Jackal: a Java-based Tool for Agent Development *

R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff
Laboratory for Advanced Information Technology
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland

James Mayfield
Research and Technology Development Center
Johns Hopkins University Applied Physics Laboratory
Laurel, Maryland

Akram Boughannam
CIIMPLEX Project Office
Advanced Manufacturing Solutions Development
IBM Corporation
Charlotte, North Carolina

Abstract

Jackal is a Java-based tool for communicating with the KQML agent communication language. Some features which make it extremely valuable to agent development are its conversation management facilities, flexible, blackboard style interface and ease of integration. Jackal has been developed in support of an investigation of the use of agents in shop floor information flow. This paper describes Jackal at a surface and design level, and presents an example of its use in agent construction.

Introduction

Jackal is a Java package that allows applications written in Java to communicate via the KQML (Finin, Labrou, & Mayfield 1997) agent communication language. It is designed to be used as a 'tool' by other applications, in that it does not require that applications be modified or extend some standard shell. Additionally, Jackal is designed so that multiple instances of it, and therefore multiple agents, may be run within the same Java Virtual Machine. Jackal:

- Facilitates the sending and receipt of KQML messages.
- Implements a conversation-based approach to dialogue management.
- Presents a blackboard interface to the agent.
- Provides portability through Java.
- Supports the use of multiple transport protocols.
- Requires no modification to existing code.
- Implements a complete scheme for agent naming and addressing.

Jackal has been developed as part of a larger effort to develop an agent infrastructure for manufacturing information flow. It has been used to facilitate communication among diverse agents responsible for collecting, processing and distributing information on a manufacturing shop floor.

In next section, we introduce the application domain in which Jackal has been developed. This is followed by a discussion of Jackal's features and their value to the agent development environment. Next, we expand on Jackal's conversation management abilities. Then, the system's design and several key components are presented. Finally, an example agent is used to illustrate the use of Jackal in construct communicating agents.

CIIMPLEX

With matching funds from the National Institute of Standards and Technology, the Consortium for Intelligent Integrated Manufacturing Planning-Execution (CIIMPLEX), consisting of several private companies and universities, was formed, with the goal of developing technologies for intelligent enterprise-wide integration of planning and execution for manufacturing (Chu *et al.* 1996; Peng *et al.* 1998). CIIMPLEX adopts as one of its key technologies the approach of intelli-

*This work is supported in part by the Advanced Technology Program, administered by the National Institute of Standards and Technology, under agreement number: 70NANB6H2000.

gent software agents, and develops a multi-agent system (MAS) for enterprise integration.

One of the key objectives of CIIMPLEX is to establish a monitoring/notification architecture for enterprise integration (Dourish & Bellotti 1992). In this architecture, an application defines events in which it is interested (e.g. changes in process rates, yield, material due dates) and requests that agents monitor these events. When these events occur, the responsible agents notify the appropriate applications. A natural way to implement the monitoring/notification architecture is to use broker agents to track and identify other agents that can provide event monitoring services.

Within the architecture of the MAS for the CIIMPLEX enterprise, an Agent Name Server and Broker Agent allow agents to be located by name and advertised services, respectively. In addition, several other types of agents are required for enterprise integration. For example, data-mining/parameter-estimation agents are needed to collect, aggregate, interpolate and extrapolate the raw transaction data of the low level (shop floor) activities, and to make this aggregated information available for higher level analyses by other agents. Event monitoring agents monitor, detect, and present abnormal events that require attention. The CIIMPLEX Analysis Agents evaluate disturbances to the current planned schedule and recommend appropriate actions to address each disturbance. Scenario Coordination Agents assist human decision making for specific business scenarios by providing the relevant context, including filtered information, actions, and well as workflow charts.

All these agents speak KQML, as implemented in Jackal, and use a subset of KIF (Genesereth 1992) that supports Horn clause deductive inference as the content language. The shared ontology is an agreement document established by the application vendors and users and other partners in the consortium. The agreement adopts the format of the Business Object Document defined by the Open Application Group.

Jackal and Agent Development

Agents which will interact with one another require some method of communication in order to coordinate their activities and distribute and collect information. To this end, several agent communication languages (e.g., KQML (Finin, Labrou, & Mayfield 1997), FIPA ACL (FIPA 1997), ARCOL (FIPA 1997), ICL (Martin, Cheyer, & Moran 1998), AgenTalk (Kuwabara, Ishida, & Osato 1995; Kuwabara 1995), KaOS (Bradshaw 1996; Bradshaw *et al.* 1997), AOP (Shoham 1993) and MAP (Eriksson *et al.* 1998)), and various software tools for them (e.g., TKQML (Cost *et al.* 1997), OAA (Martin, Cheyer, & Moran 1998), JAT and JATLite (Frost 1998; Petrie 1998), Magenta (Genesereth & Katchpel 1994) and AgentBase (Eriksson *et al.* 1998)), have been developed. Jackal is a tool for the use of KQML by agents written in the Java programming language. Java is a useful language for writing agents because it is platform independent, as an interpreted language, and has good language support for multi-threading. Jackal benefits from these properties, and relies exclusively on the core Sunsoft JDK 1.1 classes. This maximizes the likelihood that Jackal-based agents can run without modification on any platform that supports Java. Not only can Jackal-based agents run on diverse or remote environments; many may coexist within the same Java Virtual Machine. This is exploited by transparent protocol adapters for shared memory message passing.

Adding communication abilities to any Java program requires no modification of existing code. This is because Jackal's functionality is accessed through a class instance, which can be shared among agent components like a portable two-way radio. This is in contrast to systems which require that a program subclass an agent shell, or otherwise restructure itself. With this Jackal instance, the agent gains more than just the ability to send and receive messages, however. Jackal's design is based in large part on, and implements, the KQML Naming Scheme (KNS), an evolving standard for resolving agent names in a hierarchically structured, dynamic environment. This means that the agent application need only deal with symbolic agent names, and may leave issues such as physical address resolution and alias identification to the Jackal infrastructure.

Two components which work together to provide the greatest benefit to the agent are the conversation management routines and the Distributor, a blackboard for message distribution. The conversation system supports the use of easily interchangeable protocols for interaction, which guide the behavior of the system. The Distributor presents a flexible, active interface for internal message retrieval by agent components. While the Distributor optimizes access to the message flow, it is the conversation system which gives it its real value; the next section will discuss in depth the rational behind the conversation-based approach.

Conversation-Based Protocols

The study of agent communication languages (ACLs) is one of the pillars of current agent research. KQML and the FIPA ACL are the leading candidates as standards for specifying the encoding and transfer of messages among agents. While KQML is good for message-passing among agents, the message-passing level is not actually a very good one to exploit directly in building a system of cooperating agents. After all, when an agent sends a message, it has expectations about how the recipient will respond to the message. Those expectations are not encoded in the message itself; a higher-level structure must be used to encode them. The need for such conversation policies is increasingly recognized by the KQML community, and has been formally recognized in the latest FIPA draft standard (FIPA 1997; Dickenson 1997).

It is common in KQML-based systems to provide a message handler that examines the message performative to determine what action to take in response to the message. Such a method for handling incoming messages is adequate for very simple agents, but breaks down as the range of interactions in which an agent might participate increases. Missing from the traditional message-level processing is a notion of message context.

We claim that the unit of communication between agents should be the conversation. A conversation is a pattern of message exchange that two (or more) agents agree to follow in communicating with one another. In effect, a conversation is a communications protocol, albeit one that may be initiated through negotiation, and may be short-lived relative to the way we are accustomed to thinking about protocols. A conversation lends context to the sending and receipt of messages, facilitating more meaningful interpretation. The adoption of conversation-based communication carries with it numerous advantages to the developer, including:

- There is a better fit with intuitive models of how agents will interact than is found in message-based communication.
- There is also a closer match to the way that network research approaches protocols, which allows both theoretical and practical results from that field to be applied to agent systems.
- Conversation structure is separated from the actions to be taken by an agent engaged in the conversation. This allows the same conversation structure to be used by more than one agent, in more than one context. In particular, two agents can use the same conversation structure to ensure that they will engage in the same dialogue.
- The standard advantages of the underlying ACL accrue, including language-independence and ontology-independence.

To date, little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Strides must be taken in the following directions:

- Potential conversations must be easy to specify.
- Conversation specifications must be easy to reuse.
- Libraries of standard conversations should be developed.
- An ontology of conversations must be developed.

To achieve these goals, we must solve three main problems:

1. Conversation specification: How can conversations best be described so that they are accessible both to people and to machines?
2. Conversation sharing: How can an agent use a conversation specification standard to describe the conversations in which it is willing to engage, and to learn what conversations are supported by other agents?
3. Conversation aggregation: How can sets of conversations be used as agent 'APIs' to describe classes of capabilities that define a particular service or capability?

Conversation specification

A specification of a conversation that could be shared among agents must contain several kinds of information about the conversation and about the agents that will use it. First, the sequence of messages must be specified. We advocate the use of deterministic finite-state automata (DFAs) for this purpose; DFAs can express a wide variety of behaviors while remaining conceptually simple. Next, the set of roles that agents engaging in a conversation may play must be enumerated. For example, a conversation that allows a sensor to report an unusual condition to all interested agents would have two roles: sensor and broker (which would in turn be specializations of sentinel and sentinel-consumer roles). Most conversations will be dialogues, and will specify just two roles; conversations with more than two roles are perfectly acceptable though, and represent the coordination of communication among several agents in pursuit of a single common goal.

DFAs and roles dictate the syntax of a conversation, but say nothing about the conversation's semantics. The ability of an agent to read a description of a conversation, then engage in such a conversation, demands that the description specify the conversation's semantics. To be useful though, such a specification must not rely on a full-blown, highly expressive knowledge representation language. We believe that a simple ontology of common goals and actions, together with a way to relate entries in the ontology to the roles, states, and transitions of the conversation specification, will be adequate for most purposes. This approach sacrifices expressiveness for simplicity and ease of implementation. It is nonetheless perfectly compatible with attempts to relate conversation policy to the semantics of underlying performatives, as proposed for example by (Bradshaw 1996; Bradshaw *et al.* 1997).

These capabilities will allow the easy specification of individual conversations. To develop systems of conversations though, developers must have the ability to extend existing conversations through specialization and composition. Specialization is the ability to create new versions of a conversation that are more detailed than the original version; it is akin to the idea of subclassing in an object-oriented language. Composition is the ability to combine two conversations into a new, compound conversation. Development of these two capabilities will entail the creation of syntax for expressing a new conversation in terms of existing conversations, and for linking the appropriate pieces of the component conversations. It will also demand solution of a variety of technical problems, such as naming conflicts, and the merger of semantic descriptions of the conversations.

Conversation sharing

A standardized conversation language, as proposed above, dictates how conversations will be represented; however, it does not say how such representations are shared among agents. While the details of how conversation sharing is accomplished are more mundane than those of conversation representation, they are nevertheless crucial to the viability of dynamic conversation-based systems. Three questions present themselves:

- How can an agent map from the name of a conversation to the specification of that conversation?
- How can one agent communicate to another the identity of the conversation it is using?
- How can an agent determine what conversations are handled by a service provider that does not yet know of the agent's interest?

Conversations Sets as APIs

The set of conversations in which an agent will participate defines an interface to that agent. Thus, standardized sets of conversations can serve as abstract agent interfaces (AAIs), in much the same way that standardized sets of function calls or method invocations serve as APIs in the traditional approach to system-building. That is, an interface to a particular class of service can be specified by identifying a collection of one or more conversations in which the provider of such a service agrees to participate. Any agent that wishes to provide this class of service need only implement the appropriate set of conversations. To be practical, a naming scheme will need to be developed for referring to such sets of conversations, and one or more agents will be needed to track the development and dissolution of particular AAIs. In addition to a mechanism for establishing and maintaining AAIs, standard roles and ontologies, applicable to a wide variety of applications, will need to be created.

There has been little work on communication languages from a practitioner's point of view. If we set aside work on network transport protocols or protocols in distributed computing (e.g., CORBA) as being too low-level for the purposes of intelligent agents, the remainder of the relevant research may be divided into two categories. The first deals with theoretical constructs and formalisms that address the issue of agency in general and communication in particular, as a dimension of agent behavior (e.g., AOP (Shoham 1993)). The second addresses agent languages and associated communication languages that have evolved to some degree to applications (e.g., TELESCRIPT (White 1995)). In both cases, the bulk of the work on communication languages has been part of a broader project that commits to specific architectures.

Agent communication languages like KQML provide a much richer set of interaction primitives (e.g., KQML's performatives), support a richer set of communication protocols (e.g., point-to-point, brokering, recommending, broadcasting, multicasting, etc.), work with richer content languages (e.g., KIF), and are more readily extensible than any of the systems described above. However, as discussed above, KQML lacks organization at the conversation level that lends context to the messages it expresses and transmits. Limited work has been done on implementing conversations for software agents, and almost none has been done on expressing those conversations. As early as 1986, Winograd and Flores (Winograd & Flores 1986) used state transition diagrams to describe conversations. The COOL system (Barbuceanu & Fox 1995) has perhaps the most detailed current finite state automata model to describe agent conversations. Each arc in a COOL state transition diagram represents a message transmission, a message receipt, or both. One consequence of this policy is that two different agents must use different automata to engage in the same conversation. We believe that a conversation standard should clearly separate message matching from actions to be carried out when a match occurs; doing so will allow a single conversation specification to be used by all participants in a conversation. This, in turn, will allow conversation specifications to describe standard services, both from the viewpoint of the service provider, and from that of the service user.

COOL also uses an :intent slot to allow the recipient to decide which conversation structure to use in understanding the message. This is a simple way to express the semantics of the conversation. We argue below that more general descriptions of conversation semantics will be needed if agents are to acquire and engage in new conversations on the fly. The challenge will be to develop a language that is general enough to express the most important facts about a conversation, without being so general that it becomes an intellectual exercise, or is too computationally expensive to implement.

Other conversation models that have been developed include those of Parunak (Parunak 1996), Chauhan (Chauhan 1997), who uses COOL as the basis for his multi-agent development system, Kuwabara et al. (Kuwabara, Ishida, & Osato 1995; Kuwabara 1995), who add inheritance to conversations, Nodine and Unruh (Nodine & Unruh 1997), who use conversation specifications to enforce correct conversational behavior by agents, Bradshaw (Bradshaw 1996), who introduces the notion of a conversation suite as a collection of commonly-used conversations known by many agents, and Labrou (Labrou 1996), who uses definite clause grammars to specify conversations. While each of these makes contributions to our general understanding of conversations, none show how descriptions of conversations might be shared by agents and used directly by them in implementing conversations.

Defining common agent services via conversations

A significant impediment to the development of agent systems is the lack of basic standard agent services that can be easily built on top of the conversation architecture. Examples of such services are: name and address resolution; authentication and security services; brokerage services; registration and group formation; message tracking and logging; communication and interaction; visualization; proxy services; auction services; workflow services; coordination services; and performance monitoring services. Services such as these have typically been implemented as needed in individual agent development environments. Two such examples are an agent name server and an intelligent broker.

Agent Name Server At first blush, the problem of mapping from an agent name to information about that agent (such as its address) seems trivial. However, solving this problem in a way that can easily scale as the number of users and amount of data to be processed grows is difficult. We believe that development of a successful symbolic agent addressing mechanism demands at least two advances:

1. A simple naming convention to place each role an agent might play in an organization at a unique point in a namespace for that organization. Currently there is no widely-accepted mechanism for universal unique agent naming (in the way that there now is, e.g., for Internet hosts or web documents).

2. An efficient, scalable name service protocol for mapping from symbolic role names to information about the agents that fill those roles.

To a large extent, the desired techniques can be modeled after existing name service techniques such as DNS (which is widely implemented) and CORBA (whose namespace mechanisms are only narrowly implemented). Such techniques are well-studied, highly reliable, and scalable. Agent name service will differ from DNS primarily in that agents will tend to appear, disappear, and move around more frequently than do Internet hosts. This will necessitate the development of naming conventions that are less rigid than those used in DNS, and algorithms for mapping from names to agent information that do not rely on the static local databases found in DNS.

Intelligent Broker A system that is to respond to the demands of multiple users, with needs that vary over time, under an ever-increasing query load must be able to do on-the-fly matching of queries to documents and services. In an agent-based architecture, this means that one agent must be able to dynamically discover other agents based on the content of their knowledge. It should exploit the research on conversations and the symbolic agent-addressing scheme described above, while at the same time fitting neatly into existing brokered systems. Such systems will continue to see a single broker where there had been a single broker all along; now, however, that broker will have the option of coordinating many other disparate brokers of varying capabilities.

An Overview of Jackal's Design

Jackal was designed to provide comprehensive functionality, while presenting a simple interface to the user. Thus, although Jackal consists of roughly seventy distinct classes, all user interactions are channeled through one class, hiding most details of the implementation.

Figure 1 presents the principal Jackal components, and the basic message path through the system. We will first discuss each of the components, and then, to illustrate their interaction, trace the path of a message through the system (that is, as it is received by Jackal, passed on to and replied to by an agent thread, and the reply sent back to the original sender).

Intercom

The Intercom class is the bridge between the agent application and Jackal. It controls startup and shutdown of Jackal, provides the application with access to internal methods, houses some common data structures, and plays a supervisory role to the communications infrastructure.

Transport Interface

Jackal runs a Transport Module for each protocol it uses for communication. Jackal 3.0 comes with a module for TCP/IP, and users can create additional modules for other protocols. A Transport Module is responsible for receiving messages at some known address, and transmitting messages out via a given protocol.

Message Handler

Messages received by the Switchboard must be directed to the appropriate place in the Conversation Space; this is the role of the Message Handler. Messages are associated with current (logical) threads based on their ID (the value of the 'reply-with' field). This directs their assignment to ongoing conversations when possible. If no such assignment can be made, a new conversation appropriate to the message is started.

Conversations

Based largely on the work of Labrou and Finin (Labrou 1996; Labrou & Finin 1997) regarding a semantics for KQML, we have created protocols, which describe the correct interactions for various performatives and subsequent messages. The protocol for ask-one, for example, specifies among other things that a reply must be a tell, untell, deny, sorry or error. These protocols are 'run' as independent threads for all current conversations. This allows for easy context management, while providing constraints on language use and a framework for low level conversation management. This is in contrast with earlier approaches (e.g., TKQML (Cost *et al.* 1997)) which require the agent to maintain context on their own.

The Conversation Space is a virtual entity, consisting of the collection of conversations started by the Message

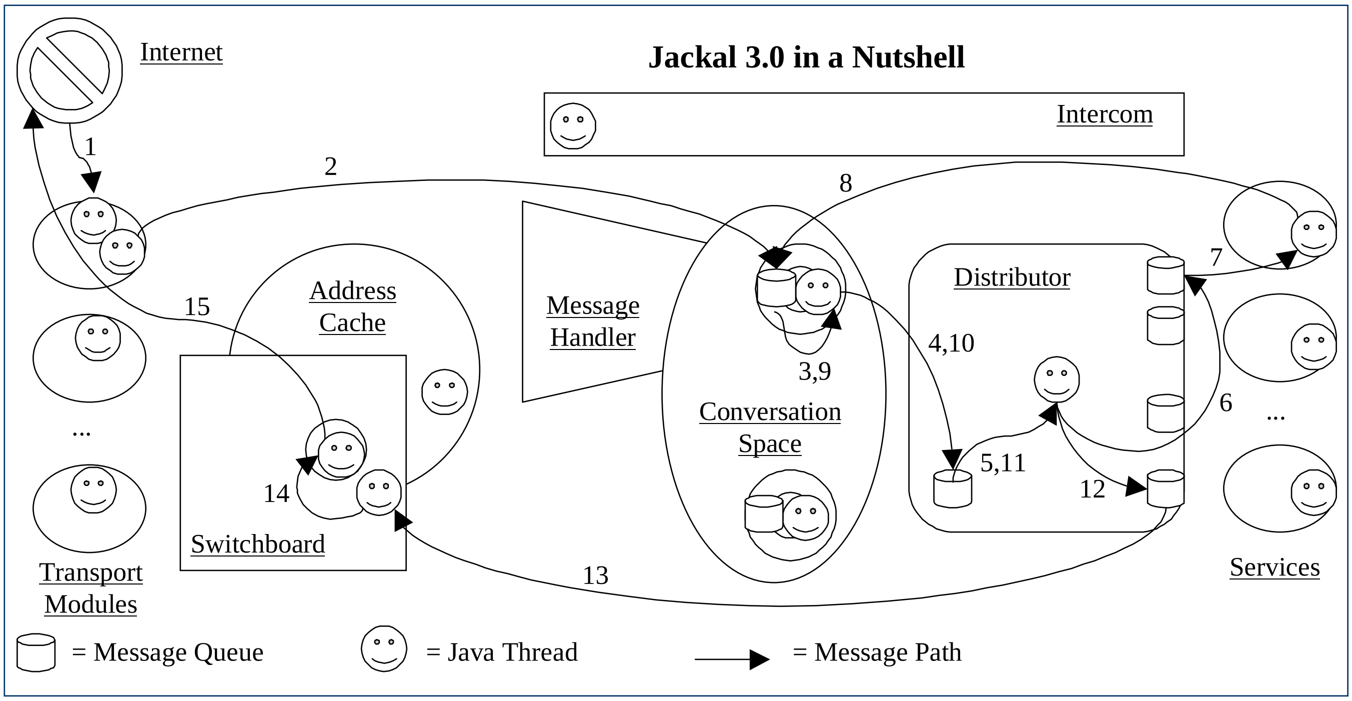

Figure 1: Jackal Architecture and Message Flow

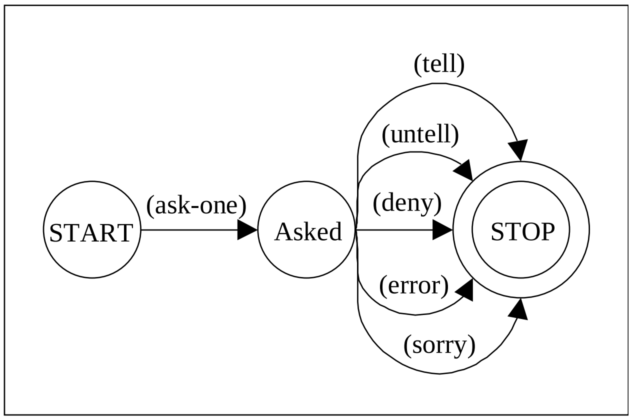

Figure 2: DFA for KQML ask-one conversation

Handler. These conversations run individual protocol interpreters.

As of Jackal 3.0.4, conversation templates (or specifications) are completely independent from the Java library. Rather, they are specified by URLs, and loaded at runtime from some remote source. Figure 3 shows the conversation template for a standard KQML conversation available from the Jackal host. This template corresponds to the finite state machine depicted in Figure 2.

The conversation management component offers a number of significant benefits to the agent:

- Running conversations in individual threads provides maximum flexibility.
- Conversations, in conjunction with the Distributor, route messages automatically to the threads which need them.
- Each conversation maintains a local store, which can be accessed by the agent via a message ID, and which serves as the conversation's context.
- Since conversations are declaratively specified, they can be loaded (remotely or locally) on demand. Our current agents download at initialization only the conversations they will need.
- The conversation mechanisms and the specification are almost completely independent of the content or message language used, and so could be easily be tuned work in a 'multi-lingual' environment.
- Actions can be associated with conversation structures, enhancing their utility. While our system supports this, the preliminary specification language does not. Extending it to include actions will require first the development of an ontology enumerating actions implemented by conversation interpreters.

Distributor

The Distributor is a Linda-like (Carriero & Gelertner April 1989) blackboard, which serves to match messages with requests for messages. This is the sole interface between the agent and the message traffic. Its concise API allows for comprehensive specification of message requests. Requesters are returned message queues, and

```
// Conversation Template
// Convention: Initial and accepting states all caps,
//             other states initial caps,
//             arc-labels lower case.
(conversation
  (name kqml-ask-one)
  (author "R. Scott Cost")
  (date "3/4/98")
  (start-state START)
  (accepting-states TOLD)
  (transitions
   (arc (label ask-one) (from START) (to Asked) (match "(ask-one)"))
   (arc (label tell)    (from Asked) (to TOLD)  (match "(tell)"))
   (arc (label deny)    (from Asked) (to TOLD)  (match "(deny)"))
   (arc (label untell)  (from Asked) (to TOLD)  (match "(untell)"))
   (arc (label sorry)   (from Asked) (to TOLD)  (match "(sorry)"))
   (arc (label error)   (from Asked) (to TOLD)  (match "(error)"))))
```

Figure 3: Conversation Template for KQML 'ask-one'

receive all return traffic through these queues. Requests for messages are based on some combination of message, conversation or thread ID, and syntactic form. They also permit actions, such as removing an acquired message from the blackboard or marking it as read only. A priority setting determines the order or specificity of matching. Finally, requests can be set to persist indefinitely, or terminate after a certain number of matches.

Services

A service here is any thread; this could be a Jackal service, or threads within the agent itself. The only thing that distinguishes among threads is the request priority they use. System, or Jackal, threads choose from a set of higher priorities than agent threads, but each chooses a level within its own pool. Jackal reserves the highest and lowest priorities for services directing messages out of the agent and for those cleaning the blackboard, respectively.

Message Routing

The Switchboard acts as an interface between the Transport Modules and the rest of Jackal. It must facilitate the intake of new messages, which it gathers from the Transport Modules, and carry out send requests from the application. The latter is a fairly complicated procedure, since it has multiple protocols at its disposal. The Switchboard must formulate a plan for the delivery of a message, with the aid of the Address Cache, and pursue it for an unspecified period of time, without creating a bottleneck to message traffic. In addition, it is equipped to handle the delivery of messages with multiple recipients or 'cc' fields.

Naming and Addressing/Address Cache

In any multi-agent system, the problem of *agent naming* arises: how do agents refer to each other in a simple, flexible, and extensible way? If the system in question employs a standard communication language such as KQML, another requirement is that agents must be able to refer to KQML-speaking agents in the outside world. Within the development of Jackal, we propose KNS, a naming scheme designed to support collaborating, mobile KQML-speaking agents using a variety of transport protocols. Jackal supports KNS transparently through an intelligent address cache.

KNS is a hierarchical scheme similar in spirit to DNS. A fully-qualified agent name (FQAN) is a sequence of agent names, separated by periods. The sequence describes registrations between agents, so that the agent preceding a period is registered with the agent following it. An example of a FQAN is

`phil.cs.http://www.umbc.edu/ans/`

In this example, "phil" is an agent registered with agent "cs", which in turn is registered with an agent that lives at the URL `http://www.umbc.edu/ans/`. The final agent in a FQAN is always a URL, providing unique, static location information for the base of an agent registration chain.

When one agent registers with another, a relationship is formed between them. The registering agent (or child) submits information to the agent being registered with (or parent) on how the child may be contacted: socket ports, email addresses, or web servers are all possible. In return, the parent becomes a repository for that information. An agent who knows how to contact agent `cs`, i.e. knows a physical address for `cs`, can contact `phil` by first asking `cs` for `phil`'s contact information. If an agent seeks to register with a new parent using a name already registered with the parent, the name is qualified by an instance number, e.g., `phil[2].cs.http://www.umbc.edu/ans/`.

Agents can register together to form teams, and can maintain multiple identities to represent roles in the multi-agent system. Multiple registrations for an agent become a network of aliases for that agent; if one name becomes inaccessible, another can be identified to fill the gap. Moreover, since agents can maintain multiple contact information for each name, agents can change locations and leave forwarding arrangements for messages while they migrate. In this way, dynamic group formation is supported.

KNS also provides a set of protocols for maintaining an agent's identity independently of any one of its registered names. This unburdens the agent and those seeking it from the need to manage dynamic sets of identities over time. For example, after a series of name registrations and unregistrations, an agent can be located through KNS via any one of the names it has previously used.

The Address Cache holds agent addresses in order to defray lookup costs. It is a multilayered cache supporting various levels of locking, allowing it to provide high availability. Unsuccessful address queries trigger underlying KNS lookup mechanisms, while blocking access to only one individual listing.

Message Path

Having described the various components of Jackal, we will trace the path of a received message and the corresponding reply, using the numbered arcs in Figure 1 for reference.

The message is first received by a connection thread within a Transport Module [1], perhaps TCP/IP, and is processed and transferred directly to the input queue of either a waiting or new conversation [2]. A unique thread manages each conversation. Methods for the initial processing of the message reside in the Message Handler, but are called by the responsible transport thread. The target conversation, awakened, takes the message from its input queue [3] and tries to advance its state machine accordingly. If accepted, the message is entered into the Distributor [4], an internal blackboard for message distribution. The Distributor examines the message [5] in turn, and tries to match it with any pending requests (placed by Jackal or agent code), in order of a specified priority. Ideally, a match is found, and the message is placed in the queue belonging to the requester [6]. The message may in fact be passed to several requesting threads. This is the point at which the agent gains access to the message flow; through services attending to the blackboard.

Once the requesting service thread picks the message out of its queue [7], it presumably performs some action, and may send a reply or new message; we assume it does. The service has two options at this point. If it does not expect a reply to the message it is sending, the message may be sent via Intercom's send_message method [8]. Otherwise, it should be sent indirectly through the Distributor. This is equivalent to sending alone and then requesting the reply, but allows the request to be posted *first*, eliminating possible nondeterminism. Either way, the message is eventually processed through send_message, which directs it into the conversation space. The message then traces the same path as the previous incoming message [9,10] to the distributor. Note that every message, incoming or outgoing, passes through the conversation space and the Distributor. The message is captured by the Switchboard's outbound message request [11], which has a special, reserved priority. The Switchboard removes new messages from its queue and assigns them each individual send threads [12]; this results in some overhead, but allows sends to proceed concurrently, avoiding bottlenecks due to wide variation in delivery times. The send thread uses the send method of the appropriate transport module to transmit the message.

Using Jackal: An Example

In this section, we will examine an agent that uses Jackal. This agent implements a simplified, working broker, and functions within a suite of Jackal demo agents that includes an agent name server, remote sensor agents and a client. The code for this agent is quite sparse, illustrating the ease with which agents can be constructed using Jackal.

Jackal agents typically consist of one main execution thread and a number of service threads. The broker's main thread is depicted in Figure 4.

```
package Agents.Weather;
import java.lang.*;
import java.util.Vector;
import J3.*;

class broker extends Thread {
  Intercom j3;
  Vector ad = new Vector();

  public static void
  main (String[] argv) { new broker(); }

  broker() { start(); }

  public void run() {
    j3 = new Intercom("broker",
      "ftp://cs.umbc.edu/common.kqmlrc");
    j3.stderr("Agent broker started.");
    BrokerServ brokerServ = new BrokerServ();
    AdvertServ advertServ = new AdvertServ();
  }

  // Advertise service goes here.
  // Broker service goes here.
}
```

Figure 4: Broker Agent

This agent has a 'main' method so that it can be started directly via the command 'java J3.broker'. Alternatively, it could be instantiated by another agent; an important option if several agents are running within the same virtual machine. As an aid to development, each instance of Jackal spawns an individual console, which it uses for standard output. The console can also be used to execute methods in the API directly, including spawning agents, via a simple command interface.

The first parameter to the Intercom constructor is the agent's initial name. As outlined above, this will be augmented at registration into a fully qualified name. The second parameter specifies the agent's resource file, which contains information needed for basic operation. Once Intercom is instantiated, Jackal performs basic startup operations, such as starting listener threads and registering with this agent's primary Agent Name Server. Note the use of the console in the next step.

Finally, the broker starts the threads that will perform its basic services; accepting advertisements (Figure 5), and matching stored advertisements with incoming requests (Figure 6). For clarity, these threads have been separated out from the main broker code.

```
class AdvertServ extends Thread {
  FIFO queue;

  public AdvertServ() { start(); }

  public void run() {
    Message msg, reply;
    queue = j3.attend(null,"(advertise)",
      null,8,false,true,0,true,false);
    while ((msg = (Message)
      queue.dequeue()) != null) {
      ad.addElement(msg);
      try {
        reply = new
          Message("(tell :content true)");
        reply.put(":language","ascii");
        reply.put(":receiver",
          msg.get(":sender"));
        reply.put(":in-reply-to",
          msg.get(":reply-with"));
        j3.send_message(reply);
      } catch (MalformedMessageX e) {
        System.err.println(e);
      }
    }
  }
}
```

Figure 5: Broker Agent: Advertise Thread

The advertisement service illustrates the basic Jackal service construction. Its first step is to place a permanent request with the Distributor for all incoming messages with the performative 'advertise'. All matches should be left for other threads, but with the write permission removed. It is returned a queue from which to fetch results. The thread then enters a cycle in which it waits (blocks) for the next message, processes it, and repeats. Our simple service merely stores the incoming message, and replies that it has been noted. Threads have the option to poll the queues, rather than block.

```
class BrokerServ extends Thread {
  FIFO queue;

  public BrokerServ() {
    start();
  }

  public void run() {
    Message msg, reply = null, a, b;
    int i;
    queue = j3.attend(null,"(broker-one)",
      null,8,false,true,0,true,false);
    while ((msg = (Message)
      queue.dequeue()) != null) {
      try {
        for (i = 0; i < ad.size(); i++) {
          a = (Message) ad.elementAt(i);
          b = (Message) a.get("content");
          if (b.equals(msg)) break;
        }
        if (i == ad.size()) reply = new
          Message("(sorry :content \"\")");
        else reply = new
          Message("(tell :content "+a+")");
        reply.put(":language","ascii");
        reply.put(":receiver",
          msg.get(":sender"));
        reply.put(":in-reply-to",
          msg.get(":reply-with"));
        j3.send_message(reply);
      } catch (MalformedMessageX e) {
        System.err.println(e);
      }
    }
  }
}
```

Figure 6: Broker Agent: Broker Thread

The broker service is only slightly more complicated, but has the same basic structure. Each message received is compared to (the contents of) each existing advertisement, using the comparison method associated with the message. Where possible, this will recursively make use of the comparison method for the given content language. If a match is found, it is forwarded to the requesting agent. Otherwise, a 'sorry' is sent.

Summary

Jackal provides developers with an easy to use facility for KQML, supporting the use of conversation based protocols. In addition, it provides basic services such as hidden address resolution. These features make it a valuable asset in developing agents for manufacturing information flow.

References

Barbuceanu, M., and Fox, M. S. 1995. COOL: A language for describing coordination in multiagent systems. In Lesser, V., ed., *Proceedings of the First International Conference on Multi–Agent Systems*, 17–25. San Francisco, CA: MIT Press.

Bradshaw, J. M.; Dutfield, S.; Benoit, P.; and Woolley, J. D. 1997. KAoS: Toward an industrial-strength open agent architecture. In Bradshaw, J. M., ed., *Software Agents*. AAAI/MIT Press.

Bradshaw, J. M. 1996. KAoS: An open agent architecture supporting reuse, interoperability, and extensibility. In *Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*.

Carriero, N., and Gelertner, D. April, 1989. Linda in context. *CACM* 32(4):444–458.

Chauhan, D. 1997. JAFMAS: A java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati.

Chu, B.; ans R. Wilhelm, W. J. T.; Hegedus, M.; Fesko, J.; Finin, T.; Peng, Y.; Jones, C.; Long, J.; Matthes, M.; Mayfield, J.; Shimp, J.; and Su, S. 1996. Integrating manufacturing softwares for intelligent planning-execution: A CIIMPLEX perspective. In *Plug and Play Software for Agile Manufacturing, SPIE International Symposium of Intelligent Systems and Advanced Manufacturing*.

Cost, R. S.; Soboroff, I.; Lakhani, J.; Finin, T.; and Miller, E. 1997. TKQML: A scripting tool for building agents. In Wooldridge, M.; Singh, M.; and Rao, A., eds., *Intelligent Agents Volume IV – Proceedings of the 1997 Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *LNAI*. Berlin: Springer-Verlag. 336–340.

Dickenson, I. 1997. Agent standards. Technical report, Foundation for Intelligent Physical Agents.

Dourish, P., and Bellotti, V. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspectives (CSCW '92)*, 107–114.

Eriksson, J.; Espinoza, F.; Finne, N.; Holmgren, F.; Janson, S.; Kaltea, N.; and Olsson, O. 1998. An internet software platform based on SICStus prolog. Position Paper, Swedish Institue of Computer Science.

Finin, T.; Labrou, Y.; and Mayfield, J. 1997. *Software Agents*. MIT Press. chapter KQML as an agent communication language.

FIPA. 1997. FIPA 97 specification part 2: Agent communication language. Technical report, FIPA - Foundation for Intelligent Physical Agents.

Frost, H. R. 1998. Java Agent Template. Online Documentation: http://cdr.stanford.edu/ABE/JavaAgent.html.

Genesereth, M. R., and Katchpel, S. P. 1994. Software agents. *CACM* 37(7):48–53, 147.

Genesereth, M. R. 1992. Knowledge interchange format version 3.0 reference manual. Technical Report Logic Group Report Logic-92-1, Stanford University.

Kuwabara, K.; Ishida, T.; and Osato, N. 1995. AgenTalk: Describing multiagent coordination protocols with inheritance. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 95)*, 460–465.

Kuwabara, K. 1995. AgenTalk: Coordination protocol description for multi-agent systems. In *Proceedings of the First International Conference on Multi- Agent Systems (ICMAS 95)*. AAAI/MIT Press.

Labrou, Y., and Finin, T. 1997. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufman.

Labrou, Y. 1996. *Semantics for an Agent Communication Language*. Ph.D. Dissertation, University of Maryland Baltimore County.

Martin, D. L.; Cheyer, A. J.; and Moran, D. B. 1998. Building distributed software systems with open agent architecture. In *Proceedings of the Third Internations Conference on Practical Applications of Intelligent Agents*.

Nodine, M. H., and Unruh, A. 1997. Facilitating open communication in agent systems: the InfoSleuth infrastructure. In Singh, M.; Rao, A.; and Woolridge, M., eds., *Proceedings of the 14th Annual Workshop on Agent Theories, Architectures and Languages (ATAL-97)*.

Parunak, H. V. D. 1996. Visualizing agent conversations: Using enhanced dooley graphs for agent design and analysis. In *Proceedings of the Second International Conference on Multi- Agent Systems (ICMAS 96)*.

Peng, Y.; Finin, T.; Labrou, Y.; Chu, B.; Long, J.; Tolone, W. J.; and Boughannam, A. 1998. A multi-agent system for enterprise integration. *International Journal of Agile Manufacturing*.

Petrie, C. 1998. JATLite. Online Documentation: http://java.stanford.edu/.

Shoham, Y. 1993. Agent–oriented programming. *Artificial Intelligence* 60:51–92.

White, J. 1995. Mobile agents. In Bradshaw, J. M., ed., *Software Agents*. MIT Press.

Winograd, T., and Flores, F. 1986. *Understanding Computers and Cognition*. Addison-Wesley.