Apeared in the Proceedings of The Second International Workshop of Central and Eastern Europe on Multi-Agent Systems CEEMAS 2001, Kraków, Poland

Rascal - a Resource Manager For Multi Agent Systems In Smart Spaces*

Krzysztof Gajos

MIT AI Lab, Cambridge, MA, USA,
kgajos@ai.mit.edu,
http://www.ai.mit.edu/people/kgajos

Abstract. Multi Agent Systems (MAS) are often used as a software substrate in creating smart spaces. Many of the solutions already developed within the MAS community are applicable in the domain of smart spaces. Others, however, need to be modified or re-developed. In particular, it has to be noted that many agents acting in a physical space domain are restricted in number and capability by the scarce physical hardware available. Those limitations need to be taken into account when coordinating agent activities in a MAS in a smart space.
In this paper we present Rascal, a high-level resource management system for the Intelligent Room Project, that addresses physical resource scarcities. Rascal performs the *service mapping* and *arbitration* functions for the system. Rascal is an implemented tool and has been partially deployed for day-to-day use.

1 Introduction

Building smart spaces requires distributing computation across a number of computers. The software components of a smart space need to cooperate robustly and the system must be able to cope with components being added and removed dynamically. For that reason a number of research groups have adopted an approach in which a multi-agent system (MAS) is the software substrate connecting all of the computational components of a smart space [10, 6, 9].

Agents in smart spaces have to deal with many of the same issues as agents in other MAS. At the same time, physical spaces are a domain with their own features and constraints that affect how the agents deal with certain situations.

In particular, agents in a smart space are heavily resource-bounded because they are embedded in a physical world where all physical resources are scarce. To illustrate this with an example, task-oriented systems, such as described in [2], focus on appropriate goal composition assuming that all particupating components can render their services at any time. In contrast, the coordination of

* 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.

multiple agents in a smart space is more difficult because physical constraints have to be taken into account. For that reason, an explicit resource management system is required in a smart space before agents' tasks can be managed effectively.

In this paper we present Rascal, a resource manager for the Metaglue agent platform. Metaglue [6] is a MAS developed at the MIT AI Lab for the Intelligent Room project. Rascal provides service mapping and resource access arbitration mechanisms for Metaglue agents. Rascal has been implemented and partially deployed for every-day use.

1.1 Definitions

What Is a Resource Manager For a Smart Space We believe a resource manager should be capable of performing two fundamental tasks: *resource mapping* and *arbitration*.

Resource mapping (i.e. match-making) is the process of deciding what resources can be used to satisfy a specific request.

Arbitration is ensuring that, at a minimum, resources are not being used beyond their capacities. Ideally, arbitration ensures optimal, or nearly optimal, use of scarce resources via appropriate allocation of resources to requests.

This paper is concerned with the management of high-level resources. As opposed to OS level management (memory, files, etc.) and load-balancing computationally intensive agents over multiple machines, these high-level resources include physical devices and large software components, for example, projectors, multiplexors, wires, displays, modems, user attention, software programs, screen real estate, sound input and output devices, CD players, drapes, and lamps.

For clarity, we define some potentially ambiguous terms that are used throughout the remainder of this paper:

Metaglue Metaglue [6] is the MAS forming the software base for all work at the Intelligent Room Project. Unlike most MAS, Metaglue provides infrastructure for close-coupling of agents (that is, it facilitates direct method calls) in addition to a message passing mechanism in order to enable faster communication among agents. Metaglue is intended for use in environments where most agents are physically close and thus good network connectivity can be assumed. Metaglue makes it easy to coordinate the startup and running of agents on any number of machines with different operating systems.
Metaglue agents are collected into "societies" which are distinct name-spaces for multiple users and spaces. A new communication and discovery model is currently being developed for inter-society communication.

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 because they provide services (see below).

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 therefore 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. For example, the `ShortTextOutput` service can be provided by the on-wall display, scrolling LED sign or a text-to-speech program. An A/V receiver is a provider of a number of services, such as an amplifier, an audio multiplexor and a radio receiver.

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 those services.

2 Summary Of Design Requirements

This section summarizes the essential requirements for designing a high-level resource management system for a smart space. Space permits only a brief overview; potential design issues are discussed in more detail in [8]. In particular, the needs for on-demand agent startup and "smart re-allocations" are motivated more extensively in [8].

2.1 Closed System Assumption

We assume that Rascal will work in a closed system, i.e. one where all agents can be assumed to be trusted (but where agents can appear or leave dynamically). We can make this assumption without reducing the scalability of the system by dividing agents into *societies.* An agent society is a collection of agents that act on behalf of a single entity, such as a physical space, a person, a group of people, an institution, an information store, etc. Rascal's job is to coordinate use of resources within a society.

In cases where agents from one society need to access resources owned by a different society, a resource manager of one society can make requests of the resource manager from the other one. The resource manager from the society that owns the resource is the sole owner of the resource and can decide to take it back at any moment if necessary. The negotiation for resources among a number of societies is a somewhat different problem from managing resources within a society. For one thing, this is now an open system and access control mechanisms need to be put in place to ensure that all requesters act within their authority. The inter-society resource management will be covered eslewhere.

A most common kind of situation where one society needs to make resource request of another is one in which agents acting on behalf of the user need resources to communicate information to the user. Agents acting on behalf of the user belong to one society and those controlling the space, belong to another (as in Figure 1). User's society usually will not contain physical devices and thus if, for example, an email alert agent acting on my behalf needs to tell me that

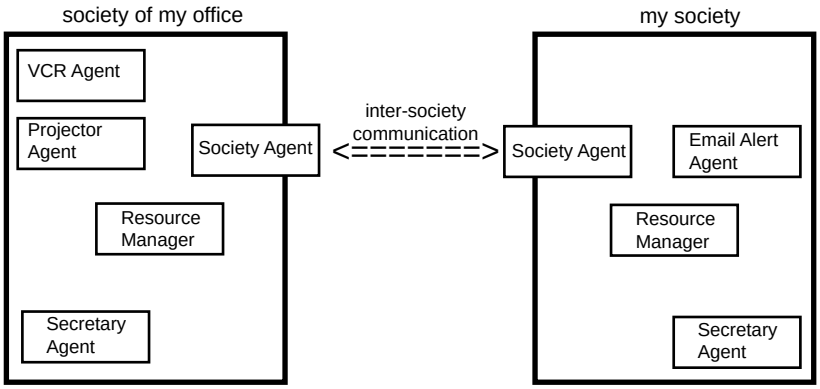

Fig. 1. Our view of the world: a society is viewed as a closed system. All agents within a society are trusted yet self-interested. A world composed of many societies is, on the other hand, viewed as an open system where, to an extent, every society can be viewed as a single agent providing a number of capabilities.

a new mail has arrived for me, it will need resources (such as speech output or a display) from my office to pass the message on to me. In such situation, the email alert agent will still make a resource request of the resource manager in my society. My resource manager, seeing that it has no good resources on its own, will make a request of the resource manager of my office's society. My office will then decide whether to fulfill the request and if so, how.

2.2 Self-interested Agents

Although we assume that all agents are trusted, we also assume that they are self-interested. All agents admitted to a society are assumed to be truthful in that they do not misrepresent their needs and capabilities. They are, however, concerned solely with performing the task or tasks they were built for. For example, an agent that controls the flow of a presentation is only concerned with ensuring that slides are visible on an appropriate display; it has no knowledge of how its actions affect the abilities of other agents to perform their tasks. The assumpiton that agents have no knowledge of their peers allows a more modular design of the system.

2.3 Tightly Tied To External Physical Resources

A special characteristic of a MAS based in a smart space, as noted before, is that it is very tightly coupled to the physical resources within that space. At the simplest level, the number of physical displays in the space is a limiting factor that determines how many visual activities (such as web browsing, sketching, watching movies, writing email) can be performed simultaneously. At a deeper level, the layout of physical connections among devices also limits their use. For

example, the output of a VCR may be connected to only one projector and a TV, while computers can be dynamically connected – via a multiplexor – to any of the available displays in the space. A resource management system, such as Rascal, is necessary to keep track of available resources and arbitrate among conflicting requests for those resources.

2.4 Reasoning About Absent Agents

In smart spaces components can be added or removed at any moment. More often than not, however, components that are available one day, are also available the next. We believe that our system should not only be able to cope with dynamic changes of configuration, but also that the stability and predictability of the physical environment should be used to the system's advantage. One consequence of a predictable enviroment is the plausibility of reasoning about agents even before they have been started. In other words, in smart spaces, agents can be started when needed using the resources that at a given moment can be spared. For example, when one of the users within a space needs to make a slide presentation, an appropriate agent will be started on an available computer that has the appropriate presentation software, available screen space, and can be connected to an on-wall display device (such as a projector). If another presentation is started at the same time, another available computer and display device will be chosen. On-demand agent startup allows the system to adapt to the current set of available resources and prevents the system designer from having to predict all possible configurations that might be required in a space (such as the unusual case where two presentations need to run simultaneously).

2.5 Need For Smart Re-allocations

In our system, it happens frequently that a new request can only be satisfied by taking a resource away from a previously satisfied request. But that previous request does not have to be left resource-less – there is often an alternative resource that can be used to fill it. Suppose, for example, that I request to watch the news in an office equipped with an on-wall projector and a TV set (see Figure 2). The projector is assigned to the job because it produces the largest image and has the best resolution. Then, while watching the news, I decide to also access my email agent. This agent must use the projector because it is the only display that can be used by a computer. Therefore, the projector is taken away from the news agent; ideally, instead of stopping the news agent, Rascal moves it to the TV set.

3 Building Rascal

3.1 Centralized Vs. Distributed

Conceptually, Rascal is a centralized system. This decision was not made lightly, but we believe the advantages of a centralized system outweight its drawbacks (such as, e.g., being a single point of failure).

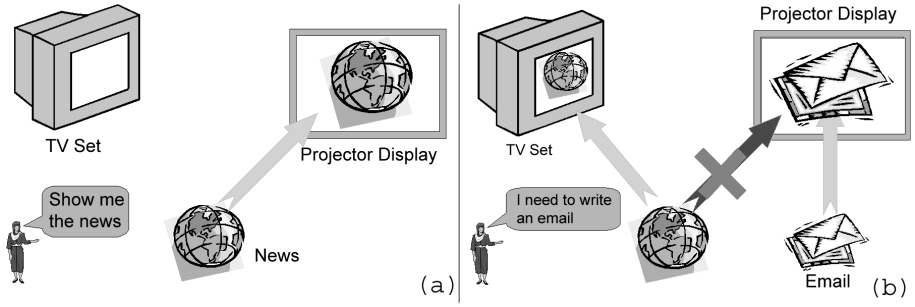

Fig. 2. Sample interaction. (a) user requests to see the news – on-wall projected display is allocated as the best resource for the task. (b) user accesses email; the only possible display for email is the on-wall projector previously allocated to the news agent. Instead of being stopped, the news agent is moved to a TV set.

Rascal was built as a separate centralized system primarily because it had to reason about absent agents. If we instead chose a distributed solution involving direct negotiation, all agents would have to be "alive" to be considered as candidates for a request. Also, a resource manager in an smart interactive, space has to be efficient. Rascal must make its decisions within a couple of seconds or less. A lot of inter-agent communication would make this goal nearly impossible.

Despite centralization, Rascal is actually not a single point of failure in the system. This is because two features of Metaglue make any agent nearly "invincible:" automatic restarting of agents and persistent storage ([6]). If any agent dies, it will be restarted the next time any other agent tries to make a call to it. The dead agent will be restarted on any available computer, so even if the original computer hosting the agent fails, the agent will be restarted somewhere else. The persistent storage mechanism allows agents to save changes to their state whenever such changes occur. Consequently, if the agent dies and gets restarted, it can retrieve its state from before the failure and continue as if nothing had happened.

3.2 Structure

Rascal performs two major functions: *service mapping* and *arbitration* among requests for services (as defined in Section 1.1) and it is composed of three major parts: the knowledge base, the constraint satisfaction engine, and the framework for interacting with other Metaglue agents.

Service mapping is performed entirely by the knowledge-based component of Rascal. Arbitration begins in the knowledge-based part (where relative cost and utility of various resources are determined) but most of the work on arbitration is done by the constraint satisfaction engine.

The components for interacting with the rest of the Metaglue agents facilitate communication with service providers and requesters, and enable enforcement of Rascal's decision (i.e., taking previously allocated services away from requesters).

In the following sections, we present these major components of the system.

Representation And The Knowledge Base Upon startup, information about all available resources is loaded into Rascal's knowledge base (if more resources become available later on, they can be added dynamically). It is important to reiterate here that Rascal relies on all resources having descriptions of their needs and capabilities separate from the actual code. Those external descriptions contain a list of services that the resource can provide. Agents who provide services may in addition specify what other resources they will need in order to provide that service. For example, the `MessengerAgent` that provides a message delivery service will need one or more resources capable of providing text output service. Agents may also specify their startup needs, i.e. a list of requests that need to be fulfilled for the agent to exist. For example, an agent providing speech recognition service will need a computer, with appropriate speech recognition software installed, in order to be able to start and configure itself properly.

When Rascal considers candidates for a request, it not only makes sure that those candidate services are adequate and available – it also makes sure that the needs of those candidates can be satisfied, and so on recursively. The final selection of candidates for requests is performed by the constraint satisfaction engine. Therefore, the knowledge-based part evaluates all possible candidates for all possible requests. This request chaining proves to be extremely valuable: when the email alert agent, for example, requests a text output service, several different agents may be considered, including the LED sign and the speech output agents. The email alert agent may have its own preference as to what kind of rendition of the text output service it prefers. However, if the communication link with the actual LED sign is broken, the needs of the agent controlling the LED sign will not be satisfied and so it will not be assigned to the request.

Rascal's knowledge base is implemented in a rule-based system (JESS [7]) written in Java. The role of this component of the system is to find all plausible candidates for all requests. In addition, for each request-candidate pair, a measure of utility has to be calculated (see the next section).

In terms of representation, services provided by agents are described by the names of the Java interfaces that can be used to access them. For services provided by entities other than agents, similar hierarchical names are used (e.g. `hardware.Computer` for a computer). In addition, attribute-value pairs are used to describe services in more detail and to refine requests.

Cost-Benefit Analysis 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 (e.g. [4]). In Rascal, self-assigned need levels are used in conjunction with the concept of utility of a service to the requester and its 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 Rascal, because of our assumption that all agents within a society can be trusted.

The basic assumption of this schema is that, given a request, each candidate resource has some utility to the requester. This utility depends on how badly the requester needs a particular request r fulfilled and on how well the resource s matches the request (Equation 1). A variety of monotonically increasing functions can be used as f_u.

$$utility(r, s) = f_u(need(r), match(r, s)) \quad (1)$$

The same method is used to calculate the utility of the already allocated resources. When a resource is taken from its current user, the system as a whole incurs cost equal to the utility of that resource to that user. Also, when a resource s_i, currently allocated to fulfill request r, is replaced with a different resource s_j, a cost is incurred. This cost is a sum of a fixed "change penalty" and the difference in utilities between the new allocation and the old one (if this difference is negative, it is set to zero) as shown in Equation 2.

$$cost(r, s_i, s_j) = changePenalty(r) + max\{0, utility(r, s_i) - utility(r, s_j)\} \quad (2)$$

The arbiter has to make sure that whenever it awards a resource to a new request, the cost of doing so should never exceed the utility of the awarded resources to the new requester.

Rascal provides a number of methods for calculating utilities and evaluating matches between requests and resources. Each resource or request description can also be accompanied by its own custom tools for performing those calculations.

Finding The Right Solution – The Constraint Satisfaction Engine When the knowledge-based subsystem selects and rates all candidates for requests, a constraint satisfaction engine (CSE) is invoked to find an optimal or nearly optimal configuration that fulfills the new request without breaking any of the previous assignments.

Rascal uses a Java-based CSE (JSolver [5]) in order to enable close coupling with its other components. In order to find the right solution, a number of constraints and heuristics are involved:

- respecting limits – there are limits on how many requests can share a service.
- only some requests need to be satisfied – CSE needs to find services only for some of the requests that it knows about: the newly made request, the needs of the services assigned to satisfy this new request and all the previously satisfied requests.
- preference to local solutions – As explained in Section 2.5, it is sometimes necessary to change the assignment to a previously satisfied request. However, it is necessary to minimize such changes to the absolute minimum.

Rascal's CSE has been set up in such a way that changes to old requests are only made as a last resort and have to be limited in scope. That is, it should not be possible for a new request to cause changes to a large number of other assignments. For that reason, Rascal's CSE uses following heuristics:

- the first service considered for any previously satisfied request is the service previously allocated to the request;
- if a different service has to be assigned, the cost of service substitution is calculated and added to the overall cost of the current new request – if the cost exceeds a preset limit, CSE backtracks;
- the CSE is run several times, each time with a different limit to the overall cost: the first time CSE runs, the limit is set to zero in hope that a solution can be found that does not disturb any of the previously assigned requests. If this fails, the CSE is run again with a higher limit. The process is repeated until a solution is found or until the CSE is ran with a limit equal to the need of this request. In Rascal, the cost of satisfying a request cannot exceed the need to have it satisfied.

Rascal-Metaglue Connection There are two major components to the Rascal-Metaglue connection mechanism: the `RascalAgent` and the `ManagedAgent`. The former makes Rascal's methods available to the rest of the Metaglue agents. The latter is a simple implementation of a Metaglue agent that all other "managed" agents inherit from. That is, all agents that want to make their services available through Rascal, or that wish to make requests through it.

4 Related Work

The Facilitator Agent in Open Agent Architecture (OAA) [9] performs *task* not *resource* management. Implicit in the OAA design is the assumption that each agent has sole control over all of the resources it might need.

Applications in Hive [10] agent platform 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 Rascal-enhanced Metaglue.

Jini [3] is a framework with a number of discovery and description tools but no arbitration capabilities. The arbitration component is supposed to be provided by the user.

Intentional Naming System (INS) [1] provides an extensive naming mechanism and a mechanism for choosing the best available service but it does not provide explicit arbitration mechanisms or tools for smart re-allocations.

5 Contributions

Multi-Agent Systems constitute a very powerful programming paradigm. Applying MAS to new domains often poses a number of challenges. This paper

shows how the MAS approach can be applied in the domain of smart spaces, where agent coordination is constrained by the availability of physical resources. Rascal—an implememnted and tested tool for managing such resources—is presented.

6 Acknowledgments

Luke Weisman did most of the fundamental work on the Rascal-Metaglue interface. He has also done a lot of research on other approaches to resource management in Metaglue-based MAS. Dr. Howard Shrobe and prof. Patrick Winston have provided invaluable advice throughout the duration of the project. Finally, the author would like to thank Mark Foltz for his thorough and insightful comments on this paper.

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. Stanislaw Ambroszkiewicz, Tomasz Nowak, Dariusz Mikulkowski, and Leszek Rozwadowski. A concep of agent language in agentspace. In *this volume*, 2001.
3. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Addison-Wesley, Reading, MA, 1999.
4. 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.
5. 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.
6. 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.
7. Ernest J. Friedman-Hill. Jess, the Java Expert System Shell. Technical Report SAND98-8206, Sandia National Laboratories, 1997.
8. Krzysztof Gajos, Luke Weisman, and Howard Shrobe. Design principles for resource management systems for intelligent spaces. In *Proceedings of The Second International Workshop on Self-Adaptive Software*, Budapest, Hungary, 2001. To appear.
9. 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.
10. 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.