JATLite: *A Java Agent Infrastructure with Message Routing*

HEECHEOL JEON, CHARLES PETRIE, MARK R. CUTKOSKY
Stanford Center for Design Research

JATLite is a tool for creating agent systems. It includes a message router that supports message buffering, allowing agents to fail and recover. Message buffering also supports a name-and-password mechanism that lets agents move freely between hosts.

A variety of software tools have been developed to facilitate construction of agent-based systems. JATLite is a tool that lets users quickly create new systems of *typed-message agents*,[1] which are defined in terms of agent communities that perform a distributed computation using typed messages of an agent communication language (ACL) like the Knowledge Query and Manipulation Language (KQML),[2] in which some semantics are defined prior to runtime. These typed-message agents must be able to volunteer messages that are not in direct response to a query; in fact, they are agents only to the degree that removing this initiative—by restricting communications to a client-server protocol—impairs the joint computation.

This "sliding scale" definition is distinctly operational. It does not depend on subjective measures of autonomy, sensing, or intelligence; nor does it rely on a notion of "doing something for the user," a notion that does not distinguish agents from most computer programs. The definition does assume that users can tell if their computational results are better or worse. (It also implies that not all systems built using tools like JATLite are necessarily agent-based.)

Agent tools that emphasize ACL communication, like JATLite, typically provide essential functionality such as an ACL parser, an agent identity/address directory—usually called an agent name server (ANS), and direct message exchange through TCP/IP socket transport. Agents communicate with each other by looking up an IP address on the ANS and making a connection with the desired agent. JATLite enhances this approach by making it unnecessary for each agent to track the IP address of other agents. Instead, an agent message router (AMR) ensures that a distributed computation dependent on messages is not disrupted by lost messages; the AMR allows agents, especially applets, to change IP addresses during a computation.

JATLite provides the communication functionality for agent systems with no other restrictions. It is agnostic about the agents' internal architecture and construction (for example, imperative or rule-based). It is also agnostic about whether the agents are mobile.

In this article, we describe the JATLite system, focusing on general problems of agent-based system development that it was designed to address. We have prepared a brief survey of related systems, "ACL-Based Agent Systems," which appears separately in *IEEE Internet Computing*'s companion webzine, *IC Online*, at http://computer.org/internet/<<Add permanent URL>>.

JATLITE MOTIVATIONS

One of our earliest motivations was to address the lack of "foreign agent" connection standards. Above the layer of standard Internet protocols, such as TCP/IP, the agent community has found it difficult to agree on standards. In the KQML developer community, there is even a view that this issue should not be defined at the language level, but we disagree. Standardized connect/register performatives support reliable connections between different agent systems, and low-level connections between agents allow a more task-oriented dialogue.

The newer Foundation for Intelligent Physical (FIPA) ACL[3] standards based on interoperability tests are a welcome change with which we agree. However, JATLite was designed to meet two requirements that are still lacking:

- *Reliable message delivery*. Some tools address the issues of lost messages in terms of the individual agent coordination strategy, but there has been little attempt to devise a safe, correct, and robust architecture for reliable message delivery.
- *Migrating agent communication*. Java applets were initially restricted to making TCP socket connections only with the server that spawned them. Thus, applets could not be typed-message agents, which must be able to communicate with any other agent using the standard ANS mechanism. Even the current trust mechanism for applets requires special user intervention to anticipate which applets may be trusted and what capabilities they can have. And since the point of Java applets is to build upon a ubiquitous infrastructure, we did not want to depend upon client-side installation of special mobile agent software.

Meeting these requirements does mean that some additional performatives are needed and so JATLite functionality and ACL elements remain novel even after three years.

OVERVIEW OF JATLITE

JATLite consists of Java classes and programs for creating new systems of typed-message agents that communicate over the Internet to perform a distributed computation. The agents may be either new or "wrapped" legacy software; they send and receive messages using KQML, an early ACL standard, although other languages such as FIPA's ACL can also be used.

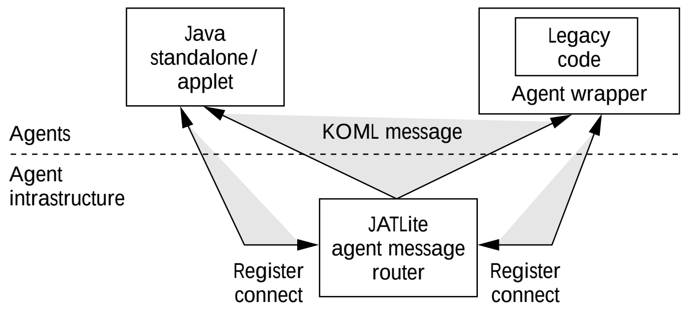


Figure 1. JATLite infrastructure.

Like earlier KQML application programming interfaces (APIs), JATLite does not endow agents with specific capabilities beyond those needed for communication and interaction. It provides front-end templates for communication with other agents; developers use the templates to add software that defines agent reactions to received messages. Figure 1 illustrates the wrapper concept in the JATLite infrastructure. JATLite templates provide all of the message communications part of both new Java agents and the wrapper for legacy code, as well as an infrastructure for managing message communications.

JATLite does add basic infrastructure functionality over and above that of earlier systems, supporting buffered-message exchanges and file transfers with other agents on the Internet (some of which may be Java applets), as well as connection, disconnection, and reconnection in the joint computation.

The Agent Message Router

JATLite's most novel service is the AMR, which allows agents to fail and recover, to migrate, and to be applets. A traditional ANS only provides the address of an agent upon request from another agent. Individual agents are responsible for keeping track of the IP addresses of all other agents with which they correspond. In addition, they must handle message failure, whether the failure is caused by a changed IP address or a simple failure of the intended recipient agent software; such handling includes recovery of messages by a restarted agent.

Message buffering. In contrast, the AMR buffers and forwards messages on a file system, like an e-mail server. Each agent makes a single socket connection to the AMR (this is the only IP address each agent knows, besides its own). The AMR then forwards the message to the recipient's correct IP address. If the recipient agent has no active socket connection, the message is delivered when it does connect again. The messages are saved on the AMR until the recipient agent sends a delete signal.

pull quote here. pull quote here. pull quote here. pull quote here. pull quote here. pull quote here.

This simple idea eliminates lost messages caused by temporary agent failure, along with the necessity for agents to track IP addresses and the restriction on applet communications. It gives an agent the flexibility to control its process and to coordinate outgoing messages. For instance, if an agent expects to take a long time to process a message, it may not want to accumulate messages in its own memory. The agent can disconnect from the AMR at will, and the AMR buffers received messages while the agent is disconnected; when the agent reconnects, the AMR sends the buffered messages to the agent. The buffered messages are deleted from the file system only on request from the recipient agent. Also, the agent can always send a message to other agents, regardless of the receiver's connectivity, just as you can send e-mail to someone without knowing if they are currently connected.

As long as an AMR is installed on the server that spawned a Java applet agent, that agent can use the AMR forwarding mechanism to exchange messages with any other registered agent. This scheme violates no browser security mechanism and requires no special setup on the user's part. The user can simply click on the URL for the agent applet, download it, connect to the AMR if the applet does not do so automatically, read the outstanding messages, and respond or send new ones as appropriate.

The AMR uses the KQML :receiver field to route the message, rather than a forwarding special performative. If the receiver is connected to the AMR and able to accept socket connections (as in the case of stand-alone applications), the AMR tries to send the message to the receiver by initiating a connection. If the receiver is intentionally disconnected or the AMR cannot initiate a connection (as is the case in an applet agent connection), the AMR buffers the messages until the receiver is connected.

E-mail could be used instead of persistent socket connections but would require each agent to have its own e-mail address. This poses a system installation issue that restricts the movement of agents from one machine to another. Applets would need access to an e-mail server on the host to which they were downloaded. Non-applet agents would need a special e-mail address to be installed on each host. Firewalls that prohibit socket connections might not allow applets and might make e-mail the only reliable transport mechanism for non-applet agents, thus restricting Internet functionality, as firewalls do.

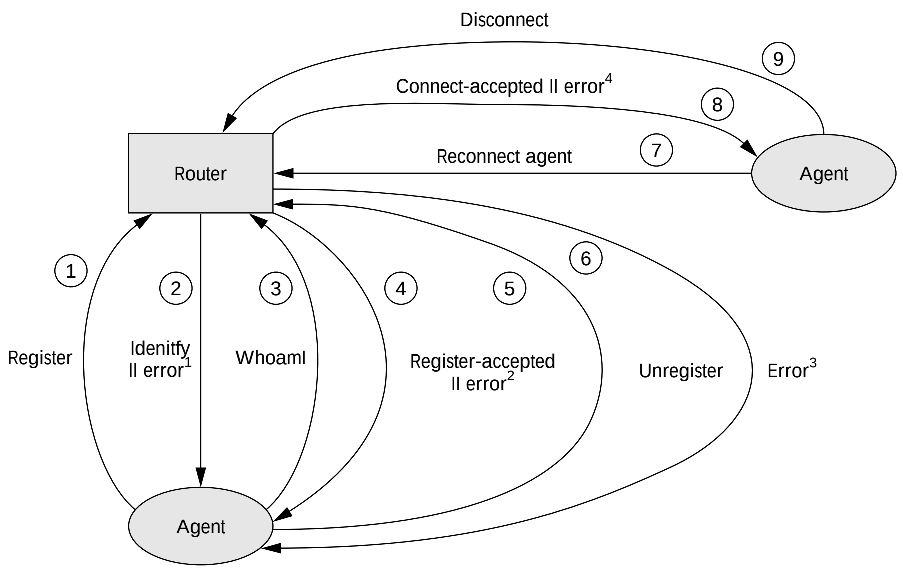
Security. With one exception, JATLite message security depends on current open standards for encryption and authentication. The one simple feature that JATLite adds is a password associated with the agent name. The AMR allows agents to change IP addresses. When an agent disconnects or times-out from the AMR, it may reconnect from a different IP address. To prevent "spoofing", the agent must prove its identity by providing a password from the original registration. This security measure is not sufficient for business applications or for more sophisticated encrypted passwords.

Joerg Schreck of the German National Research Center for Information Technology has developed Secure Socket Layer (SSL) message encryption for JATLite to produce a "mix" for KQML, for use in so-called "anonymous remailers" and "mixmaster" e-mail delivery, as described first by David L. Chaum.[4] The mix is a general tool that might be important for all applications requiring pseudonymity or anonymity; the homepage for KQMLmix is http://www.KQMLmix.net.

Both the password and the AMR functionality facilitate migrating agents. Saved messages represent part of the state of the distributed computation. This allows the state to follow a migrating agent, which might be either an applet or application software that is executed on different machines at different times. For instance, a mechanical engineer may want to run the same thermal analysis program on different computers at different times. Wrapping this software with JATLite-based code allows the program to be used with the same identity, playing a consistent role in the distributed computation, at different times on different computers.

Registration and Connection. JATLite uses nonstandard KQML performatives for registration and connection (REGISTER/UNREGISTER for registration, RECONNECT/DISCONNECT for connection, and IDENTIFY/WHOIAM for authentication). Some of these notions, in particular REGISTER, can be found in both the latest KQML and FIPA specifications. However, distinguishing between registration/connection and reconnection/disconnection depends on having the message-buffering functionality of the AMR, and so is not yet in these specifications.

Figure 2 shows the sequence of message exchanges for an agent registration. An agent that wants to register initiates a connection to the AMR and sends a REGISTER message with its name and password. If there is a registered agent with the same name, the request is denied and an error message is sent.



① "Register" protocol specifies an agent name and password.
Error^{1}: If an agent name specified in the Register protocol is already existent, "Error" will be sent.
③ "Whoiam" protocol specifies contact-information (host, port, e-mail address), message-method, KQML-extensions, and descriptions.
Error^{2}: If message-method specified in the Whoiam protocol is not supported by the router, "Error" will be sent.
⑦ "Reconnect-agent" protocol should include the agent name and its password. Contact information (host, port,e-mail address) can be redefined.
Error^{3}: If the password in "Unregister" protocol is not matched, "Error" will be sent.
Error^{4}: If the agent name in "Reconnect-agent" protocol is not registered, "Error" will be sent. If the password in "Reconnect-agent" protocol is not matched, "Error" will be sent

Figure 2. Connection protocol diagram for JATLite. The sequence shows the message exchanges for an agent registration.

Otherwise, the AMR sends an IDENTIFY message to the agent. The agent must then introduce itself using the WHOIAM performative, which includes the agent's address, description, e-mail address (optional), and so on. Upon successful registration, the AMR sends a REGISTER-ACCEPTED message back to the agent. To unregister, an agent sends the UNREGISTER performative with its password. If the password is incorrect, unregistration will be denied. Once registered, the agent can connect to the AMR by sending a RECONNECT message with its password and address. If the address is different from the previous address, the AMR replaces the address information. If a connection from the same agent is still alive, the connection will be disconnected and a new connection will be established. (Note that a loss of connection is detected only when a message needs to be sent to the connection. Once detected, the connection is closed by the AMR.) An agent can intentionally disconnect from the AMR by sending a DISCONNECT message.

FTP and SMTP support. The AMR also supports FTP and SMTP for both stand-alone and applet agents. Using built-in FTP support, an agent can save and retrieve a large amount of data respectively to and from FTP servers elsewhere in the Internet. SMTP support makes it possible for the agent to send data, including KQML messages, to any SMTP server. In both cases, for Java applets the AMR takes the role of a proxy to create a connector that links the applet and servers without caching the data to the AMR.

JATLite Architecture

JATLite exploits Java's object-oriented nature so that programmers can extend classes, building atop the basic template and specializing it for their own purposes, or omitting Java classes as desired. JATLite features modular construction consisting of increasingly specialized layers, as shown in Figure 3. Each layer can be exchanged with other tech-

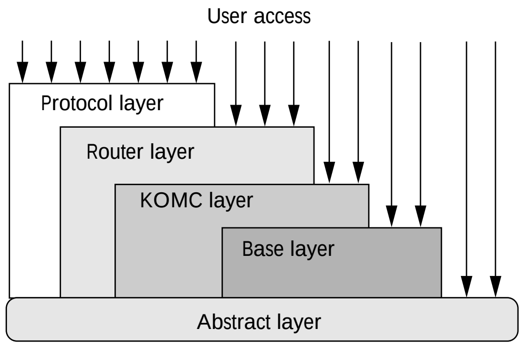


Figure 3. The JATLite architecture is a hierarchy of increasingly specialized layers. Developers can select appropriate layers to begin building their systems.

nologies without affecting the operation of the rest of the package. Developers can select the appropriate layer to start building their systems. Thus, a developer who wants to utilize TCP/IP communications but does not want to use KQML can select only the abstract and base layers as described below:

- Abstract layer—provides the collection of abstract classes necessary for JATLite implementation. Although JATLite assumes all connections to be made with TCP/IP, the abstract layer can be extended to implement different protocols such as UDP. (Note: TCP/IP socket connections are a multithreaded operation, with multiple server sockets and message receiver sockets with socket connections being persistent and having timeout provisions.)
- Base layer—provides communication based on TCP/IP and the abstract layer. There is no restriction on the message language or protocol. The base layer can be extended, for example, to allow inputs from sockets and output to files. It can also be extended to give agents multiple message ports.
- KQML layer—provides for storage and parsing of KQML messages.
- Router layer—provides name registration and message routing and queuing for agents via the AMR.
- Protocol layer—provides several open Internet protocols such as FTP and SMTP.

The reader may wonder why the KQML layer isn't on top of the router to facilitate the use of other languages. First, no good alternative ACLs existed when JATLite was originally designed. Second, all other typed-message systems used an agent-to-agent direct connection technique, so we made the router optional; this meant that KQML seemed more basic than the routing functions. Furthermore, the router depends upon ACL parsing provided by the KQML layer.

A more technical reason has been the lack of consensus on an administrative protocol. The protocol is different for KQML, FIPA, and other implementations, yet it should be in the router layer, which is the layer that makes connections and transfers messages. This makes it more difficult than it first appears to have a router layer that would interoperate among different ACLs and agent management systems.

At the AI Lab of the Swiss Federal Institute of Technology (EPFL), Monique Calisti and others building JATLite applications have developed a FIPA ACL parser and router for JATLite. The homepage for their work is at http://liawww.epfl.ch/~calisti/intagents.html.

It may also seem unintuitive for the protocol layer to be on top. But the protocol layer needs to use the router as a proxy for applet agents. FTP, SMTP, and other Internet protocols must open a socket to a host other than the host from which the applet was downloaded. But security restrictions on applets generally restrict such connections, so the router must be used as a proxy. Therefore, the protocol layer contains the agent-side code that uses router functions.

System Assumptions and Limitations

JATLite can run on any platform that supports the Java development kit JDK 1.1x from Sun Microsystems. Modifications may be needed for other Java development environments, such as the GNU free software Kaffe. Applet agents can be run using any Web browser compatible with Java 1.1.

Message delimiter. JATLite assumes TCP/IP as underlying transport but additionally uses a message delimiter, the default being the KAPI \04 character. This special escape character works only when declared to be a string. Any other ASCII character can be used within the message. For instance, a new line character that is a message delimiter for numerous Internet protocols such as FTP or SMTP can be a part of a JATLite agent message. JATLite does not support any transport mechanism that lacks a specified message delimiter, such as those mechanisms that use a message length indicator in the message header instead of a terminating character.

JATLite allows an agent to set an expected maximum message size for each connection to other agents. JATLite makes no guarantee about message order. The AMR will distribute messages in the same order it receives them. It does not inspect timing stamps to determine if one agent actually sent a message before another. There is no notion of a distributed synchronized clock.

Distributed performance. As with any Internet server, the AMR's average message processing time may restrict the speed of the distributed computation. JATLite is designed for applications in which agents typically perform significant computation between messages. If the AMR's average message processing time exceeds the average latency between agent messages, the system will bog down as messages accumulate on the AMR socket buffer. So far, our own agent systems perform more like an e-mail system, so the server is not overrun. However, it is easy to anticipate systems of many small, fast-responding agents that would overrun a server of any given speed. Our preferred method for dealing with the scaling issue is to add capabilities that would connect multiple AMRs hierarchically, much the same as Internet domains.

Since all messages are assumed to be processed through an AMR, the JATLite architecture inherits the chief disadvantage of any server application: single-point failure. The AMR is vulnerable to both file system and networking malfunctions. However, we believe that diverse, well-proven server technologies can be applied to the AMR for robust operation and recovery. JATLite makes no assumptions that would restrict such technologies from being adopted.

A related issue is control over message delivery. Whenever an agent connects (or reconnects), the AMR delivers all messages not previously deleted by that agent. We assume all agents will be able to buffer their incoming messages and decide—independently of the AMR—which to read in what order. All of our agent applets written in native JATLite code have such queues.

Compatibilities and incompatibilities. By design, JATLite is compatible with the older KAPI system. The JATLite distribution includes a KAPI release with patches to allow use with a JATLite AMR. We routinely use KAPI agents.

The newer FIPA system is more problematic for JATLite. The router layer depends upon KQML ACL constructs, and so would have to be rewritten for the FIPA ACL. It may be better to redesign JATLite to allow greater flexibility at the router layer. However, anyone can use any part of the exiting code freely, and certainly the principles of the AMR can be used in any future infrastructures.

FIPA also defines an Agent Management System. Most AMS functions are already included in the JATLite router layer. Although FIPA specifications indicate that AMS functions are performed by different agents, there seems to be no objection to a single agent performing them. Finally, current FIPA specifications simply do not address message buffering, so they do not conflict with the AMR. Future specifications may, however. A new version of the April system, called ICM,[5] is intended to be a basis for a FIPA-compliant AMS with message buffering and as support for intermittent agents, like JATLite's.

pull quote here. pull quote here. pull quote here. pull quote here. pull quote here. pull quote here.

The most serious incompatibility between FIPA and JATLite seems to be on the point of mobility. There is no direct conflict, but JATLite provides functionality requiring protocol messages not in the FIPA specification. In particular, FIPA seems to assume that if an agent changes an IP address, then it is a "mobile agent," and FIPA handles it by requiring such agents always to have a home system that can relay messages to the migrating agent. This is also the case for ICM: agent names must include a "home" host name. JATLite is agnostic on the subject of mobility: the CONNECT performative does not care whether the agent moved itself or was moved manually.

APPLICATION EXPERIENCES

The Java Agent Template[6] and JATLite grew out of experiences with several agent-based engineering systems starting with PACT,[7] an early attempt to integrate heterogeneous engineering agents using a common communication language and protocol, and continuing with NextLink[8] and ProcessLink.[9] The NextLink system was based on the Shade KAPI package and revealed several shortcomings of this approach. The current JATLite package includes a more robust version of KAPI plus patch-

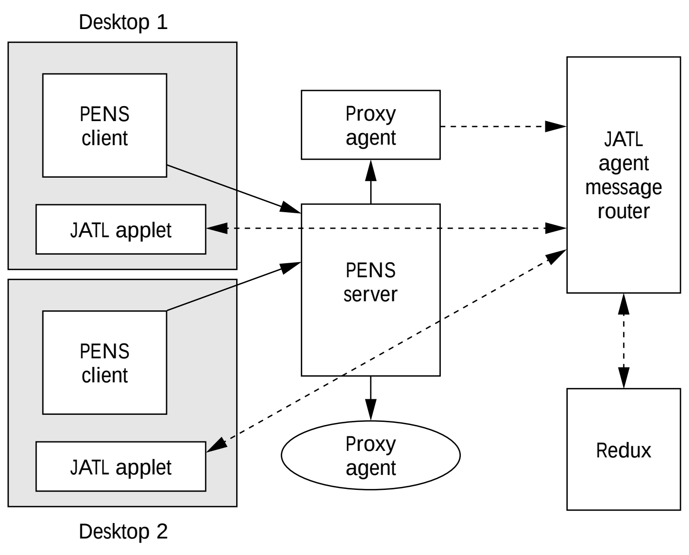


Figure 4. PENS-Redux architecture. Authors publish pages with XML-like embedded tags. The proxy agent interprets the tags as messages that should be sent to Redux as if they were composed directly by the author.

es that allow it to work with the AMR.

The package has been tested and refined over the past three years and seems to be as reliable as the platform on which it runs. In our case, the platform has been a Sun Sparc running Solaris; the only failures have occurred when the Sparc was rebooted and the AMR had to be restarted. Even though new connections and messages were blocked, no messages were lost because they were saved on the file system.

ProcessLink: An Example Application

ProcessLink is agent-based technology for computer-aided coordination of engineering teams working over the Internet. It provides automated tracking of constraints, goals, conflicts, and dependencies among team members. The team members interact through engineering tools, such as CAD programs, that have been wrapped with JATLite to send and receive messages automatically. JATLite applets that support project planning are also available.[12] All of these user-driven agents interact with a special coordination agent, Redux,[11] that coordinates the engineering design via a simple ubiquitous design model based on depth-first search and backtracking.

ProcessLink includes a generic applet for agent system development. It can display the design as a Redux goal-and-decision graph. The graph can be used to answer queries about the state of the design and also to change the design. Agent messages are produced either automatically or by hand (one option lets the user type agent messages character by character for debugging purposes). Users can run scripts of predefined messages for testing.

Other applets are specialized for constraint management, project management, and special design functions as desired. Any generic applet can be used as an individual agent by specifying the user ID and password. By disconnecting and reconnecting, the same applet can be used as a separate agent, with a different message and action.

The usual agents are applets, stand-alone Java agents, and engineering software converted into agents via wrapping with both KAPI and JATLite code. In addition, we have tested the AMR with an electronic engineering notebook program. We used the Personal Electronic Notebook with Sharing (PENS) HTML authoring system for design documentation,[10] and altered it so that the user can add structure to the text. The structure is created by highlighting text and clicking on icons to embed XML-like tags in the text. When the text is posted, a proxy agent parses the text and then sends out corresponding agent messages as if they had come from one of the existing registered agents, as shown in Figure 4.

Although this is a form of "spoofing," the authentication comes from the HTML authoring system and the results are very practical: users can create and send agent messages as a by-product of documenting their designs. The AMR treats these messages like any other and passes them on to Redux, which uses its coordination model to generate messages that can be sent to other agents.

Figure 5 shows the overall ProcessLink architecture with an example set of domain-specific agents using the generic agents as project management and design resources with JATLite as the agent substrate. Any of these agents can be applets and can disconnect and reconnect later from a different IP address.

ProcessLink supports joint computations that depend strongly on no loss of messages. If one agent fails to receive a message that a particular design change has occurred, it may try to make inappropriate design decisions. The Redux coordination agent will prevent inconsistencies and will inform an agent when and why a design change cannot be accommodated because of global constraints and the interactions of other designers.

The ProcessLink application domain is not a real-time one. The engineering software and people that constitute the "thinking" part of the agents typically have a reaction time of minutes to hours. The delay time for messages is less than two seconds. The only latency experienced by users occurs with Java applets themselves when the local host has too few resources to run them quickly. JATLite is appropriate for applications where the agents require significant "think time," but not for those requiring simpler, fast-acting, reflex-driven agents.

Related to throughput is the issue of number of agents. Our application environment typically includes fewer than 10 agents contributing to the engineering project. In a system with significantly more agents, where typically each agent performs relatively little processing, one central AMR would be a bottleneck. We suggest that a federation of AMRs, such as described in the FIPA specifications or in existing e-mail servers, will be sufficient, but for now we can claim only that the AMR is not a bottleneck for a relatively small group of agents with large processing times and, further, that this is characteristic of some large classes of agent systems, especially in engineering domains.

Our agent-based engineering work using JATLite has included the development of "agentified" engineering software modules, such as finite element analysis[11] and various CAD tools.[12] We have also interfaced with various ontological tools from the Stanford Knowledge Systems lab.[13] All of these agent systems were created using JAT or JATLite to add agent message-passing software to existing engineering tools or software modules created elsewhere. Some analysis of the software operation was necessary to characterize design decisions. The characterizations were then automatically parsed into the appropriate agent messages at runtime.

The strategy of placing the agent communications management in one AMR rather than distributed in more complex agents strongly facilitated the agent-based integration of these systems. The software development for each agent was not as complicated as it would have been if each agent had to be responsible for maintaining its cache of other agent addresses, for tracking the success and/or failure of messages, or for incorporating a scheme that was failure-tolerant.

We have not yet found another agent infrastructure that would provide the functionality we require for such engineering applications. Even the new FIPA specifications are lacking in this regard, and the compliant implementations so far do not include the message-buffering and connect/reconnect functions.

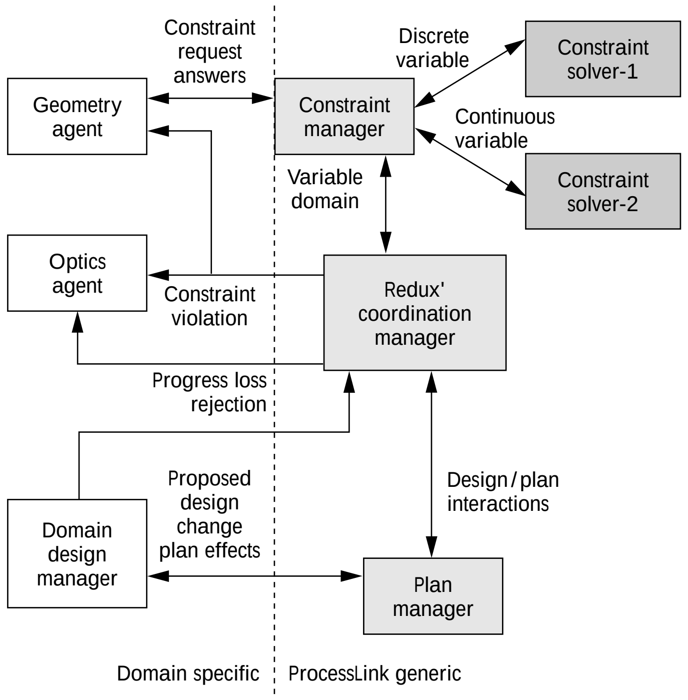


Figure 5. ProcessLink architecture for distributed engineering design. Domain-specific agents use the generic agents as management and design resources with JATLite as the agent substrate.

Debugging

Agent systems pose at least three levels of debugging:

- At the communications level: Are the agents and the AMR actually sending and receiving messages when they should?
- At the ACL level: In the case of JATLite, is each agent (including the AMR) properly parsing KQML messages?
- At the task level: Is each agent parsing each message's content properly and acting appropriately?

Each level of debugging can be done on each machine where the agent is running by watching or logging messages in response to a known good stimulus message. The AMR mechanism facilitates such debugging because it buffers the messages. If we are testing one agent, we need not repeatedly run the other agents involved in the distributed computation. Once the others have sent their messages via the AMR, the AMR will resend the messages each time the agent being tested reconnects. The same scenario can be replayed indefinitely without the other agents' participation and the corresponding possibility of slightly different messages being sent.

The AMR also simplifies debugging by centralizing communication management services. Centralized coordination obviates the need to implement a coordination scheme for each agent. Instead, we can rely on the AMR for IP address caching and message buffering. Centralization also allows debugging tools to be developed that can "single-step" message delivery, which is not possible when each agent is responsible for direct connections and message delivery to other agents.

Such centralized communications support makes agent development largely a matter of message agreement. As the people involved in the project change their agents, they post changes only to the messages to others. They do this by changing the formal description of the message on a Web page and publishing notice of the change and the URL via e-mail. Usually, such a change is preceded by a long discussion among the affected parties. The process is simplified by the restriction that agents do not send code, such as methods, to each other, but only text messages. Other agent developers change their parsers if needed.

CONCLUSION

In December 1998, we made JATLite open-source software available under the GNU general public license. Users can download the current beta version from http://java.stanford.edu/ by filling out a form giving their name, address, e-mail address, and affiliation. Over 2,000 researchers from around the world have downloaded the current version, and are using or evaluating it to create agent interfaces in a variety of fields. We encourage users to send technical questions and comments to jat-develop@cdr.stanford.edu. The archive of these messages, available at http://cdr.stanford.edu/ABE/java_agent_template/hm/, reveals a high level of experimentation. There is a JATLite user forum at jatLite-users@list.stanford.edu.

We have also developed a C++ JATLite-compatible agent template for Windows95 and Window NT platforms. The C++ versions is easier to connect to commercial software written in C and C++ than the native Java version of JATLite, but we have not released it publicly because we cannot afford to support it.

We are currently working with industrial partners on several issues related to lightweight, but robust and secure agent communication for applications in engineering and business. Three points characterize the novel aspects of the JATLite approach:

- *Performing administrative functions in the agent language—in the case of KQML, in performatives*. The KQML community has frequently argued that existing performatives, such as Ask and Tell, could push administrative functions down into the content language, but content language varies from application to application. The point of having a standard outer language like KQML is to have a single standard for standard functions. Administrative functions like registration, connection, and agent identification are important standard functions.
- Buffering messages as a viable means of robust message delivery in the face of unstable and migrating standalone and Java applet agents. Message buffering also facilitates agent construction because individual agents need not manage failed communications and the addresses of other agents. A side benefit is improved debugging.
- *Using message buffering to support a novel concept of agent identity*. Rather than identify an agent by a name and IP address, we use a name and password. Together with the time-stamping of agent names in JATLite, this method provides for unique agent naming that allows the agents to move between hosts at will.

While the JATLite primitives are only one design, these principles—administrative functions in the language, message buffering, and a password-based agent identity that allows disconnecting and reconnecting from different hosts—may be used by other designers, either directly or as criteria in developing better alternatives. ■

ACKNOWLEDGMENTS

Stanford Center for Design Research (CDR) work was funded by Navy contract SHARE N00014-92-J-1833 under the U.S. DARPA RaDEO program. Andreas Raquet wrote the current ProcessLink applets and a new KQML parser for JATLite as well as working with several other people at CDR on JATLite's development. Theresa Webster made the patches on the original NextLink system and Vikash Marda wrote the new administrative applet. A longer version of this article, including an appendix detailing the nonstandard performatives, is available at http://cdr.stanford.edu/ProcessLink/papers/jat/jat.html.

REFERENCES

1. C. Petrie, "Agent-based Engineering, the Web, and Intelligence," *IEEE Expert*, Vol. 11, No. 6, Dec. 1996, pp. 24-29; also available online at http://cdr.stanford.edu/NextLink/Expert.html.

2. Y. Labrou and T. Finin, "A Proposal for a new KQML Specification," Univ. of Maryland Computer Science and Electrical Engineering Dept., tech. report CS-97-03, Feb. 1997; also available online at http://www.cs.umbc.edu/kqml/.
3. FIPA specification, "Agent Communication Language," Draft 2-1999, Foundation for Intelligent Physical Agents, 1999; see http://www.fipa.org/.
4. D.L. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Comm. ACM*, Vol. 24, No. 2, Feb. 1981, pp. 84-88.
5. F.G. McCabe, *InterAgent Communications Reference Manual*, v0.3.3a, Publisher and city of publication?, Year?; available online at ftp://www.nar.fla.com/pub/documentation/icm/icmmanual.html.
6. H.R. Frost, and M.R. Cutkosky, "Design for Manufacturability Via Agent Interaction," Paper No. 96-DETC/DFM-1302, *Proc. 1996 ASME Computers in Engineering Conf.*, 1996, pp. 1-8; also available online at http://cdr.stanford.edu/ABE/pubs/Frost-Cutkosky.ps.
7. M. Cutkosky et al., "PACT: An Experiment in Integrating Concurrent Engineering Systems," *Computer*, special issue on Computer Support for Concurrent Engineering, Vol. X, No. Y, Jan. 1993, pp. 28-37.
8. C. Petrie, T. Webster, and M.R. Cutkosky, "Using Pareto Optimality to Coordinate Distributed Agents," *J. Artificial Intelligence for Engineering Design, Analysis and Manufacturing* (AIEDAM), Vol. 9, 1995, pp. 269-281; also available online at http://cdr.stanford.edu/ProcessLink/papers/pareto.html.
9. C. Petrie, S. Goldmann, and A. Raquet, *Agent-Based Project Management*, to be published by Springer Verlag, LNAI 1500, 1999; //Please update for 2000.// partially available online at http://cdr.stanford.edu/ProcessLink/papers/DPM/dpm.html.
10. J. Hong, G. Toye, and L. Leifer, "Using the WWW for a Team-Based Engineering Design Class," *Electronic Proc. of the Second World Wide Web Conf. 94: Mosaic and the Web*, 1994; available online at http:www.ncsa.uiuc.edu/SDG/IT94/ Proceedings/Educ/hong/hong.html.
11. P. Dabke, "Developing a Finite Element Analysis Agent," CDR tech. report, Year?; available online at ftp://cdr.stanford.edu/pub/CDR/Publications/Reports/ DevelopingFEAAgent.ps.
12. T. Mori and M.R. Cutkosky, "Agent-Based Collaborative Design of Parts in Assembly," *Proc. DETC98*, ASME, 1998; also available online at http://cdr.stanford.edu/ProcessLink/papers/toshiba/MoriASME98.pdf.
13. T. Ozawa, M.R. Cutkosky, and Y. Iwasaki, "Multidisciplinary Early Performance Evaluation via Logical Description of Mechanisms: DVD PickUp Head Example," *Proc. DETC98*, ASME, 1998; also available online at http://cdr.stanford.edu/ProcessLink/papers/toshiba/OzawaASME98.pdf.

Heecheol Jeon is a principal technologist of Macroscape Inc. His research interests include Internet-based agent infrastructure, dynamic constraint management in collaborative design, process modeling, and Internet search. Jeon has a BS in naval architecture from Seoul National University and an MS and PhD in mechanical engineering from Stanford University.

Charles Petrie is executive director of the Stanford Networking Research Center. He was previously a senior research scientist at the Stanford Center for Design Research (CDR). His research interest is distributed process coordination, with emphasis on concurrent design, planning, and scheduling. He also works on the use of XML for open interoperable workflow. Petrie has a BS in mathematics and an MS and PhD in computer science. He is the editor-in-chief emeritus of *IEEE Internet Computing*.

Mark R. Cutkosky is the Charles M. Pigott professor and associate chair for design and manufacturing in the Dept. of Mechanical Engineering at Stanford University. He is also codirector of the Alliance for Innovative Manufacturing at Stanford, an industry/academic partnership for manufacturing education and research. Cutkosky received his BS from the University of Rochester and his MS and PhD from Carnegie Mellon. He is a former NSF Presidential Young Investigator and Anderson Faculty Scholar at Stanford, and a member of ASME, IEEE, and Sigma Xi.

Readers may contact the Jeon and Cutkosky at {jhc, mrc}@cdr.stanford.edu and Petrie at petrie@stanford.edu.

5. J. McGuire et al., "SHADE: Technology for Knowledge-Based Collaborative Engineering," *J. Concurrent Engineering: Applications and Research (CERA)*, Vol. 1, No. 2, Sept. 1993; also available online at http://www-ksl.stanford.edu/knowledge-sharing/papers/shade-cera.ps.
6. D. Kuokka and L. Harada, "A Communication Infrastructure for Concurrent Engineering," *J. Artificial Intelligence for Engineering Design, Analysis and Manufacturing* (AIEDAM), Vol. X, No. Y, Month?, 1995, pp. xx-yy.