Leveraging Legacy Scientific Software with Mobile Agent Technology

Shanhani Markus
Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
99-041

https://docs.lib.purdue.edu/cstech/1471
LEVERAGING LEGACY SCIENTIFIC SOFTWARE
WITH MOBILE AGENT TECHNOLOGY

Shahani Markus
Elias N. Houstis

CSD TR #99-041
November 1999
LEVERAGING LEGACY
SCIENTIFIC SOFTWARE WITH
MOBILE AGENT TECHNOLOGY

Shahani Markus and Elias N. Houstis

ABSTRACT

Growth in telecommunications and high-speed networking technology has increased the speed and reach of the "global network". This in turn has fueled a surge in net-centric, distributed applications. Scientific computing applications too have begun to follow this design and development trend. However, many of these applications have to deal with the incorporation of legacy scientific code. This difficulty has significantly reduced the use of relevant modern distributed programming technology in new scientific applications. In this paper, we present a two-tiered agent technique to insert legacy scientific code within scientific applications that use mobile agent technology. We demonstrate the viability of this approach with a prototype implementation and discuss its use in specific scientific applications.
INTRODUCTION

Rapid advances made in wired and wireless networking technologies are evolving "net-centric computing" and "distributed computing" to be indispensable tools for computational scientists involved in solving both large and small scientific problems. These paradigms provide access to, and attempt to harness the computational power of, remote supercomputers, workstations clusters and heterogeneous collections of networked machines for compute intensive tasks. Thereby, they also promote scientific collaboration and facilitate novel approaches for realizing scientific computational goals.

Scientific applications often reuse legacy code. Encapsulating such legacy scientific software in distributed applications is a non-trivial problem that depends on many factors such as implementation language, execution pragmatics, characteristics of the legacy code and the nature of the distributed application being built. Thus, the techniques utilized to incorporate legacy code into a distributed scientific application have to be evaluated carefully on a case-by-case basis.

In this paper, we describe a two-tiered agent approach to reuse legacy code within a mobile agent based scientific computing application. We discuss the details and issues related to our prototype implementation that uses the Grasshopper mobile agent platform [21] as its distributed agent environment. We then present a distributed scientific computing scenario in which our proposed technique can be used to achieve reuse of legacy computational code.

This document is organized as follows: In Section 2, we present an overview of distributed computing and legacy scientific software. In section 3, we discuss the role of distributed computing in computational science and some general issues in designing distributed applications. In Section 4 we review an enabling technology for distributed computing, the Grasshopper mobile agent system and in Section 5, we present the conceptual technique for incorporating legacy code within an agent-based scientific application. In section 6, we describe our prototype implementation and in section 7 we discuss a scientific computing scenario that can be realized with the use of this legacy code incorporation approach. We conclude our presentation in Section 8, with an analysis of the proposed technique, its implementation and future work.

SECTION 2: OVERVIEW

In general, distributed computing refers to computing that occurs across multiple address spaces. In traditional distributed computing scenarios, these separate address spaces are assumed to be in close relative proximity, such as on workstations in a local area network. Net-centric and network-based applications leverage some of these distributed computing technologies across Intranets and the Internet. The difference between the terms "net-centric computing" and "network-based computing" is very subtle. Although there is no firm definition, generally network-based applications tend to be of a peer-to-peer or client-server nature. Thus, they comprise situations in which a network is placed between two communicating components of an application. Similarly loosely defined, net-centric applications consist of several communicating components logically placed around a network such that they collaborate with each other towards the attainment of a common goal.

The rising omnipresence of the Internet and its economic implications has resulted in an explosive growth in Internet-based technology companies and solution providers, leading to a surge in net-
centric applications. Furthermore, in most instances, scalability, fault tolerance and resource sharing requirements promote distributed computing. All these factors have influenced software technology to such an extent that distributed programming is fast becoming the natural and preferred choice for software design and development in many application areas. This trend has begun to blur the former distinctions between the distributed, net-centric and network-based computing paradigms. Thus the terms “network-based computing”, “net-centric computing” and “distributed computing” are now used practically interchangeably to describe computing involving collaborating processes on multiple machines that are physically separate and networked together.

Many significant enabling technologies have facilitated this march towards ubiquitous distributed computing. These technologies are the outcome of twenty years of research in middleware and include,

- mechanisms for data portability (ex: XDR [39])
- remote procedure calls [3][19]
- distributed messaging systems (ex: MQSeries [4])
- virtual parallel environments (ex: PVM [14] and MPI [15])
- remote object invocations [32]
- distributed programming systems (ex: CORBA [41])
- network programming languages (ex: Java [1])
- mobile agent environments (ex: Aglets [26] and Grasshopper [21])
- distributed component architectures (ex: DCOM [16], EJB [35] and SOAP [6])
- network communication buffers (ex: TSpaces [44] and JavaSpaces [37])
- distributed computing substrates (ex: Jini 2).

One of the most important and fundamental developments in distributed programming has been the emergence and subsequent widespread popularity of the Java environment and programming language [1]. Java is a language for network programming and has a self-contained, virtual machine environment. It is simple, secure, platform independent, type-safe, multi-threaded and garbage-collected and it has language-level support for remote method invocation, exception handling and native methods. Above all, Java appeared “in the right place, at the right time, with the right tools” and espoused the convenience of “write once, run everywhere”. The unique collection of powerful features in the Java programming language, along with its simplicity, have catapulted it into fame and tremendously influenced software development in this Internet age.

In computational science, at the opposite end of this spectrum of distributed software lies a significant body of legacy code. It represents hundreds of man-years of scientific research, programming and testing. Much of this legacy code is highly optimized and specialized and has been used for many years to garner a great degree of confidence in their correctness, reliability and accuracy. Hence, in trying to achieve network-based scientific computing, it is essential to investigate the issues in incorporating and reusing this irreplaceable legacy code.

Most of this legacy software is written in Fortran, a long-standing language particularly noted for its ease of use in mathematical programming tasks. Both Fortran and its modern-day counterparts (Fortran90, Fortran95, Fortran2000 and High Performance Fortran) do not by themselves, have the capability to create platform-independent distributed code to realistically achieve network-based computing. Instead, we need to find ways and means to leverage this code with state-of-the-art enabling technologies in order to build useful, reliable and robust network-based scientific computing applications.
SECTION 3: ISSUES IN BUILDING DISTRIBUTED SCIENTIFIC APPLICATIONS

Distributed applications are notoriously difficult to design, implement and debug. Additionally, programmers have to deal with issues related to heterogeneity of the underlying hardware platforms. Despite these problems, distributed computing offers an elegant solution to many application designs and continue to be strongly favored choice amongst application developers.

Schroeder in [34], and Waldo et al in [43], discuss the inadequacies of classical networked systems. Their comments center on the fact that the enabling technologies for traditional distributed systems aimed at making the network "transparent" to the programmer. They point out the fallacy of this approach and the importance of being aware of the instability of a network and its ensuing problems when designing and building distributed applications. In this vein, they demonstrate the need for modern enabling technologies to make the programmer aware of network programming issues such as, performance and latency, memory access, failure modes that are radically different to stand-alone systems, concurrency and consistency.

Latency refers to the time lag between a processor speed and network communication. For example, there is an obvious time difference between a local method invocation and a remote method invocation. This difference is very significant and should be an important design consideration when building distributed applications since it has a direct impact on overall performance. Memory access issues arise from the separation of address spaces. For instance, pointers in a local address space are not valid in a remote address space. Other than programmer awareness, the only other solution to this problem is a safety net provided by the underlying environment. This can be achieved by a programming environment that disallows pointers to local memory and enforces strong type safety. These goals are achieved by languages such as Java. Partial failure, concurrency and consistency relate to sudden failure of one or more components in a distributed system. The other working components may not be able to discern or detect these events. Furthermore, even if the failure were detected, it would be impossible to distinguish between a failure due to network breakdown or machine failure or software component failure. In distributed systems, it is important to ensure that the state of the overall system remains consistent after such a partial failure. Thus, the programmer has to be acutely aware of the indeterminate nature of a distributed application and build in safeguards to address issues such as sudden partial failure.

The above reasons indicate the difference between stand-alone systems and distributed systems and the need for distributed system designers to consciously acknowledge and decide on solutions for these problems. As emphatically stated in [34] and [43], in the interest of robustness, it is unwise to attempt "hiding the network".

Within the scientific computing world, it appears that researchers have been aware of and concerned about precisely these issues. Possibly, due to the common use of parallel computing in this community, scientific application programmers have been compelled to address the issues mentioned above. This very reason may be attributed to the slow growth of distributed scientific applications despite the existence and widespread use of enabling technologies for distributed systems for nearly twenty years. Even the excitement wrought by the advent of the Java programming language and its seemingly endless distributed programming potential has not generated a sudden surge of networked scientific applications. The objectives of the Java Grande Forum [40] typify this assertion. This forum came into being within the scientific community upon recognizing the value and potential of Java, as well as its severe shortcomings in relation to high performance scientific computing. Its charter states the following:
"Java has potential to be a better environment for Grande application development than languages such as Fortran and C++. The goal of the Java Grande Forum (hereafter, JGF) is to develop community consensus and recommendations for either changes to Java or establishment of standards (frameworks) for Grande libraries and services. These language changes or frameworks are designed to realize the best ever Grande programming environment."

Our research objective is not to use Java to build high performance applications, but rather, to use Java and other distributed technologies to leverage high performance legacy software. In doing so, we will address the issues raised above and the solutions and trade-offs we have chosen in order to build robust and reliable distributed scientific applications.

SECTION 4: THE GRASSHOPPER DISTRIBUTED AGENT ENVIRONMENT (DAE)

Agent technology is a state-of-the-art middleware technology for distributed applications. The term "software agent" has many definitions. In general, it can be described as "a software component that maintains a certain degree of autonomy". Mobile software agents have, in addition, the ability to migrate to different locations in order to accomplish its tasks. An "agent platform" is an environment that provides all the necessary services to facilitate its constituent software agent’s tasks. Our research and prototype implementations are based on the Grasshopper Distributed Agent Environment [21].

International standards have been developed to ensure interoperability between agent platforms from different manufacturers. The Grasshopper mobile agent platform is OMG’s Mobile Agent System Interoperability Facility (MASIF) standard [42] compliant. Additionally, it will be compliant with the Foundation for Intelligent Physical Agents (FIPA) specification [12] in its version 2.0 release.

SECTION 4.1: DISTRIBUTED COMPONENTS

The Grasshopper platform is built on top of a distributed processing environment. It is implemented in Java to achieve machine interoperability. The Grasshopper DAE is composed of regions, places, agencies and different types of agents that may be either stationary or mobile. Agencies are the actual runtime environments for the agents and hence at least one agency should be running on each host machine. They enable and control the creation, execution, registration, communication, persistence and transport of Grasshopper agents. A place provides a functional grouping within an agency. Regions facilitate the management of all the distributed components. Region registries are associated with each region and maintain information about all the components (agencies, places and their constituent agents) for that region.

During their lifecycle, Grasshopper agents (also referred to as services) may be in one of the following states: active, suspended or deactivated. Grasshopper agents may be either mobile or stationary. Unlike traditional mobile code that usually features remote execution (where the program is sent before execution), mobile agents can migrate during execution. This implies that all non-transient objects within a Grasshopper mobile agent should be serializable (i.e.: can be packaged and transmitted over the network).

SECTION 4.2: COMMUNICATION SERVICE

The communication facilities are offered by the Communication Service which is an important part of the core agency. Grasshopper provides several of communication protocols for remote interaction
CORBA IIOP, MAF IIOP, Java RMI, plain socket connections, plain socket with SSL and Java RMI with SSL. All communication is location transparent and is achieved with the use of proxy objects. Though the Grasshopper communication service provides the means for location transparent, inter-agent communication, it does not specify the way of communication with a specific agent language.

The DAE offers a standard array of communication modes - synchronous, asynchronous, dynamic and multicast. With synchronous communication, an agent invokes a method on another agent and waits for the result before continuing its task. With asynchronous communication, the caller agent does not wait for the remote method execution and continues with its task. The caller agent may obtain the results of the asynchronous remote method invocation in several ways - by periodic polling, by blocking when necessary or by subscribing to be notified when the result is available. Dynamic communication is useful when the caller agent does not have access to the proxy class of the other agent. In this case, it can dynamically construct a message by specifying the signature of the required method and invoke it on a universal proxy object. Dynamic communication may be performed synchronously or asynchronously. With multicast communication, the caller agent can invoke the same method on several other agents in parallel.

SECTION 4.3: SECURITY AND PERSISTENCE

The Grasshopper DAE provides both external and internal security. External security is based on the use of X.509 certificates and the Secure Socket Layer (SSL) to ensure confidentiality, integrity and proper authentication of the remote interactions. Internal security is enforced via access control. It is achieved by an identity-based and group-based access control policy.

Grasshopper provides mechanisms for persistence. This is an important concept in distributed object programming. For instance, with the Grasshopper persistence service, a copy of an agent can be maintained on a persistent medium for retrieval in case of an unexpected host computer crash. Furthermore, when persistence is enabled, idle agents that are waiting for external interactions can be removed from an agency and stored to save resources. Then when a request for the stored agent arrives, it can be re-instantiated to handle the request.

SECTION 5: AGENT-BASED LEGACY CODE INCORPORATION

In this section, we discuss a technique to insert legacy scientific code within a Grasshopper agent. We assume the legacy code is implemented in either C/C++ or Fortran. This is a valid assumption, since in most cases, legacy scientific code is Fortran based.

Legacy code encapsulation within Grasshopper agents would involve the use of native methods in Java. It would be very complicated to insert such code within a mobile agent. To do so, the agent's migration pattern would have to be completely defined a priori and deviations from this pool of potential locations would have to be disallowed. The legacy code libraries would then have to be made available at all the possible locations such that they can be dynamically loaded by the agents. Due to these restrictions and requirements, it is very unlikely that such code can be inserted within mobile agents to build an effective distributed scientific application. As a workable alternative, we present a two-tiered agent architecture that will enable distributed applications to reap the advantages of mobile agent technology. Though the legacy code encapsulation technique is described below in the context of Grasshopper agents, the methodology is not restricted to the Grasshopper agent platform. It is applicable to any Java-based agent environment.
We propose incorporating the legacy code within a stationary agent that would have close interactions with a related mobile agent. The legacy code would be inserted within the stationary agent using a wrapper approach. Figure 1 illustrates the encapsulation technique within a stationary agent. The legacy Fortran code is first encapsulated within a C wrapper since there is no native method interface between Java and Fortran. The C wrapper in turn is made available to a Java wrapper via the Java Native Interface (JNI). The methods in the Java wrapper can then be invoked by the Grasshopper stationary agent.

We refer to the legacy code encapsulated stationary agent as a Legacy Code Agent (LCA). The associated mobile agent is the Compute Agent (CA). Depending on the legacy code it contains, each LCA is considered to be "typed". Similarly, the related compute agent is also typed, to reflect its association with the correspondingly typed LCA. In a distributed scientific computing application, the compute agents will generally have a 1:N, N > 1, relationship with its similarly typed legacy code agents. This enables the mobile compute agent to transfer an ongoing computation from one LCA to another similarly typed LCA when needed. By the nature of the underlying code and the overall application design, this may result in part of (or in some rare instances, the entire) computation being repeated at the new LCA location. The advantage is that in case of partial failure, this pseudo migration capability ensures the integrity of overall computational application.

The typing of legacy code agents and compute agents is achieved via a Legacy Relationship interface. This marker interface is extended to create the CA_Legacy_Relationship and the LCA_Legacy_Relationship subinterfaces. The compute agents and legacy code agents implement these subinterfaces to demonstrate their legacy code incorporation capabilities and relationships. Figure 2 shows how this interface hierarchy overlays the service class hierarchy of the Grasshopper platform to specify a convention for the creation of CAs and LCAs.

The application design architecture could introduce another level to this hierarchy to reflect a special legacy code relationship. For example, let us consider a distributed application that incorporates a particular legacy package or library named "Alpha" within its legacy code agents. Their specific functionality can be formalized as the subinterfaces, Alpha_CA_Legacy_Relationship and Alpha_LCA_Legacy_Relationship. These subinterfaces would contain the constants and any abstract methods that are needed to provide a clear interface for the legacy code incorporation within the specialized CA and LCA agents that implement them.
SECTION 6: LINEAR SOLVER AGENTS – A PROTOTYPE IMPLEMENTATION

The technique described in the previous section has been used to implement a distributed linear solver agent application. The legacy code used in this example is the Itpack software package [25] for the iterative solution of linear systems of equations. This package is implemented in Fortran. The Itpack library contains several modules such as Jacobi CG, Jacobi SI and SOR. The iterative algorithms in these modules feature automatic parameter definition and stopping tests and they are not guaranteed to converge for all linear systems.

The distributed linear solver agent application consists of mobile linear solver agents (i.e. Itpack compute agents), stationary linear solver agents (i.e. Itpack legacy code agents), linear system objects and solution objects. This agent application facilitates the solution of linear systems using the legacy Itpack solver library and can be deployed on any hardware platform that has a Java Virtual Machine and the Grasshopper DAE with a running agency.

The prototype implementation allows a user to create a mobile Itpack CA or collaboratively request a remote user to send such an agent to his or her local machine. In the latter case, the collaborator would simply invoke the agent to enter the remote location via its graphical user interface and make it migrate. Alternatively, this application can be extended to allow a user to request such a mobile linear solver agent from a central Web site that hosts them.

Once the mobile linear solver agent arrives at the local machine, the user can specify an input linear system via the agent’s graphical user interface. The Itpack CA accepts linear systems that are represented in XML (eXtensible Markup Language) format based on a proprietary DTD (Document Type Definition). This format specification is only applicable to meta-data. For instance, instead of storing the linear system elements in XML-format, a URI (Universal Resource Identifier) to the linear system data is specified.
After the input linear system is successfully parsed into its internal object data structures, the Itpack CA has two alternative ways to solve it. It can migrate to the machine on which the Itpack LCA is located and obtain the linear system solution via local interactions or it could communicate remotely with the Itpack LCA to solve the linear system. These migration/communication patterns are illustrated in Figure 3. Since Grasshopper provides location transparent communication, the underlying interaction methodology is the same for both scenarios. The Itpack CA interacts with the Itpack LCA through asynchronous remote method invocations. For this purpose, the Itpack CA instantiates a Communicator class to setup the necessary interactions with the Region Registry such as looking up the associated Itpack LCA.

To solve the linear system, the Itpack CA invokes the Itpack LCA's "process" method with the linear system object as a parameter. Since this invocation is done asynchronously, the Itpack CA is not blocked until the solution is completed and is even free to migrate to another location. If the Itpack CA does migrate at this time, the region registries in the Grasshopper DAE ensure that it would receive the notification once the Itpack LCA completes its solution process. This capability would be useful in the case of a sudden network slowdown. If the user were to detect such an anomaly, it could migrate the Itpack CA to a more appropriate location to receive the solution object.

Once the Itpack LCA receives the linear system object, it invokes the "solve" method of the Itpack Wrapper class. This method sets up the Itpack input parameter arrays and invokes the native C method that encapsulates the calls to the Fortran-based Itpack library modules. When the native method returns, the solution (or an error status) is extracted in the Itpack Wrapper and returned to the Itpack LCA agent. The LCA then creates a solution object with all the relevant computational results. When the LCA returns from its solve method, the Grasshopper DAE notifies the Itpack CA agent via its event listener method. Depending on the contents of the solution object, the Itpack CA then displays the solution or in the case of a failed computation, an error message.
By design, this distributed application is very interactive. All error conditions and exceptions are reported to the user via the Itpack CA’s graphical interface. This nature of the application effectively deals with many of the distributed computing issues discussed in section 2. For instance, partial failure is detectable by the user because he/she can investigate any unusually long response delays. Since this application is based on a peer-to-peer communication pattern, even if one component (agent) were to fail, the system’s integrity is not compromised provided the 1:N, N > 1, relationship between the Itpack CA and Itpack LCAs is maintained. For instance, if an Itpack LCA were to fail either due to a software crash, network downtime or machine failure, at the most, the current linear system solution computation would have to be redone by another Itpack LCA. In the case of an Itpack CA failure, the user would be made aware of such a problem due to its interactive nature. In this case, the system can be made consistent again by restarting the Itpack CA if possible (i.e. if the failure was software related and not due to a hardware crash). A solution computation may have to be redone at this point if the CA failure occurred while the LCA was performing a remote computation.

This prototype implementation has been successfully tested and experimental results indicate that the overhead of the wrappers is negligible. This wrapper approach is used as a basis for the two-tiered agent architecture within the scientific application scenario described in the following section.

SECTION 7: AN AGENT-BASED FRAMEWORK FOR MULTIDISCIPLINARY PROBLEM SOLVING ENVIRONMENTS (MPSE)

A Problem Solving Environment (PSE) is a computer system that provides all the computational facilities necessary to solve a target class of problems. An MPSE is a framework and software kernel for combining PSEs for flexible multidisciplinary applications.

A physical system in the real world normally consists of a large number of components that have different shapes, obey different physical laws and constraints, and interact through physical interfaces. Mathematically, the physical behavior of each component is modeled by a PDE or ODE system with various formulations for the geometry and constraint conditions. It is difficult to imagine creating a monolithic software system to accurately model such a real world problem. Therefore, a multidisciplinary mathematical/software framework is needed. It should be applicable to a variety of practical problems and allow software reuse in order to achieve lower costs and high quality.

SECTION 7.1: PROTOTYPING PHYSICAL SYSTEMS

Most physical systems and manufactured artifacts can be modeled as a mathematical network whose nodes represent the physical components. Each node has a mathematical model of the physics of the component it represents and a solver agent for its analysis. Individual components are chosen so that each node corresponds to a simple PDE or ODE problem defined on a regular geometry. There exist many standard, reliable PDE/ODE solvers that can be applied to these local node problems. In addition, there are nodes that correspond to interfaces that model the collaborating parts in the global model. Since the analysis of an artifact changes through time, some of the interfaces appear and disappear during the analysis session.

To solve the global problem, the local solvers collaborate with each other to relax (i.e. resolve) the interface conditions. An interface controller or mediator agent collects boundary values, dynamic/shape coordinates, and parameters/constraints from neighboring subdomains and adjusts boundary values and dynamic/shape coordinates to better satisfy the interface conditions. Therefore, the network abstraction of a physical system or artifact allows us to build a software system that
is a network of well-defined numerical objects collaborating through a set of interfaces. This architecture can be combined with an agent-oriented paradigm and collaborating solvers [8] to create an MPSE as a powerful prototyping tool for physical systems.

In this section, we propose such an MPSE framework that provides the architecture and model infrastructure for the agent-based simulation of a multidisciplinary physical system or artifact. This MPSE framework can then be used to prototype MPSE applications that simulate complex multi-physics phenomena that are governed by PDE network models. For example, let us consider the simulation of a gas turbine engine.

The gas turbine engine is an engineering triumph. It has more than 1,300 parts with rotational speeds up to 16,000 rpm for the axial and 50,000 rpm for the radial flow components. For aircraft applications, it operates with maneuver loads of up to 10g, with flow path pressures and temperatures to 40 atmospheres and 1400°F. The important physical phenomena take place on scales from 0.001 to 1 meter. For a realistic gas turbine simulation, there are perhaps 100 million variables and many different time scales. This problem has very complex geometry and is very non-homogeneous. GasTurboLab [10] is a multi-physics application for the simulation of such gas turbine engines. The primary goal of the GasTurboLab research project is to advance the state-of-the-art in very complex scientific simulations and their validation.

The proposed MPSE framework can be applied towards the design and implementation of this GasTurboLab MPSE application to study physical phenomena such as stall, surge and turbine blade fatigue in a gas turbine engine [11].

SECTION 7.2: ENABLING TECHNOLOGIES

The main distributed computing technologies utilized in the MPSE framework design are the Java-based Grasshopper DAE which was described in section 3, JavaSpaces [37] and the Jini Distributed Event and Notification System [36]. The latter two technologies are part of the Jini technology system [2] from Sun Microsystems.

The Jini distributed event and notification system defines a set of interfaces, conventions and protocols that allow objects in separate Java virtual machines to notify each other of changes in state. In this remote event model, all objects that are interested in receiving events implement a generic "RemoteEventListener" interface that has a single method - "notify". The event and notification system defines only one class of events, the "RemoteEvent", and does not specify a way register interest in such an event. Instead, each event source object is expected to define and implement methods for event listeners to register their interest. As with other Jini services, the event registration is limited to a particular time duration based on the notion of a lease. An important feature of this remote event model is the provision to specify a delegate to respond to any event type. This allows the implementation of "generic third-party event listeners" and "third-party filters". This introduces the ability to add new behaviors to the process of sending, storing and delivering events such as guaranteed delivery delegates and event storage delegates.

JavaSpaces is a Jini service that provides persistent object storage spaces for remote Java processes to coordinate and exchange data. It provides a space-based programming model for distributed computing applications. In this model, a distributed application is viewed as a group of collaborating processes that cooperate via the flow of objects through one or more spaces (a shared repository). Thus, instead of communicating directly, the distributed processes coordinate by using these spaces as persistent object storage and exchange mechanisms. This indirect interaction via spaces leads to loosely coupled protocols since the processes do not have to know each other's identities or even be
active at the same time. The objects in the spaces are regarded as passive data and processes perform simple operations such as "read", "write" and "take" on them. The necessary objects are matched by a simple associative lookup based on a template. If a matching object is not found immediately, the process has the option of waiting until its arrival. Alternatively, a process may register a "notify" request to declare interest in a future incoming entry to the space. Then, when a matching entry arrives, the registered event listener is alerted by the invocation of its notify method. The service makes a "best effort" attempt to deliver notifications to registered event listeners. JavaSpaces guarantees a transactionally secure mode of operation, ensuring that an operation on a space is atomic. This feature is particularly helpful in dealing with partial failure in distributed applications.

Figure 4: The MPSE framework architecture.

SECTION 7.3: ARCHITECTURE OF THE MPSE FRAMEWORK

As its hardware infrastructure, the proposed MPSE framework assumes a network of distributed machines (computational grid), each with the Grasshopper DAE and a running agency. The framework is designed for multidisciplinary simulation problems where the natural or artificial geometric boundaries can be used to split the problem and the underlying simulation into many smaller sub-problems. Each sub-problem is assigned to a machine on the computational grid and solved independently, with mediator interactions along the boundaries for interface relaxation. Thus, an implementation of the MPSE framework for PDE simulations must support domain decomposition with geometric objects, use a network of PDE solver agents, and utilize interface relaxation techniques.

Enabling the use of legacy software and utilizing existing technology is an important goal in the design of this MPSE framework. In addition to the distributed enabling technologies discussed in the previous subsection, the MPSE framework uses the IRIS Explorer application builder and visualization system to implement its graphical user interface. The overall framework architecture is
shown in Figure 4.

**User Interface Layer:** As illustrated in the diagram, this layer contains IRIS Explorer modules for problem specification, data dispatch and computation. The tools in the Problem Specification Module are used to specify the domain decompositions. The formatted output from these modules is directed to the Dispatcher Module. This module distributes the partitioned data to the storage repositories of the selected hosts on the available computational grid. The output from the dispatcher module (host allocation tables) is directed to the Compute Module which controls the launch and execution of the computational agents.

**Enabling Service Layer:** Each host agency has an active Resource Agent (RA) which is implemented as a stationary Grasshopper agent. This RA monitors execution performance and gathers local machine load and network congestion information. The local RA notifies other interested remote RAs of any significant changes in its local network or its host performance levels. The RAs utilize the Jini remote event model and the JavaSpaces services to implement this resource information synchronization. Thus, each RA will have dynamic access to the overall network performance information such as load, congestion and machine reachability.

**Computational Layer:** The primary “workers” within the compute module are the Compute Agents (CA) and the Mediator Agents (MA). The CA is responsible for the simulation computation of a single subdomain. Thus, one CA is assigned for each domain partition, implying that each Grasshopper agency (on each host) contains only a single CA. The MAs reside on a target host that is assigned a domain partition or on an intermediate host in close network-related proximity to the target hosts with the associated neighboring domain partition assignments. These agents are implemented as mobile grasshopper agents and utilize a two-tiered agent architecture to achieve mobility even when the simulation computation is performed by legacy code.

### SECTION 7.4: COMPUTATIONAL FRAMEWORK

The compute module begins its task by launching a Simulation Controller Agent (SCA). This agent controls the entire computational simulation process by monitoring the distributed CAs and MAs on each host. The SCA interacts closely with the RAs on the target hosts to ensure the dynamic integrity of the selected computational grid. Thus, if a particular host connection deteriorates, the SCA may instruct the corresponding CA to migrate to another host and continue its computation. Furthermore, for highly compute intensive simulations, the SCA may employ load-balancing techniques to dynamically redistribute the ongoing computations on the computational grid. The CA and MA mobility makes this operation possible without major disruption to the simulation.

A two-tiered agent architecture is used to facilitate this CA and MA mobility within the MPSE framework. The legacy code associated with the CA is encapsulated within a Legacy Code Agent (LCA). The legacy code associated with the mediator agent is encapsulated within an Interface Code Agent (ICA). This legacy code encapsulation and the CA-LCA and MA-ICA relationship conventions are implemented based on the techniques in section 4. This second tier of LCA and ICA agents exists transparently within the MPSE framework. Thus, all other agents in the framework interact solely with the CA and MA agents and not the LCA and ICA agents. Likewise, the LCAs and ICAs communicate only with their related CAs and MAs. Figure 5 illustrates these interaction patterns.

When a CA needs to migrate to another host (agency), it requests its related LCA to stop computation of the current iteration. It then migrates with the last completed iteration data object (LCID object), to its new location. The CA then restarts the iteration computation based on the LCID object, with a newly related LCA. Clearly, to make such mobility possible, the CA needs to update and save the LCID object at the end of each compute iteration, in a persistent repository.
(JavaSpace service). The MA migration is achieved in a similar manner.

Figure 5: MPSE Framework agent/user interactions

SECTION 7.5: IMPLEMENTATION ISSUES

The MA has to synchronize with the CAs of the neighboring subdomains to receive their simulation output at the end of each iteration. This is achieved by using a JavaSpaces service (space) as the data exchange mechanism. Since the MA cannot begin its interface relaxation computation until it receives data from all the neighboring subdomains, it issues consecutive blocked "reads" to the space. This results in an efficient synchronization, since the order of input is not important and the latency is relatively insignificant in comparison to the simulation compute time. The relaxation computation begins soon after the final CA data is received. Likewise, for the CA to start a new compute iteration, space-based synchronization is used to signal the arrival of the MA output. For this purpose, the CA issues a template associated notification request to the space at the end of its iteration. The CA is then alerted when the MA finishes its computations and places its output in the space. The algorithms that formalize the computational agents' tasks and collaboration mechanisms are illustrated in Figure 6.

The output synchronization via the JavaSpace services is done by exchanging URI objects that essentially contain pointers to the corresponding data files. These redirection objects are used instead of actual data objects since, in most cases, the legacy code reads its input directly from a file. Furthermore, this allows the CA or MA to transfer the indicated files to locations local to the corresponding LCA or ICA if necessary. This feature is particularly useful in the case of a migrated CA or MA.

The CA-LCA and MA-ICA communication is done via asynchronous remote method invocations (RMI). The close CA-LCA and MA-ICA relationships, and the transparency of the second tier of agents within the MPSE framework, characterize these interactions as one-to-one, asynchronous and tightly coupled. Thus, asynchronous RMI is preferred over space-based communication as it better matches the nature of the interaction. The URI object for the input data is given as the argument to the remote method. The corresponding LCA or ICA then extracts the input filenames and passes them to the legacy code via the legacy code wrappers. This design insulates the encapsulated legacy software from the overall collaboration complexities without any code re-engineering.

From the
The overall design of the MPSE Framework has many safeguards to identify and deal with partial failure to ensure the consistency of the distributed application. The RAs continuously monitor the network integrity and notify the SCA of any problems. The CA and MA mobility enables the SCA to effectively handle any hardware or network related partial failure by moving the affected simulation component to another host on the computational grid. In the case of software related partial failure, the RAs (for CA failure) or the CAs (for LCA failure), would alert the SCA. In this case, the SCA can either notify the user or take an intelligent action based on a set of pre-programmed rules. If a RA failure occurs, the SCA could create another RA to continue the resource monitoring for the affected host. If the SCA were to fail, the user would immediately be made aware of the event, which

...
would be analogous to a total failure in a stand-alone system. In this case, the user could directly take appropriate actions to recover the application.

SECTION 8: FUTURE WORK

The approach proposed in this paper, to incorporate legacy software within a mobile agent system, can be applied towards an agent-based Legacy Knowledge Acquisition (LKA) Framework. Domain-specific knowledge bases built with this LKA framework could then be used as the basis for powerful recommender systems for scientific applications.

SECTION 8.1: MOTIVATION

A knowledge base is a very important and crucial component in expert systems. This importance is concisely reflected in the popular slogan "knowledge is power". In [9], Edward Feigenbaum said, "the real power of an expert system comes from the knowledge it possesses rather than the particular inference schemas and other formalisms it employs". Thus, the quality of the expert system and the results it generates are greatly influenced by its knowledge acquisition capabilities. Despite this importance, as expressed in [31], collecting knowledge to build a knowledge base is a serious bottleneck in constructing expert systems. This fact is especially true in the context of knowledge bases for legacy scientific software.

There are many collections and repositories for legacy mathematical and scientific libraries such as NetLib [29], NHSE [28] and CSIR [7]. These repositories feature systematic software cataloging mechanisms. Many are built with the Repository in a Box (RIB) [30] toolkit for convenient setup and maintenance. Most are based on the IEEE standard for software cataloging, the Basic Interoperability Data Model (BIDM) [20]. However, they do not support an effective framework to permit the submission and execution of small sample problems using the legacy code, with the purpose of obtaining performance and other relevant data on the legacy package. Instead, the scientist is required to download and install the legacy software package from these repositories in his/her local environment in order to run any sample problems. In many instances, the scientist realizes the non-applicability of a particular package, only after such an installation process.

An ideal scenario would be the ability to launch a compute agent with a sample suite of problems. The compute agent would then interact with the relevant legacy code agents and determine the encapsulated legacy code's applicability to the problem domain. For this purpose, they would utilize the software cataloging mechanisms that are already available at the repositories. They would then interactively and autonomously perform the necessary input data filtering to obtain the legacy code's required input format and execute the problem suite. The compute agent would then gather the resulting output and performance data and enter this information into the local scientific knowledge base. The proposed LKA framework would enable the implementation of such a scenario.

SECTION 8.2: RESEARCH ISSUES

Many issues need to be addressed in order to build the LKA framework. Most important amongst these is the identification of an appropriate knowledge representation scheme for the performance data in the knowledge bases. For the sake of interoperability, this knowledge representation scheme should be an extension of the IEEE standard, BIDM. A standard representation for the transmission of the sample problems and their solutions also needs to be determined. An XML representation based on a suitable DTD would probably be the best approach for this purpose.

The distributed LKA framework would encompass mechanisms for the legacy software
"suppliers" to place their legacy code agents and the knowledge "acquirers" to access them via their compute agents. In addition to the research issues already addressed in the context of this paper, the design and implementation of the LKA framework would have to address many security issues such as authentication and access control. A convenient user interface to the compute agents that would allow the autonomous construction of the knowledge bases will also have to be determined.

Appropriate conventions to deem the viability of a particular legacy code agent to solve a given problem, the application of filters to convert the input data into the requisite formats and problem selection methodologies also need to be addressed.

SECTION 8.3: CONCLUSION

In summary, as a future project, we propose a knowledge acquisition agent framework that is based on the techniques described in this paper. It can be used to encapsulate legacy software that is already available in many repositories, within legacy code agents. These agents would interact with knowledge acquisition compute agents for the purpose of building domain-specific scientific knowledge bases. With the assistance of an appropriate expert system, scientists would then be able to obtain recommendations for legacy software that would be suitable to solve their target problems.


19. N.C. Hutchinson, L.L. Peterson, M.B. Abbott and S. O'Malley. "RPC in the x-


22. IRIS Explorer Toolkit, IRIS Explorer Center (North America), 1400 Opus Place, Suite 200, Downers Grove, IL 60515-5702, USA. http://www.nag.com/IRIC.


30. NHSE Technical Team at University of Tennessee. “Repository In a Box (RIB)”. http://www.nhse.org/RIB/.


38. Sun Microsystems, Inc. The Network is the Computer, Trademark.