Terraforming Cyberspace: Toward a Policy-Based Grid Infrastructure for Secure, Scalable, and Robust Execution of Java-Based Multi-Agent Systems

Jeffrey M. Bradshaw,[1] Niranjan Suri,[1] Martha Kahn,[2] Phil Sage,[2] Doyle Weishar,[2] and Renia Jeffers[1]

1. Institute for Human and Machine Cognition (IHMC)
40 South Alcaniz Street
Pensacola, FL 32501
850-202-4400
{jbradshaw, nsuri, rjeffers}@ai.uwf.edu

2. Global InfoTek, Inc. (GITI)
156 E. Maple Ave.
Vienna, VA 22180
703-319-3994
{mkahn, psage, doyle}@globalinfotek.com

ABSTRACT

The CoABS Grid arguably provides the most successful and widely used infrastructure for the large-scale integration of heterogeneous agent frameworks with object-based applications, and legacy systems. In this paper we describe how we are extending Grid capabilities by integrating the NOMADS agent environment for strong mobility and safe execution and the KAoS framework for policy-based management of agent domains to support long-lived agents and their communities.

Keywords

Multi-agent systems, frameworks, infrastructure, policy, domains, Java, security, resource management, grid, mobility

INTRODUCTION

During the 1940s, under the pseudonym of Will Stewart, Jack Williamson published a series of fictional stories describing a process for attaching atmospheres to planets in order to make them capable of sustaining life. 'Terraforming,' the term he coined for this activity was first picked up by other science fiction writers. Eventually, it captured the imagination of a small but zealous core of scientists, space advocacy groups, and segments of the public who began focusing on Mars as the most likely target for transformation and eventual colonization. The May 1991 issue of Life Magazine ran a cover story describing a 150-year plan for a Martian metamorphosis through orbiting solar reflectors that would melt polar water, surface factories that would produce needed gases in the atmosphere, and the ultimate planting of hearty plant species as the temperature approached the freezing point of water. Today many articles, books, and Web sites continue to develop the theme.

Cyberspace is currently a lonely, dangerous, and relatively impoverished place for agents [1]. Consequently, most of today's agents are designed for short insignificant lives and a small and relatively static world. Though promoted as collaborative, agents do not easily sustain rich long-term peer-to-peer relationships, let alone any semblance of meaningful community involvement. While their features for secure reliable interaction are often touted, there is no social safety net to help agents out when they get stuck, or worse yet to prevent them from setting the network on fire when they go off the deep end. Despite the fact that agent designers want them to communicate at an "almost human" level, agents are cut off from most of the world in which humans operate. Though capable of self-directed mobility, they are hobbled by severe practical restrictions on when and where they can go. Ostensibly endowed with autonomy, an agent's very existence can be terminated unceremoniously by the first passerby who happens to find the power switch.

In short, the kinds of agents that we want—full-fledged citizens of the wired world, equipped with their own stamped passports and Berlitz traveler's guides explaining foreign phrases and places that allow them to hail, meet, and greet comrades of any sort in an open networked landscape and, if not able to team up on a project, at least able to ask intelligibly for directions—these kinds of agents, alas, exist today only in our imaginations (and, of course, in the vision sections of our research proposals).

Fortunately, the basic infrastructure on which we can build the solutions to these problems is becoming more available. Designed from the ground up to exploit next-generation Internet capabilities, grid-based approaches aim to provide a universal source of dynamically pluggable, pervasive, and dependable computing power, while guaranteeing levels of security and quality of service that will make new classes of applications possible [7]. By the time these approaches become mainstream for large-scale applications, they will also have migrated to ad hoc local networks of very small devices [8].

The CoABS Grid (hereafter referred to simply as the "Grid"), developed at Global InfoTek (GITI) under DARPA's Control of Agent-Based Systems (CoABS) program, arguably provides the most successful and widely used infrastructure to date for the large-scale integration of heterogeneous agent frameworks with object-based applications, and legacy systems [13; 14]. Based on Sun's Jini services, it includes a method-based application programming interface to register and advertise capabilities, discover services based on capabilities, and provide the necessary communication between services. Systems and components on the Grid can be added and upgraded without reconfiguration of the network. Failed or unavailable components are automatically purged from the registry and discovery of similar services and functionality provides fail over. In addition to its use to integrate components produced by some two dozen independent CoABS projects and large-scale joint experiments, it has been used successfully in a series of naval Fleet Battle Exercises (FBEs). It is also being used by the Army Communications and Electronics Command, the Air Mobility Command, and in the Air Force's Joint Battlespace Infosphere project. A bridge to the Grid was built for DARPA's agent-based Advanced Logistics Program (ALP).

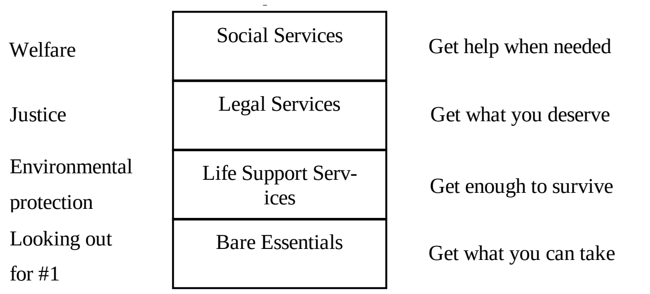
However, we must go far beyond current Grid capabilities to enable the vision of terraforming cyberspace (Figure 1). Current infrastructures typically provide few resource guarantees and no incentives for agents and other components to look beyond their own selfish interests. At a minimum, future infrastructures must go beyond the *bare essentials* to provide pervasive *life support services* (relying on mechanisms such as orthogonal persistence and strong mobility [18; 19]) that help ensure the survival of agents that are designed to live for many years. Beyond the basics of individual agent protection, long-lived agent communities will depend on *legal services*, based on explicit policies, to ensure their rights and help them fulfill their obligations [4; 9]. Benevolent *social services* will also eventually be provided to offer help when needed. Although some of these capabilities exist in embryo within specific agent systems, their scope and effectiveness has been limited by the lack of underlying support at the platform level.

In this paper we describe how we are working toward extending the Grid to provide support for rudimentary initial “terraforming” services. We will first describe the current Grid implementation. Then we will show how we are adapting and exploiting capabilities of the NOMADS and KAoS agent frameworks to provide Grid life support, legal, and social services.

THE COABS AGENT GRID

The Grid is built using the Sun Microsystems’ Jini services. The robust and dynamic nature of the Grid is derived from Jini. Grid software is written in Java and uses Java Remote Method Invocation (RMI) as a default for communication among Grid components and for transport of agent messages represented in the agent’s language of choice. The Grid runs on Unix, Linux, Windows NT/2000, and Mac OSX operating systems. Although many of the agent systems using the Grid are written in Java, members of the research community have created proxies that integrate the Grid with agent systems written in C++, Lisp, Prolog, SOAR and the Palm Pilot KVM. Legacy systems written in other programming languages can also be easily integrated with the Grid using the Java Native Interface. With respect to our incorporation of Jini services, we had two complementary design objectives: 1. to make the use of Jini services easier for beginning developers, and 2. to intentionally expose advanced Jini features for sophisticated developers In this way, basic and Grid-enhanced Jini features are made available to a wider community than ever before.

The Grid supports a wide variety of applications, from simple monitoring and information retrieval to complex, dynamic domains such as military command and control. Using the Grid, agents and wrapped legacy systems can (1) describe their needs, capabilities and interfaces to other agents and legacy systems; (2) find and work with other agent components and legacy systems to accomplish complex tasks in flexible teams; (3) interact with humans and other agents to accept tasking and present results, and (4) adapt to changes in the application domain, the task at hand, or the computing environment. The Grid does this by providing access to shared policies and ontologies, mechanisms for describing agents’ capabilities and needs, and services that support interoperability among agents and legacy systems with simple or rich levels of semantics—all distributed across a network infrastructure.



Although most agent frameworks provide some of the interoperability and other services that the Grid provides, each framework typically supports specialized constructs, communication, and control mechanisms. This specialization is desirable because particular systems can use mechanisms appropriate to the problem domain/task to be solved. The Grid is *not* intended to replace current agent frameworks but rather to augment their capabilities with services supporting *trans-architecture teams*. Agent technologies support semantically rich conversations among these agents (and wrapped legacy systems), which allow them to interoperate with agents outside their “community”. An analogue is the Internet’s bridging of heterogeneous networks by gateways and protocols. Programmers will make their components “Grid Aware”, much as many network applications are now made “Internet ready” or “Web ready” by supporting protocols and languages such as TCP/IP, HTTP, HTML, and XML. Furthermore, programmers will *want* to make their components “Grid Aware” to enable them to participate in dynamic teams that leverage other components discovered at runtime.

After discovering needed services and applications, an agent or an application does not need to use the Grid communications services: communication can be established point-to-point. For this reason, it scales to a large number of agents with no restrictions beyond those imposed by network bandwidth. Agent registration and discovery, on the other hand, are reliant on one or more lookup services.

The Grid takes advantage of three important components of Jini™:

1. the Jini™ concept of a service, which is used to represent an agent,
2. the Jini™ Lookup Service (LUS), which is used to register and discover agents and other services, and
3. Jini™ Entries, which are used to advertise the capabilities of an agent or service.

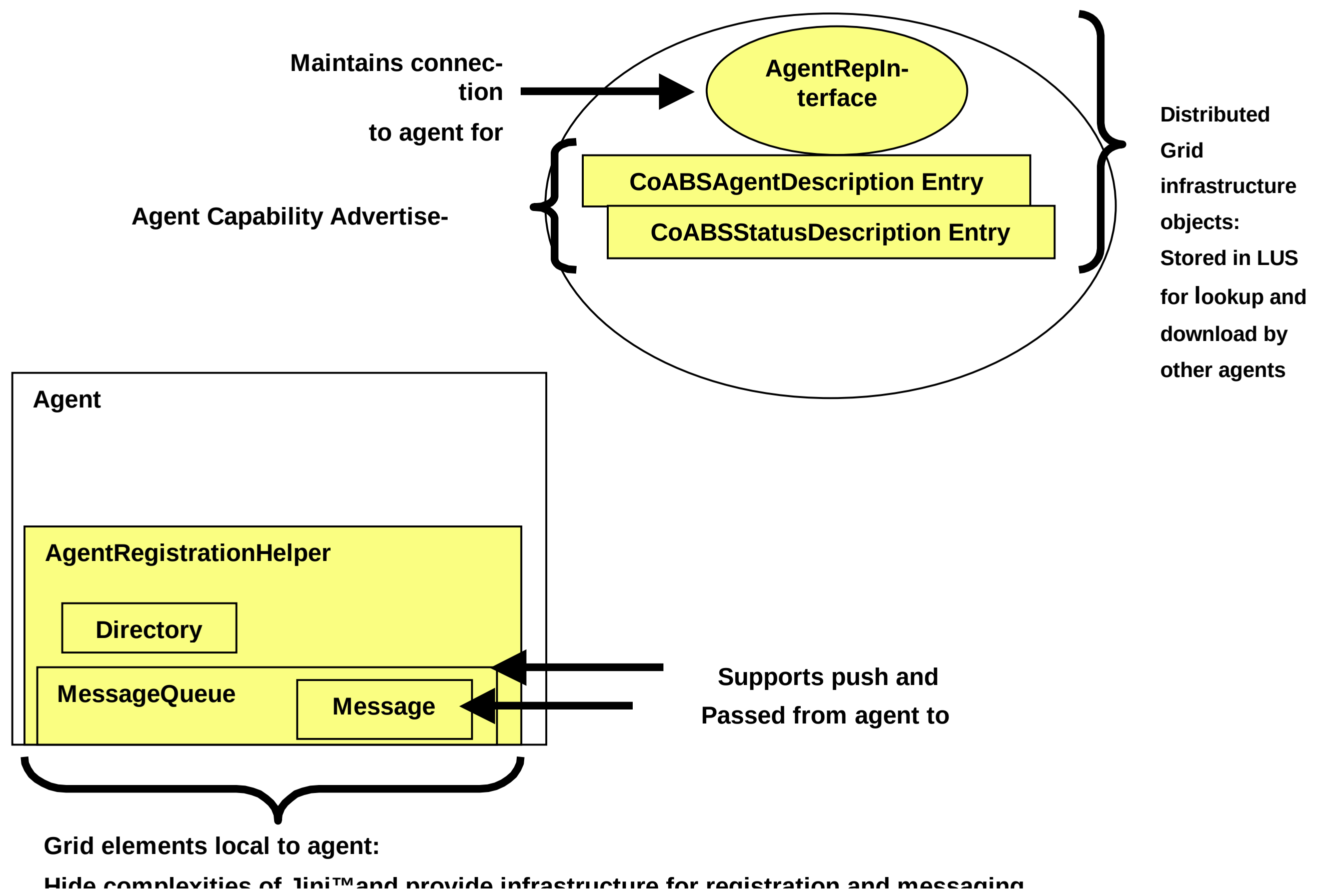
A Jini™ service is a Java object that is serialized and stored in the LUS. The LUS supports lookup of services based on type, attribute values, and unique identifier. When a Jini™ client performs a lookup through the LUS, the service object is returned to the client. The service may optionally be a proxy that uses a remote connection to communicate back to the true service at a different location. The remote connection is transparent to the client and can be of any type, e.g. RMI, CORBA, or secure socket. An important property of Jini services is that they are described by service interfaces. Thus, clients do not need to have local knowledge of how a service is implemented, but only local knowledge of the interface. The interface describes the semantics of the service. The actual code for the service implementation is downloaded to the client on an as-needed basis. This is what allows Jini services to be installed without local configuration.

The LUS grants leases to registered services, assigns globally unique identifiers to services, and supports lookup of services. It is the service’s responsibility to maintain its lease with the LUS, however Jini™ provides helper classes to do this automatically.

If a service can’t maintain it’s lease because of either failure of the service or failure of the network connection between the service and the LUS, the service will be purged from the LUS, so that the LUS contents remain current. LUS security is also a concern. Secure LUS products are offered by some commercial companies.

Jini™ provides helper classes that use a multicast protocol to find any LUSs that are running within a local area network. No prior knowledge of the machine name or port that the LUS is running on is required. Jini™ provides a unicast protocol to find LUSs outside the local area network. Service registration is maintained in all local and distant LUS. The registration is automatically propagated to any new LUS processes that are started. Multiple LUSs can be run for robustness and scalability. If one goes down, the others will still maintain registration and lookup. A sample LUS is provided in the Jini™ Development Kit and is currently used by the Grid.

Jini™ services are described in the form of a Jini™ Entry. An Entry is a collection of service attributes that is stored in the LUS along with the service. Many Entries can be stored for a single service. An Entry is an object that has public fields, cannot contain primitive types, and has a no-parameter constructor. Any Serializable object that meets these criteria can be an Entry as long as it implements the marker Jini™ Entry interface. Entry templates are used in Jini™ and Grid lookup methods to match registered services. A null Entry field is a wildcard. Non-null fields are used for exact matching. Entry templates and service types can be used to filter the number of services that are downloaded from a LUS over the network. Predicates can than be used local to the client to further restrict the number of services returned.


Maintains connection
to agent for
AgentRepInterface
Agent Capability Advertise-
CoABSAgentDescription Entry
CoABSStatusDescription Entry
Distributed Grid infrastructure objects: Stored in LUS for lookup and download by other agents
Agent
AgentRegistrationHelper
Directory
MessageQueue
Message
Supports push and
Passed from agent to
Grid elements local to agent:
Hide complexities of Jini™and provide infrastructure for registration and messaging

The Grid uses Jini™ Entries for agent or service capability advertisements. Currently, the Grid provides three classes that implement the Entry interface, though more are being added and Grid users are encouraged to add new ones that are relevant to their applications. The CoABSAgentDescription Entry has fields for agent name, description, organization, architecture, ontologies, content languages, display icon URL, documentation URL, and unique ID. The CoABSStatusDescription Entry is used to advertise agent performance and status information. The Location Entry has fields ranging from room number to latitude and longitude. The user fills in only those attributes that are relevant to a particular agent or service.

The Grid provides both local and distributed components as shown in Figure 2. The Grid provides AgentRegistrationHelper utility classes that are local to an agent and that hide the complexity of Jini™. These classes automatically find any LUS in both the local area network and user-designated distant machines. The Grid supports agent and service discovery based on Jini™ Entries and arbitrary predicates as well as by service type. The Grid also provides event notification when agents register, unregister, or change their advertised attributes. The Grid defines a Jini™ service interface called the AgentRepInteface, which is a proxy to the agent. This proxy is distributed to clients throughout the network. The AgentRepInteface interface defines a method called addMessage(), which uses a remote connection to deliver a message back to the agent. Thus, when a client agent calls a Grid lookup method, a proxy that allows immediate direct communication back to the agent is returned. The client agent can include its own AgentRepInterface in the message it delivers, so that two-way communication can be established with no further lookup. The Grid is transport neutral in terms of agent communication. The Grid defines the interface, but the agent proxy is free to use any transport.

The Grid currently provides an AgentRep implementation that uses RMI for message transport. Other transport mechanisms are anticipated. An AgentRep downloaded to a client is connected to a MessageQueue object local to the agent using RMI. A MessageListener interface is also defined to allow agents automatic notification of incoming messages. Several classes of Grid messages are provided. Some include text messages only, while others allow data attachments. The Grid is language neutral; any agent communication language can be used. It is up to the communicating agents to decipher the contents. The Grid also provides methods to send a message to a group of agents matching a particular template or satisfying a particular predicate.

Agent communication is fully distributed, in that each agent sending a message communicates directly with the receiver, using the proxy registered by the receiver. The sender is unaware of the transport mechanism being used, although currently RMI is the default. Thus, as the number of agents on the Grid increases, agent communication performance is only affected by the distribution of the agents in the network and the network bandwidth.

Although agent-to-agent communication is peer-to-peer, and thus scales to a large number of agents with no restrictions beyond those imposed by network bandwidth, agent registration and discovery are reliant on one or more Jini™ lookup services. Initial experiments of sequential registration and lookup have found no degradation of performance with up to 10,000 agents registered.

To address security concerns secure versions of the AgentRep and ServiceRep are being developed. The secure AgentRep or ServiceRep will act as a privileged-code wrapper between the client and an agent or service written to take advantage of Jini. What makes this approach possible is that the Rep is provided by the service and is thus under the control of the service – not the client. The secure Rep will be able to authenticate the client. It will also be able to associate service method calls with the client. Finally, based on the basic NOMADS and KAoS capabilities described later on in the paper, it will be able to enforce and dynamically effect changes of security policy of arbitrary complexity, for components that are running in any combination of standard Java and Aroma VM's. The secure Rep will be tied to the KAoS Policy-based Administration Tool (KPAT), Domain Managaer, and Guards described later in the paper to provide these capabilities.

The ability to place limits on the communication between the client and service is especially important in the case of hand-held devices or phones. The Jini™ community is in the process of developing a Jini Surrogate Architecture for devices that aren't able to currently run Jini. One of the premises behind the development of this architecture is that even if Jini could run on a small device, you would still need to protect the device from being overwhelmed by data from a client, due to memory constraints. The secure Rep mechanisms proposed here would be an alternate way to protect such small devices. Denial-of-service and data overload would be prevented at the client side.

NOMADS LIFE SUPPORT SERVICES

Java is currently the most popular and arguably the most mobility-minded and security-conscious mainstream language for agent development. However, current versions fail to address many of the unique challenges posed by agent software. Security services for Jini are also similarly limited, although there is an initiative underway for the adoption of a secure version of RMI. While few if any requirements for Java mobility, security, and resource management are entirely unique to agent software, typical approaches used in non-agent software are usually hard-coded and do not allow the degree of on-demand responsiveness, configurability, extensibility, and fine-grained control required by agent-based systems.

We are interested in particular in extending the Grid with basic *life support services* that will provide environmental protection for agents that need:

- guaranteed availability of some quantity of system resources, even in the face of buggy agents or denial-of- service attacks;
- protection of agent execution state, even in the face of unanticipated system failure.

Given the current limitations of Java, such protection cannot be provided by merely bolting on new services on top of the Grid; it must be built directly into the Java Virtual Machine.

Our approach for life support services has thus far been two-pronged. For standard Java Virtual Machines (VMs), we create software-based Guards, which enforce policies by relying on the capabilities of the Java 2 security model (including permissions and privileged code wrappers) and the Java Authentication and Authorization Service (JAAS). In contrast to other implementations of Java security, our enhanced JAAS-based approach allow revocation of access permissions under many circumstances as well as the granting of different permissions to different instances of agents from the same code base. For the Aroma VM, we can create a guarded environment that is considerably more powerful in that it not only provides the capabilities described above, but also supports access revocation under all circumstances, dynamic resource control and full state capture on demand for any Java agent or service. NOMADS is the name we have given to the combination of IHMC's Aroma VM with its Oasis agent execution environment [18; 19].

To understand the features of Aroma and NOMADS, some understanding of the current Java security model is needed. Early versions of Java relied on the sandbox model to protect mobile code from accessing dangerous methods. In contrast, the security model in the current Java 2 release is *permission-based*. Unlike the previous "all or nothing" approach, Java applets and applications can be given varying amounts of access to system resources.

Unfortunately, current Java mechanisms do not address the problem of resource control. For example, while it may be possible to prevent a Java program from writing to any directory except /tmp (an *access* control issue), once the program is given permission to write to the /tmp directory, no further restrictions are placed on the program's I/O (a *resource* control issue). As another example, there is no way in the current Java implementation to limit the amount of disk space the program may use or to control the rate at which the program is allowed to read and write from the network.

Resource control is important for several reasons. First, without resource control, systems and networks are open to denial of service attacks through resource overuse. Second, resource control lays the foundation for quality-of-service guarantees. Before any quality-of-service guarantees can be made about the availability of resources, the system must be able to limit resource utilization of other tasks (which is currently not possible in the Java environment). Third, resource control presupposes resource accounting, which allows the resources consumed by some component of a system (or the overall system) to be measured for either billing or monitoring purposes. Monitoring resource utilization over time allows the detection of abnormal behavior as part of the system.

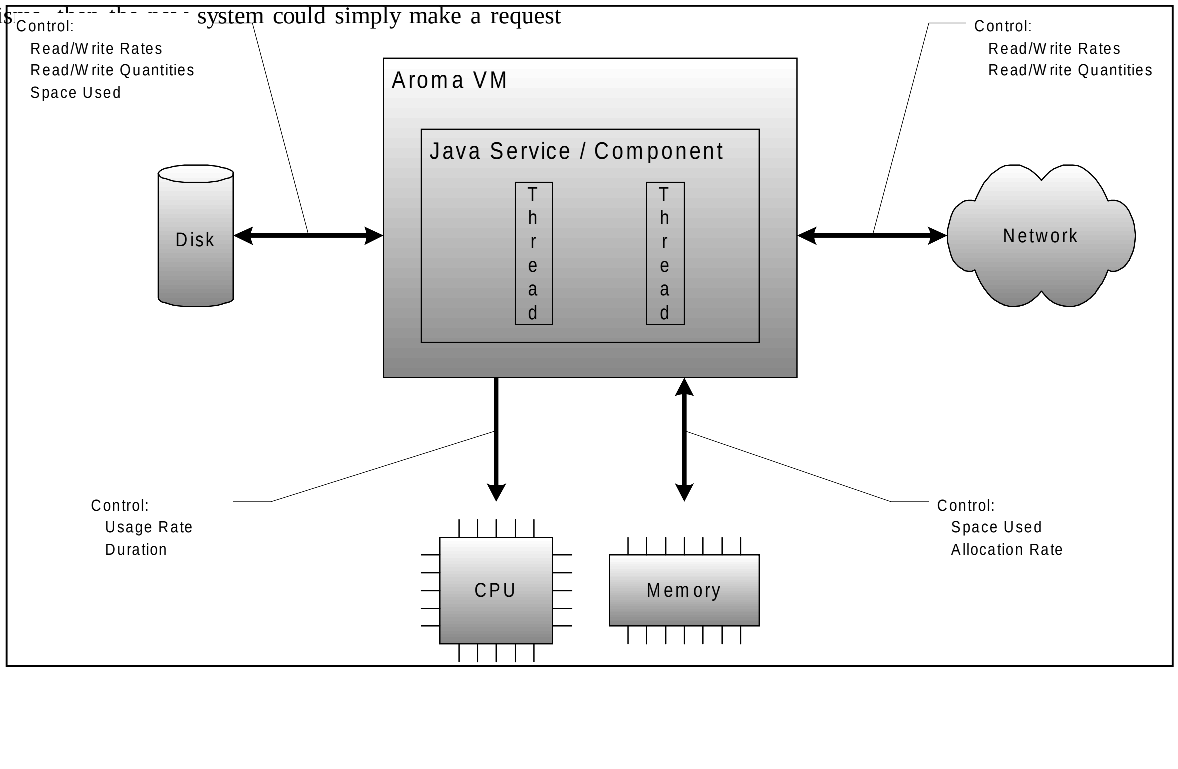
Finally, the availability of resource control mechanisms in the environment simplifies the task of developing systems for resource-constrained situations. Consider the task of developing and deploying a new system requiring concurrent execution and resource sharing with existing systems. In such scenarios, the developer of the new system often has to limit the resource utilization of the new system in order to not interfere with the operations of the existing systems (for example, maybe the new system can only use 500 Kb/sec of network bandwidth because the rest of the available network bandwidth is required by the existing systems). Providing such a guarantee requires significant effort on behalf of the developer of the new system. However, if the underlying environment were to provide resource control mechanisms, then the new system could simply make a request

nisms, then the new system could simply make a request to the underlying environment, which can then provide the necessary guarantees.

Aroma currently provides a comprehensive set of resource controls for CPU, disk, and network (Figure 3). The resource control mechanisms allow limits to be placed on both the rate and quantity of resources used by Java threads. Rate limits include CPU usage, disk read rate, disk write rate, network read rate and network write rate. Rate limits for I/O are specified in bytes/millisecond. Quantity limits include disk space, total bytes written to disk, total bytes read from the disk, total bytes written to the network, and total bytes read from the network. Quantity limits are specified in bytes. One of the major benefits of the Aroma VM is that resource controls are transparent to the Java code executing inside the VM. In particular, the enforcement of the resource limits does not require any modifications to the Java code. Also, the existence of rate limits (and their enforcement) is completely transparent to the Java component or service.

CPU resource control was designed to support two alternative means of expressing the resource limits. The first alternative is to express the limit in terms of bytecodes executed per millisecond. The advantage of expressing a limit in terms of bytecodes per unit time is that given the processing requirements of a thread, the thread's execution time (or time to complete a task) may be predicted. Another advantage of expressing limits in terms of bytecodes per unit time is that the limit is system and architecture independent. The second alternative is to express the limit in terms of some percentage of CPU time, expressed as a number between 0 and 100. Expressing limits as a percentage of overall CPU time on a host provides better control over resource consumption on that particular host.

Rate limits for disk and network are expressed in terms of bytes read or written per millisecond. If a rate limit is in effect, then I/O operations are transparently delayed if necessary until such time that allowing the operation would not exceed the limit.



Threads performing I/O operations will not be aware of any resource limits in place unless they choose to query the VM.

Quantity limits for disk and network are expressed in terms of bytes. If a quantity limit is in effect, then the VM throws an exception when a thread requests an I/O operation that would result in the limit being exceeded.

Within the Grid, a number of uses of the NOMADS-based resource control mechanisms are possible. First, the VM-level Guard will be able to utilize the resource control capabilities in order to place limits on the resources consumed by services and components running within the Aroma VM. The Guard will be able to vary the resource limits to accommodate changes in policy or level of service guarantees. The Guard will also be able to take advantage of the resource accounting capabilities to measure and report back on the resources consumed by services and components and to look for patterns of resource abuse that might signal denial-of-service attacks.

With respect to protection of agent state, we need a way to save the entire state of the running agent or component, including its execution stack, anytime so it can be fully restored in case of system failure or a need to temporarily suspend its execution. The standard term describing this process is checkpointing. Over the last few years, the more general concept of transparent persistence (sometimes called “orthogonal persistence”) has also been developed by researchers at Sun Microsystems and elsewhere [12]. The goal of this research is to define language-independent principles and language-specific mechanisms by which persistence can be made available for all data, irrespective of type. Ideally, the approach would not require any special work by the programmer (e.g., implementing serialization methods in Java or using transaction interfaces in conjunction with object databases), and there would be no distinction made between short-lived and long-lived data.

We have used the state capture features of NOMADS extensively for agents requiring anytime mobility, whether in the performance of some task or for immediate escape from a host under attack or about to go down (we call this scenario “scram”). We have also put these features to use for transparent load-balancing and forced code migration on demand in distributed computing applications. To support transparent persistence for agents and components on the Grid, we are implementing scheduled and on-demand checkpointing services that will protect agent execution state, even in the face of unanticipated system failure.

KAoS LEGAL AND SOCIAL SERVICES

Terraforming cyberspace involves more than regulation of computing resources and protection of agent state. As the scale and sophistication of agents grow, and their lifespan becomes longer, agent developers and users will want the ability to express complex high-level constraints on agent behavior within a given environment. It seems inevitable that productive interaction between agents in long-lived communities will also require some kind of *legal services*, based on explicit enforceable policies, to ensure their rights and help them fulfill their obligations. Over time, it seems likely that benevolent *social services* will also eventually evolve to offer help with individual agent or systemic problems.

In both legal and social services, it is clear that preventive initiatives are nearly always superior to after-the-fact remedies, as the following verse by Joseph Malins illustrates:

> ‘Twas a dangerous cliff, as they freely confessed,
>
> Though to walk near its crest was so pleasant;
>
> But over its terrible edge there had slipped
>
> A duke and full many a peasant.
>
> So the people said something would have to be done,
>
> But their project did not at all tally;
>
> Some said, “Put a fence around the edge of the cliff,”
>
> Some, “An ambulance down in the valley.”

We are basing our approach on the assumption that preventive policy-based ‘fences’ can complement and enhance after-the-fact remedial ‘ambulance in the valley’ mechanisms. The policies governing some set of agents aim to describe expected behavior in sufficient detail that deviations can be easily anticipated or detected. At the same time, related policy support services help make compliance as easy as possible. Complementing these policy support services, various enforcement mechanisms operate as a sort of ‘cop at the top of the cliff’ to warn of potential problems before they occur. When, despite all precautions, an accident happens remedial services are called as a last resort to help repair the damage. In this manner, the policy-based fences and the after-the-fact ambulances work together to ensure a safer environment for individual agents and the communities in which they operate.

Policy-based approaches have grown considerably in popularity over the last few years. Unlike previous versions, the Java 2 security model defines security policies as distinct from implementation mechanism. Access to resources is controlled by a Security Manager, which relies on a security policy object to dictate whether class X has permission to access system resource Y. The policies themselves are expressed in a persistent format such as text so they can be viewed and edited by any tools that support the policy syntax specification. This approach allows policies to be configurable, and relatively more flexible, fine-grained, and extensible. Developers of applications no longer have to subclass the Security Manager and hard-code the application’s policies into the subclass. Programs can make use of the policy file and the extensible permission object to build an application whose security policy can change without requiring changes in source code.

The basic policytool Java currently provides, assists users in editing policy files. However, to be useful and usable in realistic settings, policy-based administration tools should contain domain knowledge and conceptual abstractions to allow applications designers to focus their attention more on high-level policy intent than on the details of implementation. Moreover, while Java provides only for static policies, critical agent applications will require tools for the monitoring, visualization, and dynamic modification of policies at runtime.

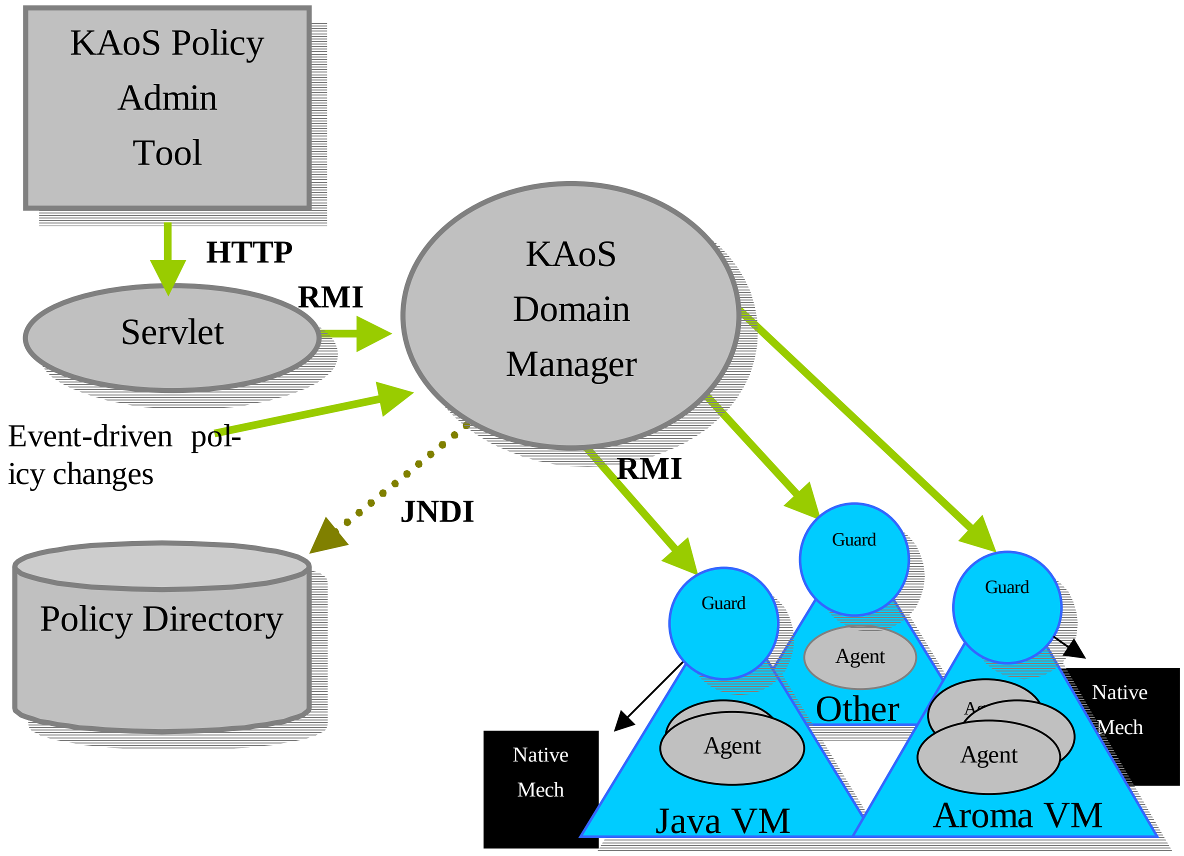
In principle, a variety of languages can be used to express policies. At one extreme they may be written in some propositional or constraint language. At the other extreme are a wide variety of simpler schemes, each of which gives up some types of expressivity. The choice of language for a particular application is affected by considerations of composability, computability, efficiency, expressivity, and amenability to the detection of equivalence and the discovery of conflicts. With funding from the DARPA CoABS program, we have begun the development of an implementation-neutral policy language expressions in DAML (DARPA Agent Markup Language, http://www.daml.org), which will used to represent both simple atomic policies (e.g., Java permissions) and complex constructions.

In KAoS, we currently distinguish between two kinds of policy constraints: those relating to *permissions* and those relating to *obligations*. These policies are often related: by entering into particular obligations an agent or component may acquire specific permissions; and vice versa: when an agent is given permission to access a shared resource, it may incur obligations as a result. We refer to the binding of a particular set of policies with a given set of agents or components as an *agreement*.

Our concept of policy-based management of agents extends beyond typical security concerns. For example, KAoS pioneered the concept of agent conversation policies [3; 9]. Teams of agents can be formed, maintained, and disbanded through the process of agent-to-agent communication using an appropriate semantics [5; 6; 20]. Conversation policies assure coherence in the adoption and discharge of team commitments by heterogeneous agents of different levels of sophistication [3; 4]. These conversation policies are designed to assure robust behavior and to keep computational overhead for team maintenance to an absolute minimum [9; 11; 17]. As a generalization of this work on conversation policies, we have demonstrated links between KAoS, NOMADS, and Java security mechanisms for access control and resource management (http:// www.aiai.ed.ac.uk /project/ coax/). Development of libraries of policy and enforcement mechanisms for mobility management [15], registration management, and various forms of obligation management are also underway.

Groups of agents are structured into KAoS domains to facilitate policy administration. A given domain can extend across host boundaries and, conversely, multiple domains can exist concurrently on the same host. Policies can be scoped variously to individual agent instances, agents of a given class, agents running in a given instance of a platform (e.g., a single Java VM), or agents in a given domain or subdomain. The policy language and KAoS management and administration tools described below are intended to work identically across different execution environments (e.g., Java VM, Aroma VM, and potentially non-Java environments), however Guards, which enforce policies, are necessarily designed for a specific execution environment (which we will call for our purposes a platform). Our approach enables policy uniformity across multiple platforms and hosts, as long as semantically equivalent monitoring and enforcement mechanisms are available as part of those platforms and hosts. Under these conditions, it follows that behavior of agents written in different frameworks and running in different languages and platforms and on different hosts can be kept consistent through the use of these policy-based mechanisms.

The KAoS Policy Administration Tool (KPAT), a graphical user interface to domain management functionality, has been developed to make policy specification, revision, and application easier for administrators without specialized training. Figure 4 shows how KPAT interacts with other components. Using KPAT, an authorized user may make changes over the Web to agent policy using a secure http connection. Alternatively, trusted authenticated components (such as Guards) may propose policy changes autonomously based on their observation of system events.



The KAoS Domain Manager serves as a policy decision point to determine whether agents can join their domain and for policy conflict resolution. The KAoS Domain Manager is responsible for ensuring policy consistency at all levels of a domain hierarchy, for notifying Guards in the event of a policy change, and for storing policies in the repository.

Policies are stored in an implementation-neutral format, currently very simple but soon to be based on our DAML policy representation. Available in a secure library repository such as an LDAP directory, we intend to allow these policies to be accessed by authorized entities in accordance with policy disclosure strategies [16]. For example, agents may need to understand domain policies in advance of submitting a registration request to a new domain. Because the policies in the library are expressed declaratively, they can be analyzed and verified in advance and offline, permitting execution mechanisms to be as efficient as possible.

The various VM-level Guards interpret these abstract policies and enforce them with appropriate native mechanisms as described previously.

Analogous to the AgentRegistrationHelper utility classes currently provided by the Grid, KAoSAgentRegistrationHelper utility classes are under development. Unlike the current aproach, which requires each domain-enabled agent to be wrapped as a KAoS agent, the KaoSAgentRegistrationHelper will allow domain management functionality to be made available to any Grid-ready agent or component framework with little or no modification required to the agent itself.

The combination of the use of libraries of pre-analyzed policy sets, separate policy decision and conflict resolution mechanisms, and efficient policy enforcement mechanisms make the use of policy-based administration tools maximally effective and performant. A policy-based approach has the additional advantages of *reusability, efficiency, context sensitivity,* and *verifiability*:

Reusability. Policies encode sets of useful constraints on agent or component behavior, packaging them in a form where they can be easily reused as the occasion requires. By reusing policies when they apply, we reap the lessons learned from previous analysis and experience while saving ourselves the time it would have taken to reinvent them from scratch.

*Efficiency***.** In addition to lightening the application developers' workload, explicit policies can sometimes increase runtime efficiency. For example, to the extent that policy conflict resolution and conversion of policy to a form that can be used by appropriate enforcement mechanisms can take place in advance, overall performance can be increased.

Context-sensitivity. Explicit policy representation improves the ability of agents, components, and platforms to be responsive to changing conditions, and if necessary reason about the implications of the policies which govern their behavior.

Verifiability. By representing policies in an explicitly declarative form instead of burying them in the implementation code, we can better support important types of policy analysis. First—and this is absolutely critical for security policies—we can externally validate that the policies are sufficient for the application's tasks, and we can bring both automated theorem-provers and human expertise to this task. Second, there are methods to ensure that program behavior which follows the policy will also satisfy many of the important properties of reactive systems: liveness, recurrence, safety invariants, and so forth. Finally, with explicit policies governing different types of agent behavior, we can predict how policies may compose.

FUTURE DIRECTIONS

As we develop and enhance the agent infrastructure described in this paper, we plan to leverage our contacts and experience to engage the open-source community in continued collaboration to accelerate enhancements to these technologies and make them available more widely than ever before. To this end will also continue involvement in the Jini community, and in related efforts in FIPA, the OMG, and Java standards and commercialization efforts.

Tomorrow's world will be filled with agents embedded everywhere in the places and things around us. Providing a pervasive web of sensors and effectors, teams of such agents will function as cognitive prostheses—computational systems that leverage and extend human intellectual, perceptual, and collaborative capacities, just as the steam shovel was a sort of muscular prosthesis or the eyeglass a sort of visual prosthesis. Thus the focus of AI research is destined to shift from Artificial Intelligence to Augmented Intelligence [2; 10].

Once we have terraformed cyberspace, agents will be freed from their current role as short-lived visitors on the wire to permanent colonists in a virtual world where we can feel comfortable with not knowing or caring exactly where they are being physically hosted. They will truly live among us and we will wonder how we ever lived without them.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of the DARPA CoABS and Ultra*Log Programs, the NASA Cross-Enterprise and Intelligent Systems Programs, the National Technology Alliance, and The Boeing Company while preparing this paper. Thanks also to many other colleagues who have assisted in this work, including David Allsopp, Patrick Beautement, Maggie Breedy, Alberto Canas, Todd Carrico, Rob Cranfill, Ken Ford, Mark Greaves, Pat Hayes, Jim Hendler, Greg Hill, Robert Hoffman, Heather Holmback, Wayne Jansen, Mike Kerstetter, Mike Kirton, Gerald Knoll, James Lott, Thomas Reichherzer, Austin Tate, Alex Wong, and members of the DARPA CoABS Coalition Operations Experiement (CoAX), the Java Agent Services Expert Group (JSR 87) and the FIPA Abstract Architecture Working Group.

REFERENCES

[1] Bradshaw, J. M. (2001). Steps toward the permanent colonization of cyberspace. In J. M. Bradshaw (Ed.), *Handbook of Agent Technology*. (pp. in preparation). Cambridge, MA: AAAI Press/The MIT Press.

[2] Bradshaw, J. M., Canas, A., Ford, K. M., Hayes, P., Hoffman, R., & Suri, N. (2001). Terraforming cyberspace. *IEEE Computer*, in press.

[3] Bradshaw, J. M., Dutfield, S., Benoit, P., & Woolley, J. D. (1997). KAoS: Toward an industrial-strength generic agent architecture. In J. M. Bradshaw (Ed.), *Software Agents*. (pp. 375-418). Cambridge, MA: AAAI Press/The MIT Press.

[4] Bradshaw, J. M., Greaves, M., Holmback, H., Jansen, W., Karygiannis, T., Silverman, B., Suri, N., & Wong, A. (1999). Agents for the masses: Is it possible to make development of sophisticated agents simple enough to be practical? *IEEE Intelligent Systems*(March-April), 53-63.

[5] Bradshaw, J. M., Sierhuis, M., Gawdiak, Y., Jeffers, R., Suri, N., & Greaves, M. (2001). Teamwork and adjustable autonomy for the Personal Satellite Assistant. *Proceedings of the IJCAI-01 Workshop on Autonomy, Delegation, and Control: Interacting with Autonomous Agents*. Seattle, WA, USA,,,

[6] Cohen, P. R., & Levesque, H. J. (1991). *Teamwork*. Technote 504. Menlo Park, CA: SRI International, March.

[7] Foster, I., & Kesselman, C. (Ed.). (1999). *The Grid: Blueprint for a New Computing Infrastructure.* San Francisco, CA: Morgan Kaufmann.

[8] Gershenfeld, N. A. (1999). *When Things Start to Think.*, New York: Henry Holt and Company.

[9] Greaves, M., Holmback, H., & Bradshaw, J. M. (2001). Agent conversation policies. In J. M. Bradshaw (Ed.), *Handbook of Agent Technology*. (pp. in preparation). Cambridge, MA: AAAI Press/The MIT Press.

[10] Hamilton, S. (2001). Thinking outside the box at IHMC. *IEEE Computer*, 61-71.

[11] Holmback, H., Greaves, M., & Bradshaw, J. M. (1999). A pragmatic principle for agent communication. J. M. Bradshaw, O. Etzioni, & J. Mueller (Ed.), *Proceedings of Autonomous Agents '99,* (pp. 368-369). Seattle, WA,, New York: ACM Press,

[12] Jordan, M., & Atkinson, M. (1998). *Orthogonal persistence for Java—A mid-term report*. Sun Microsystems Laboratories,

[13] Kahn, M., & Cicalese, C. (2001). CoABS Grid Scalability Experiments. *AAMS (Special Issue on Infrastructure Scalability)*, under review.

[14] Kahn, M., & Sage, P. (2000). DARPA Control of Agent-Based Systems Grid Tutorial. J. M. Bradshaw (Ed.), *PAAM 2000*. Manchester, England,,,

[15] Knoll, G., Suri, N., & Bradshaw, J. M. (2001). Path-based security for mobile agents. (pp. under review).,,,

[16] Seamons, K. E., Winslet, M., & Yu, T. (2001). Limiting the disclosure of access control policies during automated trust negotiation. *Proceedings of the Network and Distributed Systems Symposium*.,,,

[17] Smith, I. A., Cohen, P. R., Bradshaw, J. M., Greaves, M., & Holmback, H. (1998). Designing conversation policies using joint intention theory. *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98),* (pp. 269-276). Paris, France,, Los Alamitos, CA: IEEE Computer Society,

[18] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., & Jeffers, R. (2000). Strong Mobility and Fine-Grained Resource Control in NOMADS. *Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA 2000).* Zurich, Switzerland,, Berlin: Springer-Verlag,

[19] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A., Jeffers, R., Mitrovich, T. R., Pouliot, B. R., & Smith, D. S. (2000). NOMADS: Toward an environment for strong and safe agent mobility. *Proceedings of Autonomous Agents '99*. Barcelona, Spain,, New York: ACM Press,

[20] Tambe, M., Shen, W., Mataric, M., Pynadath, D. V., Goldberg, D., Modi, P. J., Qiu, Z., & Salemi, B. (1999). Teamwork in cyberspace: Using TEAMCORE to make agents team-ready. *Proceedings of the AAAI Spring Symposium on Agents in Cyberspace*. Menlo Park, CA,, Menlo Park, CA: The AAAI Press,