Infrastructure for an Intelligent Kitchen

by

Matthew Konefal Gray

S.B., Massachusetts Institute of Technology (1997)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning in partial fulfillment of
the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author .....
Program in Media Arts and Sciences,
School of Architecture and Planning
May 7, 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 .....
Stephen A. Benton
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Infrastructure for an Intelligent Kitchen

by

Matthew Konefal Gray

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on May 7, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

ABSTRACT

In a world of growing numbers of "things that think", a software infrastructure for managing complex systems of these things is a necessity. This paper presents such a software system, Hive. Hive is a decentralized distributed mobile agents platform, addressing the requirements of an infrastructure for "things that think". Hive addresses the need for ways to manage highly heterogeneous sets of devices, methods for describing and discovering resources, and an approach toward constructing applications.

To effectively evaluate this architecture, the particular testbed of a networked kitchen is examined. The kitchen provides a dynamic and compelling application domain to explore the Hive system. The kitchen described in this paper is capable of assisting a user in the preparation of recipes, through use of a variety of sensing and actuation technologies.

In addition to the kitchen, a number of other Hive-based systems are discussed, including a networked jukebox and a localization infrastructure for wearables. Hive is evaluated in the context of the networked kitchen and these other applications. Hive is compared to other distributed software systems, in particular Jini. Finally, areas for future work are suggested, in both the Hive infrastructure and the networked kitchen application.

Thesis Supervisor: Michael Hawley
Title: Assistant Professor of Media Arts and Sciences
Alex W. Dreyfoos ('54), Jr. Career Development Professor
of Media Arts and Sciences

Infrastructure for an Intelligent Kitchen

Thesis Committee

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

Thesis Reader .....
Neil Gershenfeld
Associate Professor of Media Arts and Sciences

Thesis Reader .....
Jim Waldo
Sun Microsystems

Acknowledgments

Without the support of my mother, father, Carrie, and all my friends, this would not have happened.

CONTENTS

List of Figures

1 INTRODUCTION

The Author
Photo credit: Peter Menzel

Distributed networks of "Things That Think"[TTT] are far too rare. Things That Think is used as a term to include essentially anything with computation, sensing or actuation capability. Typically, however, it implies items that have embedded computation, sensing, or actuation that are not normally thought of as computational or thinking devices. Networks of these things are not common, despite the growing presence of the things themselves. Distributed software networks are ubiquitous. Distributed networks of things, however, are not.

Domestic environments provide a rich domain in which networked systems of things can be built. The kitchen in particular is an area ripe with devices that could be networked together in a useful manner. Specifically, a kitchen capable of aiding in the execution of a recipe, particularly in the planning, instruction, and sequencing of activities would be very useful. Further, a natural extension of such a system would include inventory management to allow questions like "What can I make with what I've got?" to be readily answerable.

There are a number of problems associated with building a distributed network of things, in the kitchen or otherwise, that are not addressed by either disconnected things or most existing distributed software systems. One such problem is a wider range of capabilities of components of the system. In most distributed software systems, while there may be some heterogeneity, most of the participants are similar computers. In a network of things, capabilities are much more variable. Another problem is that the description of the components of the system is not necessarily limited to the capabilities accessible through an API.

This work will focus on issues of managing a range of heterogeneous devices, semantic labeling of those devices, and building a recipe assistant for the kitchen using this infrastructure. Some of the other issues that are particular to distributed networks of things will also be discussed.

1.1 APPROACH

A software system and application test bed to ease the construction of these networks has been built. This system, called Hive[HIV], has been developed within the Media Lab by Matthew Gray, Nelson Minar, Oliver Roup and Raffi Krikorian. Hive is intended to be a flexible, powerful standard architecture to provide connectivity for distributed networks of things that think. A system for solving the above problems will be built into Hive.

Nelson Minar

Using the Hive system, a networked kitchen was built, and the "recipe assistant" scenario implemented. Construction of this system highlighted the strengths and weaknesses of the Hive architecture. Particularly, the kitchen provided a good test bed for utilization of Hive's semantic labeling system.

Further deployments of Hive within the MIT Media Laboratory, and development of other applications altogether, such as the new "Honey, I Shrunk the CDs" system[1] (a jukebox demo), has allowed evaluation of the utility of the Hive system.

Oliver Roup
Photo credit: Webb Chappell

1.2 ORGANIZATION

This work will cover the background that has led to the exploration of the networked kitchen and background in distributed computing systems. It will then describe the Hive system which was developed, and then how it was applied to the kitchen. Other applications of the same infrastructure will also be discussed, followed by an analysis of the effectiveness of Hive at addressing the problems associated with building complex systems of "things that think".

1.2.1 BACKGROUND

A brief discussion of prior work in the domain of networked kitchens appears in Section 2. This includes prior work at the Media Lab, commer-

[1] See Section 7

cial development work, and work on the "Universal Kitchen" project at the Rhode Island School of Design. A description of other distributed systems that were examined and share qualities with Hive appears in Section 3.

1.2.2 HIVE AND SEMANTIC LABELING

Section 4 describes the architecture of the Hive system, including details on cells, agents and shadows. It also covers the specifics of the implementation. Section 5 covers the semantic labeling system used in Hive, where it comes from, and how it is applied.

1.2.3 KITCHEN DEMONSTRATION SYSTEM

In Section 6, details of the kitchen demonstration system are covered. The architecture and design of the system, and details of the implementation are covered. A comparison to prior work in the area is made, and the effectiveness of Hive is evaluated for this application.

1.2.4 OTHER SCENARIOS

Hive has been applied in a number of other scenarios, and some of these are covered in Section 7. The scenarios discussed include "Honey, I Shrunk the CDs", an RF based location system, and the Hive demonstration system.

1.2.5 JINI

While Hive shares many qualities with other distributed systems, it is more similar to Jini than any other. These similarities warrant particular attention. Section 8 discusses the similarities, differences, and how these systems can interact.

1.2.6 ANALYSIS

Section 9 analyzes the overall effectiveness of Hive, and identifies lessons learned, and suggests areas for future work.

2 KITCHENS

Kitchens are a rich, compelling and very personal, yet social domain in which to explore networked systems of things that think. Within the Media Lab, projects such as "Mr. Java"[MrJ] and "Counter Intelligence" (CI)[CI] have begun scouting this territory. In particular, CI, initially developed by Joseph Kaye, Andy Wheeler, and Niko Matsakis, demonstrated some basic concepts of an intelligent kitchen through a scripted demo, but was not actually a functioning prototype.

"Counter Intelligence"

The "Counter Intelligence" system ran into the problems of developing large networked systems of things. While individual components worked reasonably well, creating a well integrated, functional recipe assistant application proved very difficult. Further, the system design ran into obstacles, as it was not clear what sensor technology was going to be available. An underlying framework for connecting a diverse and possibly changing set of things that think was needed.

The kitchen of the future
Photo credit: Peter Menzel

Additionally, controlling all of the devices in a kitchen in a centralized way is not necessarily possible, and in a practical sense is undesirable. The control system for a stove, refrigerator, and cabinets should be capable of being separate, but communicating, systems.

Finally, a system that allows for a flexible security architecture is valuable. While in a prototype system, security is not a paramount concern, in an actual networked kitchen, malicious break-ins to one's appliances can pose a serious threat. These could occur if the kitchen network were connected to the Internet at large, or through intentional or accidental problems with software installed in the kitchen. Therefore, some allowance for security is needed.

There has been little scholarly research on networked or future kitchens in general. The little work that has been done has either been commercial development and prototypes, or design oriented work that does not directly consider the implications of new technology.

In the commercial development category, there are a number of examples of novel work in the kitchen that have appeared recently. Ariston

Digital, a division of Merloni, has created a line of home appliances that network over power lines. They provide services such as warnings in case of power failure, telediagnostics, power consumption management, and through the “Home Smart Monitor”, recipe access and Internet connectivity. Liebherr has developed an inventory tracking refrigerator. Electrolux has produced the “Screen Fridge”, which is essentially a refrigerator with a laptop attached. Zanussi has a set of “intelligent” household appliances named “LIVE-IN”. In non-technologically motivated design of a future kitchen, the Rhode Island School of Design (RISD) has designed the Universal Kitchen prototype. None of these systems has yet approached the vision of a truly networked kitchen.

3 DISTRIBUTED SYSTEMS

The need for a system like Hive has evolved from experience with a number of projects that demand useful connectivity between things. Hive provides a distributed system of software components designed with the "Things That Think" application domain in mind. The "Counter Intelligence" project in the kitchen domain that this work is addressing has been one motivating application, but it is worth mentioning other, earlier systems that inspired the need.

The "Marathon Man"[Red98] project involved creating a belt that collected a variety of biometric data from a person. While this project was successful in doing so, it lacked a solid architecture for collecting and presenting this data, especially for multiple simultaneous or consecutive users. Other projects using variants of this hardware, including the Everest Expedition[Eve98] and the "C2C Bike Ride"[C2C] required reimplementing this data collection and presentation layer each time.

Everest Expedition team

Other projects such as the Tangible Media Group's Pinwheels[DWI98] and the "Net Weight" scale have suffered from the problem that despite the fact that they are designed assuming network connectivity, this connectivity is in fact absent. This missing connectivity, which many application builders have chosen not to implement, leaves the applications unsurprisingly incomplete.

Coast to Coast Bike Ride

Within the MIT Media Lab, there have been some projects which have begun to address connectivity issues for things that think. Steve Gray's work on Bit Bags[Gra] focused on SNMP-based management of collections of things. The Black Box projects (Marathon Man and Everest Expedition) addressed some of the low level issues, using a serial hub made from an IRX[Poo99] board, and later an I^2C bus.

Outside of the MIT Media Lab, most exploration of distributed systems has been focused on, if not confined to, software systems. Distributed object systems such as CORBA[OMG95], OpenDoc, and DCOM are examples. These systems could be adapted to provide distributed services in an environment of networked things, however there are certain weak-

nesses due to the assumption that the components are software.

Sun Microsystems has developed a system called Jini[Wal98] described as a system that "enables spontaneous networking of a wide variety of hardware and software" and as a "distributed system designed for simplicity, flexibility, and federation." Jini provides a number of the necessary pieces for a distributed network of things. The implications of Jini with regard to this work are discussed in section 8.

Hive provides this needed connectivity layer. There are three conceptual layers to the Hive architecture. These are the devices themselves, a device driver layer called shadows, and the agents layer where intentionality and applications are built.

Hive is implemented in the Java language[AG96] using version 1.1 of the Java API[CLK98], and from the point of view of the Hive system, all things are considered to run Java. In reality this is not currently the case. One or more Java incapable devices may be connected to a Java capable machine that will act as a proxy for it. The method by which these devices connect to the Java capable machine is unimportant in terms of the Hive system. Particularly, latency or bandwidth considerations caused by limitations in the communication channel are simply considered to be a limitation of the capability of the device in question. In practice, most Java incapable systems communicate with the Java machine via a wired or wireless serial communication channel.

On top of the actual device functionality, there is the shadow layer. Shadows are local pieces of code that manage access to a local resource, usually a device. Each shadow corresponds to a particular kind of device, and provides a programmatic interface to the underlying hardware, whether through an intermediate protocol or not. Further, the shadow manages concurrent access and provides any other mediation of access or control that may be necessary. The separation of the shadow and agent functionality is meant to clarify the distinction between local access and remote communication.[WWWK97]

Finally, at the top layer of Hive is a mobile agents architecture that allows agents to move from one computer to another, access shadows (and

correspondingly, devices) as necessary, and communicate with one another to provide the desired application-level functionality. These agents are easily upgradable and modifiable by their nature as mobile code, greatly enhancing the flexibility of the system.

Hive meets the needs of this emerging class of applications of networked devices, such as the "Counter Intelligence" project. One additional capability that is needed above and beyond this basic connectivity architecture is a way for the devices, shadows and agents to communicate semantic descriptions of themselves to others. This is described in Section 5.

4 HIVE

Hive is a decentralized distributed mobile agents platform designed for "Things That Think". Hive enables connectivity for devices, allowing interactions without substantial reengineering, and creation of interactions that needn't be anticipated at the time of the design of the original devices.

4.1 ARCHITECTURE OVERVIEW

The Hive architecture is composed of three pieces and a lookup scheme to connect them. First, there are Hive "cells", or servers, which are the environment in which Hive runs. They provide the infrastructure for locating resources, basic agent mobility and a bootstrap for initiating inter-agent communications. Second, there are Hive "shadows", which are essentially device drivers for local resources, whether that may be a display, a piece of hardware, or any other manageable local resource. Third, Hive "agents" are the pieces of mobile software that engage in all communication and interact to create applications. Finally, a combined syntactic/semantic lookup scheme provides a mechanism for agents to discover shadows and each other via a remote method provided by the cells. A visual representation of the Hive architecture appears in Figure 1.

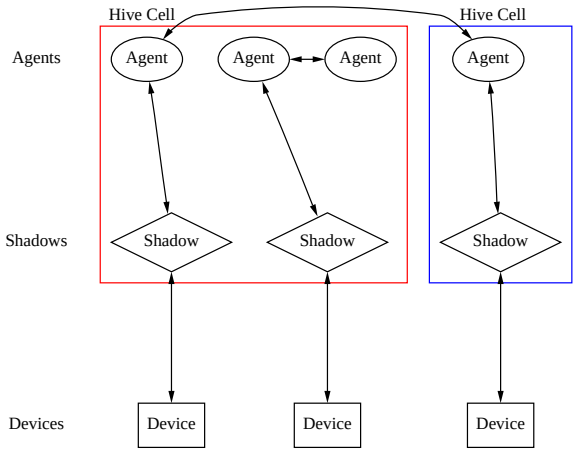


Figure 1: The Hive Architecture

4.1.1 CELLS

A Hive cell[2] provides a minimal interface to enable communication between agents and to allow agents to access local resources. Figure 2 shows the primary methods provided by a Hive cell.[3] The first two methods, `getAddress()` and `queryAgents(...)` are the only methods that are accessible remotely. These allow remote components to discover agents in the cell. The second group of methods relate to management of the agent life cycle and mobility, and the last two methods provide access to the local resources known as shadows.

```
public class Server
   public CellAddress getAddress();
   public DescSet queryAgents(Object sender,
                              String[] syntactic, String[] semantic)

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

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

Figure 2: Methods on a Hive Cell

4.1.2 SHADOWS

Shadows provide a software interface to a local resource, typically a piece of hardware. The responsibilities of a shadow are small, so as to avoid pitfalls related to security, particular application concerns, and maintainability. A shadow exposes an API to access the functions of the device. It is responsible for managing concurrent access to the device as well as protecting the device. It should not be possible to damage or otherwise harm a device, despite the sequence, timing, or parameters of calls

[2] "Hive cell" and "Hive server" are often used synonymously. The term cell was chosen to make clear the fact that Hive is a completely peer-to-peer system. The label "server" implies a distinct client, which does not exist in Hive.

[3] A number of utility methods are not included in this list.

to the shadow. Finally, a shadow is self-describing. This is described in more detail in Section 5. Beyond this, a shadow should be as minimal as possible.

The shadow model helps maintain a strong distinction between local resources, and remote operations[WWWK97]. In this capacity, it provides added security as well, which is necessary in an environment where full agent mobility is possible. In a traditional sand box or capabilities-based security model, a given agent could be restricted from communicating with hardware, or only to particular hardware, but neither of these models is well suited to constraining the actual contents of that communication. For example, a motor controller may be connected via a serial port. While a sand box or capabilities based model could prevent communication with other devices, there would be nothing to prevent an agent from overdriving the motor. With a shadows based abstraction, all communication with local resources is mediated.

4.1.3 AGENTS

Agents are pieces of mobile code that reside in cells, locate and export or otherwise utilize the functions of shadows, and communicate with other agents. Applications built on top of the Hive architecture are constructed as a set of interacting agents, or a so-called "distributed ecology of agents"[Min98]. Some agents will essentially be direct proxies for shadows, while others will be pure software services, and yet others will coordinate the behavior of other agents.

By their mobility, agents can readily be used to modify the abstraction by which a device is operated. A camera can be changed from a image exporter to a motion detector by sending a new agent to communicate with the camera shadow. The details and implications of Hive as a mobile agents system are discussed in more detail in [Rou99] and [MGR+99].

4.1.4 LOOKUP

In a small scale system, hard coding the connections and interactions is reasonable, but as a system gets larger, it becomes desirable to have a scheme for dynamically creating these connections. This is accomplished in Hive via a combined syntactic and semantic lookup scheme.

The distinction between syntactic and semantic lookup is intended as follows: An agent or shadow is *syntactically* described by its programmatic interfaces or API, and the *semantic* description is composed of qualities that are not accessible via the objects API, such as (in most cases), location, physical description, or ownership. In Hive, syntactic lookup is tied to the Java type system, specifically to the `Remote` interfaces of the object in the case of agents.

In all cases, the user of the lookup system is an agent, though the objects being looked up may be agents or shadows. The cells form a default form of federation, where it is possible to locate all of the agents inhabiting a particular cell by using the `queryAgents(...)` method on the cell. Other agents may collect these descriptions and generate other federations, such as a larger area federation, or those based on syntactic type or based on a particular of the semantic description.

The semantic lookup scheme is described in detail in Section 5, and compared with Jini's[Wal98] attribute[JLA99] based lookup scheme in detail in Section 8.

4.2 IMPLEMENTATION

The Hive system is implemented in Java[AG96], using JDK 1.1.7[JDK]. In excess of 22,000 lines of code have been written for Hive with under 10,000 lines composing the core of the system, and the remainder being application agents and shadows. Hive utilizes a number of third party packages as well: SiRPAC[Saa99], AElfred[AEl] and SAX[SAX] for processing of XML-encoded RDF files, and javax.comm[Jav] and rxtx[RXT] for serial communications. Figure 3 shows the GUI for Hive (which is itself an agent). Each icon represents an agent, and lines between agents

imply connectivity between those agents.

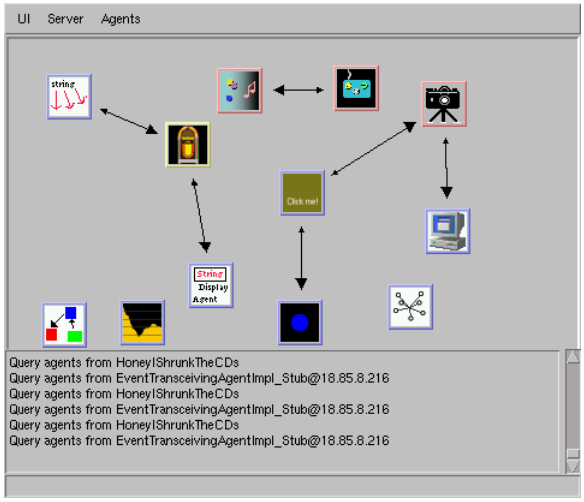


Figure 3: The Hive Graphical User Interface

4.3 APPLICATIONS

Hive was released internally within the Media Laboratory in January, 1999. A number of applications were built with Hive, including the primary application discussed in this work, the networked kitchen, which is discussed in detail in section 6. Other applications included the "Honey, I Shrunk the CDs" scenario[4], a set of interfaces by the Tangible Media Group, a vision system and electrostatic tracking system for a MOMA installation, a personal location service for wearables[5], and a number of sample applications developed by the Hive developers.

4.4 SUMMARY

The Hive system has been built and addresses many of the problems associated with constructing networks of things that think. The core ar-

[4]Described in section 7.
[5]Also described in section 7.

chitecture has proved flexible and useful in a number of particular applications discussed in the following sections.

5 SEMANTIC LABELING

5.1 BACKGROUND

A universal requirement of distributed systems is a mechanism to find components. Systems such as search engines for the World Wide Web[WWW] utilize a combination of the semantic mark up of HTML[HTM] as well as textual analysis to determine whether a component (document) matches a query. Many software systems, such as CORBA[OMG95] utilize a naming service[OMG94] where components are mapped 1-to-1 with names. CORBA also utilizes a "trading service" which allows lookup by interface (what Hive calls syntactic lookup).

In a environment where the components are "things that think", neither of the above described approaches work well. While a direct naming scheme could be used, this primarily defers the problem to one of creating a directory service so those names may be usefully associated with descriptions of the services they provide. Hive utilizes two conceptually distinct ways of locating agents and shadows: "syntactic lookup" and "semantic lookup".

Syntactic lookup is the ability to locate components based on their Java type. In the case of agents, these are completely composed of `Remote` interfaces, and in the case of shadows, are composed of the shadow's type and any interfaces it implements. This sort of lookup is necessary to assure programmatic compatibility. Many very different devices, however, may have identical interfaces.

A reasonable remote interface example is `Toggleable`, which would apply to devices such as lamps, door locks, or any other simple output device. While there is no programmatic distinction between these devices, their actual role in an application would be substantially different. This issue could be addressed by the creation of a number of so-called "tag interfaces", such as `Lamp`, which extend `Toggleable`, but implement no new methods themselves. This approach, however, has problems in that these tag interfaces cannot be dynamically extended or modified at run

time, and their use creates substantial clutter in the Java type system.

5.2 UNDERLYING TECHNOLOGIES

In lieu of extending syntactic lookup this way, Hive implements a separate semantic lookup system. Associated with each agent or shadow is a `Description` which describes the object. Hive semantic descriptions are expressed using the Extensible Markup Language (XML)[BPSM97] serialization of Resource Description Framework (RDF)[LS98] model in the same style as those described in the Composite Capability/Preference Profile (CC/PP)[RHDS98] system.

The canonical example for an application of CC/PP is a cell phone. The "capabilities" portion would indicate how much memory the phone had, if it was analog or digital, and what sort of screen it has. The "preferences" portion would indicate that the user wants a particular ring style, certain numbers on the speed dial, and the like. This naturally extends to the broader world of things that think.

5.2.1 XML

The Extensible Markup Language provides a simple but flexible basis for sophisticated semantic descriptions. An sample XML document, shown in figure 5.2.1

```
<thesis>
        <author>
                <name>Matthew Gray</name>
                <email>mkgray@mit.edu</email>
        </author>
        <title>Infrastructure for an Intelligent Kitchen</title>
</thesis>
```

Figure 4: Sample XML Document

shows the basic form. Through use of SGML Document Type Definitions (DTD)[SGM86] and Document Definition Markup Language (DDML)[BCMS99] schemas, the structure and content of document can be validated.

5.2.2 RDF

The Resource Description Framework (RDF) is a framework for representing a directed labeled graph data structure and specifies a representation in XML. For example, the above XML document example, reformulated in RDF (which requires substantially additional syntactic structure) would produce the graph shown in Figure 5.

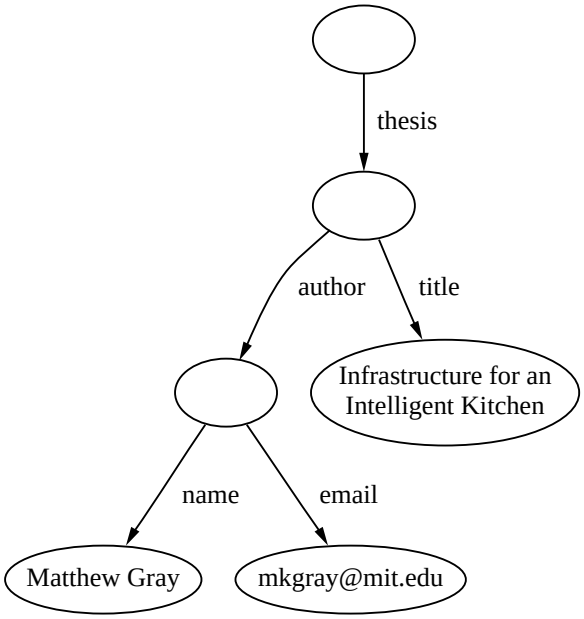


Figure 5: RDF graph of sample document

RDF is primarily distinguished from generic XML by the fact XML is considered "document-centric" and RDF is "data-centric". This means that two different RDF documents can correspond to the identical underlying data representation, while in XML, differences in formatting correspond to differences in the resulting document. Additionally, RDF has extensive syntactic requirements, making representations entirely unambiguous, but at the cost of simplicity and easy human readability. The use of RDF versus plain XML for semantic labeling is discussed below.

5.2.3 CC/PP

The "Combined Capability/Preferences Profile" specification, as the name suggests is a way of representing the capabilities and preferences that correspond to a device. It provides a standard structure for representing default values, where those values may reside in an externally referenced document, rather than requiring all metadata to be replicated in each description.

The particular kinds of data CC/PP aims to describe are well suited to the primary purpose of semantic descriptions within Hive. While it is valuable to be able to include traditional metadata such as the owner of a resource in Hive, the primary goal for these descriptions is to describe capabilities and preferences. CC/PP is built on top of RDF.

5.3 SEMANTIC DESCRIPTIONS IN HIVE

The semantic description lookup scheme for Hive was implemented utilizing RDF, CC/PP and XML. In a typical configuration, each Hive agent or shadow would have a description that contained information about the type of device it represented, its location, a "nickname", configuration data, and any other metadata relevant to potential users of the service.

This approach has certain similarities with Jini's attribute[JLA99] based lookup, which is compared in detail in Section 8.3. Other systems, such as Ontolingua[FFR96] and KQML[KQM] have the disadvantage of substantial complexity, even in comparison to RDF, as they are targeted more toward knowledge representation problems.

5.3.1 DESCRIPTION OF DESCRIPTIONS

A sample RDF description appears in Figure 6. It describes a "QuickCam" camera located in room 468 of building E15. The representation of the RDF structure that this represents is shown in Figure 7.

A schema, or particular set of conventions for describing an object, is needed if multiple agents are going to interact. If one agent calls the

```
<?xml version="1.0"?>
<RDF
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about=""
   thing:nickname="Pia Quickcam">
      <thing:config thing:command="cqcam"/>
      <thing:location
            thing:building="E15"
            thing:room="468"/>
      <thing:role>
            <Description>
                  <thing:camera thing:kind="QuickCam"/>
            </Description>
      </thing:role>
</Description>
</RDF>
```

Figure 6: Sample RDF Description

place that something is located its "place", which has parameters of "latitude", "longitude" and "elevation", and another calls it a "location", with a "building" and "room", they will have a hard time usefully interacting.

Defining a particular schema is outside of the scope of this work. Initially, the goal, instead is to provide a sufficiently flexible description system for exploring different schema, and how those schema might be constructed. In practical application, a few loose conventions were applied as a preliminary Hive schema.

Hive does not provide a notion of a unique name for agents; an agent is used by reference, and need to be discovered by the lookup mechanism. Part of the reason for this is that it is hard to define the notion of identity in a mobile distributed objects context. If an agent moves from one host to another, is it the same agent? If an agent duplicates itself and both copies move, which is the "original"? However, it is often useful to be able to refer to an agent by name, or ask an agent its name so the "same" agent can be found later. To avoid the semantic ambiguity of "same", Hive utilizes the semantic description system to allow an agent to give itself a name, and it can decide when its identity changes or remains the same.

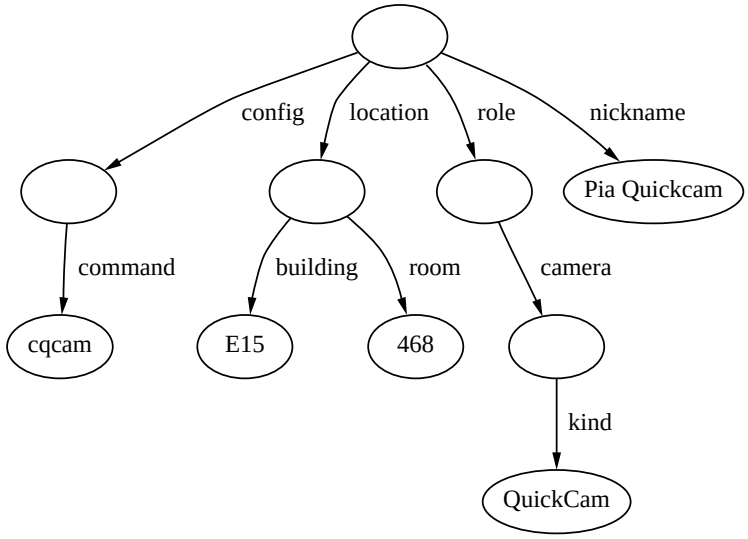


Figure 7: RDF Structure of description shown in Figure 6

Due to the fact that the agents assign themselves these names, there is no guarantee of uniqueness, and uniqueness may not even be desired. Consider the case of a "phone book agent". If there are multiple identical instances of this agent, they are all equally good. They may as well share a common name. If there are multiple instances of a light bulb agent, however, they should have very different names. This name is called the "nickname" and in the case of the example in figures 6 and 7 is "Pia Quickcam".

In the "Honey, I Shrunk the CDs" demonstration described in Section 7, the central agent locates both the tag reader and the jukebox by nickname, since it is looking for a particular pair of agents, the "Pia Demo Tagreader" and the "Pia Jukebox". Most of the time, though, agents would be found based on their other parameters.

In the kitchen demonstration described in Section 6, one agent collects references to all of the tag-readers available. This is accomplished via semantic lookup, without the use of nicknames or any other individualized identifier. Specifically, the concentrator agent performs a query for all agents which have as one of their "role"s the type "tag-reader". The parameter "role" is another piece of the preliminary Hive schema. It allows

a device to have multiple arbitrarily parameterized labels. In the example in figure 6, the device has the role of "camera", and that role has the parameter "kind" which in the example is "QuickCam".

Another example of roles, where a single agent may have multiple roles, is shown in the `AutoWiringAgent`. This agent finds a pair of two on screen agents to connect together. First, it locates all agents that have the role of "screen-widget" to limit the selection to screen based agents. Second, it selects an agent with the role of "button", and an agent with the role of "event-display", each with appropriate syntactic types and connects them together.

Lookup based on a combination of parameters is possible as well. An extension to the kitchen lookup could include restricting the tag-readers that are selected to those in a particular location (i.e., the kitchen). An example of a scenario that uses the location portion of the preliminary Hive schema for use by wearable computers is described in Section 7.2. Further work on defining schemas and managing their creation is needed, and Hive's semantic labeling approach provides the necessary flexibility.

Finally, in the example in figures 6 and 7, there is a portion of the description labeled "config". This is configuration information rather than actual semantic description. This is implemented to use CC/PP style defaults, to easily allow a number of descriptions to share defaults, such as having a default location. Further comments on the use of RDF for configuration appear below in the analysis.

5.3.2 DESCRIPTION API

The fundamental unit used in queries is a set of zero or more descriptions called a `DescSet`. The `DescSet` allows flexible manipulation of these descriptions to identify the agents or shadows of interest to a particular application.

The API for querying a `DescSet` is primarily composed of the `select(String parameter, String value)` method. The effect of this call is, in each description, to traverse the path on the graph with the label `parameter` that connects to a node with value `value`. If `value` is

null or absent, it will traverse the path regardless of the value of the node it leads to.

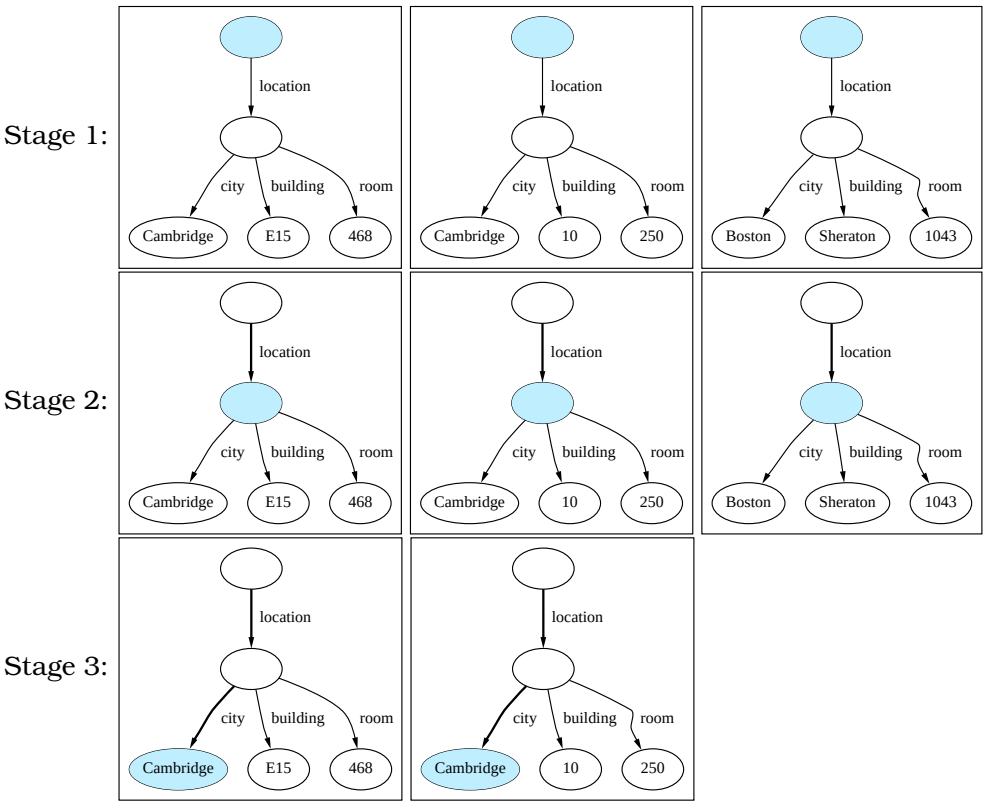


Figure 8: Progression of a sample query

Figure 8 shows the steps in a simple query of a `DescSet` starting with with 3 descriptions. In this example, the goal is to select all descriptions that describe an object located in the city of Cambridge. Unrelated portions of the graph are not shown for simplicity.

Stage 1 of Figure 8 shows the state of the `DescSet` before anything is done. A `DescSet` may be reset to this state by calling `DescSet.noContext()`. A call of `select("location")` changes the state to that shown in stage 2. A final call of `select("city", "Cambridge")` changes to what is shown in stage 3. Note, the third description, which described an object in Boston, was removed from the set.

In order to obtain a `DescSet` in the first place, an agent calls the

`queryAgents(...)` method on a cell, or constructs it from a pre-existing list of agents or shadows. A number of other methods exist on `DescSet` to retrieve matches following a set of `selects`, count matches, add and remove descriptions, merge with other `DescSets`, and reset the graph query. A practical guide to using RDF descriptions in Hive appears in Appendix A.2.

5.3.3 ANALYSIS

This approach toward semantic labeling has proved flexible, extensible and useful. The use of RDF has been a bit cumbersome, however. In the small number of actual applications built so far, it has been capable of usefully describing all the necessary components. As larger numbers of components are constructed, and the interaction increase, better evaluation will become possible.

The system's flexibility and extensibility has been evidenced by two particular extensions: nicknames and configuration data. In the early design of Hive, a decision was made not to assign unique names to objects; all objects would be the result of lookups. As discussed above, it became clear that allowing agents to name themselves had utility. Adding a "nickname" as part of a description was easy, but the use of RDF allows for structured nicknames if so desired. That is, a nickname may have the string value "Pia Quickcam", but it might have a parameter "assigningAuthority" of "Pia", or any other substructure desired. Another agent doing a lookup that does not use this structure would not be interfered with.

During the development of Hive, the need for a method of doing per agent configuration arose. Adding new per agent configuration files was considered, but dismissed when it was realized that the configuration could readily be put into the semantic description. The configuration information in the semantic description is automatically translated into JavaBeans[JB] method calls to configure the agent or shadow. Further, more complex configuration structures can be put into the description and utilized by the agent on its own terms.

Unfortunately, RDF has evolved into a very complicated specification,

and has become more difficult to use. While this complexity provides substantial customizability and a strong underlying data model, it is unclear whether the tradeoff is worth it. Alternative approaches would include a non-RDF, but still XML based representation, or a more complex knowledge representation system.

Future work, particularly if RDF is kept as the description representation, should include implementation of a suite of utilities to generate, modify, and view these representations. To some extent, this is motivation to continue to use RDF, as many such tools will be created for RDF in general, independent of a particular application domain.

Details of how the semantic labeling system were applied in various Hive applications are discussed in Sections 6 and 7. User documentation for doing RDF based agent configuration appears in Appendix A.1.

6 THE NETWORKED KITCHEN

Imagine a kitchen in which all of the appliances are networked together. The pantry, cupboards, and refrigerator know their own contents. When preparing a recipe, the kitchen would preheat the oven, identify substitutions in recipes if the user were on a restricted diet, and would reorder ingredients from the supermarket as supplies got low.

Mike Hawley, Matthew Gray, and Andy Wheeler at the prototype kitchen of the future
Photo credit: Peter Menzel

Specifically, consider a recipe assistant, capable of walking a cook through the preparation of a recipe. An important sub-application of this is a system to actually schedule the individual steps of a recipe.

The level of assistance provided by the kitchen would vary depending on user preferences, the particular recipe, and the devices and sensors available in the kitchen. The ideal system would be able to take advantage of as much or as little capability as a kitchen might offer.

Particular hardware features built for use in the kitchen include a tag reading system for identifying and locating cooking utensils, dishes, and ingredients, a digital scale, a microwave oven, a display, and speech output. The software for the system uses Hive's semantic lookup scheme to take advantage of new, previously unknown hardware that matches certain descriptions.

6.1 SYSTEM DESIGN OVERVIEW

This demonstration system for the spring Things That Think consortium meeting was constructed. The demonstration system for the kitchen is composed of nine kinds of agents. The basic architecture is shown in Figure 9.

Wiring for the demonstration
Photo credit: Peter Menzel

At the top level in an agent which manages the execution of the recipe, in concert with the other service agents. At the second level are a set of "manager" agents which coordinate interactions with individual or sets of devices. Finally, there are the agents that correspond to the physical devices and services available in the kitchen.

Although not utilized in the demonstration, an additional agent could

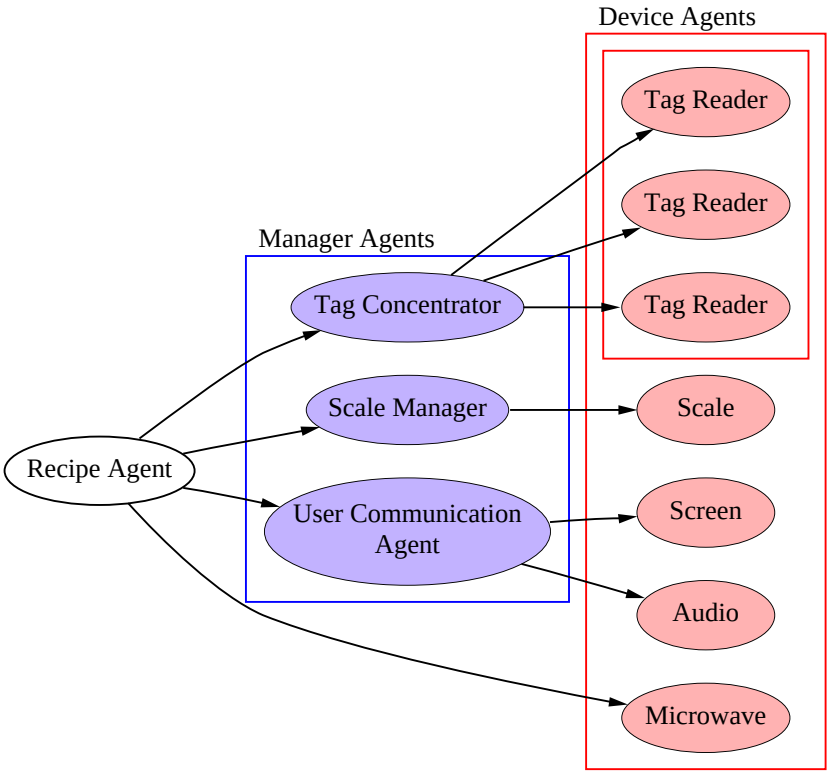


Figure 9: Network Kitchen Architecture

be implemented above the the recipe agent to determine which recipes are being used. Implemented, but not used in the demo system is a generic recipe scheduler capable of managing multiple simultaneous recipes and managing resource contention. A more detailed description of this scheduler appears below in Section 6.2.5.

6.2 SYSTEM DESIGN DETAILS

6.2.1 RECIPE AGENT

The recipe agent oversees interactions and activities in the kitchen by interacting with the manager agents. This agent does not necessarily plan the sequencing of the recipe itself, as that may come from an independent scheduler, such as the one described below.

A separate recipe agent provides the flexibility to manage the execution of a recipe in a way suitable to a particular user. Users might prefer the use of different schedulers, or the ability to have custom recipe agents that don't rely on a scheduler at all, as was used in the demonstration scenario.

6.2.2 MANAGER AGENTS

The manager agents provide the the underlying "kitchen logic". These agents manage an inventory of food and cookware, manage complex interactions with hardware, and coordinate communication with the user through whatever means are available. These agents provide an interface to the above described recipe agents, without the recipe agent needing to be concerned with the actual hardware available in the kitchen.

Further, these agents will also take care of identifying resources that may become available that were not anticipated of at the time of the initial design. Sufficiently unexpected improvements in the functionality available may require updates of individual agents to take advantage of the new capabilities, however the agents will communicate via a set of well-defined interfaces to ease expansion of the system. Individual users are able to dramatically change the behavior of the entire system by either tuning parameters of these agents, or replacing them individually, without any pervasive change to the remainder of the system.

The *UserCommunicationAgent* manages all communications to the user or users of the kitchen. In the demonstration system, this agent communicated with the user through verbal prompting as well as graphical and textual on-screen feedback. This agent could be readily modified to provide location based projection, communication via a wearable computer[6], or via an ambient display. In the demonstration system, the display systems were reusable generic display components.

The *ScaleManager* managed complex interactions with the scale. The *ScaleAgent* itself (described below) reports only the current weight on the

[6]See Section 7.2 for a particular application of Hive with wearables

scale. The *ScaleManager* maintains a recent history, to identify if an item has been added or removed, how much has been added since the last step in the recipe, and how much more needs to be added, when it is notified of the goal by the recipe agent. In a kitchen environment with other devices that mandated complex interactions, similar management agents would be created.

Canister with RFID tag
Photo credit: Peter Menzel

The inventory management agent, called the *TagConcentrator*, manages data from any number of tag readers, of any types. Using the semantic lookup system described in Section 5, the *TagConcentrator* discovers all of the tag readers available to it, and organizes the presence, absence, appearance, and disappearance of any and all tagged items. Additionally, should new tag readers be introduced, the *Tag Concentrator* will dynamically discover them.

These management agents provide a convenient abstraction for interacting with varying hardware configurations. Beyond an abstraction, however, they utilize Hive's semantic lookup scheme to automatically identify the best piece or pieces of hardware to use in a given scenario. This ability to dynamically utilize available hardware is one of the key strengths of Hive, and this is accomplished through separation of functionality into agents, and use of semantic descriptions.

6.2.3 DEVICE AGENTS

Digitally controlled microwave
Photo credit: Peter Menzel

Agents to manage each of the individual tag readers, the microwave, the scale, the speech output system, and the on-screen display were used. The *TagReaderAgent* and *ScaleAgent* are both not specific to the kitchen scenario, and are used in other scenarios.[7] The speech output was accomplished using an agent previously used for playing music, called the *JukeboxAgent*, and the on-screen text display utilized the *StringDisplayAgent*. The *MicrowaveAgent* is currently only utilized in the kitchen scenario.

This reuse of components is straightforward in Hive. Above and be-

[7] See section 7

yond code reuse, however, is actual service reuse. In the demonstration setup, a separate audio system was used for the kitchen demo than for the jukebox demo[8], however if the second audio system had not been set up, the kitchen software would have automatically located another audio player, and utilized that.

6.2.4 SHADOWS

Shadows for each of the device described in the previous section were used as well. Some of these shadows were particular to hardware only utilized in the kitchen, such as the Transcell scale[9] and the microwave, while others such as the Swatch tag readers are identical to those used in other scenarios.

Digital scale
Photo credit: Peter Menzel

6.2.5 RECIPE SCHEDULER

A general recipe scheduler is necessary for any regular use of the networked kitchen. While it is possible to implement individual recipes, as was done in the case of the demonstration peanut brittle recipe, it becomes impractical to do so for large numbers of recipes.

Peanut Brittle, in process

Most recipes are presented as a fixed schedule, however it is clear to a person using such a recipe that in fact there are interdependencies, but rarely is the sequence strict. For this reason it is useful to have a dynamic scheduler that is capable of determining what sequence tasks may be done in, to find an optimal sequence, to manage resource utilization, and to adapt to unanticipated changes. Eventually, it would be useful to build an entire planner, such as CHEF[Ham86], rather than merely a scheduler to the system to allow creation of new recipes from a set of well established rules.

A scheduler based in part on STRIPS[FN71] and on Sacerdoti's procedural nets[Sac75] was implemented. Due to the highly constrained nature

[8] See Section 7.1

[9] While the shadow for the scale is particular to the kitchen demonstration, the agent for the scale merely requires a shadow with a particular interface, and can be utilized with both a kitchen scale, and a bathroom scale, such as the one used in the Net Weight[Gei99] demonstration.

of the recipe scheduling problem, and the fact that the initial implementation was to be a scheduler only, and not a generic planner, substantial simplifications were possible. Further, in 1971 Fikes and Nilsson write in [FN71]

> However, since we envision uses in which the number of operators applicable to any given world model might be quite large, such a simple system would generate an undesirably large tree of world models and would thus be impractical.

In the nearly 30 years since then the "impractical" has become quite practical, and the application domain distinctly limits the number of operators. For this reason, the scheduler can reasonably employ a "simple system" of an exhaustive search rather than requiring a the more complex approach described in [FN71].

The scheduler accepts recipes as input which specify a list of steps, their interdependencies and resource requirements. These interdependencies can include requirements that certain steps be completed as prerequisites, that certain steps be started within a certain time bound after the completion of another step, and any combination of these timing constraints. The resource requirements may require a particular resource, or a resource of a particular type for a flexibly specified time bound.

For the demonstration system, the scheduler was not integrated with the system, as a single recipe was being demonstrated repeatedly, and identically. Integrating the scheduler should be straightforward. Such a combined system would then be capable of managing multiple simultaneous recipes as well.

Currently, the scheduler requires recipes in a fairly unnatural form, with constraints explicitly stated. Future work could include attempts to construct this internal constraint based representation from recipes in a natural language form. Intermediate possibilities, such as a recipe that is human-readable, but also contains some added constraint information for the scheduler, should be readily doable.

As mentioned above, further work to develop a full featured recipe planner, rather than simply a scheduler, would be a rich area for new

research. Simple planning features such as the ability to perform substitutions and modify recipes for different cooking equipment is a natural next step. Beyond that, it is not unreasonable to suggest a planner that is capable of constructing recipes from scratch, given a basic set of requirements.

6.3 ANALYSIS

Hive proved a valuable toolkit in developing the kitchen. A particular example of how Hive showed its flexibility in this development occurred when the initially planned tag reading system was not available. Originally, the demonstration was intended to use a single large antenna polyphonic tag reader, in combination with a couple of the Swatch tag readers. Two days prior to the demonstration, the polyphonic tag reader ceased functioning. Without any code changes, the demonstration was capable of substituting a larger number of Swatch tag readers to accomplish the same functionality. This was possible due to the fact that Hive utilized a modular agent architecture and performed all of its lookup based on semantic parameters, such as requiring that something be a "tag-reader", rather than a particular kind, or a particular number of tag readers.

One useful comparison to make to help determine the efficacy of Hive, is to compare the kitchen demonstration system built with Hive to the one built without Hive. Comparison of functionality, implementation complexity and code and component reuse show conspicuously some of the advantages of Hive for applications such as this.

Vanilla on a tag reader
Photo credit: Peter Menzel

The following table shows an itemization of the features present in both the Hive and pre-Hive systems.

The Hive based system implemented the same feature set as the previous demo as well as a number of new features. The Hive recipe selection GUI was substantially less attractive than the pre-Hive system, though the pre-Hive system only allowed for the selection of a single recipe. Further, while the pre-Hive system contained support for the microwave, it was not actually used in the demonstration. Additionally, in the Hive

Feature	Hive	pre-Hive
On-screen prompting	yes	yes
Audible prompting	yes	no
Digital scale	yes	yes
Tag readers	yes	yes
Arbitrarily many tag readers	yes	no
Microwave	yes	yes
Ability to execute a full recipe	yes	no
Ability to execute more than one recipe	yes	no
Inventory tracking	yes	no
Recipe Selection GUI	yes	yes

Figure 10: Pre- and post-Hive feature comparison for the kitchen demonstration

version, each system component is usable easily outside the context of the specific demonstration. As an example during the demonstration, the microwave was connected dynamically to an on-screen button to control it.

Measuring implementation complexity and code reuse is difficult, however "lines of code" provides one metric. The following table shows the number of lines of code used in each system, and the portion that is shared with other applications.

	Hive	pre-Hive
Non-shared lines of code	˜450	˜3000
Shared lines of code	˜350	˜300

Figure 11: Code complexity comparison

The Hive scenario, of course, requires a large amount of generic infrastructure. The implementation complexity for the kitchen demonstration however, is clearly much less under Hive. Additionally, more code is capable of being used in multiple applications under Hive.

The kitchen is clearly a domain which is open to substantially more exploration. Immediate extensions of this system include upgrading to the originally planned polyphonic tag system, adding new appliances, and extending software capabilities. Future possibilities are discussed further in 9.

7 APPLICATION SCENARIOS

Hive has been utilized in a number of other applications in addition to the kitchen demonstration described in Section 6. These systems have provided further examples of Hive's capabilities.

"Honey, I Shrunk the CDs" demo
Photo credit: Peter Menzel

7.1 "HONEY, I SHRUNK THE CDS"

Consider a bowl full of small discs, each representing a song or artist. One of these discs is selected and placed on a surface in front of the bowl. The corresponding music plays, and the disc is thrown back in the bowl.

As a demonstration of Hive, a music playing system, "Honey, I Shrunk the CDs"[10], was developed. This was a system of three agents and two shadows constructed to provide a novel interface to playing music.

The demonstration was composed of a Radio Frequency Identification (RFID) tag reader which had a series of small circular tags with a resemblance to small CDs (hence the name of the system) and an MP3 based jukebox. Figure 12 shows the basic software configuration for the scenario represented by "Hive Cell 1" and "Hive Cell 2". "Hive Cell 3" shows the extensions made to the system, as discussed in Section 7.1.1.

The three agents, shown in cells 1 and 2 of the figure were the *TagReaderAgent*, which watched the tag reader for the appearance of disappearance of tags, the *JukeboxAgent*, which was responsible for playing requests, reporting songs played and reporting when the end of a song was reached, and the *HoneyIShrunkTheCDs* agent, which managed the entire application. Each of the *JukeboxAgent* and *TagReaderAgent* communicated with a corresponding shadow.

The *HoneyIShrunkTheCDs* agent contained all of the real application intelligence for this scenario. On startup, it would do a series of lookups using the lookup system described in Section 5 to locate the demonstration system tag reader and the jukebox, and would connect itself to each. If one or neither was available, it would wait until they were. Once these

[10] In reference to the film "Honey, I Shrunk the Kids"[GS89]

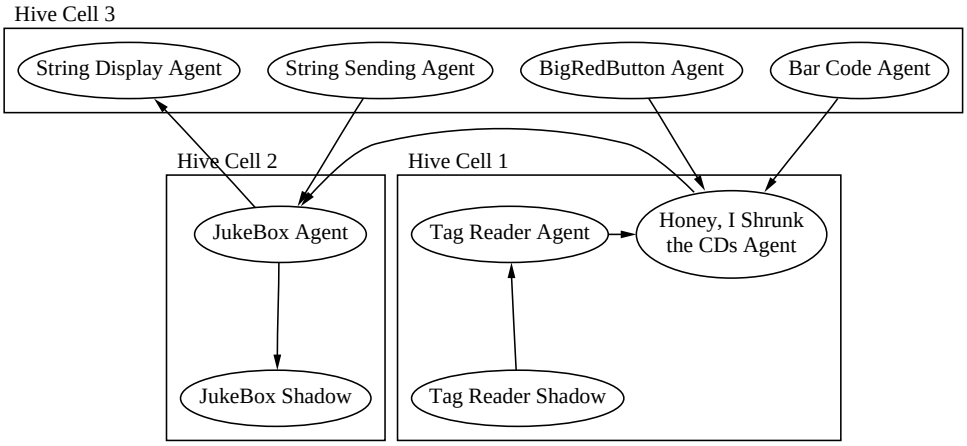


Figure 12: "Honey, I Shrunk the CDs" software configuration

connections were established, it would translate incoming information about the tags on the tag reader into either changes in its internal state (such as whether the request should be continuous or one time only), or requests to the jukebox to play a song.

While a very simple application, it proved to be a good test of the stability of the Hive software over long periods of time. The demonstration system ran continuously for weeks under regular use, without any substantial problems, including successfully handling numerous interactions that were intended to crash the system, as well as many less maliciously intended extensions to the system, as described below.

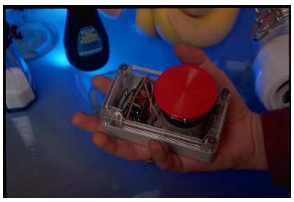
7.1.1 EXTENDING THE SCENARIO

Once the scenario was in regular use, a number of extensions were implemented. Some of these were for actual use, to make the system more useful as a jukebox in a public space, and others were simply to show what was possible.

The first extension was string based control of the jukebox. Often, it was desirable to request a particular song without associating a particular RFID tag with that song. This was accomplished without writing any new code; A *StringSendingAgent*, which allows a user to enter a string, could

simply be connected to the *JukeboxAgent* and songs could be requested by name. This exhibited one of the key features that Hive was designed for: ability for interaction between components that weren't conceived of as interacting when they were first designed. The Jukebox was designed with only the tag reading system in mind, and the string sending agent was a simple standard utility agent.

Second, as a demonstration of the ease of interoperability, a bar code reader was able to be put into the setup in the same role as the RFID tag reader. Just as the *HoneyIShrunkTheCDs* agent translated RFID tags into song names, it could do so (once again, with no code modifications) for bar codes.



The "Big Red Button"
Photo credit: Peter Menzel

Finally, the "Big Red Button" was hooked up to cause the jukebox to play a random song. This required minimal code modifications, as the button agent did not generate string events, as the tag reader, bar code reader and string sending agents did. It also could have been accomplished with no code modification and the creation of a new agent which would monitor the button and submit new requests to the jukebox.

In order for this system to be useful as a mechanism for playing music as well as as a demonstration system, even more convenient interfaces need to be constructed. Currently, control of the system is either through the tag reader, or by starting a Hive cell with local agents, such as a string sender, and connecting to the jukebox. More convenient for most users would be a web browser based interface. The beginnings of a web based mechanism for controlling agents has been implemented and it will be straightforward to utilize in this scenario.

7.2 PERSONAL LOCATION USING RF LOCUSTS

Brad Rhodes with wearable
Photo credit: Webb Chappell

Another application of Hive was the implementation of a personal location service using RF Locusts[SKA97] and a wearable computer, by Brad Rhodes. A "locust" is a device which transmits a location beacon. The original locusts did so using infrared, and a later revision[Lof] transmitted RF at 416 MHz. A user with a wearable computer receives this transmis-

sion, and can identify its location. The wearable computer then chooses how to utilize this information, whether to publish it for consumption by outside applications, or to act on it.

Hive enables sophisticated actions to be performed based on this data. A simple initial application caused a particular song to be played on a jukebox (the same code as the one in the previous example) when a person arrived in a particular room. Additionally, Hive allows for easy selection of whether or not to make the location available to the outside world. The case of the arrival song is an example of limited publication. In this case, entering a particular room, causes an action, which may been caused by another source, and entering other rooms may not cause any effect.

Additionally, further development of this scenario will utilize the location component of semantic descriptions to allow a wearable to identify resources available to it when it enters a particular room. That is, a room may have a jukebox, display, or other devices that the wearable could utilize if it was "aware" of the fact that it is in the same room. Hive and semantic descriptions enable this. This is discussed in more detail in [RMW99].

7.3 HIVE DEMONSTRATION SYSTEM

A Hive demonstration system was constructed with a variety of generic devices to use to exhibit Hive's features. These included a number of screen based agents as well as a number of physical devices. The screen based agents included buttons, image displays, flashing dots, and string inputs. The physical devices included the "Big Red Button", a toggle switch, a "Tickle Me Cookie Monster", a AC relay used to toggle a light, a number of cameras, a motion detector, and a cricket[MMB+99] (a small lego robot).

"Tickle Me Cookie Monster" with electronic guts

These devices were connected together in a wide variety of ways. Most of the connections were one-to-one connections of the various event sources to the event receivers. Examples included the big red button causing cookie monster to laugh, using the toggle switch, or cookie monster's

tickle sensor to trigger a light, using the motion detector to trigger movement of the cricket, and use of the button in the Cookie Monster doll to cause a picture to be taken. This demonstration showed the flexibility of Hive in connecting devices in ways not anticipated in the device's original design.

7.4 OTHER AND FUTURE WORK

A number of other projects within the Media Lab are utilizing Hive. Craig Wisneski has built a variety of tangible interfaces that are connected to Hive. Brygg Ullmer is developing "Strata", a project exploring the design of layered, electronically-augmented physical models that serve as tangible interfaces to specific dataspaces, that uses Hive to connect displays to data sources.

Future applications of Hive may include connecting Brad Geilfuss' Net Weight and Inner View[Gei99] projects, as well as providing low-level transport mechanisms for Hive, such as via Hyphos, by Rob Poor.

Net Weight prototype

8 JINI

Jini and Hive share many qualities. Both are distributed software systems designed to be simple and flexible and targeted at small applications involving spontaneous networks of small devices. Additionally, both are implemented in Java, and share many of the modes of interaction that this implies. Finally, both have a concept of component services or agents which are mobile. While both systems utilize code mobility, Jini has a somewhat different paradigm of operation: the services export their interfaces and stubs, while in Hive there is this mode of mobility as well as the ability for an agent to entirely move from place to place. This section discusses the basic Jini architecture, compares it with Hive, and explores their differences, inter-compatibility and strengths.

8.1 BASIC ARCHITECTURE

Jini was developed by Sun Microsystems to "enable impromptu networking of a wide variety of devices"[JFS]. As with Hive, Jini is implemented in the Java language, however, as it was developed within Sun, utilized a later version of Java, JDK 1.2 (aka "Java 2"). Jini was initially released in January, 1999.

The basic components of a Jini system are called services and clients. These services join a Jini federation via the discovery and lookup protocols, which are used to locate other services. Jini also incorporates specific notions of leases and events into its architecture. A client uses the discovery and lookup protocols to locate services, but does not join a federation itself.

The discovery and lookup protocols provide a mechanism for Jini services to find one another. "Discovery" in the Jini sense, is the process of a new service locating a nearby lookup service. This is done by sending a broadcast or multicast packet which contains enough information for the Jini lookup service to establish communication with the new service. The new service then registers itself with the lookup service, and other

services can then use the lookup services to find it. After this, all communication between services is currently via RMI[RMI], though individual services may implement their own protocol, if desired.

Jini services, in addition to interacting with each other, may be connected to directly by a client. A client uses the discovery and lookup protocol to find a desired service, but does not register itself as a service.

Leases are the mechanism Jini utilizes to deal with distributed failure. One service's interaction with another is specified for a limited lease time, which must be renewed. If the other service, or the intermediate network has failed, this lease renewal fails, providing an opportunity for the service to gracefully respond to the outage.

Jini also defines a mechanism for sending "Distributed Events"[JDE]. These events are closely tied to the standard Java Event Model, but are adapted for use in a distributed environment. Events provide a convenient mechanism for devices to send general notifications, such as the availability of a new service.

Though not strictly part of Jini, Sun provides two standard and generally useful Jini services: JavaSpaces and a Transaction Manager. JavaSpaces provide a network "location" for interaction between distributed components. Requests, responses, or other types of data may be posted in a JavaSpace for examination and potential use by other services. The Transaction Manager provides a way to execute complex transactions in the face of potential failures. This sort of service is critical for many applications, and while it could readily be implemented within each application, Sun chose to provide it as a general Jini service.

8.2 ARCHITECTURE COMPARISON

At a fundamental level, Hive and Jini are closely related; both are Java based distributed object systems implemented in Java that communicate principally via RMI, and provide mechanisms for discovery of other services. In a number of ways, such as Hive's notion of a cell, the particular approach toward lookup, and how code mobility is treated, they are quite

different.

Hive includes the notion of a "cell", in which agents and shadows reside. In Jini, there is no notion of collocation. Correspondingly, Jini lacks the notion of a shadow. The shadow abstraction becomes more useful when software components are fully mobile, for the reasons discussed in Section 4.1.2. Jini services can act as a proxy for a local device, in much the same way a Hive shadow does. In Jini, if a full software mobility layer were added, services would have to implement all communications with a device themselves, or an equivalent of the shadows abstraction would have to be created. Additionally, in Hive, all components that communicate over the network are expected to be agents, and there is no distinct concept of a client.

In Hive, the structure of a cell provides a default discovery mechanism. When an agent is "born", it is told where (which cell) it is, and correspondingly does not need to engage in any sort of discovery protocol directly. To some extent, this simply defers the discovery problem to the level of the cells: How do cells discover one another? Certainly a broadcast/multicast approach such as Jini could be utilized, or a more mobile-agents-oriented approach could be taken, such as having "discovery agents" hopping from cell to cell, identifying other cells that are known, and sharing this information as it travels. This does not completely solve the problem, as it requires bootstrapping via either centralized or hierarchical long-lived/well known cells, or via a Jini like discovery protocol. However, even without this, the cells themselves provide a convenient form of limited federation.

The way actual lookup occurs is somewhat different between Hive and Jini. In Hive, each cell acts like a Jini lookup service, but individual agents may provide lookup services for other agents collected from multiple other cells. Further, the Jini approach of "Attributes" and Hive's semantic labelling share many qualities, and are worth exploring separately. This comparison appears below in Section 8.3.

Hive lacks Jini's feature of leases altogether, though the Jini leasing system could be readily used within Hive with minimal changes to Hive. Further, Hive uses the identical distributed event model as Jini. Hive pro-

vides a number of default services, such as an agent that allows creation of a user interface, a server list agent which manages a list of currently known cells, but no JavaSpaces analog or transaction manager is used.

One of the more significant differences between Hive and Jini is the approach taken toward code mobility. In Hive, agents are fully mobile, and may fully transfer their code and thread of execution to another machine, at will. Then, any network communication with this agent is done via dynamically downloaded RMI stubs. In Jini, a service is not capable of initiating movement on its own, and only implements the latter form of mobility. In either case, there is nothing to prevent the dynamically downloaded stubs from being a full implementation of the service, rather than just an RMI proxy.

8.3 SEMANTIC DESCRIPTIONS

Semantic descriptions in Hive play the same role as Jini Lookup Attributes[JLA99]. In Jini, services are registered with a lookup service as a `ServiceItem`, which is composed of a reference to the service, and a set of attributes. These attributes are Java classes. When lookup is performed, exact match checks are performed against the public instance variables in the attribute classes, with null acting as a wild card. One key difference between this approach and the Hive system is the lack of hierarchical descriptions.

The Jini Attribute system has the advantage of being strongly typed, due to the fact that it utilizes the Java type system. This advantage is lessened, however, by the lack of hierarchical descriptions. Because of this, the fields of many attributes will be represented as `String`s, rather than actual data structures. For example, an object may have an attribute of "Owner" which has the fields "name", "email", and "department". It would be convenient if the "department" field could then have fields like "supervisor". In the Jini attribute approach, it is necessary for the "department" to be a `String` or equivalent for the matching to work. Looking up all services whose owners work for a department supervised by a particular

person becomes impossible within the lookup system. This requirement is mitigated by the fact that it is often easy to just flatten such structures, but this does not scale well.

As mentioned in Section 5, Hive's semantic labeling system can alleviate some of the weaknesses of having no strong typing through use of DTDs and DDML schemas. Further, this allows Hive descriptions to more readily interoperate with non-Java based descriptions.

In Jini, the description, or attributes, is not directly tied to a service. Due to the fact that the attributes and the service reference are wrapped together into the `ServiceItem` bundle, it is possible that the same service could be referred to by multiple `ServiceItem`s, each of which has different attributes. This has certain advantages, in flexibility, and certain disadvantages, in the case of an object listed in multiple lookup services that wants to make a single change.

8.4 COMPATIBILITY

Once Hive is moved to Java 2, it should be straightforward to allow a Hive agent to act as a Jini service. It would need to participate in the discovery and lookup processes on its own, and manage leases. Other than this, communications from other Hive agents and other Jini services could proceed as usual, including event generating agents using a single receiver list composed of Hive and Jini components.

Similarly, a Jini service would require minimal modification for it to be able to be instantiated inside a cell as a Hive agent. The Jini service would need to implement methods related to the Hive agent life cycle. Event oriented agents would also utilize Hive's mechanism for event subscription, although even within Hive this is not a strict requirement.

It would be further possible to enhance Jini and Hive interoperability by recasting the Hive cell itself as a Jini service. This would provide a mechanism for adding Hive-style mobility to Jini, where services could initiate movement from one location to another. The cell service would provide the mobility methods and the Hive lookup methods. Additionally,

Jini's lookup service would provide a way for cells to initially discover one another.

Without any changes, Hive an Jini share the same distributed event model. Utilizing this commonality may be difficult due to the bootstrapping problem of a Jini service getting a reference to a Hive agent or vice versa. If this bootstrap problem were solved, however, Jini and Hive services could readily obtain references to one another, as both systems use the distributed event system to notify interested parties of new agents/services.

8.5 ANALYSIS

When compared to Jini, Hive has a number of advantages and disadvantages. Most of its advantages make it better suited for use in a research environment, and less well suited for commercial deployment, which is appropriate given the context of the development of each system.

The advantages of Hive include the approach toward semantic labeling, Hive-style code mobility, and an explicit distinction between local and network resources, in agents and shadows. The semantic labeling approach used in Hive is especially useful in exploration of ontologies for the things that think domain. By allowing arbitrarily structured, dynamically modifiable descriptions, new techniques and schema can be tried out somewhat more flexibly and quickly than the Jini Lookup Attribute approach. This flexibility is at the cost of some added complexity, however.

Hive also provides complete mobility. This provides a number of advantages, most of which have not yet been thoroughly explored. Jini's stub mobility provides much of the needed features in common application domains, and with custom generated stubs, some of the possibilities of full mobility become available. Further, as mentioned above, full mobility could be added to Jini without a total redesign.

Hive explicitly includes a notion of locality. This is necessary for its

approach toward mobility; if an agent wants to move, it needs a place to move to. This concept of locality allows a way for agents to find localized resources. Hive's shadows layer provides a standard way for these agents to communicate with local devices, such as described in the Jini Device Architecture Specification[JDA99], but in such a way to accommodate the mobility of the agents. Jini can accomplish some of the same goals by using a capability based security model for untrusted services.

Jini currently has substantial advantages in terms of initial discovery. Hive cells have no bootstrap mechanism other than specifying a well-known cell or cells, to join the larger network. Additionally, Jini provides the stability one would expect from a commercial system, in contrast to Hive's lesser stability, as would be expected from a research system.

9 ANALYSIS AND CONCLUSIONS

9.1 KITCHENS

Only the very beginning of the exploration of a networked kitchen has begun in this work. The Hive system proved useful in development of a flexible, decentralized demonstration networked kitchen. The implementation should also provide a good platform on which to extend and further examine possible applications in the kitchen.

A recurring theme in all attempts to build a "kitchen of the future" have been that opinions as to how it should work are strong, varied and often diametrically opposed. A componentized system for kitchens, such as the one built with Hive, will be necessary to experiment with scenarios to determine which approaches are successful, which are not, and which depend on the particular user preference.

9.2 HIVE

Hive has succeeded in allowing a number of complex systems of things that think to be built. The kitchen demonstration system alone is the largest individual Hive application built so far, and it did not run into any scaling related problems itself. During the Spring TTT meeting, a number of other projects within the lab utilized Hive, and some systems experienced problems related to too many RMI sockets being produced. In both the fall and spring demonstrations, many unanticipated ways of connecting devices together were tried with no other difficulties.

Further, the Hive system proved to be relatively straightforward for other users to utilize, despite only minimal documentation and personalized help being available at the time of the demonstrations. A number of the other projects, particularly the wearables project described in Section 7.2 continue to use Hive extensively.

9.2.1 SEMANTIC LABELING

The RDF-based semantic labeling approach also proved successful in application in a small number of systems. In the kitchen demonstration, it allowed for flexibility in the use of hardware. This became especially relevant when the tag reader planned on being used turned out to be broken, as described in Section 6.3.

Continuing work using semantic labels will help evaluate whether RDF is a suitable solution, givens its disadvantages of complexity. This may be mitigated or eliminated by the production of RDF tools, either as part of the Hive project or as general RDF tools.

9.3 FUTURE WORK

There are many directions in which to continue this research. Both the kitchen demonstration and the Hive infrastructure have shown substantial promise.

9.3.1 KITCHENS

In the kitchen, an initial area for work is the addition of new devices to the repertoire of the system. This will include new devices similar to those that exist, such as a superior tag reading system, as well as currently unimplemented devices, such as blenders, stoves, sinks, and ovens. An important step in pursuing this is moving the research into a real kitchen rather than the prototype mock up that has served so far. Further, integration of a scheduling system into the recipe manager, as well as a more sophisticated inventory manager will provide challenging software projects.

Once an actual kitchen with a standard array of appliances is available, experiments in usability and stability can begin. User interface in the kitchen is of especial importance, as a traditional screen display is not well suited to the environment. Various input and output forms, including projection, speech in and out, use of wearables, and novel display

devices should be explored.

9.3.2 HIVE

The Hive infrastructure should continue to be developed in a number of ways. The current mobility layer is incomplete, and completing this will allow for a variety of interesting explorations into uses of mobile code. Experience with users of Hive has shown that while Hive is useful and approachable to most users, there is room for a number of utilities to manage semantic descriptions and configurations, more documentation for implementors, and a more straightforward method to install Hive.

Other future developments are likely to include some integration with a web server to allow "web application server"-like systems. A variety of novel applications of code mobility can be examined, including a mobile-code-based discovery system and implications to security. Partial or full integration with Jini is a further possibility. A release outside of the Media Lab is also planned.

9.4 CONCLUSION

Hive has provided the necessary infrastructure to construct a successful prototype kitchen that can be extended and developed further. Additionally, Hive has achieved initial success in making it substantially easier to build complex networked systems of things that think.

A DOCUMENTATION

A.1 RDF FOR CONFIGURATION

```
------------
| OVERVIEW |
------------

Hive configuration files allow you to do a number of things:

  o  Specify which shadows and agents get automatically started
  o  Specify communication channel parameters (baud rate, etc)
  o  Specify other agent or shadow configuration
  o  Specify a different PPM icon for a particular instance of an agent
  o  Specify semantic descriptions for agents and shadows

This makes it easy to create an application built on top of Hive that
just involves running Hive, and requires no manipulation or
configuration through the GUI.  This is useful for demos so that you
can avoid clicking through multiple PropertySheet's and the like.

These files are stored in a single directory, be default .hive in your
home directory.  A different directory can be specified on the
command line with the -configdir option.  For example:

"java edu.mit.media.hive.server.Server -configdir=/home/mkgray/democonfig"

-------------------------
| STANDARD CONFIG FILES |
-------------------------

There are two default hive configuration files plus one per shadow or
agent that you want to have custom configuration for.  This section
describes the two default files.  The format of the per shadow/agent
config files is in the next section.

The default filename for the first is "agentConfig", but can be
specified to be something else with the -agentconfig command line
option.  The default filename for the other is "shadowConfig", and can
be specified to be something else by the -shadowconfig command line
option.

Both of these files have the same format.  On each line, an agent or
shadow class is specified and the name of an RDF configuration file is
listed.  A line beginning with a '#' is considered a comment.  Class
names must be fully qualified.  For example, an agentConfig file to
start up a running graph agent with the configuration file mygraph.rdf
would look like this:
```

```
------- cut here-------
# Just start up my running graph agent
edu.mit.media.hive.agent.desktop.RunningGraphAgentImpl mygraph.rdf
------- cut here-------
```

If you wanted to start up multiple running graph agents, with different configurations, you'd do something like this:

```
------- cut here-------
# Start up three running graph agents
#
# Start two graphs with my standard settings
edu.mit.media.hive.agent.desktop.RunningGraphAgentImpl mygraph.rdf
edu.mit.media.hive.agent.desktop.RunningGraphAgentImpl mygraph.rdf
#
# And start another with my other settings
edu.mit.media.hive.agent.desktop.RunningGraphAgentImpl myothergraph.rdf
------- cut here-------
```

The shadowConfig file is the same. For example, if you wanted to start two cricket shadows, running on different serial ports:

```
------- cut here-------
# Start up our cricket shadows
edu.mit.media.hive.shadows.CricketShadow cricket-com1-config.rdf
edu.mit.media.hive.shadows.CricketShadow cricket-com2-config.rdf
------- cut here-------
```

One added important piece is that if you use a shadowConfig file, these are THE ONLY SHADOWS that will ever be allowed to be started. For example, if you use the above configuration and then try to start an agent that tries to use another shadow (say, a QuickCam shadow), it will be forbidden. So, be sure to specify all shadows you want started. You are still free to start arbitrary agents.

```
-------------
| RDF FILES |
-------------
```

The RDF files specified in the agentConfig and shadowConfig files determine the configuration for each individual agent or shadow. These files are found in your hive config directory.

All Hive RDF files should contain the following at the beginning of the file:

```
------- cut here --------
<?xml version="1.0"?>
<RDF
```

```
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about="">
------- cut here --------
```

and at the end of the file:

```
------- cut here --------
</Description>
</RDF>
------- cut here --------
```

The rest of this section will describe what goes in between these. In order to specify a configuration parameter for a agent or shadow, you use something like the following:

```
<thing:config thing:iconPPMName="scale.ppm"
      thing:sampleRate=".5"/>
```

This specifies that the iconPPMName is "scale.ppm" and that the sampleRate is ".5".

To configure a serial port is slightly more complicated:

```
<thing:config thing:parameterFoo="blah">
<Description>
<thing:channel thing:baudRate="9600"/>
</Description>
</thing:config>
```

This sets parameterFoo to "blah", and the baud rate of the serial port used by this shadow to 9600.

Now, how does an agent or shadow use the values specified in these config files? By using Java Beans. For example, in the previous example, an agent would need to implement two methods:

```
public String getParameterFoo();
public void setParameterFoo(String s);
```

and, automatically setParameterFoo would be called with the value from the config file. This happens between agent construction and the call to arriveAt(), so the agent will not even have a Server reference yet when it is configured, so best practice is to store away the value when setParameter(...) is called, and use it in arriveAt() or doBehavior(). Configuration parameters may be String's, int's, float's, double's, boolean's and anything else there is a

```
PropertyEditor for (java defines the ones listed by default).

To allow serial port or other LocalChannel configuration, you don't
need to do anything other than specify the parameters in the config
file, if you've used the standard Hive serial port and external
process support.

The configurable parameters for a serial port channel are:

Parameter Value
--------------
baudRate 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
serialPort 1, 2, 3, ... (note these correspond to COM1, COM2, ...
or ttyS0, ttyS1, ttyS2, ...)

The configurable parameter for a process channel is:

Parameter Value
--------------
processCommand a string specifying the external command to run

Additionally, AgentImpl has one default configurable parameter:

Parameter Value
--------------
iconPPMName A string specifying the filename of a 32x32 raw PPM
(this file should be in your hive config dir)

Finally, an RDF file may be used to provide a semantic descrip-
tion of
the agent or shadow.  This is a substantial topic in and of it-
self,
and will be addressed separately, at another time.

------------
| EXAMPLES |
------------

Here is a set of sample configuration files that would work to-
gether,
with descriptions of what they do.

===== agentConfig =====
# Create a couple of agents
edu.mit.media.hive.agent.desktop.RunningGraphAgentImpl mygraph.rdf
edu.mit.media.hive.agent.thing.ScaleAgent scale.rdf
======================

===== shadowConfig ====
# Start the shadow for the scale
```

```
edu.mit.media.hive.shadows.TranscellScale transcell.rdf
=======================

This configures the graph to update every half second, and have a
scale of .0625 (1/16) so that when hooked up to the scale it will show
a graph of ounces over time.  "updateInterval" and "scaleFac-
tor" are
parameters that the RunningGraphAgentImpl accepts.
===== mygraph.rdf =====
<?xml version="1.0"?>
<RDF
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about="">
        <thing:config
                thing:updateInterval="500"
                thing:scaleFactor="0.0625"/>
</Description>
</RDF>
=======================

This just specifies an alternate PPM for the scale
====== scale.rdf ======
<?xml version="1.0"?>
<RDF
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about="">
        <thing:config
                thing:iconPPMName="scale.ppm"/>
</Description>
</RDF>
=======================

Configure the scale to be on port 1 at 9600 baud.
This file isn't actually necessary, since port 1 and
9600 baud are the defaults.
==== transcell.rdf ====
<?xml version="1.0"?>
<RDF
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about="">
        <thing:config>
                <Description>
                <thing:channel
thing:baudRate="9600"
```

```
thing:serialPort="1"/>
</Description>
</RDF>
=======================
```

A.2 HOW TO USE SEMANTIC DESCRIPTIONS

```
HOWTO use semantic descriptions in Hive
---------------------------------------


INTRO
-----
Semantic descriptions in Hive are based on RDF.  This means everything
is a directed labeled graph.  Using the Lookup utility class or the
DescSet.select() method, you can query these graphs and find resources
that match your requirements.

SAMPLE
------
Here's a sample RDF file, describing a camera in room 468:

<?xml version="1.0"?>
<RDF
     xmlns='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:RDF='http://www.w3.org/TR/WD-rdf-syntax#'
     xmlns:thing='http://www.media.mit.edu/hive-syntax#'>
<Description about=""
   thing:nickname="Pia Quickcam">
      <thing:config thing:command="cqcam"/>
      <thing:location
            thing:building="E15"
            thing:room="468"/>
      <thing:role>
            <Description>
                  <thing:camera thing:kind="QuickCam"/>
            </Description>
      </thing:role>
</Description>
</RDF>

The rdf statements this makes are:

The "nickname" of (the described object) is "Pia Quickcam"

The "config" of (the described object) is FOO
The "command" of FOO is "cqcam"

The "location" of (the described object) is BAR
```

```
The "building" of BAR is "E15"
The "room" of BAR is "468"

The "role" of (the described object) is BAZ
The "camera" of BAZ is QUUX
The "kind" of QUUX is "QuickCam"

Note, it says about="".  Leave this as is, and Hive will automatically
generate the correct object associations.  Putting something else in
the about field may produce strange results.

This may seem like a mess (and it is), and in general you won't need
to think about things this way to generate descriptions, or query
them, but it is useful to know when starting out.

IN GENERAL
----------
In general, to discover a new agent, you would do one of the following
two things.  Say you wanted to find a camera, such as the one
described above.  You might do:

Agent a = Lookup.findAgentMatch(myServer, this,
                  "edu.mit.media.hive.EventTranslatingAgent",
                  Description.HIVE+"camera");

This will give you the first agent it finds that is of syntac-
tic type
EventTranslatingAgent and has the role of camera.  The
"Description.HIVE" prefix, is the RDF name space stuff, which I explain
in the details below.  If you don't care about the details, always
prefix types with it.

If instead you want to find all agents in room 468, you might do:

DescSet set = Lookup.findAgentMatch(myServer, this, null, null);
set = Lookup.require(set, Description.HIVE+"location",
                    Description.HIVE+"room",
                    "468");

These lines say, correspondingly "Give me all the agents on this
server", and "Give me all the agents whose location has a room of
468".  (note, this code will give agents in room 468 of any building)
Then if you want an enumeration of the matching objects you would do:

Enumeration e= set.elements();

Or, if you want the first match, you would do:

Object o = set.elementAt(0);
```

```
If you want the number of matches, you would do:

int n = set.matches();

DETAILS
-------

So, how do you discover a particular agent?  For agents, you do so by
calling queryAgents(...).  For example, to get all of the
EventSendingAgent's on the local cell, you could do the "raw" query:

String[] esa = { "edu.mit.media.hive.agent.EventSendingAgent" };
DescSet mySet = myServer.queryAgents(this, esa, null);

or, more likely, to save you the trouble of creating the tem-
porary array,
you use the utility method in Lookup:

DescSet mySet = Lookup.getAgentSet(myServer, this,
                           "edu.mit.media.hive.agentEventSendingAgent",
                           null);

queryAgents and getAgentSet each take arguments of the caller, a
string array or string corresponding to the agent syntactic type you
want, and a string array or string corresponding to the agent semantic
role you want.  Either of these may be null to match any.  getAgentSet
also has as a first argument the RemoteServer to query.

[Note: in all the examples from here on in, I'm not prefacing types with
      Description.HIVE or any other name space prefix.  This is because
      it's messy looking and I don't feel like it, not because it isn't
      needed]

Now you have a DescSet object which is a set of 0 or more descriptions
which can be queried and manipulated.  Now, say you want to se-
lect the
agent from this set that has the nickname "Pia Quickcam".  To do so,
you would do:

DescSet mySet = mySet.select(Description.HIVE+"nickname", "Pia Quickcam");

Select takes two arguments, the label of the RDF graph edge to fol-
low,
and a second optional argument of the required value of that node.  To
clarify, here's another more complicated example:

set = set.select("owner"); // This selects the owner of each item
set = set.select("birthday"); // This selects their birthdays
set = set.select("month", "July"); // This selects those whose birth-
day is in
```

```
                          // July
```

Each select causes the new, potentially pared down, set to be returned. If a description in the set does not have an owner at all, the first call will eliminate it. The second will eliminate all owners that don't have birthdays, and the third will eliminate all owners whose birthdays whose months are not "July". Get it?

As you may have inferred by now, there is a "context" in a set, so that you can do structured queries like the above one. What if in the above scenario we wanted to, after all this, select only objects in
E15. If we immediately did:

```
set = set.select("location");
set = set.select("building", "E15");
```

That would be equivalent to having said: Give me all objects which have a owner which has a birthday which has a month equal to "July" which has a location which has a building equal to "E15". Since a month does not have a location, and that's not what you meant, you have to do something else. So, you have to reset the context. You do this by doing:

```
set.noContext()
```

Note, this has no return value, it actually changes set, rather than return a new one. Eventually I'll clean up the API to be more consistent, but deal for now. So, the total sequence of:

```
set = set.select("owner"); // This selects the owner of each item
set = set.select("birthday"); // This selects their birthdays
set = set.select("month", "July"); // This selects those whose birthday is in
                          // July
set.noContext();
set = set.select("location");
set = set.select("building", "E15");
```

This has the total effect of what was originally intended. Additionally, DescSet has another method upContext(), which causes the context to be moved up one level.

Now, say you want to do something with the values rather than exact match, like get all the agents in even numbered rooms. To do that, you would have to do the following once you got your DescSet mySet:

```
mySet = mySet.select("location");
mySet = mySet.select("room");
```

```
DescSet newset = new DescSet();
for(int i=0; i < mySet.matches(); i++){
      if(isStringEven(mySet.getValue(i))){
            Object m = mySet.elementAt(i);
            // Do whatever you want with the match m, like
            // add it to a Vector or whatever, or maybe put its
            // description back into a new descset for more queries:
            newset.addDesc(m.getDescription());
      }
}
```

Now, what if you want to look at the description of an object that you already have? At the moment, this requires putting it into a descset and manipulating it as above. Say I have an Agent foo (or any object that
implements Describable), I do:

```
DescSet set = new DescSet();
set.addDesc(foo.getDescription())
```

If you have a Vector of Describables, you can just throw that in the constructor:

```
Vector foo = new Vector();
// Add some describables to foo
DescSet set = new DescSet(foo);
```

From this point you can manipulate the set as above.

Now, what do you do if you have multiple DescSet’s and you want to mush them together? (eg, you query multiple servers, and now want to find all objects in room 468, on all the servers you queried) You use DescSet.merge():

```
DescSet everything = new DescSet();
DescSet foo = Lookup.getAgentSet(server1, this, null, null);
everything.merge(foo);
foo = Lookup.getAgentSet(server2, this, null, null);
everything.merge(foo);
foo = Lookup.getAgentSet(server3, this, null, null);
everything.merge(foo);
foo = Lookup.getAgentSet(server4, this, null, null);
everything.merge(foo);
```

This will cause the DescSet "everything" to contain the merged results from all 4 queries. merge(...) will purge duplicates should the occasion arise where you try to merge a set which contains repeats.

That is, set.merge(set) is a no-op.

```
The Lookup class has a number of utility functions to make do-
ing this
sort of thing slightly easier, but this utility class is incomplete.
But, you can do anything with DescSet, if in a few more API calls.
Here's the relevant/useful methods on DescSet:

DescSet(Vector v)
      Construct a DescSet from a Vector of Describable's
DescSet()
      Construct a new empty DescSet

int matches()
      Number of objects that match the current query
Enumeration elements()
      An enumeration of the objects that match
Object firstElement()
      Get the first match
Object elementAt(int i)
      Get the object match at index i
Object pickOne()
      Randomly select one of the matches.
String getValue(int i)
      Get the value of the current query in the _description_
      (ie, if you've selected the name of the owner, this will give
       you the name of the owner, while elementAt() would give you
       the object itself)
DescSet select(String p)
      Prune the set to objects that have a parameter p in their
      description and current context
DescSet select(String p, String v)
      Prune the set to objects that have a parameter p with value v
      in their description and current context
void merge(DescSet s)
      Merge s into the current set
void addDesc(Description d)
      Add a description to the set
void removeDesc(Description d)
      Remove a description from the set
void upContext()
      move the context "back" one
void noContext()
      Reset the context
```

REFERENCES

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

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. ISBN: 0–201–63455–4.

[BCMS99] Ronald Bourret, John Cowan, Ingo Macherius, and Simon St. Laurent. Document Definition Markup Language. Technical report, W3C, 1999. http://www.w3.org/TR/NOTE-ddml

[BPSM97] 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

[C2C] Coast to Coast. http://www.media.mit.edu/c2c/

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

[CLK98] Patrick Chan, Rosanna Less, and Douglas Kramer. *The Java Class Libraries*, volume 1. Addison Wesley, 2nd edition, 1998.

[DWI98] A. Dahle, C. Wisneski, and H. Ishii. Water Lamp and Pinwheels: Ambient Projection of Digital Information into Architectural Space. In *Summary of Conference on Human Factors in Computing Systems (CHI '98)*. ACM Press, 1998. http://tangible.www.media.mit.edu/groups/tangible/papers/Ambient_Fixtures_CHI98/Ambient_Fixtures_CHI98.html

[Eve98] 1998 Everest Expidition, 1998. http://everest.www.media.mit.edu

[FFR96] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua Server: A Tool for Collaborative Ontology Construction. Technical

Report KSL-96-26, Knowledge Systems Laboratory, Stanford University, September 1996. ftp://ksl.stanford.edu/pub/KSL_/_Reports/KSL-96-26.ps

[FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Readings in Planning*, pages 189–208. 1971.

[Gei99] 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.

[Gra] Steve Gray. Bit Bags.

[GS89] Stuart Gordon and Tom Schulman. Honey, I Shrunk the Kids!, 1989. http://us.imdb.com/Title?Honey,+I+Shrunk+the+Kids+(1989)

[Ham86] Kristian J. Hammond. CHEF: A Model of Case-based Planning. In *AAAI 1986 Proceedings*, pages 261–271, 1986.

[HIV] Hive Project Web Page. http://hive.www.media.mit.edu/projects/hive/

[HTM] HyperText Markup Language. http://www.w3.org/MarkUp/

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

[JB] JavaBeans: The only component architecture for Java. http://java.sun.com/beans/index.html

[JDA99] Jini Device Architecture Specification. Technical report, Sun Microsystems, Inc., 1999. http://ww.sun.com/jini/specs/deviceArch.pdf

[JDE] Jini Distributed Event Specification. http://www.sun.com/jini/specs/ev.ps

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

[JFS] http://www.sun.com/jini/factsheet/

[JLA99] Jini Lookup Attribute Schema Specification. Technical report, Sun Microsystems, Inc., 1999. http://www.sun.com/jini/specs/schema.pdf

[KQM] Knowledge Query and Manipulation Language. http://www.cs.umbc.edu/kqml/

[Lof] Alex Loffler. http://wearables.www.media.mit.edu/projects/wearables/locust/rf-locust/

[LS98] 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/

[MGR+99] 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/

[Min98] 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/

[MMB+99] B. Mikhak, F. Martin, R. Berg, M. Resnick, and B. Silverman. In Alison Druin and James Hendler, editors, *Robots for Kids*. Morgan Kaufmann Publishers, Inc., 1999.

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

[OMG94] Common Object Services Specification. Technical report, 1994. http://www.cs.wustl.edu/~schmodt/CORBA-docs/coss.ps.gz

[OMG95] Object Management Group OMG. *The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0*. Object Management Group (OMG), 2.0 edition, 1995.

[Poo99] Robert D. Poor. The iRX 2.1 ...where atoms meet bits, 1999. http://www.media.mit.edu/~r/projects/picsem/irx2_/_1/

[Red98] Maria Redin. Marathon Man. Master's thesis, MIT Department of Electrical Engineering, 1998. http://ttt.www.media.mit.edu/SF/

[RHDS98] Franklin Reynolds, Johan Hjelm, Spencer Dawkins, and Sandeep Singhal. Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation. Technical Report NOTE-CCPP-19981130, W3C, November 1998. http://www.w3.org/TR/NOTE-CCPP/

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

[RMW99] Bradley J. Rhodes, Nelson Minar, and Josh Weaver. Ubiquitous Computing Meets Wearable Computing: combining localization with personalization. submitted to The Proceedings of The Third International Symposium on Wearable Computers (ISWC '99), 1999.

[Rou99] Oliver Roup. Hive: A Software Infrastructure for Things That Think. Master's thesis, MIT Department of Electrical Engineering and Computer Science, 1999.

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

[Saa99] Jaane Saarela. SiRPAC - Simple RDF Parser and Compiler, 1999. http://web1.w3.org/RDF/Implementations/SiRPAC/

[Sac75] Earl D. Sacerdoti. The Nonlinear Nature of Plans. In *International Joint Coferences on Artificial Intelligence*, 1975.

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

[SGM86] ISO 8879 – Standard Generalized Markup Language, 1986.

[SKA97] Thad Starner, Dana Kirsch, and Solomon Assefa. The Locust Swarm: An environmentally-powered, networkless location and messaging system. In *The Proceedings of The First International Symposium on Wearable Computers (ISWC '97)*, pages 169–170, 1997. `http://lcs.www.media.mit.edu/projects/wearables/locust/`

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

[Wal98] Jim Waldo. Jini Architecture Overview. Technical report, Sun Microsystems, Inc., 1998. `http://java.sun.com/products/jini/`

[WWW] The World Wide Web. `http://www.w3.org/WWW/`

[WWWK97] 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`