Proceedings of the 2nd International Workshop on Self-Adaptive Software, Budapest, Hungary, 2001, to appear.

Design Principles for Resource Management Systems for Intelligent Spaces

Krzysztof Gajos, Luke Weisman and Howard Shrobe[⋆]

MIT Artificial Intelligence Laboratory
{kgajos, luke, hes}@ai.mit.edu

Abstract. 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. There are two major obstacles that prevent this technology from spreading. First, different smart spaces are equipped with very different kinds of devices (e.g. a projector vs. a computer monitor, vs. a TV set). Second, multiple applications running in a space at the same time inevitably contend for those devices and other scarce resources. The underlying software in a smart space needs to provide tools for self-adaptivity in that it shields the rest of the software from the physical constraints of the space, and that it dynamically adjusts the allocation of scarce resources as the number and priorities of active tasks change.
We argue that a resource manager can provide the necessary functionality. This paper presents a set of guiding principles for building high-level resource management tools for smart spaces. We present conclusions we arrived at after two years of exploring the topic in the Intelligent Room Project at the MIT AI Lab. The paper is based on a number of implemented and tested tools.

1 Introduction

For several years, our research group in the MIT AI Lab has been developing an "Intelligent Room" [8, 9, 5], a space that interacts with

⋆ This work was supported by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, and by DARPA through the Office of Naval Research under contract number N66001-99-2-891702.

its users through sensory technologies such as machine vision, speech recognition and natural language understanding. Our room also is equipped with a rich array of multi-media technologies. These technologies are intended to provide a natural, human-centered interface to its users.

The Intelligent Room is designed to be a utility that must always be available and it must provide reasonable services to its users even though their needs are not easily predicted. It must continue to provide these services even if there are equipment failures or if there is contention for the use of resources among the users or applications. It is also desirable that it be able to provide improved and additional services if higher quality equipment is added.

Finally, and most crucially to be truly human-centered it must be able to do all these things seemlessly while running, without intervention by programmers and systems wizards. In other words, the Intelligent Room must be a self-adaptive system in the spirit of [17, 16]. It must monitor the environment as well as its own state, have a variety of techniques for accomplishing its goals, and make intelligent choices about which technique to use in the current context.

This paper describes our experience with building such a system. The key insights are:

1. People should interact with the Intelligent Room not in terms of resources, but rather in terms of abstract services (e.g. “show me this information” rather than “print this on that printer”)
2. The Intelligent Room should be capable of mapping a service request to a variety of solutions (“project the information,” “display it on a PDA,” “print it on a printer”)
3. The Intelligent Room should choose a solution based both on how well the solution meets the users’ needs and how well it minimizes the use of costly or rare resources and
4. It should make this decision at runtime so that it can respond to a changing set of requests and a changing environment.

1.1 What is a resource manager for a smart space

What we mean by a resource manager is a system capable of performing two fundamental tasks: *resource mapping* and *arbitration* .

By resource mapping (a.k.a. match-making) we mean the process of finding out what actual resources can be taken into consideration given a specific request.

By arbitration we mean a process of making sure that, at a minimum, resources are not being used beyond their capacities. At best, arbitration ensures–via appropriate allocation of resources to requests–optimal, or nearly optimal, use of scarce resources.

This paper is concerned with management of higher-level resources. While OS level management (memory, CPU time, etc.) is of course important, and load-balancing of computationally intensive agents over multiple machines is also, we limit our focus to higher-level resources such as physical devices and large software components (see [28] and [22] for an example of a system that deals with resources in a smart spaces at the OS level). Our concerns lie with, for example, projectors, multiplexors, wires, displays, modems, user attention, software programs, screen real estate, sound input and output devices, CD players, drapes, and lamps.

1.2 Some definitions

For clarity, we define here some potentially ambiguous terms.

Metaglue Metaglue [10, 21, 26] is the multi-agent system forming the software base for all work at the Intelligent Room Project. Metaglue manages agent-to-agent communication via Java's RMI system. Agents can start and obtain references to other agents via a `reliesOn` method. All agents have unique IDs; part of an ID is the "occupation" which is the top-level interface the agent implements. Agents are also collected in societies so multiple users and spaces can have distinct name-spaces. Metaglue makes it easy to coordinate the startup and running of agents on any number of machines with differing operating systems.

Agent Agents are distinct object instances capable of providing services and making requests of the resource manager. This means agents themselves are considered to be a type of resource (see below) because they provide services.

Device A physical or logical device is something akin to a projector, screen, or user-attention; devices are often, but not necessarily represented by agents. Devices provide services and so are resources.

Service Services are provided by agents and devices; a single agent or device can provide more than one service and any kind of service can be provided by a number of agents or devices. This is explained in more detail in Section 4.1.

Resource A resource is a provider of a service. Both agents and physical devices are resources. For example, a physical LED sign is a resource (providing the LED sign hardware service) obtained and used by the `LEDSignText` Agent which is in turn a resource (providing `TextOuput` service and `LEDSign` service) that can be obtained and used by any other agent needing such a service.

2 Our work to date

This paper is based on our work on the Intelligent Room Project [8, 9, 5] at the MIT AI Lab, including otherwise unpublished research on resource management. Below we summarize our results relevant to this paper in order to give the reader a better idea of how we arrived at our observations.

Over the past two years we have developed several resource management tools for the Intelligent Room. The tools differed in approach and level of sophistication. At the two extremes we have Namer and Rascal [15]. Namer only does context-based name resolution (i.e. some service mapping but no arbitration). Rascal, on the other hand, is a very complex system that uses a rule-based language (JESS, [13]) for representing knowledge about agents' services and needs, as well as for service mapping, and uses a constraint satisfaction engine (JSolver, [7]) for arbitration. All of our resource managers implement a common interface which allows us to interchange them without changing any of the other code in the system. The reason for having several different resource management schemes was motivated by more than just the need to find the right solution: it is our assumption that different resource mangers will be used in people's various mobile personal spaces (with one or two devices and where computation is scarce) and large and well-equipped static spaces.

As mentioned before, Rascal is our most complex resource manager, conforming to most of the design principles laid out in this paper. Currently Rascal does not deal with issues of privacy and access control and we have only just began work on cooperation mechanisms.

Rascal relies on agents having external descriptions of themselves. Such descriptions include a list of startup needs, a list of provided services (each with a list of its own needs) and descriptions of all possible requests for resources the agent may make in its life-cycle.

Knowing startup needs and needs for providing particular services allows Rascal to ensure that before it assigns a particular agent to provide a service in response to a request, all of the needs of that new agent (and its underlings) can be satisfied. For example, if some agent requests a TextOutput device and the possible candidates are SpeechTextOutput and GuiTextOutput, Rascal will ensure that either speech generation is available for SpeechTextOutput or a computer screen is available for GuiTextOutput, before assigning either candidate to the requester. Additionally, knowing agent's startup needs also allows us to dynamically choose what particular machine an agent should be started on.

So far our software has been installed in six spaces of four different kinds: three small offices, one small living room, one large office (also used for small meetings), and a twelve-seat conference room. The instrumentation of these spaces varies widely; we have a single dilapidated projector and a couple of lights in one of the small offices on one hand, and six projectors and a large number of A/V devices in the conference room on the other. Our living room has two projectors, a TV, several cameras, and A/V equipment.

3 On-demand agent startup - reasoning about absent agents

An agent system in a smart space should have a way of autonomously starting agents on-demand and consequently the resource manager should be able to reason about agents that are not alive right now but could be brought to life if needed.

Because on-demand agent startup is one of the basic features of Metaglue, we have taken it for granted but many other agent systems do not support it. Hence we will now briefly argue why on-demand agent startup is a desirable feature of an agent system in charge of a smart space and then discuss the consequences for resource management.

3.1 Why support on-demand agent startup in smart spaces

Most agent systems deal with very dynamic, spontaneously created and often unstable collections of agents. Therefore, creators of such systems have to refrain from making assumptions about what is available in the system at any given time and usually have to resort to dynamic discovery, direct negotiation or other such techniques when an agent looks for a service or resource (e.g. [18, 23, 12]). This general attitude has been assumed by creators of agent systems controlling smart spaces. Standard Jini [2] implementation and Hive [20] are good examples of such systems.

Smart spaces, by the virtue of being based on stable physical environments, impose a special set of constraints on the underlying software infrastructure. It is true that a lot of adaptivity is still needed – new components can appear and disappear, people come and go, devices are brought in and removed–but at the same time we benefit from assuming certain level of persistence.

It is a feature of a physical space that most of its components are static in the sense that they are usually there. If one day a space contains lights, A/V equipment, projectors, and telephones, it is reasonable to expect that those devices would be present the next day as well. They will be there whether we are using them or not. This level of predictability can (and should) be reflected by the underlying software infrastructure. This is not to say, of course, that the software should not be capable of dynamically accepting new components.

On-demand agent startup is highly useful in any flexible space in a variety of ways. For example, it allows us to make multiple instances of an agent when we want to perform several versions of the same task. Furthermore, it allows us to have very complex interrelationships between agents and very large numbers of agents. Without on-demand startup one needs to craft elaborate startup scripts or hand-start all the agents in the system; both of these are infeasible when talking about collections of forty agents or more, especially when considering that the particular agents change depending on who is starting the system, the various tasks the system is to accomplish, and the room the system is being started in.

With on-demand startup, starting a single high level agent is sufficient to obtain a service provided by that agent. This agent will then request and, cause to be started, all other agents it needs in order to do its job well.

Even with the convenience argument set aside, the following example illustrates some additional benefits of being able to start agents dynamically.

Example 1. Let us assume that the phone service is provided by the phone agent. The agent needs a computer with a voice modem hooked up to a phone line in order to provide its services. Imagine a system consisting of several machines with voice modems hooked up to a single phone line (e.g. in a shared graduate student office). If we did not allow for on-demand agent startup, we would have to do one of the following:

1. Start the phone agent on a prespecified machine, running a risk that if that machine goes down the service is no longer available.
2. Start an instance of the phone agent on every machine with a voice modem and a connection – a rather misleading solution because each of the agents would be advertising phone service but only one of them would be able to provide it at a time because all of the machines share a single phone line.

Another immediate use for automatic agent startup has to do with robustness and recovery: if an agent providing a computational service goes down because of computer failure, it can be automatically restarted at a new location.

We understand that this point has much potential for debate, and so we will not dwell it as other aspects for and against it lie outside the realm of resource management.

3.2 Impact on resource management

If we assume that on-demand agent startup is supported by the underlying software architecture, then it stands to reason that the resource manager for such a system has to be able to reason about absent agents.

To the best of our knowledge, it is uncommon in agent architectures, even those in charge of smart spaces, to have non-alive agents be taken into account during any coordination efforts. It is our belief that taking potentially available agents into account allows a resource management system to make intelligent decisions about resource allocation as in Example 1 in the previous section. (See also Example 2 in Section 4.3.)

An important consequence of embracing on-demand agent startup is that we cannot rely on agents themselves to provide descriptions of their needs and services as they might not be running. The resource manager has to have access to such descriptions without having to instantiate any of the agents. Rascal requires agent programmers to create separate description files but other solutions could easily be created (e.g. descriptions could be cached by the resource manager).

Implicit in Example 1 in the previous section is the assumption that the system, and in particular the resource manager, has a way of starting agents on a specific computer or virtual machine. Metaglue provides such capability as one of its two main primitives. It is unclear to us at the moment to what extent other systems support it.

4 Representation

In this section we concentrate on what knowledge should be contained in the resource manager but not on how that knowledge should be encoded. In particular we argue that when building a resource manager for a smart space, the following key points should be observed:

– Represent resources in terms of the services they provide (e.g. text output) as well as their type (e.g. scrolling LED sign).

– Ensure that representations are rich enough to allow the requesters to get *the best* tools for the job. In particular we caution those using Java against using only interface names for describing resources.
– Ensure that the representation is capable of describing resources that are not represented within the agent systems by agents or other special proxy objects. Examples of such resources would be hardware that is not directly controlled by the agent system but yet is crucial for system's performance (e.g. wires, low level computer components such as modems, third party software modules, etc.)

4.1 Services not devices

To be truly useful, smart spaces 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 packaged software for our rooms and offices just as today we get it for our desktop computers. 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 for desktop computers can require that a computer should be equipped with a display, a CD-ROM and a sound card, they certainly cannot require the same level of uniformity among smart 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 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. As well as providing uniform interfaces to devices, as is done on desktop computers, we propose providing uniform interfaces to the 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. For example, on one hand, the "short-text-output" service may be rendered by a computer display device, a speech output device or by a one line scrolling-LED display. On the other hand, the LED sign, as well as providing short text display, can provide simple graphics and animation.

We are not unique in suggesting that devices represented by device drivers are insufficient for a smart space; a somewhat different approach was suggested by Winograd [27]. Schubiger-Banz et al. [25] argue for "addressing by concept" in all ubiquitous computing environments (both spaces and/or collections of mobile devices). INS [1] uses "intentional names" for all networked resources. EasyLiving [6] also seems to represent resources in terms of services they provide.

4.2 Rich representation - rich requests

We now examine how the services should be represented by the resource manager. The details are, of course, dependent on the particular implementation.

Open Agent Architecture OAA [19], which relies on a facilitator agent for all inter-agent communication and task brokerage, uses a PROLOG-based ICL (Interagent Communication Language) for describing agents' needs and capabilities. The language allows service providers to describe the agents in terms of tasks they can perform and not really in terms of resources they represent.

Decker [11] uses KQML for communicating needs and abilities of agents.

A common tendency among Java-based systems (e.g. Jini [2], Hive [20], Rascal [15, 14]), is to use the name of the interface (or interfaces) that the resource implements, and a list of attribute-value pairs for describing agents' capabilities.

In this last case, an agent's interface provides information on how the agent's capabilities should be invoked. It also often provides most of the information on what the agent does. One of the advantages of using interface names for describing agents is that interface "ontologies", i.e. APIs, are easily understood by programmers and some of them get adopted by large communities. But it has to be stressed again, that the interfaces should provide access to agent's services. Thus an agent can advertise a number of interfaces, one for each services it provides.

We agree with designers of Hive and Jini in that the types of interfaces implemented by an agent provide a lot of valuable information about agent's capabilities and expected behavior. We also agree with them in that the interface names are not sufficient for describing any agent fully.

Just having interfaces and nothing else be an agent's description is not enough. Often a number of parameters, some of them continuous, contribute to a service's full description. Display services need to be described in terms of resolution, size, color depth, brightness, etc. Having detailed descriptions of services allows for more precise requests: an agent that needs to show a map with a lot of detail, will request a high-resolution color display, not just a display. At the same time, a mail alert agent could deal with a very low resolution display as long as it is visible and so would ask for the display without additional parameters.

4.3 Abstract resources

One important feature that distinguishes multi agent systems in charge of smart spaces from other multi agent systems is that they reside on the frontier between the physical and computational worlds. To function well, those systems have to not only accept but also embrace the physical world around them (we refer to this point again in Section 8).

As a consequence of this, it becomes necessary for the system to explicitly describe not only the services provided by its agents but also those provided by physical hardware and non-agent software present on available computers.

A common approach to this problem is to add agents to represent all needed physical and computational capabilities of the host environment. Hive, for example, uses "shadows" to represent physical devices accessible on or from particular computers. Metaglue has agents that represent individual devices. But how do we know where to start those shadows or agents? An unsatisfactory way is when startup has to be done by a human or by a script leaving the system with no way of reasoning about it or taking action on its own. In case of Metaglue, the device-controlling agents upon startup retrieve the name of a computer they should tie themselves to. In our view, the agents that directly interact with hardware or other software should be able to start dynamically (see Section 3) and dynamically find the computers with all necessary equipment and software.

Example 2. Currently in our system, the main way of providing the speech-input service is with personal wireless microphones connected to computers running third party speech recognition software. In our conference room we have several computers with the right software, several microphones, and an audio mixer that allows us to route microphone signal to any of the computers.

When any of our agents requests speech-input service, Rascal, our resource manager, checks the description of our speech input agent for all of the services that it will need to provide the service. Those will include a computer with a speech recognition engine, a microphone, and a connection between the two. Neither the speech recognition engine nor the microphones have software proxies in our agent systems yet the resource manager is able to reason about them. Rascal ensures that the speech-input agent starts on a computer with the right speech recognition engine and will award a microphone that is not being used for other tasks (e.g. teleconferencing) to the agent, and will ensure that there exists a connection between the two (see Section 8 for discussion of connections).

5 Arbitration

At the heart of resource management is arbitration. By our definition of a resource manager, when two or more agents vie for the same limited resource, the resource manager has to evaluate which gets what.

In this section we argue that arbitration is essential in any larger system embedded in a smart space because it allows individual agents and applications to be written without having to take other agents' and applications' needs into considerations. It also provides for the most basic (but not the simplest) apparently smart behavior of a space. Some arbitration schemes applicable in open agent systems, such as marked-based resource allocation, will prove less effective. Cost-benefit based on self-reported needs and preferences has proven a good solution especially when combined with access control (which limits requests by untrusted and potentially malicious or non-conforming agents).

In addition, in cases where a resource needs to be taken away from a requester to satisfy a new, more urgent, request, every effort should be made to find a replacement for the withdrawn resource.

5.1 Why arbitrate

Arbitration allows for easier implementation of individual agents: the agent writer can view the world more selfishly than if there was no arbitration mechanism. With arbitration in place, the agent programmer can be sure that if any other agent needs resources more, the system will take care of necessary re-allocations (just like in properly multi-tasking operating systems programmers do not need to worry about yielding to other processes).

Another (obvious) benefit of arbitration is achieving apparent "intelligent" behavior of a space. Just like animals are expected to use their body parts intentionally and in a coordinated fashion, we also expect computer-steered spaces to be "aware" of the interface devices.

5.2 How to arbitrate

The simplest way of resolving ties among requests is to award a resource to the most recent request. For many reasons this may prove to be insufficient. For example it would not be desirable for a new email notification to take over a screen during a video conference with one's boss. Hence there exists need for some analysis before allocating resources. Rascal, for example, uses a simple cost-benefit analysis (details in [15, 14]) to decide who should be awarded a particular service. This scheme relies on agents accurately and honestly reporting how urgently they need a resource. This approach is potentially problematic in that it allows for malicious or inaccuarate representation of one's needs.

A more natural and simple approach to arbitration in potentially open systems in smart spaces seems to be one in which some access control mechanism is used in conjunction with some priority-based scheme. In such a situation the access control mechanism would weed out requests from untrusted and unauthorized agents and then a priority mechanism would decide which of the trusted and authorized requesters should get what resources. In a model where agents can act on behalf of spaces or people, the role-based access control model [24] seems a viable option. We discuss the need for access control further in Section 10.1.

Other approaches had been developed with open systems in mind, notably some based on market mechanisms [3, 4]. Those approaches require existence of a central "bank" and some sort of currency. Such approaches, in their natural form, are not well suited for smart environments. It should not be possible, for example, for someone thousands of miles away to buy control of the room with their extra virtual currency.

Because resource managers can take resources away from requesters, it is reasonable for a requester to keep a resource even after finishing a task if it expects it may need the resource again in near future. For example, a email notification agent may want to keep its output channel as it is desirable for the sake of consistency in space's behavior for those notifications to come through the same channel unless there is a good reason to change.

5.3 Arbitration should allow for clever re-allocations

Consider the following scenario: a space is equipped with a TV set, LCD projector, VCR, video mux, and some computers. The VCR (that also acts as a TV receiver) can be connected to either the projector or to the TV set through the video mux. The computer, however, can only be connected to the projector.

The user is watching the news on the projector, this being the best resource to satisfy a request for a large display. Then the user hears some really important news and decides to share it with a friend while watching the rest of the newscast and so she requests her email agent. With our simpler resource management schemes in place, the projector would be taken away from the news and allocated to the email agent. The more desirable behavior, in this situation, would be for the newscast to be moved over to the TV set and the email to be then displayed on the projector.

The point here is that in many cases the only way to accommodate a new request is to take a resource away from one of the currently active requests. The disturbance can often be minimized, however, by reallocating the old request to a different service. The insight here is that the sets of services that can satisfy various requests overlap only partially and the relationships are often more complex than just proper inclusion (see Figure 1). The reason for it is two-fold: first, different kinds of devices can provide different sets of services; second, physical connections for different kinds of signals are routed differently (so, for example, in one of our spaces the video signal goes through a multiplexer and thus can be connected to either of the projectors or to a TV set, while VGA connections are hard wired).

Allowing for re-allocations makes arbitration among requests much more complex: whenever a resource manager receives a request for a resource, it has to look for a solution that satisfies not only the new request but all of the old ones as well (as far as possible). In other words, a simple task of selecting the best resource for a request turns into a global constraint satisfaction problem.

One point to keep in mind, of course, is that re-allocations are costly. If we move the newscast from a projector to a TV,the user is bound to find it distracting. In some cases disturbance will be minimal, for example a mail notification agent will not mind a re-allocation if it

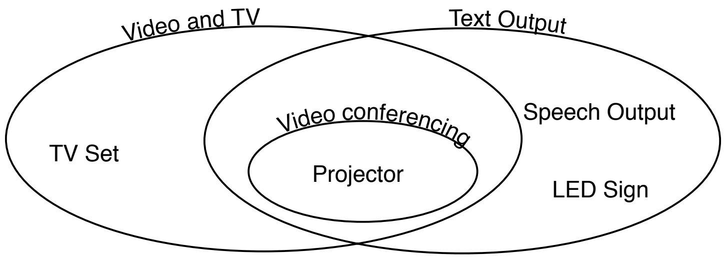

Fig. 1. Different kinds of tasks can be performed with different but overlapping possible sets of devices: both the TV set and the projector can be used for watching videos or tv while only the projector can be used for teleconferencing. At the same time, the projector, the LED sign or the speech output can be used for text output.

happens between notifications—the next time it will simply use a different output device. It's more serious in case of agents that may have to rebuild a lot of their state after re-allocation. Some examples of this would be a camera which was carefully focused on a face or area of the room, or an Internet browser with all of its browser history.

Reasoning about the cost of a re-allocation has to be a part of the overall arbitration process. In Rascal, requesters can specify how costly a re-allocation would be to them and the cost can vary between zero and the cost of taking the service away altogether. The cost of a re-allocation in Rascal has two components: the fixed cost specified by the requester and the difference in utility between the new request and the old request (with the stipulation that it cannot be smaller than zero).

6 Ownership of resources over time (resources vs. tasks)

In this section we argue that many of the services provided by agents in a smart space (e.g. display service provided by a projector) are not tasks and therefore they should be managed differently from tasks. In particular requesters should be given ownership of resources over periods of time. Agents need to own their resources as they are often engaged in long-term jobs that can be changed or modified. We discuss all of this by comparing what we mean by resource management with task management performed by the faciliator agent in the Open Agent Architecture (OAA) [19].

Open Agent Architecture (OAA) is a good example of an agent system that could control a smart space and that also has a complex inter-agent facilitation scheme. What needs stressing, however, is that the OAA “facilitator agent” actually performs *task* management and not *resource* management. That is, the facilitator agent will break down a task into simpler sub tasks and allocate those to individual agents who can fulfill them best. It will not, however, ensure that all of the resources needed for the tasks are available and not in use by other agents. Hence OAA is well suited for a task like sending the current Boston weather report to all of requester’s friends. The task will be broken into components, appropriate information obtained and message sent. OAA is not well suited for tasks that cannot be thought of as point-like in time. Implicit in the OAA model is the assumption that agents can never conflict over the use of scarce resources. Task management is, of course, very important but in a system that controls a physical space with a large number of scarce resources task management should work hand in hand with a resource manager.

It is more natural to think of many agents as having a life cycle, and going independently about their own long-term jobs. For example, an agent listening to and recording conversations in the room in order to be able to bring back audio snippets via keyword searching, needs resources over an extended period of time to complete its job. Showing a movie can be thought of as a task but it can be interrupted, modified, or abandoned in the middle. It can also prevent other agents from using a display for their jobs. In that sense, showing a movie is different from the OAA view of a task.

7 Third party resource request annotation

Once the core resource management system is in place, it should be easy to write modules that do specific types of reasoning and then use that reasoning to annotate requests to limit or reassess possible matches. Often knowledge of the world has direct impact on the appropriate nature of a specific resource to a specific request–this knowledge being outside either the resource manager’s realm of expertise or the requesting agent’s knowledge–and it is imparative that it be easy to have a third party entity contribute such knowledge.

For example, say a user is in a particular room and desires to play a song via his SongPlayer agent. The user would start his agent that would then ask for and locate the bits of the given file, and then attempt to gain access to another agent which would play the actual file. This, of course, would be a resource request for an agent with the ability to play sound.

However, there is also another criterion to the desired agent: location. If the user has been wandering from room to room, it is important that the sound playing agent used be in the room the user is in. This is knowledge that needs to be appended to the request, but neither the resource manager nor the song requesting agent would appropriately have this knowledge.

It would be a violation of normal notions of modularity if the SongPlayer agent had to check the user's location and annotate its resource request. It also seems unwieldy for the resource manager to be responsible for finding and maintaining this knowledge; certainly if this knowledge were in the resource manager's domain, then much other knowledge would be as well. Furthermore, the nature of a flexible agent system is knowledge itself is unlikely to be codified in a universal standard, and so the resource manager would be responsible for translating the output of various other agents into proper resource request annotations. Solving this problem is definitely an active area of research, but in this case it make for a massively large and unwieldy project in the writing of the resource manager.

The best solution we found is to have third party agents that extend the functionality of the resource manager. Authors write agents or functions which pattern match on resource requests and add then additional criterion to those requests as appropriate. In the example above, a distinct other agent which tracks the user eavesdrops on all resource requests and annotates any relevant ones to only consider physically local possibilities.

Request annotations should be able to happen in two ways. The first is modifying the request before a list of possible matches is generated. The second method is filtering the possible matches at the tail-end of the process, after the list of possible resources has been generated. Regardless of method, third party annotators allow for a real componotization of the agents; without them either one or the other agent on any given transaction needs to know too much about the significance of the job at hand. The idea is to have dumb objects wired together smartly to get thinking results, not to have heavyweight objects that are hard to write or maintain.

A further advantage of third party annotators is being able to provide the room with a way of dynamically adapting to equipment failures by writing modules that extended reasoning about certain particular resource allocation problems. For example, if we had an agent that could tell if a projector was broken by looking at the screen with a stearable camera, we could easily have an agent update the resource manager so all resource requests for projectors automatically remove that projector from consideration.

Furthermore, having the ability to have third party annotators should, we hope, serve nicely in the future when contemplating adding large features to the system such as access control (see Section 10.1). Once the model of requesting resources and receiving them is established, pretty much anything can be thought of as modifying or changing the appropriateness of a given resource to a given request–namely annotating a preexisting request.

8 Connections

One style of resource that deserves special attention are connections. Connections are a vital piece of the background of a smart space, and a system with a resource manager that fails to manage them is bound to end up in serious trouble.

The way our room is wired, we have several muxes and switches allowing for information to flow from source devices (cameras, VCRs, microphones) to output devices (projectors, TV sets, modems). Computers are also integrated into this web as either sources or sinks. We also have some trunk wires connecting muxes to muxes, for example, which can only carry one signal at a time. This, of course, is a limited resource. We are a long way from the time when the optimal carrier of all information signals (audio, video, etc.) is the same Ethernet, and until then we need to take into account the specialized wires in an intelligent space. This often means we do not have a fully connected graph of signal sources and sinks, and so the physical connections themselves are a limited resource that needs management.

Due to this, we enter all our connections into the manager as "connection resources". When an agent requests, say, a VCR and projector combination, they also request the collection of resources consisting of the path of connections leading from the VCR to the projector.

We keep the connection aspects of the system very much behind the scenes as an extension to the resource manager. Just because they are a crucial piece does not mean that they need to be in the forefront of a high-level agent programmer's attention. Agents can just ask for resources with the caveat that they are connected, and do not look at the resources involved in the connecting itself at all. The connection extension to the resource manager forges the actual path.

9 Special requests

We have discovered a need for a few "special requests" for resources that seem to lie a bit outside the parameters discussed above. Happily, these are extensions of above, and can be added layers on top of the existing system. We will briefly discuss them in the following sub-sections.

9.1 "Screen saver"

Many agents may want to use resources for a low-level background effect if the resources are not being used for something else. For example, the news ticker or weather forecast agent may want to use the LED sign if there is no better use for it.

The "Screen Saver" type of request gets automatically re-filled after the resource is taken away, used, and then released by some other agent. It is a way of the agent saying, in effect, "I want these resources whenever they are free. If you take them away, then give them back when they become free again."

The advantage of this approach is that it prevents a busy wait on the agent's side. Without "Screen Saver" requests, an agent would have to poll the resource manager from when it has lost its resource until it obtains it again.

An alternative solution would be to have blocking requests, which would also work. We have not closely examined this option, however.

9.2 Auto upgrade

When a resource being used by an application is released, it is worth checking to see if other agents would be better served by getting that resource now that it is available. Agents can specially request that they do not mind being switched to a better resource at any time.

9.3 High-urgency short-term loans

Some requests are for more task-oriented reasons. In these cases, a resource may be needed only for a brief moment. For example, an alert agent might briefly need the speakers of the room to inform a room occupant that there is a call waiting. If the occupant was watching a movie, it would be much more smooth if the alert agent could just borrow the audio for a moment and then give it back. Without borrowing, the original agent would have to re-request the lost resources, and again we would have the polling situation described in the previous sub-section 9.1.

Loans, of course, make cost analysis in the resource manager even more difficult and we have found no easy answers as of yet.

10 Future

In this section we talk about two issues in resource management in smart spaces that we have identified as important but have not yet researched in depth.

10.1 Access control

The real world is full of access control mechanisms. In particular, there are many ways in which access to spaces and enclosed equipment is restricted to certain people. The same is true of information. It stands to reason that agents acting on behalf of people should be subject to similar constraints their owners. If Alice does not have a key to Bob's office, then she is probably not supposed to be able to use his VCR either. We can take this parallel a step further and introduce some more interesting problems.

All members of our lab have a right to enter our conference room. They also have the right to control all of the a/v equipment, the lights, etc. To what extent should this right be extended to their electronic proxies? Should people be granted access to the devices when they are not physically present in a space, e.g. while on a trip to a faraway country? Should the access to the devices only be granted to authorized people on the condition that they are physically present in the space?

If we assume that physical presence is required of most people, let us take another scenario into consideration. Our research group has a meeting and one of the members is in a different city and needs to teleconference with us. During the meeting she needs to show us some of her results. Should she then be allowed to control our projectors and our slide show software? Should telepresence be treated equally with physical presence? Should perhaps one of the people physically present at the meeting grant her the permission? If so, who should have the rights to grant permissions to others?

As we said before, we are not clear yet how access control should be performed in a smart space but we are quite certain that the resource manager would have to be a part of the process. After all, it is the resource manager that grants agents access to particular resources. Thus the resource manager needs to be able to find out what resources the requester has rights to.

10.2 Cooperation

As research on smart spaces progresses, it becomes more and more likely that several spaces will be controlled by the same software. A number of people will be moving from one smart space to another and will expect to be able to make various requests in those spaces. They will also expect some of their agents to “follow” them. Building a single resource manager that would manage resources of all the spaces and all the people is clearly impractical. hence, there will have to be a number of resource managers, each representing a particular collection of resources and requesters. Given that spaces may border with each other or be enclosed by one another, and also given that agents acting on behalf of people will need to use resources provided by spaces, it is necessary for resource managers to communicate with one another to perform optimal resource allocation.

11 Contributions

We have outlined a number of issues that we found to be important in the design of high-level resource management systems for smart spaces. Smart spaces are a relatively new research area and few projects have reached a point where resource management would become critical. We believe, however, that all projects will eventually face these problems once their basic infrastructure is in place and multiple, independently developed, applications are being ran in a space at the same time.

12 Acknowledgments

Many people have contributed to the research presented here. In particular Stephen Peters has provided us with a lot of help in implementing and testing some of the experimental resource management software. Stephen is also working on representing multiple users and multiple spaces in our system. Rattapoom Tuchinda is currently researching the issues of privacy and access control in smart spaces.

References

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM*

Symposium on Operating Systems Principles (SOSP), Kiawah Island, SC, December 1999.

2. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Reading, MA, 1999.
3. Jonathan Bredin, David Kotz, Daniela Rus Rajiv T. Maheswaran, Çagri Imer, and Tamer Basar. A market-based model for resource allocation in agent systems. In Franco Zambonelli, editor, *Coordination of Internet Agents*. Springer-Verlag, 2000.
4. Jonathan Bredin, Rajiv T. Maheswaran, agri Imer, Tamer Basar, David Kotz, and Daniela Rus. A game-theoretic formulation of multi-agent resource allocation. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 349–356, June 2000.
5. Rodney Brooks. The intelligent room project. In *Proceedings of the Second International Cognitive Technology Conference (CT'97)*, Aizu, Japan, August 1997.
6. B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. Easyliving: Technologies for intelligent environments. In *Handheld and Ubiquitous Computing*, September 2000.
7. Hon Wai Chun. Constraint programming in Java with JSolver. In *First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, London, April 1999.
8. Michael Coen. Building brains for rooms: Designing distributed software agents. In *Ninth Conference on Innovative Applications of Artificial Intelligence (IAAA97)*, Providence, RI, 1997.
9. Michael Coen. Design principles for intelligent environments. In *Fifteenth National Conference on Artificial Intelligence (AAAI98)*, Madison, WI, 1998.
10. Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The Metaglue system. In *Proceedings of MANSE'99*, Dublin, Ireland, 1999.
11. Keith Decker, Mike Williamson, and Katia Sycara. Matchmaking and brokering. In *The Second International Conference on Multi-Agent Systems (ICMAS-96)*, 1996.
12. Jacques Ferber. *Multi-Agent Systems - An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, Reading, MA, 1999.
13. Ernest J. Friedman-Hill. Jess, the Java Expert System Shell. Technical Report SAND98-8206, Sandia National Laboratories, 1997.
14. Krzysztof Gajos. A knowledge-based resource management system for the intelligent room. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2000.
15. Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS 2001*. Springer, 2001. To appear.
16. Robert Laddaga. Active software. In Paul Robertson, Robert Laddaga, and Howard Shrobe, editors, *Self-Adaptive Software*, number 1936 in Lecture Notes in Computer Science. Springer, 2001.

17. Robert Laddaga, Paul Robertson, and Howard Shrobe. Results of the first international workshop on self adaptive. In Paul Robertson, Robert Laddaga, and Howard Shrobe, editors, *Self-Adaptive Software*, number 1936 in Lecture Notes in Computer Science. Springer, 2001.
18. Victor Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. *Autonomous Agents and Multi-Agent Systems*, 1:89–111, 1998.
19. David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999.
20. Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In *Proceedings of ASA/MA'99, the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, August 1999.
21. Brenton Phillips. Metaglue: A programming language for multi agent systems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.
22. Manuel Roman and Roy H. Campbell. Gaia: Enabling active spaces. In *Proceedings of 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.
23. Tuomas W. Sandholm. Distributed rational decision making. In Gerhard Weiss, editor, *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.
24. Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
25. Simon Schubiger, Sergio Maffioletti, Amine Tafat-Bouzid, and Bat Hirsbrunner. Providing service in a changing ubiquitous computing environment. In *Proceedings of the Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality, HUC 2000*, September 2000.
26. Nimrod Warshawsky. Extending the Metaglue multi agent system. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.
27. Terry Winograd. Interaction spaces for 21st century computing. In John Carroll, editor, *Human-Computer Interaction in the New Millenium*. Addison-Wesley, 2001.
28. Tonomori Yamane. The design and implementation of the 2k resource management system. Master's thesis, University of Illinois, Urbana, Illinois, 2000.