# Agent Development with Jackal

R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff
University of Maryland Baltimore County
Baltimore, Maryland 21250
cost@acm.org, {finin, jklabrou, xluan1, ypeng, ian}@cs.umbc.edu

James Mayfield
Johns Hopkins University Applied Physics Laboratory
Laurel, Maryland 20723
james.mayfield@jhuapl.edu

Akram Boughannam
IBM Corporation
Boca Raton, Florida 33431
akram@us.ibm.com

**Abstract**

Jackal is a Java-based tool for communicating with the KQML agent communication language. Some features that 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 enterprise-wide integration of planning and execution for manufacturing.

## 1 Introduction

Jackal is a Java package that allows applications written in Java to communicate via the KQML [1] agent communication language. It is designed to be used as an add-on component, rather than a framework or shell. This facilitates its integration into existing systems. Jackal has been developed as part of an effort to apply agent technology to problems of manufacturing integration. The Consortium for Intelligent Integrated Manufacturing Planning-Execution (CIIMPLEX) was formed in 1995 with the primary goal of developing technologies for intelligent enterprise-wide integration of planning and execution for manufacturing [3]. CIIMPLEX has adopted as one of its key technologies the approach of intelligent software agents, and has experimented with multi-agent systems (MAS) for various difficult tasks involved in enterprise integration.

## 2 Jackal and Agent Development

Agents that will interact with one another require some method of communication in order to coordinate their activities and distribute and collect information. To this end, a number of agent communication languages, and various software tools for them, have been developed. Jackal is a tool for the use of KQML by agents written in the Java programming language. Because it is a stand-alone component, rather than an extendible shell, it can easily be integrated into existing systems. Jackal provides more than just the ability to send and receive messages, however. Its design is based in large part on 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 in Jackal that 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 that 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 that gives Jackal its real value; the next section will discuss the rational behind the conversation-based approach.

## 3 Conversation-Based Protocols

A notion gaining in popularity is 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. A conversation lends context to the sending and receipt of messages, facilitating meaningful interpretation. The adoption of conversation-based communication carries with it numerous advantages to the developer. For one thing, there is a better fit with intuitive models of how agents will interact than is found in message-based communication, and a closer match to the way that network research approaches protocols. Also, conversation structure can be separated from the actions to be taken by an agent engaged in the conversation, allowing the same conversation structure to be used by more than one agent, and in more than one context.

To date, little work has been devoted to the problem of conversation specification and implementation for mediated architectures. Strides must be taken to develop standards and methods for conversation specification, sharing, manipulation and reuse.

## 4 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

To appear in *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*, May 1999, Seattle, Washington.

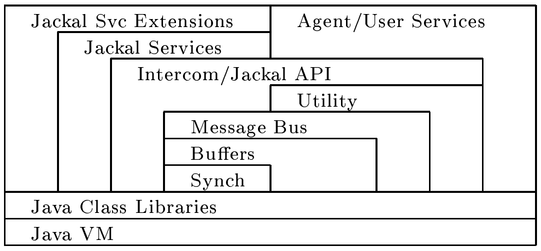

Figure 1: Jackal Architecture

most details of the implementation. Typical usage is as an accessory to an individual agent, although shared use by a collection of agents is not precluded.

## 4.1 Architecture

Figure 4 depicts Jackal's layered architecture. Its native execution environment is standard Java. Enhanced synchronization primitives and buffers tie together Jackal's very loosely coupled components. Consisting principally of the conversation interpreters and a message redistribution system, the *Message Bus* is the common path for all message traffic. This Bus, wrapped with some additional utilities by the Jackal API, is referred to as the Jackal *Core*. Both Jackal and agent services interact with the Core and each other through the API, provided by the *Intercom* class, an entity which plays a supervisory role to the rest of Jackal's components. The Jackal *Package* as it is typically distributed consists of the Core and a set of standard services.

### 4.1.1 Message Bus

Based largely on the work of Labrou and Finin [2] regarding a semantics for KQML, we have created protocols which describe the correct interactions for various performatives and subsequent messages. Currently active conversation are managed in protocol interpreters by individual threads. Messages are associated with current (logical) threads and assigned to new or ongoing conversations as appropriate. This allows for easy context management, while providing constraints on language use and a framework for low-level conversation management. Declarative conversation specifications are downloaded as needed at runtime from an online repository. In conjunction with an ontology of well-known actions, these conversations can be made to implement a wide range of agent behaviors. The conversation management component offers a number of significant benefits to the agent:

- Running conversations in individual threads provides maximum concurrency.
- Conversations, in conjunction with the Distributor, route messages automatically to the threads that need them.
- Each conversation maintains a local store, or 'context', which can be accessed by the agent.
- Since conversations are declaratively specified, they can be loaded on demand. Our current agents download only the conversations they will need.
- The conversation mechanisms and the specification are independent of the content or message language used.
- Actions can be associated with conversation structures, enhancing their utility.

The Distributor is a Linda-like 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, which persist for a given number of matches. Requesters are returned message queues, and receive all return traffic through these queues.

### 4.1.2 Services

A service here refers to either components of the controlling agent, or a subthread of Jackal itself. Two services packaged with Jackal are the Message Transport Service (MTS) and the Naming and Addressing Service (NAS).

The MTS is responsible for conveying messages into and out of a Jackal instance. The MTS runs a Transport Module for each protocol it uses; users can create and add additional modules. A Transport Module is responsible for receiving messages at some known address, and transmitting messages out via a given protocol. Utilizing an intelligent address cache, the MTS formulates a delivery plan for each outgoing message, and pursues it concurrently with other executing transmissions.

The NAS provides a powerful address resolution service, as an implementation of KNS. Jackal supports KNS transparently through an intelligent address cache. Standard services exist to allow any agent to register with any other agent, facilitating the formation of relationships or teams. Agents can hold multiple identities, and choose which to use in different situations. Protocols implemented by the NAS allow agents to easily discover other agents, regardless of the their current location or chosen identity.

## 5 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 MASs. More information is available from http://jackal.cs.umbc.edu/Jackal.

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

## References

[1] Tim Finin, Yannis Labrou, and James Mayfield. *Software Agents*, chapter KQML as an agent communication language. MIT Press, 1997.

[2] Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*. Morgan Kaufman, August 1997.

[3] Y. Peng, T. Finin, Y. Labrou, R. S. Cost, B. Chu, J. Long, W. J. Tolone, and A. Boughannam. An agent-based approach for manufacturing integration - the CIIMPLEX experience. *International Journal of Applied Artificial Intelligence*, 13(1–2):39–64, 1999.