

Using Agent Control and Communication in a Distributed Workflow Information System

M. Brian Blake

Department of Computer Science, Georgetown University, 234 Reiss Science Building, Washington, DC 20057

blakeb@cs.georgetown.edu

Abstract. Agent communication has developed widely over the past decade for various types of multiple agent environments. Originally, most of this research surrounded simulation systems and inference systems. Subsequently, agents are expected to adapt to, dynamically create, and understand evolving conversation policies. This concept of agent communication is not completely necessary in some domains, especially in domains where the policy of interaction is essentially static. One such domain is that of distributed workflow management with implications into Electronic Commerce. In this domain, agents are “middle-agents” that represent the distributed components that implement each individual workflow step. By representing the component-based services of each step, multiple distributed agents can essentially manage a workflow or supply chain that spans several on-line businesses (B2B). The WARP (Workflow-Automation through Agent-Based Reflective Processes) architecture is a multi-agent architecture developed to support distributed workflow management environments where distributed components are used to implement each of the workflow steps. This paper describes a software engineering process for integrating new component-based services into a static workflow-based ontology. Furthermore, the interaction protocol and supporting implementation based on the Knowledge Query and Manipulation Language (KQML) are presented. This agent communication architecture is implemented with the latest in Sun Microsystems’ Jini technology.

Keywords

Object-oriented Ontology, Agent Communication, Workflow Management

1. Introduction

Electronic markets are becoming increasingly popular with higher expectations in the future. Many business interactions occurring over the Internet follow either workflow or supply chain models. Moreover, some on-line businesses are adopting the use of components to implement their services. Currently, components are being designed and developed with greater modularity. Independent components can fulfill substantial tasks in these on-line environments. On-line transactions occur both across distributed servers within a single company’s intranet as well as across multiple companies via the Internet (sometimes considered business-to-business or B2B [1]). Subsequently, transactions are no longer the interaction of human-controlled business modules, these transactions are defined more by the configuration and coordination of independent components, regardless of where they are housed. When these transactions interact using workflow or supply chain paradigms, there must first be some way to designate policy information and

secondly some architecture to sequentially invoke the independent components as dictated by that policy.

The WARP architecture was conceptualized to operate in this environment specifically where on-line workflow enactment consists of the coordination distributed components[2] [4]. The WARP architecture uses a two-phased approach. In the first phase, the WARP architecture has semi-automated functionality where humans interact with workflow manager agents in the process of designing a workflow schema. In the second phase *relevant to this paper*, multiple agents collaborate to manage a workflow of on-line distributed components. Essentially, this is an approach that uses an agent-based middleware layer to coordinate internet-based workflow. One example might be an on-line stock purchasing scenario that requires the workflow coordination of an on-line broker, an on-line trader, and an on-line banker. Each of these on-line businesses may have independent components to perform such tasks as collection of customer requests, stock trade, and payment services, respectively. WARP role agents can act as proxies for the components located at the distributed sites of the independent companies. These agents collaborate on the pre-determined workflow schema to manage the interaction among the components. A high-level architecture in context of the on-line stock-purchasing domain is shown in Figure 1.1.

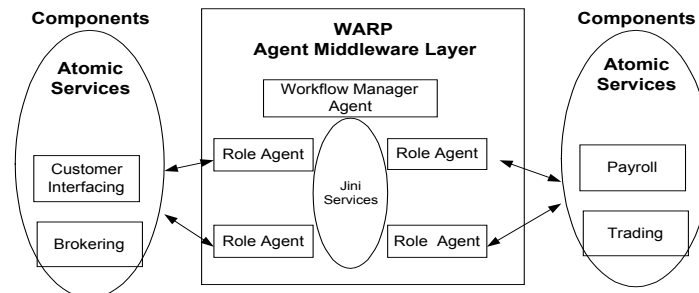


Figure 1.1 The Agent Middleware Phase: WARP Architecture

The main focus of this paper is the communication among the role agents as a component of the workflow coordination of the distributed component-based services. This research uses a “tuple-spaces” (as first seen in the Linda project [10]) approach to communication among a group of agents. The implementation makes use of Sun Microsystem’s implementation of tuple-spaces, Java Spaces. This implementation also uses the reflective capabilities in the Java programming language. The architecture is built on JavaBean component-based services. Hereafter this approach to agent communication will be referred to as KOJAC (KQML over Jini for Agent Communication). To clarify, WARP is the overall agent architecture for the workflow enactment, while KOJAC is the protocol and software underlying WARP that handles

agent communication and interaction. This paper continues in the subsequent sections with a brief background of agent communication. Next there is a brief overview of the WARP architecture. The following sections discuss the ontology and software engineering approach to KOJAC that provides communication within the WARP system. The final sections discuss the actual agent communication architecture and results.

2. Agent Communication and Electronic Commerce

This approach to agent communication gets its origins from KQML. This section gives an introduction of that part of KQML and other agent communication efforts that are pertinent to this work.

2.1 Correlation with KQML

The motivation for KQML [15][16] was to formalize a method by which agents can communicate effectively and efficiently. The message format supplies the agent with knowledge of which agent it is communicating to, a protocol for establishing dialogue, the language by which agents are communicating, terms by which other agents will interpret expressions, and exception handling. It is not within the scope of this paper to cover the KQML specification in entirety but to introduce the portions of the protocol that may assist later interpretations

KQML is separated into 3 layers, content, message and communication layers. The content layer allows agents to communicate which language is going to be used in a particular message. The message layer contains the message to be communicated in the form of content messages and declaration messages. The final layer is the communication layer, which exchanges packages to specify communication attributes. The message layer is of importance to us. The message layer, more specifically in content messages, is what is emulated in this work. Performatives are specialized KQML message types. The specification of a performative can increase system-wide transactions and functionality. In later sections, the KOJAC approach presented in this paper will be used to implement a subset of the common reserved performatives [15] as in Table 2.1.

<i>Category</i>	<i>Name</i>
Basic query	ask-one, ask-all
Generic Informational	tell-one, tell-all
Capability-definition	advertise, subscribe, monitor
Networking	register

Table 2.1 A Subset of Reserved Performatives

2.2 Other Relevant Agent Communication Efforts

Over the last decade, there have been several efforts to create a data format that is acceptable to all software environments. The most notable effort is the work developing the Extensible Markup Language (XML). Currently, XML is the best choice for a

language for representing data across multiple platforms. As described in the previous section, KQML has been used to represent data in agent communication. However, KQML uses a Lisp-based text representation that is not widely accepted for business transactions. If agents are to be used in electronic commerce, there is a need for a consolidation of the accepted industry-based representations like XML and the languages that the agents can understand. The latest research trends in agent communication have taken it one step further. Currently there has been the creation of an *Agent Communication Markup Language (ACML)*, which combines the traditional agent communication concepts of KQML with the industry acceptable universal format of XML [12]. Underlying the ACML is the Business Rule Markup Language (BRML). BRML is the B2B-specific content language [13].

The Foundation of Physical Agents (FIPA) has specifications for interaction protocols, communicative acts, and content messages for agent communication [8]. KQML is a subset of both the complexity and completeness of these specifications. Early in the project, we decided that the interaction protocols of KQML presented enough functionality to support this workflow-based communication. In future efforts, there is a plan to further evaluate the benefits of the evolving FIPA standards. Moreover, this research is toward implementation-level approaches to agent communication. The goal is toward the connection of agent implementation practices and those currently accepted in industry. This work has set the foundation for upcoming work that unites KOJAC with ACML-type approaches. In our most recent research, XML-based schema formats are translated into the object ontology and vice versa. Furthermore, software objects can be constructed based on individual XML documents by which agents can use for communication. The goal of this research is toward the consolidation of agents and electronic market software systems development.

3. Background of WARP

The approach to automated compositional configuration is called Workflow Automation through Agent-Based Reflective Processes (WARP). This approach is based on the use of an agent-based middleware architecture here after called the WARP architecture. This WARP architecture consists of software agents that can be configured to control the workflow operation of distributed services. The WARP architecture is divided into two layers. These layers are the application coordination layer and the automated configuration layer.

The application coordination layer is the level in which the workflow instances are instantiated and the actual workflow execution occurs. The application coordination layer consists of two agents, the Role Manager Agent (RMA) and the Workflow Manager Agent (WMA). The RMAs have knowledge of a specific workflow role. The WMA has knowledge of the workflow policy and applicable roles. When a new process is

configured, the workflow policy is saved in a centralized database. The RMA plays a role in the workflow execution by fulfilling one or more services as defined by the workflow policy in the centralized database. The RMA registers for workflow step-level events in the event server based on its predefined role. When an initiation event is written into the event server, the RMA is notified. Subsequently based on its localized knowledge of services and its workflow role, the RMA invokes the correct service. The WMA has similar functionality, but instead registers for overall workflow level events (i.e. workflow initiation and nonfunctional concerns). The WMA does not control the workflow execution, but in some cases it adds events to bring about non-functional changes to the execution of the entire workflow.

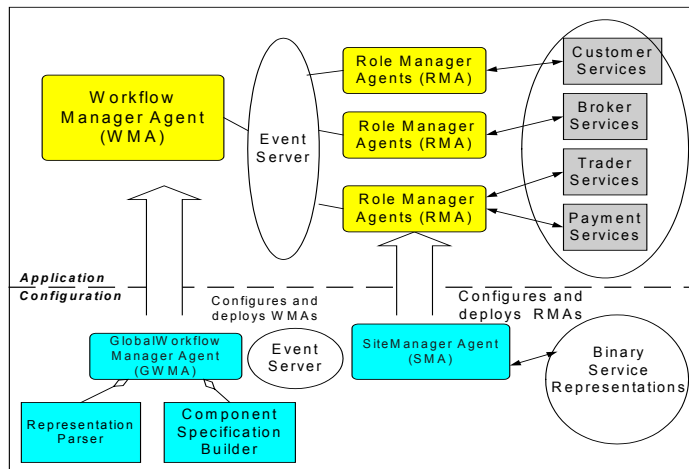


Figure 3.1. The WARP Architecture.

At the automated configuration layer, agents accept new process specifications and deploy application coordination layer agents with the new corresponding policy. This layer consists of the Site Manager Agents (SMA) and the Global Workflow Manager Agent (GWMA). The GWMA accepts workflow representations from a workflow designer as input. The SMAs discover available services and provides service representations to the GWMA. The GWMA accept both of these inputs and writes the workflow policy to the centralized database. The GWMA then configures and deploys WMA to play certain aspect-oriented roles. At the completion of workflow-level configuration, the SMA configures and deploys RMAs to play each of the roles specified in the workflow database. A general view of the WARP process is shown in Figure 3.1.

The application coordination layer is where the agent communication occurs (pertinent to this paper). To consider this operational environment, again let's use the on-line stock-purchasing domain (Figure 1.1.). A configured WARP system would have a Role Manager Agent (RMA) for each of the roles. RMAs act as middle agents [6] for

components. The RMAs obtain system aspects of the component through introspection and are able to invoke component functions through the process of reflection. The three roles are the Customer Interface Role, the Broker Role, and Trading Role. There is a RMA for each role. There is one Workflow Manager Agent (WMA) that helps in the coordination of the entire workflow. Each RMA would subscribe for service completion events prior to their affiliated services. For example, the Broker (Portfolio Management) Role would monitor for the completion event of a `getTradeRequest` service. Suppose a customer invokes the `getTradeRequest` service. The Customer Interface RMA would receive a completion event from the component (actor) and would broadcast the pertinent data for this service completion. The RMA for the Broker Role would be notified of this completion. First it would check to see if this service is pertinent to any of its workflow policy responsibilities. If so, the RMA for Broker Role would wait for the ready event to be written to the server by the WMA. The WMA would have also been monitoring and notified of the `getTradeRequest` service completion. The WMA would post any amendments to the workflow based on nonfunctional concerns at the process level. Subsequently, the WMA would publish a ready event to the pertinent RMA. Through reflection, the RMA would invoke the proper service (`searchPortfolio` service) for this step (reflection) in the workflow policy. Subsequently the output data, and the service completion would be broadcast. This process sequence is shown in Figure 3.2 for the stock purchase process.

KOJAC is the approach to agent communication needed to manage the sequence of actions described in Figure 3.2. Agents must be able to understand service completions in addition to have the knowledge of resulting actions. Agents also need to communicate other nonfunctional workflow management concerns like exception-handling, atomicity, and performance.

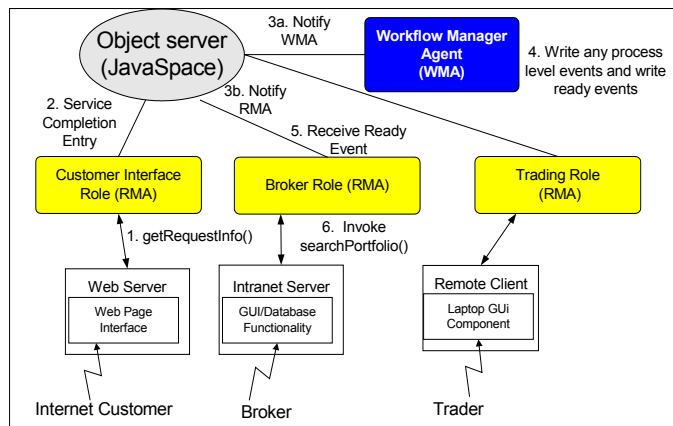


Figure 3.2 WARP Operational Environment

4. The Static Workflow-Oriented Ontology

The agent communication in this domain relies very heavily on the concepts of workflow management. In fact, the communication protocols used in the approach are built on a workflow-based ontology. In the following section, the pertinent workflow terminology is defined. Subsequently, there are technical details of the workflow-based object-oriented ontology.

4.1 Workflow Terminology

The workflow language here follows mainstream workflow terminology used presently by researchers [17]. In order to set the nomenclature for further discussion, the following set of definitions are adhered to throughout this paper.

- A *task* is the atomic work item that is a part of a process.
- A task is implemented with a *service*.
- An *actor* or resource is a person or machine that performs a task by fulfilling a service.
- A *role* abstracts a set of tasks into a logical grouping.
- A *process* is a customer-defined business process represented as a list of tasks.
- A *workflow* (instance) is a process that is bound to particular resources that fulfill the process.

4.2 The Workflow-Based Object-Oriented Ontology

The object-oriented ontology is a shared knowledge-based among agents. This solution is practical in the context of object-oriented domain analysis [11], since agents reason about a particular domain when they communicate. E-market designers can use traditional object-oriented analysis and design techniques to construct a domain model using object-oriented structural diagrams[14]. This domain model later translates into a physical set of classes. Objects from these domain classes are later specialized to particular types of Jini/JavaSpace entry objects. This is discussed in greater detail in the operational semantics of KOJAC in 6.2.

The first implementation of KOJAC is for the WARP agents. WARP agents communicate based on a domain that considers workflow policy, roles, services, and data flow. This business process-based ontology is reusable across most Emarket domains that implement a workflow of distributed components. The static view of the workflow-based ontology is illustrated in Figure 4.1. The workflow policy is the heart of this ontology. Agents that coordinate component-based services first need to know the workflow policy. Each step in the workflow policy correlates to a role and the completion of a specific service. Each service has one or more parameters (pre-conditions) or return values (post-conditions). The workflow policy further defines the subset of parameter and returns that are populated between each individual step as a dataflow. The reason for defining data flow is because one service may return more information than the subsequent service requires. Also, multiple concurrent services may precede a single service. In this case, a combination of returns from multiple services would precede the subsequent service.

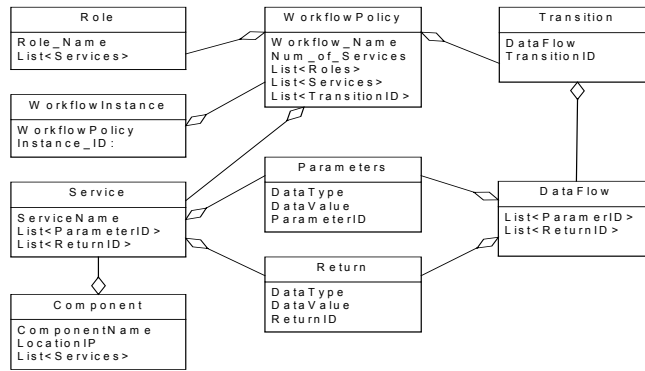


Figure 4.1 Workflow-based Object-Oriented Ontology

5. A Software Engineering Development Approach

A common second step in object-oriented analysis and design is translating the object-oriented domain model into a software design model. In this translation, implementation classes (classes that only pertain to the software implementation domain) are added to the model such as servers, queues, stacks, etc. Also, some domain classes are translated into “proxy” classes (i.e. software classes that represent domain entities). Furthermore, some domain classes are directly transferred to the software design model. The software design model is the basis for the software implementation and development.

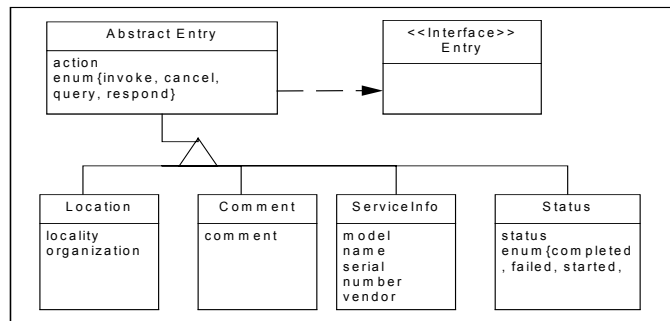


Figure 5.1 Hierarchy of KOJAC-specific Jini Classes

KOJAC specifically isolates the original domain and proxy class implementations. The agents use the software implementations of these classes for communication. In order to facilitate this process, the software designer needs to specialize these classes to the specific set of abstract classes that adds additional communication based information. These set of abstract classes, we take from the set of classes defined in the Jini API that

support JavaSpace functionality. JavaSpace communication relies heavily on the instantiation and use of objects that either implement *Entry* interfaces or subclass the *AbstractEntry* class. These objects can be written, taken, read or notified in the JavaSpace server. Jini further specializes these Entry classes. These derived classes are *Address*, *Comment*, *Location*, *Name*, *ServiceInfo*, *ServiceType*, and *Status*. The structural view of the Entry classes is shown in Figure 5.1. In some cases, the KOJAC approach adds another layer of specialization to include some functionality that was not included in Jini's set of specializations. For example, the native *AbstractEntry* class does not have an action attribute. So, this attribute was added so this class would be more consistent with the WARP workflow environment. As this idea of agent communication expands into other domains, other additions may have to be made to the native Jini classes.

In order to incorporate the domain and proxy classes with KOJAC, the designer must specialize those classes with the pertinent Entry class. If the software development process incorporates the use of Rational Rose, these specializations can be easily made with a few keystrokes. Finally, these agent communication-specific set of classes with the new specializations are compiled into a Java package. This package acts as the shared ontology for the agents. In the run-time environment, agents will reflectively access this ontology using introspection and reflection as defined in the Java development environment. The KOJAC-specific steps as they relate to a typical software development life-cycle is illustrated in Figure 5.2.

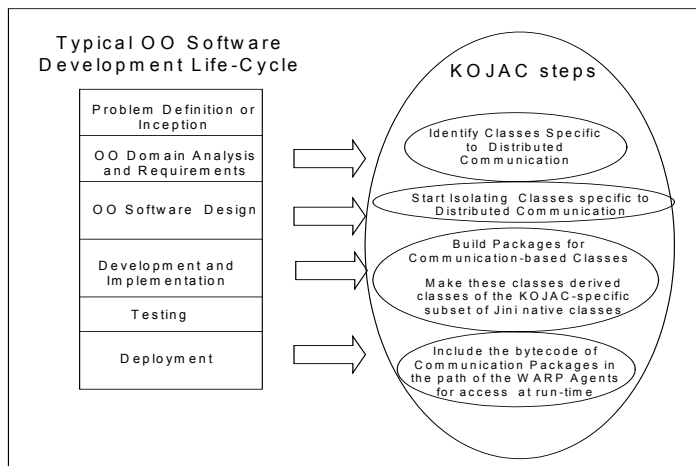


Figure 5.2 KOJAC Steps Integrated with the Software Life-Cycle

KOJAC in the WARP Environment

In order for KOJAC to work in the WARP environment, the classes in Figure 4.1 were used as the distributed communication based classes. These workflow-oriented classes

should derive from the native Jini classes illustrated in 5.1. The WARP approach translates the domain classes illustrated in Figure 4.1 to be derived of Jini classes. In Figure 5.3, we use stereotype notation (i.e. << >>) to show from which of Jini-based classes that each of the workflow-oriented classed are derived. The Service, Parameter, Return, DataFlow, and Transition are all Status Entry classes that get passed among the RMAs and WMAs. The Component class is a Location Entry class because it reveals the location of the components that the RMAs will be invoking. Roles, WorkflowPolicy, and WorkflowInstance classes are ServiceInfo Entry classes. Interpretations of the type of Entry Classes will vary from domain to domain. Later discussions of the JavaSpace will show how sub-classing the domain classes is important for object matching.

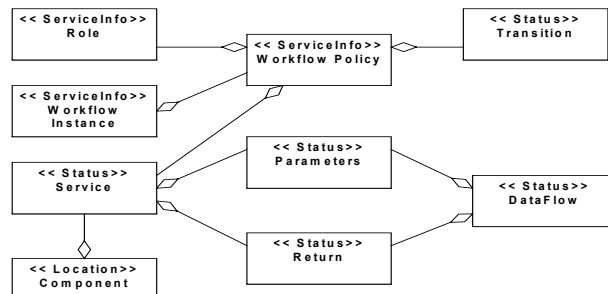


Figure 5.3 Entry Class Specializations for the WARP Ontology

6. KOJAC

The KOJAC approach has a set of semantics and an operational environment that extensively incorporates the operations of Sun Microsystems' JavaSpace technology [5]. This section gives an introduction of JavaSpace technology and then describes the operational semantics and tools associated with the KOJAC approach.

6.1 Using Jini / JavaSpace Technology

Jini technology is a suite of services developed by Sun Microsystems that provide a simple substrate for distributed computing [7]. Jini supports most common principles surrounding distributed coordination (i.e. remote objects, leasing, transactions, and distributed events). It is not in the scope of this paper to give an in-depth description of Jini but to describe those services that are used for agent communication, specifically JavaSpaces [9]. JavaSpace technology is based on "TupleSpaces". TupleSpaces, first introduced in the context of the "Linda" project in 1982, allows distributed software processes to communicate autonomously. The tupleSpace emulates a data storage server. The server receives entries from independent components and stores them for retrieval. Exterior components can be notified when an entry of a certain pattern or tuple is entered. Components can also read and take matching entries based on a tuple-based pattern they submit. Though JavaSpaces technology was motivated by tupleSpace, it is slightly

different. JavaSpaces is an “object” storing service. It supports read, write, take, and notify on actual software objects. A few basic JavaSpaces interactions are illustrated in Figure 6.1. Distributed components can register to a JavaSpaces server. Distributed component A can request notification when a test object is inserted into the server as in step 1. When that test object is written into the server by component B, component A will receive an event notification. Component C can read the test object and receive a copy, while component D takes the test object and removes it from the server.

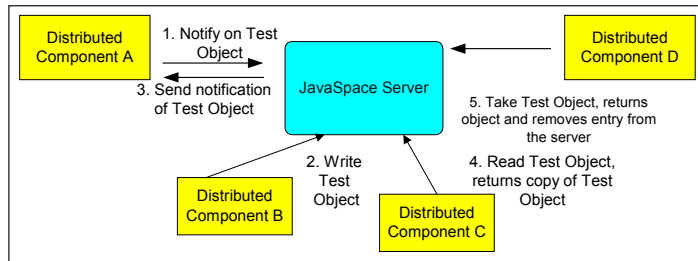


Figure 6.1 Basic JavaSpace Functions

6.2 KOJAC:Operational Semantics

KOJAC’s interaction protocols are based on a subset of the reserved KQML performatives from Table 2.1 using the WARP environment as an example. It is not the intent to detail all possible interactions but more to show how typical interactions would occur.

Let’s consider the scenario when an agent registers its interest. An agent can *register and subscribe* by connecting (Line 1) to the JavaSpace server and setting notify commands for all entries that it is interested in. For example, a Broker (Portfolio Management) RMA as in Figure 3.2 would first connect to the JavaSpace server, then it would set notifications for Status entries (Service Class) on services that it can perform. This will enable notifications to be sent to that agent when there are status messages pertinent to its services. Also, the Broker RMA would set notifications for ServiceInfo entries (Workflow Instance Class) that include services that it encapsulates (Line 8,9,14, and 15). Therefore, when an WMA distributes new workflow instances that contains a service that can be fulfilled by the Broker RMA, then that agent is notified. The Java-based syntax to register is detailed in Figure 6.2. This code shows hard-coded service names for demonstration only, however in operational environments this information is dynamically imported from a relational database.

An agent can *tell-all* or *broadcast* a message with the JavaSpace write command. All agents interested in certain information would have naturally subscribed to that type of message as above. After an agent issues the write command all interested agents would be notified by the JavaSpace server. These are just a few performatives that were implemented using Jini/Javaspace technology.

```

[1] // Get reference to Javaspace Server
[2] JavaSpace SpaceWARP = (JavaSpace)rh.proxy();

[3] //Instantiate Status Entry
[4] this_Service = new Service();
[5] Service.ServiceName = "searchPortfolioInfo";
[6] // Other fields are set to null (WILDCARDS)
[7] // Instantiate ServiceInfo
[8] this_WFInstance = new WorkflowInstance();
[9] this_WFInstance.Service = "searchPortfolioInfo";

[10] // Notify on Service
[11] EventRegistration thisReg =
[12] SpaceWARP.notify(thisService, null, null,Lease.ANY, null);

[13] // Notify on WFInstance
[14] EventRegistration thisReg =
[15] SpaceWARP.notify(thisWFInstance, null, null,Lease.ANY,null);

```

Figure 6.2 Java-based Syntax that Implements KOJAC's JavaSpace Functionality

6.3 KOJAC Components

KOJAC consists of a set of object-oriented tools that can be integrated with the Java-based agents to assist in using the JavaSpace and Jini's Entry classes. This toolkit can be incorporated into the agent communication functionality or it can be called remotely through Java Remote Method Invocation (RMI). The component diagram for the KOJAC tools is detailed in Figure 6.2.

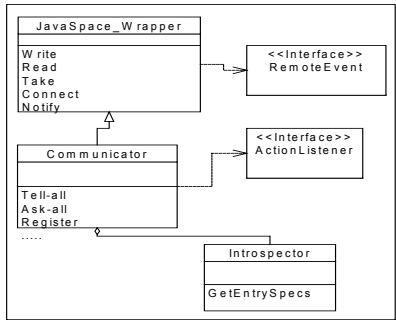


Figure 6.2 KOJAC Component Diagram

The KOJAC architecture consists of a Communicator class. The KOJAC architecture consists of a Communicator class that inherits functionality from a JavaSpace_wrapper. The JavaSpace_wrapper class implements all of the native JavaSpace commands. The Introspector class looks into the ontology-based package to construct entries used by the

Communicator class. The Communicator class also brokers events between the JavaSpace server and the agents.

When a component completes a service, it fires a completion event. The completion event is captured by the RMA. Since the WARP agents are workflow-based, they contain internal intelligence mapping workflow-based events to agent communication actions. This mapping would be different in different domains and would have to be incorporated in the agents of that domain. The RMA classifies the event as a completion. The RMA invokes the Tell-all method from within the Communicator class as in Figure 6.2. Communicator incorporates the Introspector class within its Tell-all functionality. The Introspector searches the ontology-based package (Java bytecode) for an entry class that has the same name as the completed service. The introspected class is returned and the action field is populated as a completion. Finally, the inherited write function (from JavaSpace_wrapper parent class) is called with introspected class as a parameter.

7. KOJAC Prototype

A prototype of the WARP architecture was implemented using 3 Dell workstations [1]. One workstation containing the WMA was contained on a Dell Workstation running Windows NT Server 4.0. This workstation was running Apache's Tomcat webserver and contained an Oracle 8i relational database. This workstation also contained Sun Microsystems' JavaSpace server. Two other Dell Workstations, running Windows 98 were connected as peers to the initial workstation. The peer workstations each contained RMAs. This environment was used to simulate the workflow coordination of distributed components. Each peer-level workstation contained several JavaBean components that acted as the underlying services in this prototypical workflow management scenario. The aforementioned operational environment is illustrated in Figure 7.1. Early results from the WARP architecture showed that there is a high degree of overhead when reflectively invoking the component-based services, specifically with large numbers of concurrent workflow instances. The WARP architecture dynamically accesses and invokes JavaBean components that are available on Java's RMI registry. Major overhead is associated with the introspection of components that are registered on that registry as opposed to components that are on the local disk. However, since the bytecode for the communication-based ontology is local to the WARP agents, the communication classes do not have to be registered on the registry and reflection occurs locally. These interactions do not cause a great deal of overhead even with a large number of concurrent workflow instances being executed. However, a future expectation is to distribute the agent communication knowledge to prevent maintaining multiple copies. This would entail including the agent communication byte code on the registry, which is clearly a potential problem for future research in addition to being a problem widely known in the area of distributed object management.

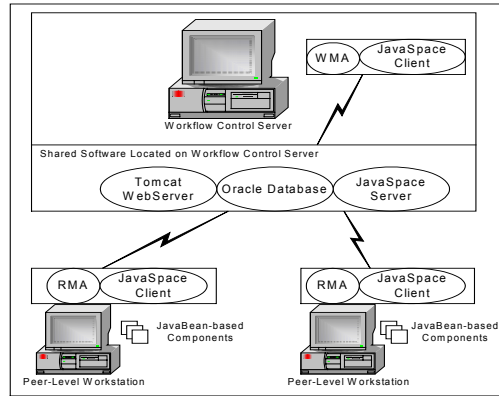


Figure 7.1 KOJAC/WARP Prototype Environment

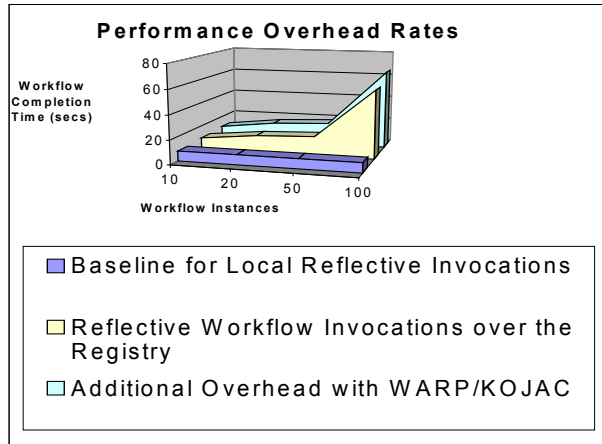


Figure 7.2 Performance Overhead Rates

Three important performance measurements are shown in Figure 7.2. The first measurement baselines the performance when a workflow of components (10 JavaBean components) are invoked locally through reflection (this is without WARP/KOJAC). This performance can be measured with differing numbers of concurrent workflows (i.e. workflow instances are initiated instantaneously). The workflow completion time in this case was almost consistently ~8.5 seconds. The second performance measurement shows the same circumstances as the first, however the components are invoked reflectively from a registry. This added substantial overhead as mentioned above. Finally, the final measurements contain all of the dynamic management ability of the WARP/KOJAC

architecture as a workflow of components are reflectively invoked over a registry. Outside of the overhead associated with reflection, WARP/KOJAC only adds an additional 15%. This 15% rate is consistent as workflow instances vary from 10 to 100.

8. Discussion and Future Work

This paper suggests an approach to agent communication that implements KQML semantics using Jini services. Two main focuses in specifying an implementation for agent communication languages are developing a standard suite of APIs that support message transfer and an infrastructure of services that support basic facilitation services [16]. The problem with this currently is that there are many different implementations that tend to deviate from the semantics. KOJAC standardizes an implementation by integrating a standard ACL into a known set of tools and services. By using the primitive structures and functions, other agent-based developers using Java-based technologies can incorporate the same semantics. By using Jini services, agent communication inherits common distributed programming features by default. This use also enforces the standardization of the agent communication semantics.

This approach integrates well with current software development lifecycles as the Rational Unified Process (RUP) [14]. In fact, tools implementing RUP can automatically generate the source code that is used as the agent communication ontology. Using bytecode as a method for storing agent communication knowledge makes a useful connection between software development processes and agent integration. In addition, this approach fits seamlessly with current distributed event-based development tools like Jini. However, this research has proven that with the distribution of this bytecode across networks and among separate networks, there is a huge amount of performance overhead. This overhead may even suggest that this approach may be impractical when large numbers of components are distributed among multiple networks.

Consequently, performance results in this work have opened avenues for future research. Initially, we plan to investigate other architectures that may efficiently support this approach to agent communication. Another area of research is storing the ontology in XML schema or even ACML schema. This research would be promising in making a connection to other relevant agent communication efforts. The best connection would be the translation of the agent ontology from the KOJAC software design steps illustrated in Figure 4.3 directly to XML/ACML-based semantics. Finally, communicative acts should be modeled using FIPA-ACL semantics.

9. References

- [1] Blake, M.B. " B2B Electronic Commerce: Where do Agents Fit In? ", Proceedings at the AAAI-2002 Workshop on Agent Technologies for B2B E-Commerce/AAAI Press, Edmonton, Alberta, Canada, July 28, 2002
- [2] Blake, M.B. " WARP: Workflow Automation through Agent-Based Reflective Processes ", *Proceedings at the 5th International Conference on Autonomous Agents*, Montreal, Canada,

May 2001 (software demonstration)

- [3] Blake, M.B. Rule-Driven Coordination Agents: A Self-Configurable Architecture, *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS2001)*, Dallas, TX, IEEE Computer Society Press, March 2001
- [4] Blake, M.B. WARP: An Agent-Based Process and Architecture for Workflow-Oriented Distributed Component Configuration. *Proceedings of the 2000 International Conference of Artificial Intelligence*, Las Vegas, NV, June 2000
- [5] Blake, M.B., KOJAC: Implementing KQML with Jini to Support Agent-Based Communications in Emarkets, AAAI-2000 Workshop on Knowledge-based Electronic Markets (KBEM2000) (AAAI Press, Technical Report WS-00-04) Austin, TX, August 2000
- [6] Decker, K., Sycara, K., and Williamson, M. Middle Agents for the Internet, *In the Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan.
- [7] Edwards, K. *Core Jini*. Upper Saddle River, N.J.: Prentice Hall 1999
- [8] FIPA Interaction Protocol Specification(2002), [http:// www.fipa.org/repository/ips.html](http://www.fipa.org/repository/ips.html)
- [9] Freeman, E., Hupfer, S., and Arnold, K. *JavaSpaces Principles, Patterns, and Practice*, Reading, MA.:Addison Wesley 1999
- [10] Gelernter, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112 1985
- [11] Gomaa H and Kerschberg, L. An Evolutionary Domain Life Cycle Model for Domain Modeling and Target System Generation, *In Proceedings of the Workshop on Domain Modeling for Software Engineering, International Conference on Software Engineering*, Austin, TX 1997
- [12] Groszof, B. and Labrou, Y., An Approach to using XML and a Rule-based Content Language with an Agent Communication Language, *IJCAI-99 Workshop on Agent Communication Languages*, Stockholm, Germany 1999
- [13] Groszof, B., Labrou, Y. and Chan, H. "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. *Proceedings of the 1st ACM Conference on Electronic Commerce (EC-99)* Denver, Colorado: ACM Press, 1999.
- [14] Krutchen, P. *The Rational Unified Process: An Introduction (2nd Ed.)*. Prentice Hall, 2000
- [15] Labrou, Y. and Finin, T. A semantics approach for KQML – a general purpose communication language for software agents. *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM-94)*, Gaithersburg, MD 1994
- [16] Labrou, Y., Finin, T. and Peng, Y. "The current landscape of Agent Communication Languages", *Intelligent Systems*, 14(2): IEEE Computer Society 1999
- [17] Lei, K. and Singh, M. A Comparison of Workflow Metamodels, *Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling*, Los Angeles, CA 1995