent mobility is still rather tricky. When code moves from one Virtual Machine to another, the new Virtual Machine needs to have access to a copy of the class definitions that the Agent might require. In many uses of mobile code this is no problem, because the various Virtual Machines can share a common and centralized code repository. Because Hive eschews centralization, no such strategy is possible. Hive implements a system whereby the class definitions needed to support an Agent can migrate from Cell to Cell.

There are still some issues: Because Java is late-binding, one cannot statically enumerate all the classes upon which an Agent depends. There is always the possibility that an Agent will suddenly attempt to load a new class, but the class will not be available locally and the Cell that the Agent came from will have gone away. Hive currently has no graceful way to deal with this problem, and any Agent that en-

```
public abstract class AgentImpl
     extends UnicastRemoteObject
     implements Agent, Serializable {
  public Description getDescription();
  public void setDescription(Description desc);
  public String getIconPPMName();
  public void setIconPPMName(String  v);
  public String getName();
  public boolean connectTo(Agent otherAgent);
  public void disconnectFromAll();
  public void disconnectFrom(Agent otherAgent);
  public abstract void doBehavior();
  public boolean moveTo(CellAddress address);
  public void arriveAt(Server s);
  public void setServer(Server s);
  public void setTimeToDie(boolean flag);
  public void diePlease();
  public PPM getIconPPM();
  public RemoteServer getServer();
  public Vector listAllConnections();
  public boolean isReady();
}
```

Figure 4-6: Local Methods for an Agent

counters this situation will simply fail. Cells are meant to be relatively long lived entities, so this problem should not occur too often.

Another problem is that of *versioning*: Given Agents' mobility, it is possible that when an Agent is revised, older versions will linger within the Hive. Java does not differentiate well between two versions of the same class. Sun has indicated that in the future they may make modifications to the Java VM to address this problem. If this does not turn out to be the case, Hive will need to develop it's own strategy. One possibility is to modify Agent names to reflect what version of a particular Agent it is.

4.3.2 Template Agents

Many devices and Agents within Hive communicate using *events*. Hive provides a number of Agents that handle the complexity of broadcasting events and managing subscribers. Many Application Agents are written by subclassing one of the template Agents provided by Hive. `EventSendingAgent`, `EventReceivingAgent` and `EventTranseivingAent` are all templates provided by Hive.

4.3.3 Common Agents

`ServerEventInfoAgent`: This Agent broadcasts changes in a Cell's internal state to any interested subscribers. Agent births, deaths and connections are all noted, as well as Cell reports and pending Cell shutdowns are all broadcast through this Agent. If this Agent is not present on a Cell, then these events are not communicated outside of the Cell.

`ServerControlAgent`: This Agent exposes the `ServerControlShadow` and provides external Agents with the ability to administer the Cell. This Agent allows remote Agents to create, kill and move Agents, as well as connect and disconnect Agents and even shutdown the Cell. If this Agent is not present, then it is not possible for a remote Agent to administer the Cell. The current implementation of ServerControlAgent does not implement any kind of permissions check. Any Agent can request that the Cell be shutdown or that an Agent be killed. This needs to be substantially revised before Hive will be ready to deploy widely.

`ServerListAgent`: This Agent maintains a list of all known Cells in the Hive. It can receive events from other Cells, and also rebroadcast information to other Cells within the Hive. Currently, Hive employs a simple *master list* scheme. A single well known Cell must be notified of any change in Cell state, and it also re-broadcasts any new information to all Cells in the Hive. This scheme is highly centralized and does not fit with the decentralized design philosophy of Hive. A more scalable scheme needs to implemented before Hive can be widely deployed.

4.4 Graphical User Interface

Although it is simply implemented by a single Agent and not necessary to run a Cell, the Graphical User Interface for Hive is the first thing a user will see when they boot up Hive. The GUI gives a graphical representation of a Hive system by using icons to indicate Hive Agents and arrows between them to indicate connections. Shadows are not visible in the User Interface, but they underly many of the Agents. The border around Agent icons differs to denote which host an Agent is currently resident on. Agents can be moved, killed, connected and disconnected simply clicking on them, and a pull-down menu allows a user to create new Agents.

The graphical user interface is very useful for examining a Hive system, understand-

ing what Agents are resident on a group of Cells, and creating a new system with a completely minimal amount of effort. We believe that being able to visualize and browse a system facilitates the construction of new systems using Hive.

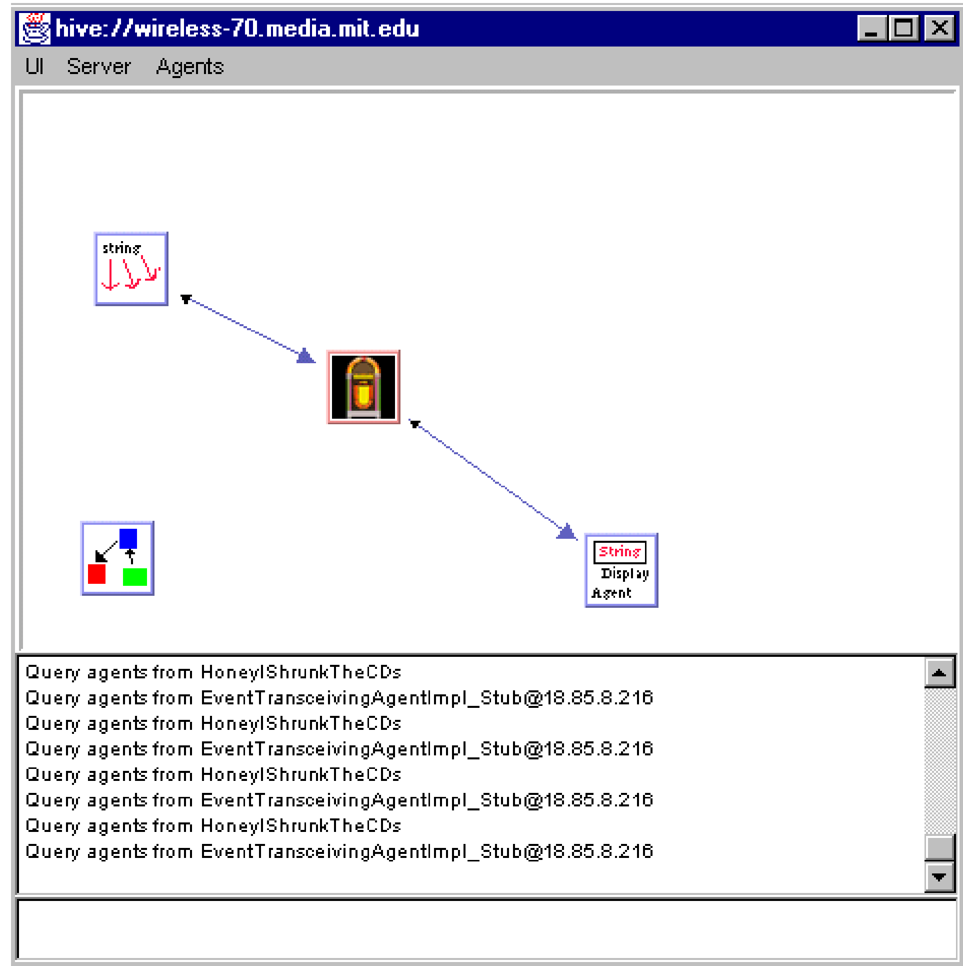


Figure 4-7: A Simple Hive Configuration

4.5 Applications

Within the Media Lab, Hive has begun to see use in a number of application domains:

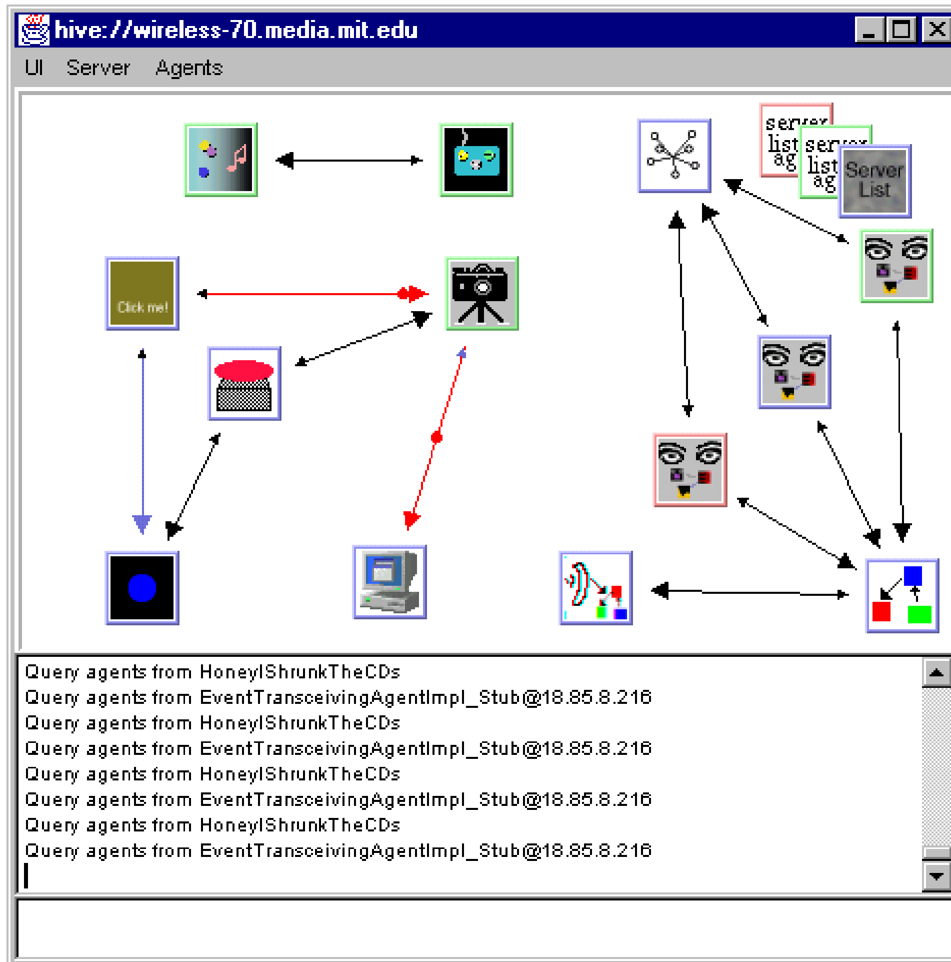


Figure 4-8: A More Complicated Hive Configuration

4.5.1 Image Manipulation

Included with the Hive distribution are a number of Agents that implement an image manipulation pipeline. A base Agent implements a generic filter: An Agent that retrieves images from a remote source, performs some manipulation on them and then makes them available for retrieval itself. A number of sample filters are provided, including a gray-scale filter, a contrast filter, and a motion filter. By chaining these filters together into pipelines, Hive provides a very simple mechanism by which image manipulation can be performed on a continuous stream of images. A few filters chained together can very quickly yield remarkably complex results. The decentralized nature of Hive makes it easy to share the burden of this filtering between numerous machines, and the ad-hoc nature of Agent communication makes it easy to reconfigure the pipeline.

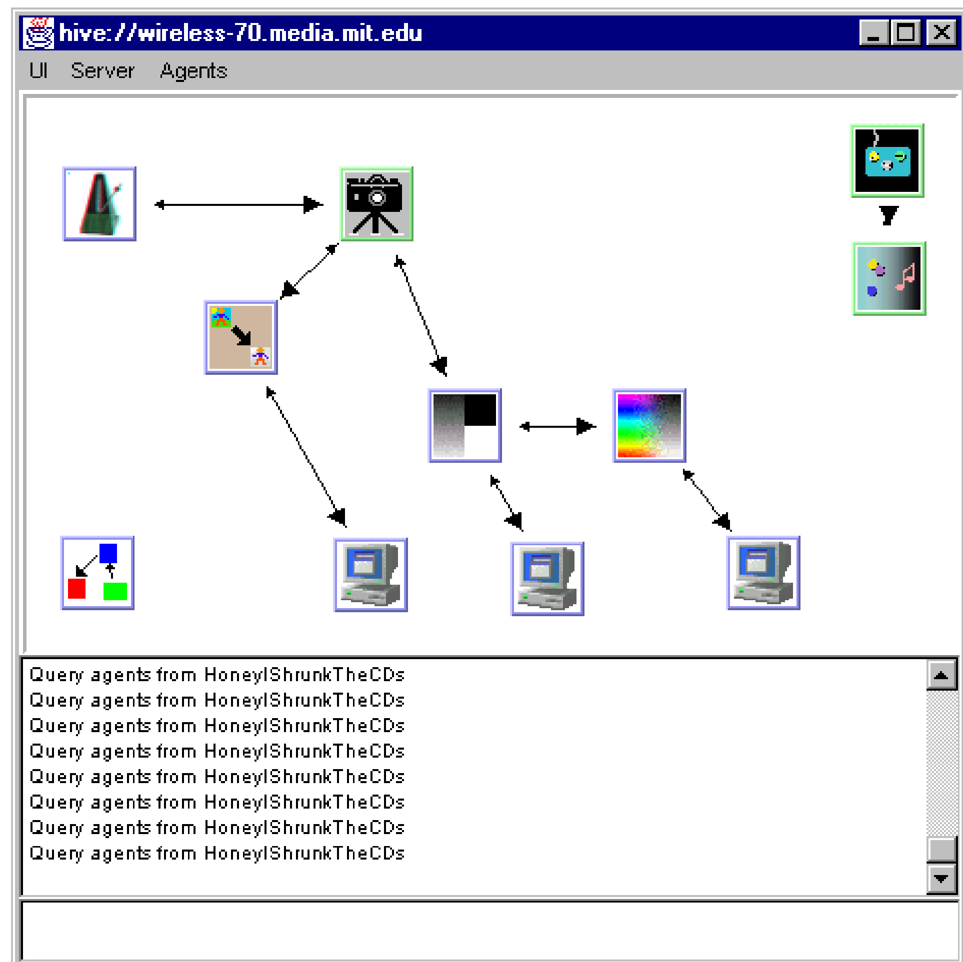


Figure 4-9: Hive Image Filters

4.5.2 Jukebox Player

Jukebox Player

A simple but very popular demonstration application is the Hive Jukebox. In one corner of a room is an RF tag reader and a number of tags. In another corner of the room is a computer with a good audio output and a large number of MP3s. Finally, a third computer contains a database mapping tag IDs to song names. When a tag is placed on the reader, an event is generated and forwarded to the database. The database maps the tag with the song name and forwards the request to the audio player which then plays the song. The result is a simple audio player with an unusually compelling interface: The "magic" tags start the audio without any visible or physical interaction.

An unusual benefit of the architecture of the system is it's stability. All three Agents in the system are constantly looking for each other, and if for some reason one of the machines in the demo fails, when it reboots, the other Agents reconnect so that the demo continues to function. The jukebox player has been successfully run for weeks at a time without interruption.

4.5.3 Where's Brad?

Where's Brad?

One of the research subjects at the MIT Media Lab is "Wearable Computing" which examines how one's relationship with a computer changes when it is always with them. Wearable research runs the gamut from head-mounted Linux machines[48] to computation embedded into fabric.[25]

Graduate Student Bradley Rhodes is investigating turning himself into a Hive Cell.[14] A number of location beacons or "locusts" are mounted throughout the lab, and the computer that Brad wears on his clothing can detect these beacons and find his location. Brad's approach is unusual because while Brad knows where he is, the room does not. Brad can then decide who to share his location with by having the Agents in his Hive Cell forward the information. Brad has used this setup to queue theme music when he walks into the building, and to display his location on a map within his research group. The Hive Cell Brad carries is run on a low powered 486 PC running Linux. This is probably the lightest weight hardware that Hive has been deployed on to-date.

4.5.4 Personal Bits

Craig Wisneski's Masters thesis, "The Design of Personal Ambient Displays" introduces a number of artifacts to communicate personally relevant information.[50] There are devices that grow warm when stock prices go up, or when somebody walks into a room, there are objects that glow when another is shaken. Craig uses Hive as the implementation layer for many of his devices because it allows him to focus on the physical design of his devices.

Counter Intelligence

4.5.5 Counter Intelligence

The most major application that has been developed with Hive is *Counter Intelligence*, a networked, intelligent kitchen.[9] Counter Intelligence integrates numerous RF tag readers, audio and visual feedback, a networked scale a even a connected microwave into a single coherent system. The Counter Intelligence system can guide a user through a recipe or even suggest one based on the ingredients on hand. The system also handles the problem of resource contention within a kitchen: It will not attempt to use more bowls than are available.

The whole Counter Intelligence system is implemented with Hive. All of the devices are exposed by Shadows and manipulated by Agents. A single recipe Agent determines what resources are available in the kitchen by querying the Cell. The system is enormously flexible. Adding another tag reader to the system requires no code changes at all: The Recipe Agent merely notices that a new tag reader has been installed and adds it to the list of resources available within the kitchen.

The Hive version of Counter Intelligence replaces an earlier version that was written as a single monolithic application. The original version ran into problems because the code for accessing the devices and the code for the behavior of those devices became difficult to separate. As the system increased in complexity, the desire for relatively minor behavioral changes in the kitchen required increasingly large modification to the system software. Also, the original application had difficulty dealing with changing or failed hardware.

Net Weight

4.5.6 Net Weight

Graduate student Bradley Geilfuss has created a networked bathroom scale and mirror.[14] When someone steps on the scale, it attempts to distinguish them

amongst a small set by looking at their toes. Their weight can then be tracked over time and the mirror is used as a display to convey information back to the user. Net Weight is currently being adapted as a Hive object so that the data it collects will be available to other devices.

4.5.7 The Future

Each of these projects would be possible without Hive. Some of them, like *Counter Intelligence* were actually developed before Hive. Most of these projects were pursued in their own right, rather than developed as an exercise to utilize Hive. In each case, Hive was used because it eases the complexity of implementation. Managing many resources between multiple machines, retrieving data from the web, handling failure are all handled by Hive.

While each of these projects are interesting on their own, the real promise of Hive is in connecting all of these systems into a single coherent entity. Wouldn't the bathroom scale be more useful if the kitchen could tell it what a user was eating? Building the kitchen would be easier if position information from the RF locusts were available. The same code that runs the tag sensors within the kitchen is used in the jukebox player. As more and more devices are connected the environment becomes more richly accessible. Imagine a world in which all of the data that ordinarily swirls around us were accessible and knowable.

Affective computing is a field where the emotional state of a user is taken into account when determining the behavior of our systems.[34] Understanding what that emotional state is is very hard if the only data we have to analyze is keyboard and mouse use. What if our computer could know that our heart was racing, or that we'd been pacing the room or that we'd just turned on fast-paced music? What if our computer knew that it was 5 in the morning and we were onto our fifth coffee? In a networked world, this information could be easy to come by.

4.6 Related Work

Hive has been influenced by a number of technologies that have either been around for some time or have arisen in a similar time-frame to Hive.

4.6.1 Jini

In January 1999, Sun Microsystems released Jini a technology layer to "enable impromptu networking of a wide variety of devices".[46] Hive and Jini share a lot in common. Both are distributed software systems intended to be small and flexible and targeted at applications involving embedded devices. Both are written in Java, and both construct applications by composing services from a distributed set of components.

Perhaps the most critical differences between Hive and Jini are those of origin. Hive is a research system developed in a research lab, whereas Jini is intended to be a commercial grade system.

Concretely, Jini is a more concretely designed system. It does not support mobility as deeply as Hive. While the stubs for a Jini service can be downloaded to a machine connecting to that service, there is no facility for a Jini service to pick itself up and move of it's own accord.

Additionally, Jini does not split it's interfaces between device capabilities and device behaviors the way Hive does using *Shadows* and *Agents*. Jini uses a single monolithic object to expose both the capabilities of a device and it's behavior. Our belief is that the separation between Shadows and Agents leads to code that is easier to maintain and re-use, but only time will tell.

Jini has a much less flexible lookup scheme than Hive. While Hive provides a rich semantic description language for describing Agents, Jini uses the Java class hierarchy to distinguish between services. Instance specific information like location or owner is exposed using Javabeans accessor methods.[42] This approach seems much more limited as it does not allow for structured descriptions of objects, but it is difficult to know whether these differences will prove critical.

Finally Jini does not provide a graphical user interface to allow users to browse the services within a federation. While one could certainly be written, we find that the graphical user interface is critical to articulating the power of Hive.

Jini has some features which Hive does not. Jini has a well defined discovery and lookup scheme for new services within a federation to announce themselves. Hive currently has no such facility and relies upon Cells to be well known to each other. Although it is not technically part of the Jini spec, Sun provides a transaction manager so that Jini transactions can remain consistent, even in the case of node

failure. Probably most importantly, Jini is supported extensively by Sun and has an enormous budget behind it.

4.6.2 Inferno

Inferno is a relatively new operating system aimed at Embedded Devices from Lucent.[26] Inferno is comprised of the Limbo programming language, the Dis virtual machine, and the Styx network protocol. Like Java, Inferno is interpreted, cross platform, and has extensive platform level support for security. Inferno requires fewer resources to run on a device, probably measured in hundreds of Kilobytes rather than thousands.

Although Inferno is an interesting system, it does not enjoy the same widespread level of support that Java enjoys. Indeed, support for PersonalJava, an embedded subset of Java was recently added.[35] Inferno offers no obvious advantage as an implementation medium for Hive. In order for Hive to be actually used, the relative obscurity of Inferno proved to be a real hurdle.

4.6.3 Corba / DCOM / RPC

Numerous efforts have been made in the past to allow computational objects interact between systems. Perhaps the best known is CORBA which is a cross platform but enormously complex object standard.[32] Also well known is DCOM which is Microsoft's "Distributed Component Object Model".[11] DCOM is also cross platform, at least in theory. Finally many operating systems support remote procedure calls, which permit procedures to be called between hosts on a network.

None of these standards offer any way to implement object mobility the way Java and RMI does. Although there is work underway to add a pass-by-value specification to the CORBA standard, this is likely to be very difficult. Because CORBA really is language independent, passing a FORTRAN object by value to a C++ System will be messy at best.

4.6.4 Universal Plug and Play

"Universal Plug and Play" is a relatively recent Microsoft initiative which attempts to address the same issues as Hive and Jini.[44] While relatively few technical details have emerged, UPnP is worth taking seriously given the power Microsoft has to

promote it's standards. It is conceivable that UPnP will look like the Semantic Description layer implemented in Hive. Whatever the case, it seems unlikely that UPnP will implement any kind of mobile code support.

Chapter 5

Conclusions

Hive is a software toolkit written in Java that facilitates the construction of systems of Things That Think. Hive has a number of unique features that differentiate it from other similar systems.

1. *Unusual Metaphor*: Hive is organized in a way that makes thinking about and designing systems easier.
2. *Mobile Code*: Hive utilizes mobile code to reduce complexity and manage the growth of systems over time. No comparable toolkit makes use of mobile code in the same way.
3. *Powerful Description Facility*: Hive provides a flexible and powerful description framework so that Agents can discover each other and perform meaningful communication.
4. *Failure Tolerance*: By composing applications from ecologies of Agents, Hive systems are tolerant of partial failure.

Since it's creation, there have been several popular styles of computer programming. Two of the most popular recently are *procedural* programming and *object oriented* programming. While technically, anything that can be done in object oriented programming can be done in procedural programming as well, in practice, things do not work out that way. Because object oriented programming is easier to conceptualize and grow over time, in practice, object oriented programming enables users to build bigger, more complex systems than were possible otherwise.

Hive hopes to makes the same leap. If a system were completely designed and thought out in advance, then any system that could be built with Hive could also be built using a traditional object scheme like CORBA or DCOM. However, because real systems change and grow over time, the organizing principles of Hive mean that we can attempt bigger, bolder projects than has ever been possible before.

5.1 Future Work

Hive continues to be a work in progress. The immediate goal for the project is that it be deployable Internet-wide. Although this has been a central design goal of Hive from the beginning, there are several things that need still need to be implemented for this to happen.

5.1.1 Security

Hive currently lacks a strong and coherent security implementation, and Hive cannot be deployed outside of a research environment without one. First, the security perimeter of the Hive Cell needs to be re-worked. The password based scheme that is currently in place is inadequate. All passwords are stored and transmitted in cleartext, and if the password for a Cell becomes known, malicious Agents can use that password to gain entry to a Cell.

A much more appropriate system would be a public-key cryptography based system where Agents are signed by their authors and a Cell can make trust decisions based on the origin of the Agent. This is similar to the Authenticode system promoted by Microsoft.[1]

Additionally, Cells need to implement a permissions based security policy. Currently, once an Agent is resident on a Cell, it has unlimited permissions. More appropriate would be for Agents to be given permission to perform certain tasks only. Implementing this policy will likely mean moving Hive from Java 1.1 to 1.2 because the more recent version of Java has a more powerful security model.

5.1.2 Privacy

Systems like Hive have startling privacy implications. When the fabric of our world becomes embedded with computation and sensing technology, managing access to

our personal data becomes incredibly important. Confidential data within Hive is threatened by a malicious Cell. The potential exists for a Cell to analyze an Agent's internal structure and extract personal information. There are a number of approaches that could be employed to mitigate this problem.

Upon receipt of personal data, an Agent could encrypt it with a users public key and discard the original. By employing this procedure, the original data could not be extracted without the users private key which, presumably the Agent would not possess. Of course this approach is not possible if the Agent needs the data to make behavioral decisions.

Another possible approach is that personal data could be split amongst multiple Agents living on different Cells. The idea is that the data could not be compromised without compromising all of the Cells that the various Agents were resident on. Implementing this strategy effectively would be very difficult and the approach requires further study.

5.1.3 Distributed Lookup

Hive's lookup facilities are currently all based within a single Cell. If an Agent wishes to find another Agent within a population of several Cells, it currently has no choice but to connect to all the Cells sequentially and perform it's query on each of them. What would be very useful is some sort of distributed lookup facility, so that a single query could locate Agents across multiple Cells. Rather than building this into the base architecture, an appropriate way to implement this would be to create "census taker" Agents that wander from Cell to Cell making lists of all the Agents available on a number of Cells.

5.1.4 Beyond a list of Cells

At present, information about the Cell population is maintained in a centrally located list. This is a temporary solution that will not scale well. Information about available Cells needs to be maintained in a database held in a more distributed manner, something along the lines of the DNS system which maps host names to IP addresses on the Internet.[12] Because Cells are intended to be longer lived than Agents, the relatively long latency times of distributed databases should not pose a problem.

5.1.5 Out of Band Channel Negotiation

Hive uses Java RMI for all inter-host communication. While this is relatively efficient, there is some overhead involved in serializing out complete Java classes. While this is fine for moderate data rates, there are some applications that are dependent on particularly low latencies or channels with specific characteristics, like UDP datagram connections. In this case, Hive could provide the discovery and lookup service for two resources to find each other and then provide facility for negotiating a private channel amongst themselves.

5.1.6 Interface to the Web

Given how popular the World Wide Web is, it would be useful to provide a bridge between Hive and the World Wide Web. This might take a number of forms: Hive could expose it's user interface so that it could be embedded into a web page through a Java Applet. This would make it easier for more people to participate in Hive because rather than having to download and install the Hive code, a user could simply connect to a web site to manipulate a Hive Cell.

There are already Hive Agents which retrieve some information from the web. This connection could be made the other way by allowing Agents to publish information as web pages. This would provide a window into the state of the Agent, but make it much more accessible.

5.1.7 Periodic Disconnection

Despite the relative ubiquity of networking, there will still be cases when certain devices need to be disconnected from the network. Hive Agents currently fail when they lose their connection to other Agents. Agents could be modified so that they deal gracefully with a lost network connection, for example by queuing up events until the network connection is restored.

5.1.8 Licensing Ramifications

Hive is intended to be released as an open source[33] toolkit. A number of open source licenses like the GPL[19] place requirements on whether software it covers can be distributed or linked against other software that it does not cover. Words like *linked* and *distributed* have well understood meanings when software resides

on a single computer, but the introduction of mobile code muddies their meanings considerably. Is software that moves around within a network being distributed? Should run-time connections be considered linking? If source code must be distributed alongside object code, then do Agents need to carry around a copy of their source code? These issues are complex and will likely be debated more heavily as mobile code becomes more common.

5.1.9 Cell Description

Just as there are web sites that are well known for particular types of information, it is possible that there will be Hive Cells that develop a reputation for the types of Agents that are usually present there, or for certain kinds of resources. To facilitate the discovery of these Cells it might be helpful to for Cells to describe themselves, much as Agents do. Rather than adding this facility to the Hive Cell, an appropriate way to implement this might by having a `CellDescriptionAgent` expose a description for a Cell.

5.2 Future Applications

Beyond the straightforward goal of spurring the proliferation of Things That Think, Hive offers the opportunity to create some really ambitious projects that would be difficult to build otherwise.

5.2.1 You Are The Star

Imagine a toy that begins as Karaoke. A user gets a basic box with a microphone and a speaker. They put it down in the middle of their living room and it acts like simple Karaoke. The magic comes when the user buys attachments, each of which has a unique effect; Some of them add effects to the sound or add harmonies, some of them control lights within the living room. None of the attachments need to be setup or configured, because of all of the devices in the system can discover and communicate with each other even though they were built at different times and perhaps by different manufacturers.

5.2.2 Air Dropped Infrastructure

Imagine that the armed forces could fly over a conflict zone and air drop hundreds of video cameras, microphones, and motion sensors. Some of them would be destroyed before they landed, their locations would be randomly distributed, and one by one pieces of the system would be discovered and destroyed. If the system were built using Hive, then the loss or failure or discovery of certain elements of the network would not incapacitate the whole system. There would be no centralized point of control to compromise the network so even if some elements of the network were compromised and destroyed, the rest of the network would continue to function.

Appendix A

TTT History

The Things That Think consortium has given rise to a wide variety of projects, and a full history is beyond the scope of this document. However, a number of particular projects influenced the creation of Hive directly and are worth noting here.

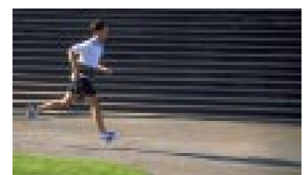
A.1 Mr. Java

Mr. Java

Mr. Java is what happens to a coffee machine in a networked world. The fusion of an Acorto 2000 automatic coffee machine and an RF tag reader, Mr. Java was installed in the Media Lab kitchen. When a user placed a tagged cup below the spout, it not only provided users with their preferred type of coffee, it also read them their favorite source of news.[10]

In practice, Mr. Java was a rather awkward device. A complete PC sat next to the coffee machine, and the system was broken as often as it was working. Still, the potential was clear: A machine that knew it's customers was something new and interesting. What if that data could be made available to other machines on the network? What if all machines were like this?

A.2 Marathon Man



Marathon Man

If airplanes can have flight recorders, then why not people?

In 1997 ASCII Corp sponsored the “Black Box” project. The idea was that people should be wired for vitals the same way airplanes are. If people could log their vital signs in a continuous and unobtrusive way, it would open up a whole new realm in health care.

Marathon Man was conceived as the first attempt to build a complete telemetry system for people. Over three different marathons, belt-packs monitored the vitals of a runner and send them wirelessly to the Internet in real time. The U.S. Army Rangers at Ft. Benning, Georgia used near identical hardware to learn about the effects of diet and fatigue on troop training.[36]

The systems used by the project were purpose-built from scratch. PIC Microcontrollers polled for data from the sensors and forwarded it to a Windows CE device on the belt. From there the data was sent via CDPD modem to a server at the Media Lab for formatting into graphs and distribution to the Internet. The web subsystem of the project was one of the first indications that the digital design of Things That Think could benefit from systematization.

Everest Extreme Expedition

A.3 Everest Extreme Expedition

The 1998 Everest Extreme Expedition consisted of an expert team of climbers, doctors, and scientists. The mission was scientific: put geological and surveying apparatus in place, and conduct experiments in physiology and tele-medicine.[13] From a technology perspective, there were a number of goals: First, build a body monitoring system similar to those from Marathon Man, but capable of functioning in the harsh environment of a high-altitude expedition. Second, create rugged, satellite connected weather stations that could function for a year unattended. Third, build cameras that could record where they were when they snapped a photo. Finally, integrate all of the data collected on the trip into a single coherent visualization.

The E^3 systems benefited from the experiences of the Marathon Man project, but unfortunately, few of the actual systems were transferable. The sensor design was modularized to reduce the implementation burden and to prevent the failure of a single sensor from taking down the whole bus. On the software side, systems were written to aggregate and visualize data automatically.

Bibliography

[1] Authenticode. http://www.microsoft.com/windows/ie/most/howto/?/windows/ie/most/howto/trusted.htm

[2] Microstar's Java-Based XML Parser. http://www.microstar.com/aelfred.html

[3] 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

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

[5] Igor Boukanov. Java Security. *Dr. Dobb's Journal of Software Tools*, 22(9):10–10, September 1997.

[6] 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

[7] W. Buxton. Absorbing and Squeezing Out: On Sponges and Ubiquitous Computing. In *Proceedings of The International Broadcast Symposium*, 1991-1996. http://www.dgp.toronto.edu/OTP/papers/bill.buxton/sponges.html

[8] The Collection of Computer Science Bibliographies. http://liinwww.ira.uka.de/bibliography/index.html

[9] Counter Intelligence. http://www.media.mit.edu/ci/

[10] Prospectus: Counter Intelligence. http://www.media.mit.edu/ci/kswp.forttt.word6.doc

[11] DCOM: A Technical Overview. http://www.microsoft.com/NTServer/appservice/techdetails/overview/dcomtec.asp

[12] Domain Names - Implementation and Specification. http://www.ietf.org/rfc/rfc1035.txt

[13] Everest Extreme Expedition. http://www.everest.org

[14] Bradley Geilfuss, Jr. Net-Weight and Inner-View Personal Health Data Monitoring and Interaction. Master's thesis, MIT Department of Media Arts and Sciences, 1999.

[15] Neil Gershenfeld. *When Things Start To Think*. Henry Holt and Company, 1999. ISBN: 0-8050-5874-5.

[16] Java GetOpt. http://www.urbanophile.com/arenn/hacking/download.html

[17] Li Gong. Java Security Architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5. http://www.javasoft.com/products/jdk/preview/docs/guide/security/index.html

[18] J. Gosling, F. Yellin, and The Java Team. *The Java Application Programming Interface Volume 2: Window Toolkit and Applets*. Addison Wesley, ? 1997.

[19] The GNU General Public License. http://www.gnu.org/copyleft/gpl.html

[20] 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.

[21] Java Communications API 2.0. http://java.sun.com/products/javacomm/index.html

[22] Java Development Kit Version 1.1. http://java.sun.com:80/products/jdk/1.1/

[23] Java Development Kit Version 1.2. http://java.sun.com:80/products/jdk/1.2/

[24] 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/

[25] Rehmi Post Maggie Orth and Emily Cooper. Fabric Computing Interfaces. In *Proceedings of ACM CHI '98*. ACM Press, 1998. http://www.media.mit.edu/ morth/fabDIS.htm

[26] 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.

[27] Nelson Minar. Desigining an Ecology of Distributed Agents. Master's thesis, MIT Department of Media Arts and Sciences, 1998. http://nelson.www.media.mit.edu/people/nelson/research/masters-thesis/

[28] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed Agents for Networking Things. Submitted to ASA/MA '99, 1999. http://nelson.www.media.mit.edu/people/nelson/research/hive-asama99/

[29] The Network is the Computer. http://www.sun.com/corporateoverview/who/vision.html

[30] Object Serialization. http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html

[31] Voyager. http://www.objectspace.com/products/voyager1.htm

[32] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0.* Object Management Group (OMG), 2.0 edition, 1995.

[33] The Open Source Definition. http://www.opensource.org/osd.html

[34] Roz Picard. *Affective Computing.* MIT Press, 1997. ISBN: 0-262-16170-2.

[35] PersonalJava. http://www.javasoft.com/products/personaljava/

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

[37] Java Remote Method Invocation (RMI) Interface. http://java.sun.com/products/jdk/rmi/index.html

[38] RXTX Home Page. http://jarvi.ezlink.com/rxtx/index.html

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

[40] 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/

[41] SAX 1.0: The Simple API for XML. http://www.megginson.com/SAX/

[42] Sun Microsystems. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1. http://java.sun.com/beans/

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

[44] Universal Plug and Play. http://www.upnp.org/resources/UPnPbkgnd.htm

[45] Uniform Resource Locators (URL). http://www.ietf.org/rfc/rfc1738.txt

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

[47] 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

[48] The MIT Wearable Computing Web Page. http://wearables.www.media.mit.edu/projects/wearables/

[49] Mark Weiser. Ubiquitous Computing. http://nano.xerox.com/hypertext/weiser/UbiHome.html

[50] Craig Wisneski. The Design of Personal Ambient Displays. Master's thesis, MIT Department of Media Arts and Sciences, 1999.

Photo Credits

The Hive Logo, ©Kelly Heaton, 1998: p.14

The Hive Development Team, ©Web Chappel, 1999: p.19

Jukebox Player, ©Peter Menzel, 1999: p.50

Where's Brad?, ©Webb Chappell, 1999: p.51

Counter Intelligence, ©Peter Menzel, 1999: p.52

Net Weight, ©Bradley Geilfuss, 1999: p.52

Mr. Java, ©Webb Chappell, 1997: p.63

Marathon Man, ©Michael Hawley, 1997: p.63

Everest Extreme Expedition, Unknown, 1998: p.64

Colophon

This document was created by the author on a Sony Vaio PCG-505TX(UC) running Windows 98. The text was edited using GNU Emacs 20.3.1 with AUC TeX bindings and typesetting was performed using LaTeX2$_\varepsilon$. A modified version of the MIT thesis style `mitthesis.sty` was used as a template. The Collection of Computer Science Bibliographies[8] was extremely valuable for creating BibTeX references. Figures were created using `dot` and converted to Postscript using `xv`.

LaTeXoutput was converted to Postscript using `dviPS`, and final conversion to PDF was performed using Adobe Acrobat. Printing was performed on an HP Laserjet 4si MX onto Xerox Image Master acid-free paper. The typeface used was Times New Roman 10pt.