A Knowledge-Based Resource Management System For The Intelligent Room

by

Krzysztof Gajos

Submitted to the Department of Electrical Engineering and Computer Science

August 11, 2000

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

Abstract

As computers become cheaper, smaller and more powerful, they begin to appear in places where until recently we did not expect to find them. The idea of ubiquitous computing and smart environments is no longer a dream and has long become a serious area of research and soon this technology will start entering our every day lives. One of the major obstacles preventing this technology from spreading is the fact that different smart spaces will have very different kinds of devices available. This lack of consistency will make it very difficult to write portable applications for intelligent spaces. In this thesis I present a set of requirements for a resource management system for intelligent spaces that will make writing portable applications for such environments possible. One of the main requirements for such resource manager is that it represents devices in terms of abstract services they provide thus making applications for smart spaces independent of the kind of equipment present in the space. I also present an actual resource management system called Realm, built according to the design requirements derived, that was implemented to solve the resource management problem in the Intelligent Room – a smart space developed at the Artificial Intelligence Laboratory at MIT. By evaluating the usefulness of Realm, I also evaluate the proposed set of design requirements for such a systems.

Thesis Supervisor: Patrick H. Winston
Title: Professor, Department of Electrical Engineering and Computer Science

Thesis Supervisor: Howard E. Shrobe
Title: Associate Director, MIT Artificial Intelligence Laboratory

Table of Contents

1 Introduction ______ 6

1.1 The Intelligent Room ______ 7

1.1.1 Metaglue ______ 8

1.2 Problem ______ 8

1.3 Why this is an important problem to solve ______ 8

1.4 What this problem is not ______ 9

1.4.1 Classical OS resource allocation problem ______ 10

1.4.2 Job shop scheduling ______ 10

1.5 Solution ______ 10

1.6 Examples ______ 11

1.6.1 Showing a movie ______ 11

1.6.2 Short Text Message ______ 12

1.7 Concepts and vocabulary ______ 13

1.8 Organization of this thesis ______ 14

2 The Problem ______ 15

2.1 Initial requirements ______ 15

2.2 Resource Management and the Intelligent Room ______ 15

2.3 Design Requirements ______ 18

3 Design Overview ______ 20

3.1 Main design decisions ______ 20

3.1.1 Structure ______ 20

3.1.2 Reactive ______ 21

3.1.3 Choice of tools ______ 21

3.2 Quick look at Realm ____ 22

3.3 The Knowledge-Based Part ____ 23

3.3.1 Meta control ____ 24

3.3.2 Knowledge expressed ____ 26

3.3.3 Ontology ____ 30

3.3.4 Reasoning ____ 30

3.4 The Constraint Satisfaction Engine ____ 31

3.4.1 Constraints ____ 31

3.4.2 Heuristics ____ 32

3.5 Cost and Utility ____ 32

3.5.1 Other approaches to cost and utility ____ 35

3.6 Connecting with Metaglue ____ 35

3.6.1 Realm Agent – making Realm accessible to Metaglue ____ 35

3.6.2 Managed Agent – making Metaglue accessible to Realm ____ 36

3.6.3 Handling Nested Requests ____ 37

3.6.4 Connections ____ 38

4 Implementation ____ 39

4.1 The Knowledge Part ____ 39

4.2 The Constraint Satisfaction Engine ____ 39

4.2.1 Constraints ____ 41

4.2.2 Heuristics ____ 42

4.3 Metaglue Integration ____ 42

5 Realm at work ____ 43

5.1 System startup ____ 43

5.1.1 Reading in the info about available agents ____ 43

5.1.2 First request – “short text output” service (with interference) ____ 46

6 Comparison With Related Systems ____48

6.1 Jini ____48

6.2 Open Agent Architecture (OAA) ____48

6.3 Hive ____49

6.4 Resource Description Framework (RDF) ____49

7 Evaluation of Realm ____51

7.1 Compliance with Requirements ____51

7.2 Impact of Realm on the Intelligent Room ____52

7.3 Lessons Learned ____53

8 Contributions ____55

8.1 Within the Intelligent Room ____55

8.2 Design Principles for Resource Management Systems for Intelligent Spaces ____55

8.3 New paradigm – user as a resource ____56

9 Bibliography ____58

Acknowledgements

I would like to thank my thesis supervisors: Prof. Patrick Winston for making me realize that I have done something of significance, and Dr. Howard Shrobe for guiding me through the process. I would also like to thank Luke and Stephen for their comments, Andy for the cookies and Michael for not bothering me much.

1 Introduction

As computers become cheaper, smaller and more powerful, they begin to appear in places where until recently we did not expect to find them. The idea of ubiquitous computing is no longer a dream and has long become a serious area of research. Some of that research is directed towards harnessing the computational power surrounding us into creating smart spaces. Smart spaces, such as intelligent houses, offices, command posts will be there to assist us in our work, help us manage and share information and facilitate communication with other people.

To be truly useful, those spaces will have to be affordable, which implies that it should be possible to build them out of mass produced, interconnected components. This includes both the hardware and the software. Hence we can imagine that in the future we will be getting software for our rooms and offices just as today we get it for out desktop computers. By that time, the “Office Suite” of programs will mean something very different from what it means today. Creating such programs, however, may prove very difficult.

It is already difficult to keep desktop computers similar enough to make it possible for the same software to run on all of them. It will certainly be even more difficult when it comes to smart spaces. People take great pride in how they arrange their work and living environments and so creators of software for smart spaces cannot impose how those spaces should be arranged or equipped. While software creators can require that a computer should be equipped with a display, a CD-Rom and a soundcard, they certainly cannot require the same level of uniformity among living spaces. Thus we have to make it possible for applications to run in a variety of spaces with diverse devices and configurations.

The differences among desktop computers have been minimized by the use of software drivers for various devices installed in those computers. Hence, it does not matter what kind of a video card or a monitor one has – the drivers are going to make all cards and monitors “speak the same language” and provide the same services to all applications.

In the intelligent spaces the situation will be even more difficult: not only will spaces have different kinds of displays, ranging from little TVs to large plasma displays, but some spaces may not have displays at all. Thus we have to express the abilities of various devices in smart environments in more abstract terms.

Instead of providing uniform interfaces to devices, as is done on desktop computers, I propose providing uniform interfaces to services provided by those devices. This distinction is more profound than it may at first appear. It comes from the fact that each service can, in principle, be provided by a number of conceptually different devices and each device can provide a number of distinct services. The "short-text-output" service may be rendered by a computer display device, a speech output device or by a one line LED display. If absolutely necessary, it could even be provided by a light device, which may flash the message in Morse code.

To achieve the above-mentioned goal, I present in this thesis a set of design requirements for a resource management for an intelligent space. I then present an actual system, called Realm, built to satisfy those requirements. Realm has been built to solve resource management problems of the Intelligent Room – a smart space developed at the Artificial Intelligence Laboratory at MIT.

1.1 The Intelligent Room

The Intelligent Room is an experiment in human-computer interaction (HCI). The goal of the project is to experiment with new kinds of applications when the traditional computer communication channels such as mouse, keyboard and monitor are replaced by new kinds of input and output devices such as speech recognition, speech generation, control of everyday electrical devices, on-wall displays and other exotic means of communication. The applications tested in the Room range from living room automation, to disaster command center to applications that help test and debug Room's other applications (thus making the Room a tool for debugging itself). The Intelligent Room project focuses on the interactions between humans and computers embedded in smart spaces, and on developing the software infrastructure necessary to implement such interactions easily and efficiently.

1.1.1 Metaglue

Metaglue [Phi99, War99, Coen99] is the software foundation of the Intelligent Room project. Metaglue is an agent platform built as an extension of the Java programming language. It provides the individual agents with numerous capabilities regarding inter-agent communication, persistent storage, execution context and management of the namespace.

1.2 Problem

The problem, which I attempt to solve here, is that of assigning abstract services provided by various devices (physical and computational) to requestors. Each device can provide a number of services. Each kind of abstract service can potentially be provided by a number of different devices with some variation of quality. The problem is to find the right devices for the requests while keeping conflicts to a minimum.

Another aspect of the problem is that we would like the Room to make "intelligent" decisions about which devices are better providers of certain services. For example, when the user is just sitting reading a book, a speech output device can be a very good candidate for rendering the "short text message" service. However, if the user is on the phone, using speech output would be too distracting; in that situation the LED sign controller would be a better candidate for providing the short text message service.

Finally, any resource manager should arbitrate among requestors making sure that no resource is being used by too many requestors at any given time while still satisfying as many requests as possible.

1.3 Why this is an important problem to solve

The purpose of this particular system is to impose a layer of abstraction in the system in order to make the applications for intelligent environments independent of the particular environment in which they run. For example, until now a movie showing applications relied directly on a particular agent controlling a particular projector. With the resource manager in place, the

movie showing application will only have to request a display and will not need to worry about what display is going to be used or what particular piece of software is responsible for controlling it. The power of this layer of abstraction becomes apparent if we take into account the fact that certain abstract services can be provided by a number of devices and each device can potentially provide a number of services. Hence, for example, if an agent needs to communicate a short message to the user, it may be granted access to the speech generator, overhead LED display, a computer display connected to a projector or, in extreme cases, a beeper agent or a printer controller. Each of those devices can provide the "short text output" service needed. It will be the role of the resource manager to decide which of those devices would best perform the service.

Resource conflicts are also a serious problem in the Intelligent Room. By now a large number of applications have been developed for the Room and when they run concurrently, they often end up trying to use the same resources at the same time. For example, information agent often takes a display away from the video showing agent.

Finally, the knowledge-based aspect of the resource manger will allow it to interact with other knowledge-based components of the smart space. Thus if the user is having a meeting with somebody, the resource manager will be advised to be discrete and to prefer visual devices over those using sound. Conversely, if the user might be relaxing or dozing off and an important message arrives, auditory clues may work better.

1.4 What this problem is not

Terms such as resource management, agent system, intelligent environment are used in a large variety of contexts. Thus people coming from various backgrounds may have various expectations of what the problem I am trying to solve might be. Below I present a few examples of problems that this system is not trying to address.

1.4.1 Classical OS resource allocation problem

The problem of resource allocation is one of optimization in a context of simple descriptions of resources. In contrast, the problem addressed in this thesis has to do with intelligent actions in a context of richly described capabilities. In OS resource allocation the requests are made for particular devices and the system has to make sure that all requests are satisfied in an expedient manner. In resource management in an intelligent space, requests are made for certain services and it is the role of the resource manger to decide how the job could be done best.

1.4.2 Job shop scheduling

Job shop scheduling deals with the allocation of resources over time to perform a collection of tasks. While this problem in the end also boils down to a constraint satisfaction problem, it is somewhat different in nature from the problem presented in this thesis. Job shop scheduling is about maximizing throughput over certain period of time. The problem being studied here is about finding the best possible configuration of resources at a particular point in time as well as deciding who are the best candidates for any request.

1.5 Solution

In this thesis I present an actual system that satisfies the design requirements derived in chapter 2. This system, called Realm, performs the resource management tasks for the Intelligent Room.

Realm is composed of three major components: the knowledge part, the constraint-satisfaction engine and infrastructure for integrating with Metaglue.

The knowledge part is mostly composed of templates for describing agents, services and requests. It also contains a number of rules and functions for matching services to requests and a set of meta control rules. The knowledge part stores information about the environment and is responsible for suggesting candidate services for all requests.

The constraint satisfaction engine uses information from the knowledge part about the current state of the room and about requests and the corresponding candidate services. On the basis of this information, it produces a model of the world with as many requests satisfied as possible and hands it back to the knowledge part. The engine uses depth-first search method with local constraint propagation.

The Metaglue interface is responsible for mediating service and information requests between Metaglue agents and Realm.

1.6 Examples

Below are some scenarios of interactions within the Intelligent Room. I first present how those scenarios were handled in the pre-Realm Room and then show how they are handled with Realm.

1.6.1 Showing a movie

In the pre-Realm Intelligent Room, an agent wishing to show a movie to the Room's occupants, had to

- bring to life the agent controlling one of the two projectors,
- bring to life the agent controlling the VCR, and
- bring to life the agent controlling the multiplexer to which both the VCR and the projector are connected,
- turn on the projector and change its input to video,
- set the inputs and outputs on the multiplexer,
- start the VCR.

Moreover, the movie agent written for one office, will not work without some modifications in the next office even if they both use similar devices. And it will require even more

modifications if, instead of a projector, the other office used a different device requiring a different piece of software to control it.

With Realm in place, the movie showing agent has to include description of the services it needs in order to provide the movie service. Those will include: a VCR service, a display service and a connection between the two. Those three requests should be put in a single request bundle named, for example, “movie needs”. Now, the agent has to do the following:

- make a request passing the name of the request bundle to be activated (here “movie needs”) – it will get all three services requested in the bunch or nothing,
- ask the connection to set itself up – this will set the multiplexer inputs and outputs correctly and switch the projector to video input turning it on first,
- ask the VCR to play the movie.

1.6.2 Short Text Message

Interactions in the Intelligent Room are communications intensive. The Room often needs to communicate short messages to the user. At the moment it is up the applications to decide how to communicate with the user. At the same time, applications have no knowledge of other applications that might also be trying to get user’s attention. Most agents choose to communicate with the user via speech out interface. Yet, several other agents provide Short Text Output service, although they are rarely used. For instance, controlled computer display agent can display short text messages. The agent controlling the LED display can place a short message on the LED display above the projected displays. The speech generator has already been mentioned. In extreme cases, the pager service or the printer service could be used to get the message to the user.

In the pre-Realm Room, most applications defaulted to speech out agent. This was inconvenient at times, for example if a meeting was going on in the Room or if the user was on the phone. With Realm in place, the Room can exert influence over what particular agents should provide what services. Realizing that the user is on the phone, Room concludes that all sound producing activity should be kept to the minimum. Hence it turns down the volume on the CD

player and it also inspects all the service requests arriving at Realm. Whenever several services match a request, the Room can reduce the likelihood of sound intensive services being used. Thus LED display or computer display will be preferred over the speech out in such situations.

1.7 Concepts and vocabulary

For clarity, I present here all the special vocabulary used in this thesis. All of the terms listed below I also explained in other parts of the thesis. I present and explain them together to enable quick reference.

Smart space, intelligent environment – a space, such as a room, house or an office, augmented with computational and communicational devices. A smart space should be aware of its inhabitants and should be designed to explicitly interact with them.

The Intelligent Room, the Room – a smart space developed at the Artificial Intelligence Laboratory at MIT.

Metaglue – an agent language based on Java; Metaglue is the software foundation of the Intelligent Room project.

Agent – a basic software unit in Metaglue.

JESS – Java Expert System Shell [Fri00], a rule-based system written in Java; syntactically based on CLIPS.

MESS – Metaglue Expert System Shell, an extension of JESS that has built-in primitives for communicating with Metaglue; MESS is used to capture a lot of "common sense" knowledge in the Intelligent Room.

Realm – a resource management system built for the Intelligent Room. It's a multi-component system: its reasoning engine is written in MESS, the constraint satisfaction engine is written in Java and the Metaglue interface is written as a Metaglue agent.

1.8 Organization of this thesis

I begin by defining the design requirement for a resource manager for an intelligent environment. I present the experiences and insights gained in working with the Intelligent Room and show how they affected the definition of those requirements. In the next chapter I present the design overview of Realm – a particular instance of a resource manager for a smart space. The fourth chapter contains some implementation details of Realm. Next chapter provides some more information about the inner workings of Realm by walking through some examples. Then I compare Realm to other related systems. Finally, I evaluate Realm and conclude with by laying down the contributions of this thesis.

2 The Problem

In this chapter I derive a set of design requirements for a resource management system for an intelligent space. I base my derivation on my experience of working with and designing the Intelligent Room's software infrastructure. I begin by reiterating the main assumptions mentioned in the Introduction. I then present more detailed requirements derived from my experiences with the Intelligent Room. I conclude by presenting the full set of requirements for a resource manager to fulfill. Those requirements will be the basis for designing Realm and the measure against which Realm will be judged later.

2.1 Initial requirements

Below I present the main requirements for a resource management system.

- *Describe resources in terms of abstract services they provide*

  This will provide a powerful layer of abstraction that will make applications less dependant on particular devices and software.

- *Allow the host system to contribute to the process of choosing services for requests*

  Which resource is the best provider of a given service depends not only on the needs of a particular requestor but also on the current context in the space (as in the "short text output" example in section 1.6.2)

- *Arbitrate among all the requestors needing services*

  This is a basic requirement for any resource management system. Here we particularly want to acknowledge the fact that many of the services provided by devices are not point-like in time (assumption the designers of OAA seem to have made [Mar99]). Instead, many services will be assigned to requestors for extended periods of time.

2.2 Resource Management and the Intelligent Room

Below I present some likely interactions observed in the Room and some aspects of Metaglue that shed some light on what may be required of a resource management system in a smart environment.

?? *Many resource intensive interactions require services with little state*

For example, movie showing agent uses a display. There is little or no state maintained by an agent controlling the display. The consequence of this fact is very important to the design. It means that if necessary, previously assigned service can be replaced with another service without too much cost connected with recreating the state. Consequently, it should be possible to resort to reallocating services if a new request cannot be satisfied otherwise. There will, of course, be situations where reallocating a new service to a request would cause a lot of disruption – requestors should be able to specify how well they can tolerate reallocations.

?? *Information about most services is known at startup time.*

This an important feature that distinguishes Metaglue from many other systems like Jini [Arn99], OAA [Mar99] or Hive [Min99]. In Metaglue agents (and thus services they provide) are started on demand and the underlying system explicitly supports it. Metaglue offers a `reliesOn` primitive that allows one agent get a handle to another agent, starting it if necessary. It is thus implied that agents have some prior knowledge of each what other agents they can expect in the system. In other systems agents are not available until they are started. On one hand it provides the system with greater flexibility because agents can be added and removed on the fly and the system makes no assumptions about them. On the other hand, the system will not function properly unless all of its components are somehow started first. I will argue that in complex environments with a large number of applications and services it would undesirable to have all components running all the time. Thus the moment a permanent component is added to the system, it should register itself with the resource manager to let it know that it is available. Any resource manager should, of course, support dynamic addition and removal of available services. But it should also know about and be able to start inactive services.

?? *Agents in the Room may have to run on the same computer as a particular piece of hardware or software.*

This is true for all systems – software components controlling particular pieces of hardware have to reside on computers to which this hardware is attached. This is a trivial issue in systems

where software components are started by hand or in some contrived manner. It becomes more difficult in systems like Metaglue where an agent needing to control a modem gets started by another agent. The modem controlling agent has to make sure that it ends up running on the right machine regardless of what computer the other agent is running on. Metaglue has a `tiedTo` primitive that agents can use during construction time to specify the name of the machine they have to run on. Because I argue for on-demand startup of agents, I also argue that a resource manager should know about such startup requirement of agents as hardware or software components they need to be tied to. Of course with a resource manager in place, agents will rarely request to be tied to particular machines – instead they will request to be put in a UNIX-like environment or on a machine with a voice modem that is connected to a phone line.

This also means that a resource manager should know not only about services provided by agents but also about hardware and low level software (such as a modem or an operating system).

Some system may use agent mobility (rather than the "tied to" concept) to deal with the problem of agents being started on a different machine from the one they should run on.

?? *New kinds of devices and interactions have to be accommodated by the Room on regular basis*

Given that smart spaces are still in a research stage, new kinds of ideas, devices and interactions have to be incorporated on regular basis. Thus any vital component of software infrastructure controlling such a space has to be easily extensible. In the case of resource management it means that it should be possible to easily extend the representation so that new kinds of devices can be described and so that new kinds of choice strategies can be incorporated.

?? *Physical connections between devices may go through a number of multiplexers.*

As a consequence of this fact, pre-Realm agents in the Intelligent Room had to not only find the right pieces of software to control right devices, but they also needed to find right pieces of software to control multiplexers to set up all the necessary connections. For example, the movie showing agent had to find a VCR controller, a projector controller and a controller for the multiplexer in between. It also had to know which input on the multiplexer corresponded to the

VCR and which output to the projector. In addition, the movie showing agent had to make sure that the projector input was set to video and not computer. Realizing that this information should not be present explicitly in the movie showing agent, I decided to incorporate it in Realm. Thus Realm now stores all the information about connections among devices and about all the input-output settings on all devices with multiple inputs or outputs.

2.3 Design Requirements

In conclusion of this chapter I present a list of high-level requirements for a resource management system for an intelligent environment. For each requirement I provide a slogan (for easier reference) and a short description.

1. **Services not devices**

   Describe resources in terms of services they provide;

2. **Dynamic knowledge cooperation**

   Allow the host system and the agents to dynamically affect candidate service selection process.

3. **On-demand agent startup**

   Support and take advantage of on-demand agent startup.

4. **Machine requirements**

   Be able to describe such requirements as living on the same machine as another service (rendered in hardware or software);

5. **Extensibility**

   Representation should be extensible.

6. **Local to global approach**

   Strive for a local solution resorting to global changes only if necessary; but consider reallocations and withdrawals when other solutions fail.

7. **Resource conflict reduction**

   Reduce resource conflicts; take into account the fact that services are owned over time.

8. **Hide connections**

   Hide connections from the agents – forcing the agent to know about all the connections among various components would contradict the requirement that agents should see services rather than devices.

3 Design Overview

This chapter provides a design overview of Realm – a resource management system for the Intelligent Room. I will first outline the major components of the system and then proceed with detailed descriptions of how those components are structured and how they function and how they interact with one another.

3.1 Main design decisions

3.1.1 Structure

In order to put as few constraints on the reasoning process within Realm, I decided to keep all the knowledge Realm acquires about the all the agents in a single place. Another reason for keeping the knowledge separate from the actual agents is so that the knowledge can be expressed in a way that is best for that purpose while agents can be written in a language that is most appropriate for them. Finally, separating knowledge from agents allows the system to reason about agents that are not alive yet.

Realm functions within Metaglue as a single agent and it runs in a single place. It is a centralized system. Metaglue, however, is a distributed system. Making Realm an integral part of Metaglue may severely reduce the robustness of the whole system. At this stage distributing Realm is out of question for two reasons: first, the complexity of the possible solution makes it a separate research topic; second, the resulting system would be very communication intensive and, likely, significantly slower. Because Realm is to control a real-time system, it has to run quickly and efficiently.

Realm can be still robust and mostly crash-proof thanks to two features of Metaglue:

1. persistent storage – Metaglue can store any kind of data on behalf of agents and keep the data even in case of agent or Metaglue crash;

2. automatic restarting of agents – if an agent in Metaglue crashes and any other agent tries to communicate with that dead agent, Metaglue will restart the dead agent. If necessary, it will even restart it on a different computer.

Hence, a reasonable level of robustness can be achieved if Realm stores its state frequently enough. Then, in case of a crash, it will automatically be restarted whenever any of the agents needs it and it will restore its state from Metaglue's persistent storage.

Other possible structures could include direct negotiation or environment marking [Fer99, Huh99] but all of them would require all participating parties to be present. This would violate the "on-demand startup" design requirement. Having a single knowledge repository allows the system to reason about all components regardless of whether they are active or not.

3.1.2 Reactive

It is sufficient for Realm to reason only about the current situation. If a request is denied, the requestor should wait and try again later or try to issue a more modest request. I argue that the state of the system changes continuously and without warning. It would be impossible to give requestor guarantees as to when they would be awarded desired services because services could suddenly become unavailable or a higher priority requests may arrive thus upsetting the order in the queue. If there can be no guarantees then maintaining a queue of requests would not benefit the system significantly.

3.1.3 Choice of tools

Given that Metaglue is an extension of Java programming language, and that Realm has to interact with Metaglue, it was desirable to use Java as a base for Realm. Java, however, is not well suited for expressing knowledge hence Java Expert System Shell (JESS) was chosen for the knowledge part of the project. JESS is a rule based language interpreter written in Java. It was a good candidate for the task because it provided adequate tools for expressing knowledge and good connectivity with programs written in Java. Integration of Java into the system was almost seamless.

A tool derived from JESS, called MESS (Metaglue Expert System Shell), was created and is now a part of Metaglue and some of Room's knowledge infrastructure is already written in MESS. Use of MESS for the knowledge part of Realm also allows for easy integration with Room's knowledge base.

The constraint-satisfaction engine is based on the JSolver[1] library [Chun99]. JSolver is a generic constraint-satisfaction engine that tries to find a solution by depth-first search with local constraint propagation after every assignment of value to variable. JSolver is easily extendible making it a convenient base for this part of Realm.

3.2 Quick look at Realm

The system is composed of three major elements:

- **The knowledge-based part**

  This is where the knowledge about all agents, services and requests is stored. This includes static, descriptive, knowledge of all the services and their corresponding needs, as well as the rules supplied by agents or the system whose role is to modify the default behavior of the system. Also, the Room's common-sense system can influence the behavior of the Resource Manager through the knowledge-based part. The Room's common-sense system may, for example, know that because of the sunlight glare, during the day users usually prefer to use the right projector display.

  When a request is made by an agent, the knowledge-based part of the system generates plausible candidates for all requests and sets up all the necessary constraints, which are later solved by the constraint satisfaction engine.

- **The constraint satisfaction engine**

  As said above, the constraint satisfaction engine takes the requests, candidate services and constraints from the knowledge-based part of the system and tries to find the best global

---

[1] JSolver version 2.0.4 distributed by AOTL (Advanced Object Technologies Ltd.); http://www.aotl.com

solution. This global approach guarantees that a configuration will be found if one exists. It also implies that sometimes previously satisfied requests may be reallocated to new services. Too much reallocation is clearly undesirable (e.g. the display for the movie showing application should not change unless really necessary) hence the engine uses heuristics minimizing reallocations (or, to be more specific, the engine tries to minimize the cost of the assignment and each reallocation comes at a cost; more about it later). The engine will also make sure that needs of services allocated to satisfy requests can also be satisfied. To take an example, there would be no point in allocating the pager service to send a message to the user if there was no modem available in the system.

✍ **Interface with the Metaglue agent system**

This component allows Metaglue agents to make requests an query and modify information about themselves. It also allows Realm to communicate its solutions back to the host system. The interface is composed mostly of two elements: the `RealmAgent`, which is the representative of Realm to Metaglue. The second component is the `ManagedAgent` class, which is used as a base for each Metaglue agent that wishes to provide or request services through Realm.

3.3 The Knowledge-Based Part

The knowledge part of the system stores the information about all agents, their needs and services they provide. It also enables various entities to reason about available resources and influence their allocation. During the process of generating candidates for requests, the candidates may be reviewed by a number of parties, namely the requestor, the service provider and the system. While scrutinizing a candidate, the parties can alter the match level of a candidate. The match level indicates how well a service corresponds to the needs expressed in the request.

It is not necessary for all those parties to scrutinize all candidates but they can do so if they so desire. Requestors may need to go over the candidates if there was some piece of logic they could not express in the syntax of a request. Services may want to review themselves as candidates to

some requests if they have reasons to believe that they may be particularly good or bad at providing the requested service or they need to check if the new request will interfere with the requests they are already serving. We can imagine that the provider of a display service, who is already serving two other requestors, may need to review its candidacy in order to try to predict if the new requestor would demand so much space that it would obstruct the objects displayed by the previous requestors. Finally, the system may want to inspect the candidates if it has some special knowledge about the state of the space or the interactions within it. It may, for example, know that the left projector is a little bit brighter than the right one and thus it would always bump its score a little.

The agents can participate in the reasoning process by submitting custom rules when they submit the description of themselves. Thus it is not actual agent code doing the reasoning but the special knowledge-oriented code that gets shipped to the resource manager.

The actual agents remain in contact with the resource manager and their own description contained within Realm. At any time they can (through Realm's Metaglue interface) query and modify values of any of their properties or submit requests.

3.3.1 Meta control

As said before, one of the goals of Realm is extensibility and making it possible for other modules in the Room to interact with Realm. Realm allows other software components to interact with it at any stage of the reasoning process. In order to clearly demarcate individual stages of the reasoning process, Realm uses the notion of context. Realm can control what kind of processes happen when by making context specification one of the predicates of each rule.

Within each context, however, not all components are always required to act. Realm uses another meta control tool – commands – to control who needs to run when. Commands are implemented as facts and rules can be guided by commands if they have command specification as one of their predicates.

To be clear, both context and commands are only guidelines for the components. It is possible, though not encouraged, for various components to act on the system without using the guidance provided by context and commands.

Both context and commands are explained in more detail below.

3.3.1.1 Context management

Structured and organized reasoning within Realm is possible due to the use of strict context management. Sets of rules get activated only in certain contexts and thus order is imposed. The system allows for nested contexts, such as `newAgent.generateSolution.engine.create`, where dots are used as context level separators. Rules can match based on the full global context (`newAgent.generateSolution.engine.create`), subset of global context (`newAgent.generateSolution` or just `newAgent`) or on subset of local context (`engine.create` or just `create`). In a way, the name of the most local context could be compared to the name of a procedure and the full context to the scope of execution. By giving rules access not only to the local context but also to the global context, execution of rules can proceed differently depending on how a particular point in the execution of a system has been reached. For example, local context `generateCandidates` could be reached when information about a new agent is loaded and the system pre-computes the candidates for all of the new agent's requests. The same local context can also be reached when an on-the-fly request is created and submitted. In the first case, the system will create candidates for the requests within the newly registered agent and will return to the idle state. In the second case, the system will generate candidates and then will proceed to run the constraint-satisfaction engine and return a solution.

The use of context is especially useful when allowing agents to provide custom rules with their descriptions. Appropriately specified context in the rule ensures that the rule becomes relevant only when it is supposed to.

3.3.1.2 Commands

Commands allow the system to regulate which agents' rules should be active during a given context. While context regulates the kinds of actions that should be taken, commands regulate which agents should participate and which should not. Commands are objects with fields: *subject*, *action*, *object* and *details*. These fields specify who should do what and to whom, and provide some extra context information.

For example, within the local context `generateCandidates` a following command can be issued:

```
subject:    projector request of the movie showing agent,
action:     generate candidates,
object:     all
details:    none
```

This command calls on the projector request of the movie showing agent to find potential candidates to fill that request.

3.3.2 Knowledge expressed

The knowledge about each Realm-enabled agent in the system is represented as a nested structure. There is an object representing an agent with service family objects linked to it. Service families are composed of individual services. Those services, in turn have properties and needs. Needs are represented as request bunches. Each request bunch, of course, has a number of requests linked to it and each request has its properties. Request bunches can also exist independently of services. The conceptual structure of those objects is illustrated in Figure 3-1. More detailed information on what various components of the representation means is presented below.

On top of the static knowledge, each agent can also provide rules and functions that will do any special-case handling on its behalf. The rules should use appropriate context as one of its predicates to ensure that they get fired at right time of execution.

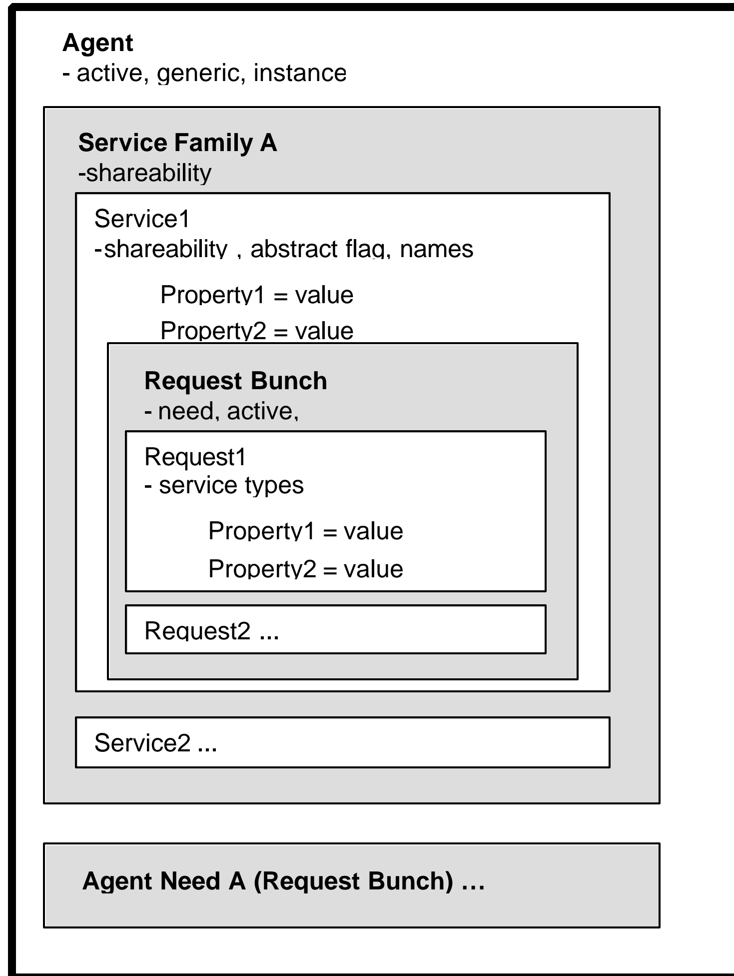

Figure 3-1 Hierarchy of elements in Realm's knowledge base.

Agents – in the knowledge part an agent object represents an entity capable of providing or requesting services. Those entities can correspond to actual agents or to abstract packages. Abstract packages cannot have any needs – they can only contain services. Those abstract "agents" are useful for representing services provided by actual devices or software that are external to Metaglue. Examples of such services may include a computer, an operating system or a physical projector. In addition, each agent object in the knowledge part can be marked with any of the following flags:

- *Active* – true by default; set to false only when for some reason the underlying agent becomes unavailable and should not be considered.
- *Generic* – some descriptions will refer to entities that can have multiple instances, e.g. a browser. In those cases the generic flag will be set to true. In cases where the description pertains to only one possible instance the generic flag should be set to false.

- *Instance* – indicates whether the description pertains to an instance of an agent that is actually running or one that could potentially be brought to life.

Agent needs – expressed as a *request bunch* (see below), lists all the needs that have to be satisfied during agent's startup. These needs have to be satisfied regardless of whether or not the agent is providing any service.

Service Families – Services have to belong to service families. The main reason for the service families to exist is so that shareability constraints could be put on groups of services and not just on individual services. This is especially useful if a device can provide the same service at several grades of quality. For example, a computer display could provide display service at three different resolutions. If we represent them as three different services without tying them into the same family, various other agents might request each of the three services at the same time while only one of them can be used at any given time. Hence imposing shareability constraints on the whole collection as well as on the individual services will solve the problem. The problem could also be solved by introducing an internal service that all three grades of display service would need. The shareability constraint on the commonly needed service would impose a limit of usage on the group of displays. The service family, however, appears to be an easier solution. Also, in the future, service families could hold requests for things that are common to all services within the family.

Services – for each service, Realm needs such information as:

- *Shareability constraints*, i.e. the number of requests that the service can serve concurrently; this information may be updated dynamically as the conditions change.
- *Abstract vs. non-abstract* – some services may be available even without running an agent – such services should be marked as abstract. Services provided by abstract "agents" should be declared as abstract.
- *Name or names* – names of services are expressed using Java interface name syntax. A single service can have more than one name.

Also, each service can have properties associated with it that contain information specific to this specific rendition of a service. Services may also have needs represented as *request bunches* (see below). Associating a request bunch with a service indicates that the requests contained in the bunch will be required by the service in order to function properly.

Request Bunches – requests are put in bunches to allow agents to make all-or-none type of requests. In most situations agents need combinations of other services in order to provide their service. For example a movie showing service needs both a source of a movie (a VCR or a computer capable of streaming video) and a display (a TV or a projector). For each request bunch Realm specifies how badly it is needed (or would be needed if it were requested).

Requests – for every request, Realm has flags that indicate if it is currently active and if it is a new request, i.e. one that has just been submitted. Active requests are those that have been satisfied already and are currently in use. New requests are yet to be satisfied. Each request also specifies the names of functions to be used for matching the request against a service and for calculating the utility. Request description also includes a flag that indicates if the requesting agent is tied to (that is has to be on the same machine as) the service it is requesting.

Requests may also have properties associated with them. Those properties will be matched against the properties present in the descriptions of candidate services.

Properties – Realm provides an abstract template for properties. Properties can be of any kind depending on what kind of information needs to be conveyed about a service. The property template includes two main fields: the name of the property and, in case of request properties, the name of a function that should be used for matching. Properties can have one or more values or can be expressed in any other manner that is appropriate. Realm provides a few standard property types and functions for matching them: single value property (for both numeric and symbolic values), multiple value property (also fir any kind of values) and range properties (only for numeric values).

Connections – as explained before, Realm needs to know about physical connections used by the devices in the system. It needs to know what devices are at the end of each connection and what inputs/outputs each connection is attached to. Connections can also be created on the fly.

Special agents – some agents are treated specially within Realm. Most of them are abstract agents representing physical resources such as computers or other physical devices that can be used, explicitly or implicitly, by the system. The only other kind of special agents is the Metaglue agent representing the substrate necessary for any other agent to exist. No other agent can run on a computer unless there is a Metaglue agent running there already.

3.3.3 Ontology

Realm relies on all agents to use the same interface names for names of services and the same property names for the semantically equivalent attributes. The problem of ensuring that all agents use the same language for describing their needs and services is called the ontology problem. Realm imposes no ontology of its own. Hence it must rely on ontologies constructed during the incremental development of a body of agents.

Leaving creation of the ontology to the future developers leaves these programmers with more freedom at the expense of guidance. Producing a consistent and complete ontology for expressing the names of all services available in intelligent spaces is a challenging task and could be a subject of a separate research project.

3.3.4 Reasoning

Realm by itself does not do much reasoning. It does, however, provide means for other knowledge-based components of the Room to reason about what it knows and to influence its decisions. The strict meta control imposed internally on Realm's components allows other systems to add themselves to the pipeline at any stage of reasoning. It is expected that normally other knowledge-based systems will try to affect Realm during the candidate generation process. They can, however, query and reason about the state of the services at any time

3.4 The Constraint Satisfaction Engine

This part of the system is based on the JSolver library. As explained before, JSolver is a generic constraint-satisfaction engine that tries to find a solution by depth-first search with local constraint propagation after every assignment of value to variable.

3.4.1 Constraints

Cost constraint – the cost of obtaining a new resource cannot exceed the utility of that resource; more about cost and utility in one of the following sections.

TiedTo constraints – some services have to be on the same computer or on the same virtual machine as another service; it is actually a very important concept overlooked by most other systems. In other systems (e.g. Hive) it is assumed that agents controlling certain hardware devices first come to life on an appropriate computer and then announce themselves to the system. The system, however, has no way of starting them in the right place because it does now know what the right place for them is. REALM, on the other hand, has enough knowledge about the hardware available in the system and about the agents and their services to start services on right machines.

The same owner constraint – services owned by the same owner have to reside on the same virtual machine hence if the services provided by this owner are tied to other services, the resource manager has to make sure that all of them end up together on the same VM;

Activate if assigned – if a service gets assigned to a request, the system has to make sure that all of the services needed for that newly assigned service are available; hence the newly assigned service gets activated and so do its needs; the initial assignment is not complete until all of the implied requests are not satisfied. For example, the answering machine service needs a phone service but the phone service, in turn, needs a modem.

Service shareability constraint – some services can be used by one requestor at a time, others can be shared by a number of users and yet others have to limit on how many users use them at a time. The system has to make sure that no service gets overused. At the moment, the shareability constraint gets set once for each request and cannot be changed during the run of the

constraint-satisfaction engine. Such ability would be potentially useful because some users take larger share of a resource than others, e.g. applications sharing the screen space. At the moment a display service has to guess how many users it can take at a time and cannot readjust the number on the basis of how much space particular applications need.

Service family shareability constraint – we can also impose a limit on a collection of services, independently of the limits imposed on each of those individual services. This is especially useful if the bunch contains several grades of the same service, for example two grades of a speech out service: one with the ability to change voices and intonations (if the Laureate server is running) and the other without those extra capabilities. In both cases, only one user can be using either service at a time hence it makes sense to put a limit of one on the entire bunch thus making sure that only one of those services is assigned at a time.

3.4.2 Heuristics

Assign new requests first – new requests are satisfied first, sometimes at the expense of the previously satisfied ones. In case of a tie, the newer request wins.

Then assign old things – when solving the global service allocation problem, try to assign old services to the already satisfied requests.

Use machines where agent system is already running – when assigning virtual machines to agents, give preference to those that have already been started. This heuristic makes sense because the system knows not only about the services that are already available but also about those that can be made available, and this includes virtual machines.

3.5 Cost and Utility

When resources are scarce, part of the arbitration process is deciding which requests are more important. This could be done with self-assigned priorities or economic models may be involved. In Realm self-assigned need levels are used in conjunction with the concept of utility of a service to the requestor and the cost to others. This is a very simple and arbitrary scheme. It could

easily be replaced by a different system should there be a need for that. This simple model is sufficient for the current implementation of Realm because of the assumption of cooperation among the developers of various software components. More complicated models would be required if code we used could not be trusted.

The basic assumption of this schema is that, given a request, each candidate service has some utility to the requestor. Services that are already allocated to request are also useful to their current users. When a service is taken from its current user, the system as a whole incurs cost equal to the utility of that service to its ex-owner. Also when a currently allocated service is replaced with a different one cost is incurred. The arbiter has to make sure that whenever it awards a service to a new request, the cost of doing so should never exceed the utility of the awarded services to the new requestor.

Below I define and explain the all the concepts in more detail:

Need – indication of how badly the requestor needs to have the request fulfilled;

Match level – expressed as percentage, indicates how well a given service would satisfy a request; each requests can pick a function for calculating the match level. One of the standard functions can be picked or requestors can define their own.

Utility – function of match level and need; indicates how much the requestor would benefit from a particular service. Each request can specify what function they want to use for deriving utility. If product is used, utility becomes directly proportional to the match. By using other functions (square root or square), requests can attach high utility to even lower matches or they can dismiss everything with the exception of best matches. The requests, however, come in bunches. The utility derived from assigning services to individual requests in the bunch, is equal to the smallest utility assigned to any request in the bunch.

$$u_{bunch} = \min\{ u(r_i, s_j); r_i \in bunch\}$$

Equation 3-1

$$u(r,s) \; ? \; f_u(need_r, match(r,s))$$

Equation 3-2

Where, *u(r,s)* is the utility of the service *s* to request *r*; $need_r$ is the need of request *r*, *match(r,s)* is the match level between request *r* and service *s*; f_u is the utility function, u_{bunch} is the utility of the bunch.

Cost – in cases where allocation of a service to a new request causes older requests to be deprived of a service or reallocated a different service, cost is incurred. If an older request is deprived of a service, the cost of that action is the utility of that service to that request. If the request is reallocated a new service, the cost is determined by the owner of the request that just got reallocated and can range from nothing to the utility of that service to that request. The total cost of satisfying a new request, is the sum of all individual costs incurred in the process. The total cost cannot exceed the utility.

$$c_w(r_i, s_j) \; ? \; u_{bunch}; where \; ?r_i \; ? \; bunch?$$

Equation 3-3

In the above equation c_w stands for cost of withdrawing service s_j from request r_i.

$$c_r(r_i, s_j \; ? \; s_k) \; ? \; p_{r_i} \; ? \; \max\{0, u(r_i, s_j) \; ? \; u(r_i, s_k)\}$$

Equation 3-4

In the above equation $c_r(r_i, s_j ? s_k)$ stands for the cost of reallocating the request r_i from service s_j to service s_k; p_{ri} is the fixed penalty for reallocating request r_i.

3.5.1 Other approaches to cost and utility

Market-based approach – requires electronic cash and a banking system [Bre97]; generally used when agents come from a variety of sources or act on behalf of a number of different entities that cannot agree on "nice guy" type of cooperation.

3.6 Connecting with Metaglue[2]

In order to be useful, Realm needs to be integrated with its host agent system. In this section I will describe how Realm integrates with Metaglue. Because Metaglue is written as an extension of the Java language, all lower level references will be using Java vocabulary.

There are several things that need to be taken into account when designing the Metaglue-Realm interface. Below I present an overview of this interface conceptually dividing it into two parts: one that allows the Metaglue agents to communicate with Realm and the other that allows Realm to controlled the Metaglue agents that agree to be managed by it. After that I present the infrastructure for handling the connections.

3.6.1 Realm Agent – making Realm accessible to Metaglue

Realm agent, the Metaglue front end of Realm allows agents make requests, query and modify descriptions of services they provide and release services when no longer necessary.

Making requests – because all the knowledge about services and requests provided by an agent is represented on the knowledge-based side, all an agent needs to do in order to make a request is to provide the name of request bunch to be activated and executed. Realm agent then passes the request to the knowledge-based side and returns the list of newly assigned services to requests within the activated bundle, provided that the request could be satisfied. Otherwise Realm agent returns a failure notification.

---

[2] Other members of the Intelligent Room project (namely Luke Weisman and Stephen Peters) have contributed to the research presented in this section. Resource management has been an issue for a long time in the Room and several other attempts had been made at solving the problem before the work on Realm started. Also, new applications for the Room were

Querying and changing service descriptions – when an agent wishes to change a description of one of its services, Realm agent will pass that information to Realm where all necessary recomputation will take place. Any agent can query any other agent's service properties. Realm agent responds to the queries by looking up the values of appropriate properties within Realm.

3.6.2 Managed Agent – making Metaglue accessible to Realm

`ManagedAgent` class is the base class for all agents that want to participate in Realm-mediated resource allocation. The `ManagedAgent` class will automate all processes common to all or most agents using Realm's services. I will refer to such agents as *managed agents*. Below I list and explain a number of actions in which `ManagedAgent` class will be involved.

Startup - As explained in the previous chapter, Metaglue agents can request to be placed on particular machines or together with some other agents. Agents may need to run on particular machines in order to be able to control particular pieces of hardware. With Realm in place, agents should be able to ask to be tied to particular pieces of hardware (described as a service provided by a particular machine) rather than to particular computers. For example, if there are several computers with modems connected to a phone line, the agent should be able to run on any of those computers.

Thus whenever a managed agent starts, it should ask Realm what computer or, more precisely, what virtual machine it should run on. Agents that need to be tied to particular pieces of software and hardware need to contact Realm to find out where their needs can be satisfied. Agents that do not have such needs also need to contact Realm to find out where they should run just in case another agent needs them. Mechanisms for handling this are built into the parent class of all managed agents thus requiring minimal effort on the part of programmers building new managed agents. When a managed agent is started, it contacts Realm to find out where it should live and then uses Metaglue's `tiedTo` primitive to make sure it is started in the right place.

---

written in a Realm-compliant style even before Realm became available and during this process many requirements for the Realm-Metaglue interactions have been clarified.

Representing services within the agent – in Metaglue, when an agent requests a handle to another agent, it gets back a proxy object that mediates all method calls. That proxy object, in turn, internally stores a handle to the actual agent. This representation is very convenient for Realm. It means that if an agent requests a service, it can be given back a proxy object representing a given service. Realm can than invisibly interchange or remove the handle to the actual agent providing the service.

Revoking a service bunch (`revoke` *primitive)* – services are not revoked individually but in a bunch. The purpose of putting several requests in a bunch is to say that this is an all-or-nothing request. Hence revoking one service from a bunch should be equivalent to revoking the whole bunch. When a service is revoked, Realm removes prevents the proxy object from relying methods calls from the requesting agent to the agent providing the service. The requesting agent can be notified about the revocation of any of its services if it extends the `serviceRevoked` method from the underlying `ManagedAgent` class.

Replacing a service (`replace` *primitive)* – while services cannot be revoked individually, they can be individually replaced. Realm does it by replacing the actual agent handler inside the proxy object with the handler to a new service provider. It then calls the `serivceReplaced` method in the `ManagedAgent`. An agent can be notified of the replacement by overwriting the `serviceReplaced` method.

3.6.3 Handling Nested Requests

As explained before, Realm takes a cautious approach to allocating services: before it allocates one, it makes sure that it can satisfy its needs. I will refer to requests activated in the process of satisfying other request as *nested requests*. When a requests is processed by the constraint-satisfaction engine, the engine creates a model of the world, in which all new requests, including the nested ones, are satisfied. This creates a slight complication because the nested requests had not been yet formally issued. They will be issued when the agents providing services allocated to first level requests are asked to deliver their services. At this point the nested requests

will formally be issued. Realm stores the results it computed earlier and uses them if possible. However, if conditions in the Room change, the agent may modify its requests somewhat before issuing it. In that case, Realm will have to recomputed the solution. If the change to the request was slight, only having to do with the ordering of candidate services, the service will still be satisfied because during previous run Realm has ensured that *some* service can be provided for that nested request. Problems may arise if agents change their requests significantly depending on conditions. In such situations, Realm will not be able to guarantee a solution.

Also, if a new request is activated before a nested request is activated, the new request may claim one of the services required by a nested request if the need level of the new request is sufficiently high.

3.6.4 Connections

A separate set of infrastructure is needed for handling connections. As explained above, connections are represented in the system in terms of the devices attached to the two ends and the names of inputs and outputs the connections are attached to.

There is a variety of devices that can choose from a number of inputs or can send signal to different outputs. A multiplexer is, of course, one of them. But so is a projector that can choose between a number of RGB inputs and a video input. It should be possible to use a standard language for addressing all devices with input/output selection capabilities. Hence, in Realm, controllers of all such devices provide the "connectible" service. Connectible devices can be asked to listen to particular inputs, send signal to particular outputs or create connections between an input and an output, depending on whether the device is a signal provider, recipient or an intermediate multiplexing device.

Realm also provides a connection-making agent that can create connections given the endpoints. It draws on the knowledge stored in the knowledge part of Realm.

4 Implementation

In this chapter I present some information about the implementation of Realm. I outline organization of main modules, show examples of some of the code and evaluate the complexity of the solution.

4.1 The Knowledge Part

The knowledge part is just a framework for storing and manipulating information specific to the host system. Without any information about Metaglue and the Intelligent Room, the knowledge part contains mostly:

?? Templates for asserting information about agents, services and requests.

?? Infrastructure for meta control: context management tools, command tools;

?? Meta control information for Realm, i.e. information on what to do when (see Figure 4-1)

?? Request matching infrastructure – tools that compare requests to available services and generate candidates – requests specify what particular functions should be used for matching.

?? Standard functions for calculating utility

Most of the code in this part of the system is composed of fact templates and functions. The few rules present in the core of the system perform meta control tasks.

4.2 The Constraint Satisfaction Engine

Bulk of the code went into the constraint-satisfaction engine.

The JSolver library, which forms the core of Realm's constraint-satisfaction engine, deals with integer constraints. Hence available services are represented as distinct integer values.

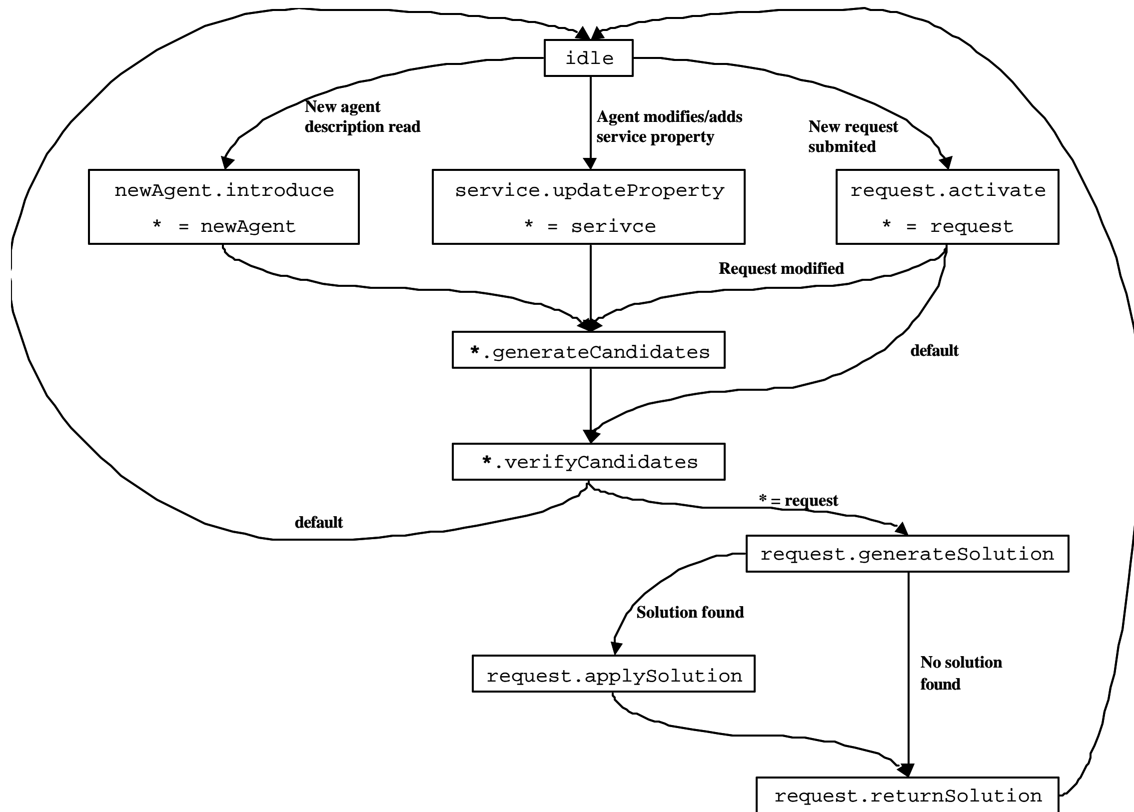

Figure 4-1 Outline of Realm's meta control flow.

Variables represent requests. Assigning a value to a variable is analogous to assigning a service to a request.

Available virtual machines are also represented as integers and each agent has a variable that represents the virtual machine this agent should end up running on. This makes the process complicated because I am using a single constraint satisfaction engine to represent two different concepts: service to request assignments and agent to virtual machine assignments.

At the setup time (engine is re-setup before each run), the ids of candidate services are put in the domains of variables representing requests. Analogously, ids of candidate virtual machines are put in the domains of variables representing virtual machines, on which particular agents should run. The id's used in the constraint satisfaction engine are different from those used internally in the knowledge part. For efficiency reasons, ids in the constraint satisfaction engine should come in a contiguous block.

By default all variables have service or virtual machine with id 0 in their domain. This represents a universal “dummy” service or virtual machine that is infinitely shareable and this is the first value that is tried by the engine for any variable. Thus inactive requests get quickly dealt with by being assigned “dummy” services and virtual machines. When a request gets activated, a “not equals 0” constraint gets imposed on the corresponding request variable thus ensuring that the request gets satisfied by a real service. The same is done for the virtual machine variable of the agent owning the request.

4.2.1 Constraints

?? Service shareability constraints are represented as cardinality constraints; if a service represented by an integer value *n* can only serve one request at a time, this information is represented in the engine by imposing a cardinality constraint of 1 on value *n*.

?? Service family shareability – these constraints are represented analogously to the service shareability constraints. If there is a limit on how many requests can be served by a family of services, a cardinality constraint is imposed on the set of values corresponding to the services in the family.

?? Cost constraint – every time a service gets assigned to a request, the total cost and utility are calculated. If the cost exceeds utility, the engine backtracks. Utility is always calculated in the most generous manner while cost is calculated in a very conservative manner ensuring that as the difference: *utility – cost* never decreases as the search progresses deeper and deeper.

?? Tied to constraint – some requests are marked as being tied to the services they get assigned to. If a value gets assigned to a variable representing such a request, an “equal” constraint gets imposed on the variables representing virtual machine assignments of the agent issuing the request and the agent providing the service. Thus an agent requesting a modem will be put on one of the virtual machines running on a computer that has a modem installed.

?? Activate if used constraint – as explained in previous chapter, if a service gets assigned to a request, the system has to make sure that all the needs of that request are satisfied. Hence, whenever a value is assigned to a request variable, the system checks if the service corresponding to that value has all its requests activated. If not, it activates them. Activating a request, as explained above, is equivalent to imposing the “not equals 0” constraint on the request variable and on the virtual machine variable of the owning agent.

4.2.2 Heuristics

JSolver allows programmers to write their own functions specifying in what order variables and values should be tried. This way I can instruct the engine to first try the “dummy” service or virtual machine for any request or virtual machine variable. I can also instruct the engine to first assign values to new request and then proceed with others. This implementations makes the engine deterministic.

4.3 Metaglue Integration

Metaglue integration was implemented just as described in the Design Overview chapter. The `RealmAgent` is a gateway for communicating with Realm. The `ManagedAgent` is a base class for all agents that want to request or offer services through Realm. Implementation of those components was fairly straightforward. Thanks to the way Metaglue passes agent handles, it was very easy to implement the reallocation feature – the requesting agent would have a pointer to a proxy object that stays the same but a pointer to the actual agent can be freely changed inside the proxy object and thus services can be replaced on the fly.

5 Realm at work

This chapter describes the workings of Realm by looking closely at various actions it performs at various stages of its lifetime. Below I present what happens at Realm's startup and then I show how a request for the "short text message" (see section 1.6.2) service gets handled.

5.1 System startup

Currently, Realm starts when any of the agents request its services (thanks to the on-demand agent startup feature of Metaglue). Upon startup, the first thing Realm does is scan the agent distribution tree looking for files containing descriptions of agents.

5.1.1 Reading in the info about available agents

Each file containing an agent description must contain, at least, the following:

- an assertion statement that informs Realm of the name of the agent described in the file,
- a rule triggered by the `introduce` command, which will assert all the descriptive information about the services the agent provides and about the needs of those services.

The file may also contain special matching functions for requests and properties, special functions for deriving utility, or rules for special handling of certain requests.

5.1.1.1 Asserting service info and needs

The moment a file is loaded and the statement with name of the new agent is asserted, the system issues introduce command addressed to the new agent. The rule containing all the descriptive information gets triggered and information about the agent, its services and needs becomes available to the system. Below is an example of a rule that asserts information about the phone agent:

```
(defrule phone-introduce
```

```
  (kcommand (verb "introduce")
          (subject "agentland.device.phone"))
  =>
  (newAgent (assert (agentInfo (name "agentland.device.phone"))))

  ;; setting up the service family
  (bind ?sbid1 (newBunchID))
  (assert (serviceFamily (name "Phone Family")
                  (owner "agentland.device.phone")
                  (id ?sbid1)))

  ;; declaring the phone service
  (bind ?sid (newServiceID))
  (assert (serviceInfo (name "Phone Service")
                   (owner ?sbid1)
                   (id ?sid)
                   (interfaces "agentland.device.phone")))

  ;; creating request bunch for the phone service
  (bind ?sbid1 (newBunchID))
  (assert (serviceNeed (name "needs for the phone")
                   (owner ?sid)
                   (id ?sbid1)))

  ;; requesting a modem for the phone service
  (assert (request (name "modem")
               (owner ?sbid1)
               (tiedTo TRUE)
               (interfaces "hardware.modem")))
)
```

This simple agent provides only one service – the phone service – and requires only one other service to run: a modem. The header of the rule makes the rule fire when the "`introduce`" command is issued with the subject equal to the formal name of this agent. In the body of this rule, first the information about the phone service gets asserted and then the information about the phone's needs. Because JESS does not allow for nested fact structures, all facts have to explicitly point to the higher-level facts that own them. Thus the `service` fact has an `owner` field that points to the `serviceFamily` fact, etc.

In this case the Java name of he agent is the same as the name of the service it provides. This is not required. In case of agents providing more than one service it is not even possible.

5.1.1.2 Generating candidates

Once the information about an agent is provided to the system, two things happen:

- the system asks all other agents to review the services offered by the newcomer

- the system asks the newcomer to look for services that will satisfy its requests.

In case of the movie showing agent, the system will ask all other agents if they need the movie showing service. If the answer is yes, the new agent's service will become a candidate for one of the other agents' request.

Then the system will ask the movie agent to look for candidates for its needs: the VCR service and the display service.

5.1.1.3 Matching and utility calculation

Every time candidates are to be generated for a request, the system looks at all possible services and does the following:

- checks if a service is of a kind that the request is asking for (Java interface names are used for naming services);
- if the answer is yes, all the properties specified in the request are matched against those specified by the service under consideration. Each request property specifies what method should be used to match it. All kinds of properties have default matching methods to make programming simple. In case of the display service request, the movie showing agent uses request properties specify the minimum resolution it needs, the actual size of the display and its color capabilities;
- after all properties are matched, the results of the property matches are combined to produce overall match; each request can specify what method should be used for doing so. A number of default methods are provided by Realm. The movie showing agent uses the minimum method, i.e. the overall match is as good as the lowest match among all properties.
- When the overall match is obtained, the requestor has to specify the utility of the candidate service. Utility is usually a function of the match. The default method makes utility directly proportional to the match. If an agent is desperate for any result, however, another function is used that assigns high utility to even less desirable

candidates. Depending on the situation, the movie showing agent will choose between these two functions.

5.1.1.4 Verifying candidates

After candidates are created, they are verified first by the owners of the candidate services and then by the system. As said before, neither the service owners nor the system are required to perform the verification but they can if they want to. More about verification will be said in the next section.

5.1.2 First request – "short text output" service (with interference)

5.1.2.1 Activating request bunch

When an agent needs a service, it uses the `request` method inherited from `ManagedAgent` to activate an appropriate request bunch within Realm. In this example, let us assume that the calendar agent needs to remind a user about an upcoming event. It needs to request the "short text output" service. It executes a call:

```
Hashtable services = request("contact user");
```

Where "contact user" is the name of a request bunch contained in the description of the calendar agent. The bunch contains only one request, namely one for the "short text output" service. The request method returns a hash table keyed by the names of the requests and contains pointers to agent proxy objects.

5.1.2.2 Verifying candidates again

When the request for a "short text output" service gets activated in Realm's knowledge base, the request gets reviewed again, this time with the current situation in the Room in mind. In this case the calendar agent does not need to perform any extra verification but the system does. Depending on what the user is doing, it affects what kinds of devices would be most likely to be used. As explained in the example in section 1.6.2, if the user is on the phone or engaged in a conversation, the Room's common sense engine will deduce that voice-intensive devices should be used only if necessary. It will thus decrease the utility of the speech generation service making it more likely that a quieter service will be awarded to the request.

5.1.2.3 Setup engine

Once the candidates are generated and verified, the constraint-satisfaction engine is reset and the knowledge about the current state of the world and current requests and corresponding services is translated into the data structure usable by the engine. The process is straightforward and quick despite the fact that a relatively large number of new objects has to be generated.

5.1.2.4 Run engine

When the engine is ready, it is ran. If there exists a solution, such that all constraints are satisfied, the engine returns the new state of the world. Otherwise it returns a failure.

5.1.2.5 Impose the solution on the system

If the engine returned a result, Realm needs to impose it on the world. If any of the requests serviced caused a reallocation, Realm first deals with this. Then it returns newly assigned services to the requests being serviced.

6 Comparison With Related Systems

In this chapter I represent several well-known related systems and show how Realm differs from them. Table 6-1 contains a summary of the information presented in this chapter.

6.1 Jini

Jini [Arn99, Edw99] is a framework for building resource management systems. It provides tools for resource discovery and resource description. It does not, however, provide any tools for actual resource management. Realm could well be used to do the resource management with Jini. The Jini could provide their communication infrastructure as well as some of its descriptive mechanisms and Realm could deal with allocating services to requests and with minimizing resource conflicts.

6.2 Open Agent Architecture (OAA)

OAA [Mar99] is an agent platform with a built-in system for describing and managing services offered by the agents. Its descriptive capabilities in some ways surpass those of Realm. Unfortunately, OAA does not address some issues that I have found to be crucial in administering services in a smart environment.

OAA uses a special language, ICL, based on PROLOG for description. It describes agents in terms of tasks they can perform. The difference between tasks and services is a basic tasks are indivisible and are treated as being point like in time. The requests are stated as goals. Often a goal will be composed of sub goals logically connected together. The strength of OAA comes from the facilitator's ability to decompose complex goals into more basic ones and then delegating them to individual service providing agents in correct order. The weakness of OAA comes from the implicit assumption that all basic tasks are indivisible and point-like in time. Thus OAA does not provide any built-in mechanisms for arbitrating among conflicting requests.

Such model is not adequate in an intelligent environment where many services have to be provided over time, e.g. display for the movie showing agent.

6.3 Hive

Hive [Min99] is an agent platform; a system for building applications by networking distributed resources together. One of the major design objectives of Hive was to make it "fully distributed." This precludes any kind of central repository of knowledge. Unlike in Realm-enhanced Metaglue, Hive agents become available to the system only after they were explicitly started externally to the system. Agents in Hive are self-describing and all decisions about who is going to perform what tasks are arrived through direct negotiation. Applications in Hive are created by explicitly connecting various components together. Thus resource conflicts are diminished because connections among agents are long-lived and pre-designed, contrary to the on-demand configurations created within Realm-enhanced Metaglue.

6.4 Resource Description Framework (RDF)

Resource Description Framework (RDF) [Las99], as the name implies, provides only means for expressing information about resources. It does not provide any tools for reasoning about those resources or brokering requests. RDF provides extensive means for expressing capabilities and properties of various resources. Unlike Realm, RDF does not provide explicit support for encoding needs of services. One of the main advantages of RDF is that it is, in principle, independent of encoding. One tested encoding is based on XML but others are theoretically possible thus making RDF platform independent.

	Discovery	Communication of needs and services (description)	Arbitration	Other
Realm	Service providers need to notify resource manager; requestors need to know how to find the manager; central registry; the manager can also know about agents that are available	Interface names and lists of properties; devices may offer many services; they can also offer the same services at different grades of quality	Full, service providers and service requestors can also influence arbiter's decisions	Can be influenced by other parts of the system.
Jini	IP multicast; fully automatic	Interface names and Attribute objects	The framework offers none – it is up to the designer of a particular implementation to provide an arbitration component if necessary	
OAA	Self-announcement	Uses ICL (based on PROLOG) for announcing capabilities and for making request – platform and language independent; requests are expressed in terms of goals (i.e. actions not services)	No concept of ownership of resources hence no resource conflicts hence no arbitration; however, good facilitation abilities include decomposition of complex goals	
Hive	Central registry or agents' own initiative	Class name + RDF to describe the details	None – when agents are arranged into applications, care has to be taken not to overuse any of the resources.	Users RDF
RDF	N/a	Rich, uses XML for encoding (other encodings possible)	N/a	

Table 6-1 Overview of various resource management and related systems.

7 Evaluation of Realm

The system is now in the final stage of implementation in the Intelligent Room. In this section I will show how Realm performed during tests with respect to the design requirements laid out in the second chapter. I will also evaluate how well Realm satisfied our expectations. Thus I will implicitly evaluate how adequate and complete the design requirements laid out in chapter two were.

7.1 Compliance with Requirements

1. **Services not devices**

   Realm's knowledge representation is in terms of services not devices or agents. Realm uses Java interface name syntax for naming all services.

2. **Dynamic knowledge cooperation**

   Realm has an extended support for allowing various parties to participate in the candidate selection process. The host system as well as the service provider and requestor can influence the decisions every time a request is made.

3. **On-demand agent startup**

   Ream can reason about agents and their services even if they had not been brought to life yet. One of the important contributions of Realm is that it does not activate a service unless it has a way of satisfying all of its needs.

4. **Machine requirements**

   Realm allows service providers to request to be on the same computer or on the same virtual machine as another service (rendered in either hardware or software). This is also a consequence of supporting on-demand agent startup.

5. **Extensibility**

   Both representations and computational functions of knowledge part of realm can be extended by any agent. Alterations do not require changes to Realm itself. It is

sufficient that an agent provides (together with its description) definitions of all new functions and representations it uses.

6. **Local to global approach**

   Heuristics implemented in the constraint-satisfaction engine, as well as the cost–utility analysis, ensure that solutions with minimal impact on the whole system will be tried first. Realm will, however, resort to such actions as reallocating new services to old requests or, even, withdrawing services from old requests, if absolutely necessary.

7. **Resource conflict reduction**

   Realm's constraint-satisfaction engine ensures that requests are services fairly.

8. **Hide connections**

   The programmer is not required to know anything about the connections among various components of the Room. The knowledge is stored by Realm and used by the connection making agent. All the programmer has to provide is the names of the end point services.

7.2 Impact of Realm on the Intelligent Room

First of all Realm makes it possible to move our technology from our research lab to other offices in the building more easily. It will no longer be necessary to customize applications for each new space. As long as most of the services are provided somehow, it is no longer important how they are rendered. For example, most offices will have at most one overhead projector (unlike our research space that has two). In fact, some of the offices will only have computers screens.

Secondly, Realm makes the application development process much easier by taking care of many lower level details such as finding what devices are available and what software should control them. Designers can now focus more on the higher-level aspects of the interactions with Room's inhabitants instead of having to deal explicitly with device choice and control.

Realm makes the Room behave in an apparently intelligent fashion by preventing applications from stealing resources from each other. Also, when appropriate, the Room now

allocates different devices to the same request depending on the state of the interaction with the user. As in the example chapter 4, if any of the agents needs to say anything to the user but the user is on the phone, the Room will recommend using one of the visual devices instead of the speech output.

7.3 Lessons Learned

The main shortcoming of Realm that we observed during the experimentation period was the assumption that it always looks for a single way to satisfy a request. With Realm, tt is not possible to make requests for several or all services of a kind (such features are present in some other system, e.g. OAA). We can imagine situations where an application needs to communicate something to the user very urgently. In situations like that, it would be desirable to make a request for two or three communication devices and use them at once to ensure that the message captures user's attention.

Further, we have discovered a need for posting "standing" requests with Realm. There are two kinds of such requests that are desirable:

1. *Automatic upgrade* – after a request is activated, the requestor should be able to mark it for automatic upgrade. What that means is that whenever a new service that could satisfy the request becomes available, the system should check if the new service is better than the current one. If so, Realm should automatically replace the old service with the new one. We can imagine that if the movie showing agent receives the TV as a display, it may want to request automatic upgrade hoping that one of the projectors will eventually become available.
2. *Idle mode handlers* – requestors should also be able to post requests for services on "whenever available" basis. We can imagine mood creating agents -- acting somewhat like screen savers -- that would take care of certain types of devices whenever those devices are idle. Background music could be played through the speakers whenever nobody else uses

them; “virtual window[3]” could be displayed on an idle projector. This kind of requests could also be used by applications that need to perform resource intensive background jobs, such as faxing copies of meeting minutes.

The “standing” requests are just going to be stored in Realm and kept active. They will not be put in a queue – the system will remain reactive and will not attempt to do future planning.

[3] The research area for the Intelligent Room has no windows. Cameras have been installed in adjacent offices and connected to projectors in the work area to create the “virtual windows.” While this was done on a whim, it proved to have a positive impact on the atmosphere in the lab and would be a perfect “screen saver” kind of application.

8 Contributions

8.1 *Within the Intelligent Room*

Realm provided the Room's software infrastructure with a powerful layer of abstraction. With Realm in place, the Room's applications became portable and plans have been made to install Room-like infrastructure in several other spaces ranging from individual faculty and student offices with very minimal equipment to a medium size conference room filled with a variety of high-tech devices. Without Realm, installing Room's applications in those spaces would have required rewriting all applications to fit the needs and abilities of each individual space.

Also, as mentioned in section 7.2, Realms simplifies enormously the process of creating higher-level applications for the Room by relieving the software designers from having to explicitly choose or control devices needed by their applications.

8.2 *Design Principles for Resource Management Systems for Intelligent Spaces*

One of the main contributions of this thesis is having laid out and tested a set of requirements for a resource management system for a smart environment. Those principles were derived from several years of interacting with and designing software for one such space, namely the Intelligent Room. Some of these principles, such as resource conflict reduction, were a simple consequence of the problem being considered. Others, I believe, were novel and were derived from having observed unanticipated aspects of interactions in the Intelligent Room. The novel ideas included the observation that on-demand service startup is a desirable feature of a smart space and that it causes a number of problems for resource management that need to be addressed explicitly. These problems include having to start some services in correct places or having to anticipate the needs of services being allocated to satisfy requests.

Another important observation reflected in the design principles presented in this thesis has to do with the fact that devices in an intelligent space are often connected by means external to the computer system. Such connections include video connections from VCR to a multiplexer to a projector, or a cable connecting a modem to phone line. Such connections have to be taken into account and they have to be represented in the system. They should also be hidden from the programmer to ensure that the programmer can deal only with abstract services and not with concrete physical devices.

8.3 New paradigm – user as a resource

Presence of Realm also opens the door to a new design paradigm in the Intelligent Room project: one in which user's attention is explicitly modeled as a resource available in the system. Together with extending the concept of connections it will allow for automatic choice of the best device to reach the user even without the intervention of the Room's common sense engine. Consider the example of a calendar application. Let us assume that in order to communicate with the user, calendar agent requests not just a "short text output" service but also "user's attention" service and a connection between the two. "User's attention" service can be provided by two "services," namely user's auditory and visual systems. The moment user enters the Room, connection is established between visual devices (such as displays and the LED sign) and user's visual system. At the same time connection gets established between the speech output service and user's auditory system. Hence when the calendar requests user's attention and user is on the phone, user's auditory system is not available and all calendar can get is visual attention. Given this, the constraint satisfaction engine will allocate one of the services that have a connection to user's visual attention.

This notion of human attention being a scarce resource is a very important and a very old one. After all, there are severe limitations on how many things we, humans, can attend to at a time. Currently, computational abilities of all kinds of devices are improving at a rate close to the one predicted by the Moore's law. The only component of the computational system that is not improving is human attention. Hence, as the spaces get better equipped and physical devices

become plentiful, there will still be a limit on how many agents can try to communicate with a single human user.

The new space allocated to the Intelligent Room project will be equipped with eight projectors (compared to the two available in the current space). Clearly, the number of screens will no longer be the main bottleneck in communicating with the user.

9 Bibliography

[Arn99] Arnold, Ken, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo and Ann Wollrath. *The Jini Specification.* Addison-Wesley, Reading, MA, 1999

[Chun99] Chun, Hon Wai. *Constraint Programming in Java with JSolver 2.0 – An Introduction.* http://www.aotl.com/binary/JSolver%202.0%20Intro.pdf

[Coen97] Coen, Michael. Building Brains for Rooms: Designing Distributed Software Agents. *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence (IAAI97).* Providence, RI, 1997

[Coen98] Coen, Michael. Design Principles for Intelligent Environments. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI98).* Madison, WI, 1998

[Coen99] Coen, Michael, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, Krzysztof Gajos and Peter Finin. *Meeting the Computational Needs of Intelligent Environments: The Metaglue System.* In submission, 1999

[Edw99] Edwards, W. Keith. *Core Jini.* Prentice Hall, Upper Saddle River, NJ, 1999

[Fer99] Ferber, Jacques. *Multi-Agent Systems – An Introduction to Distributed Artificial Intelligence.* Addison-Wesley, Reading, MA, 1999

[Huh99] Huhns, Michael N. and Mary M. Stephens. Multiagent Systems and Societies of Agents in *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence.* Gerhard Weiss ed. MIT Press, Cambridge, MA, 1999

[Las99] Lassila, Ora and Ralph R. Swick eds. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation, 22 February 1999 http://www.w3.org/TR/REC-rdf-syntax/

[Mar99] Martin, D. L., A. J. Cheyer, and D. B. Moran, "The open agent architecture: A framework for building distributed software systems*," Applied Artificial Intelligence*, vol. 13, pp. 91--128, January-March 1999. http://www.ai.sri.com/pubs/papers/Mart:Open/document.ps.gz

[Min99] Minar, Nelson, Matthew Gray, Oliver Roup, Raffi Krikorian and Pattie Maes. Hive: Distributed Agents for Networking Things. *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents.* August 1999

[Mor97] Moran, D. B., A. J. Cheyer, L. E. Julia, D. L. Martin, and S. Park, "Multimodal user interfaces in the Open Agent Architecture," in *Proc. of the 1997 International Conference on Intelligent User Interfaces (IUI97).* pp. 61--68, Orlando, FL, January 1997 http://www.ai.sri.com/pubs/papers/Mora97:Multimodal/document.ps

[Sad94] Sadeh-Koniecpol, N., K. Sycara, and Y. Xiong, *Backtracking Techniques for the Job Shop Scheduling Constraint Satisfaction Problem*, tech. report CMU-RI-TR-94-31, Robotics Institute, Carnegie Mellon University, October, 1994. http://www.ri.cmu.edu/pubs/pub_350.html

[Fri00] Friedman-Hill, Ernest J. *Jess, The Java Expert System Shell*. January 2000. http://herzberg.ca.sandia.gov/jess

[Phi98] Phillips, Brenton. *Metaglue: A Programming Language for Multi Agent Systems.* M.Eng. Thesis. Massachusetts Institute of Technology, Cambridge, MA, 1999

[War99] Warshawsky, Nimrod. *Extending the Metaglue Multi Agent System.* M.Eng. Thesis. Massachusetts Institute of Technology, Cambridge, MA, 1999