Appears in: Proceedings of the International Symposium on Agent Systems and Applications, 1999

JGram: Rapid Development of Multi-Agent Pipelines for Real-World Tasks

Rahul Sukthankar[1,2], Antoine Brusseau[1], Ray Pelletier[1], Robert Stockton[1]

[1]Just Research
4616 Henry Street
Pittsburgh, PA 15213

[2]The Robotics Institute
Carnegie Mellon Univ.
Pittsburgh, PA 15213

{rahuls,brusseau,pelletier,rgs}@justresearch.com

Abstract

Many real-world tasks can be decomposed into pipelines *of sequential operations (where subtasks may themselves be composed of one or more pipelines).* JGram *is a framework enabling rapid development of such multi-agent systems. Each agent's services are specified in the* JGram Description Language *(*JDL*), and automatically converted into Java source templates. These services may be invoked synchronously (analogous to function call) or asynchronously (analogous to message passing), in a manner that is transparent to the service's implementation. Complex tasks are created by composing several agent services into hierarchical* JGram *pipelines in which each agent may dynamically delegate its subtasks to other agents in a recursive manner, and in which errors are handled by a cross-agent, exception mechanism. Although* JGram *agents communicate using Java's Remote Method Invocation (RMI) protocol, the framework provides significant enhancements such as authentication, encrypted channels, and dynamic service specification.* JGram *has been used to develop several real-world agent systems. This paper discusses ARGUS, a visitor identification system that integrates a security camera with face detection, face recognition and user notification systems to automatically identify regular visitors arriving at the front door of our building.*

1: Introduction

Complex real-world problems can often be modeled as collections of simpler, sequential operations. For instance, the Unix environment provides a large collection of relatively simple tools (filters) that can be creatively chained together (using *pipes*) to solve non-trivial problems. This paper describes *JGram*, a generalization of the Unix pipe concept as applied to multi-agent systems, where the role of the filters is played by agent services. Aside from the immediate observation that a *JGram* pipeline is not restricted to a single machine, it differs from a standard Unix pipe in the following three respects. First, the stream oriented communication is replaced by the JGram slate, an arbitrary collection of named, typed entities. Second, agents involved in a *JGram* pipeline can dynamically alter the pipeline's structure (itinerary). Finally, unlike a Unix pipe, a *JGram* pipeline provides extensive support for error handling. Unix pipes are useful mainly because they allow rapid composition of basic functions. In the same spirit, agents in the *JGram* framework can initiate a *JGram* pipeline by specifying a sequence of recipients.

The *JGram* framework minimizes the drudgework involved in creating agent services by automatically converting high-level agent specifications (written in the *JGram Description Language*) into Java source code. These services may be invoked synchronously (analogous to function call) or asynchronously (analogous to message passing), in a manner that is transparent to the service's implementation. *JGram* is thus a Java-based agent framework specialized to enable the rapid integration of existing software into a multi-agent pipeline. This paper begins by motivating the usefulness of such a tool by briefly describing a real-world visitor identification system that was built using *JGram*. The paper then details the various aspects of the agent framework, with each feature being discussed in the context of the visitor identification scenario. An overview of relevant related work is provided, and promising directions for future research are outlined.

2: The Visitor Identification Task

Consider the following scenario. Visitors to large apartment complexes are typically screened by a security guard in the lobby before being allowed to enter. Over time, guards learn to associate frequent visitors with the tenants whom they plan to visit, and are able to immediately notify the visitor's host of the guest's arrival over the building intercom. ARGUS (named after the vigilant watchman from Greek mythology) is an automated version of such a security guard.

At a high-level, ARGUS's operation consists of the following steps, each of which is managed by one or more agents. A security camera photographs the building entrance every two seconds, and a motion detection algorithm identifies potential scenes containing visitors. Faces from these images are extracted using a neural-network-based face detector [13]. A face recognition system, ARENA [15], examines these face images and attempts to find visually similar matches in its stored database of visitors. Any user interested in receiving notification of visitors runs a user-interface agent which is automatically informed when the relevant visitors are identified. This agent also allows users to provide ARGUS with immediate corrections for identification errors. Since ARENA is capable of online learning, this feedback can be immediately incorporated into the recognition dataset. For more information on the ARGUS system, please see [16].

ARGUS is implemented as a collection of agents in a multi-agent system for several reasons. First, the components require different platforms: for instance, the camera interface is limited to Windows, while the face recognition system prefers Linux. Similarly, ARGUS users, distributed over an intranet, require notification on their individual workstations (running either Linux or Windows). *JGram* agents, which use Java RMI for communication, are well suited for this scenario. Second, the computational load imposed by some of the image processing routines is severe enough to merit splitting the task over multiple machines. Third, a multi-agent architecture offers a high degree of modularity, allowing ARGUS agents to be dynamically added to or removed from the system. For instance, interface agents can be created and killed as users arrive and leave without affecting the rest of the ARGUS system. Similarly, monitoring agents can be inserted to diagnose problems without disrupting service, and different face recognition algorithms can be seamlessly tested. Figure 1 presents an overview of the ARGUS agents and their interactions, and each agent is briefly described below.

The *delegator* agent provides several services, such as an interface to the camera, rudimentary agent name services, and manages the image processing agents which perform the bulk of ARGUS's work. The *delegator* creates a *JGram* pipeline of the form: *delegator* $\rightarrow$ *detector* $\rightarrow$ *recognizer* $\rightarrow \ldots \rightarrow$ *notifier*. The *detector* agent, a wrapper around an existing face detection system, locates human faces in its input image. If a face is not found, the detector signals this failure by throwing

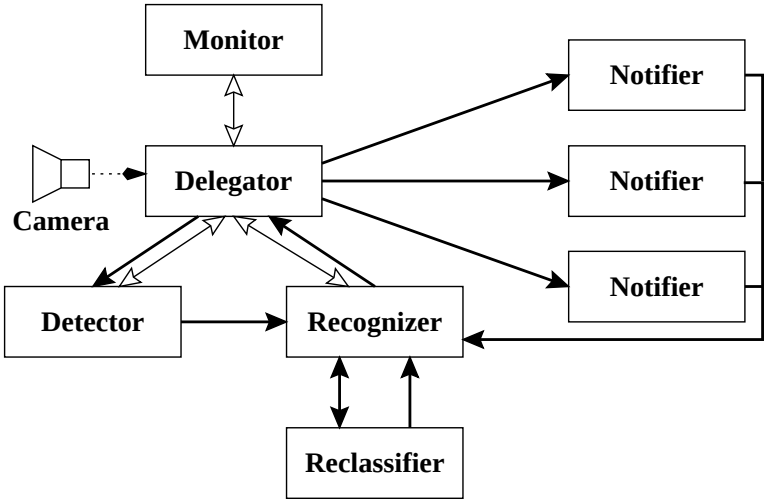

Figure 1. This diagram shows an overview of the ARGUS architecture, where each box depicts a *JGram* agent. The heavier lines show the major data pathways and the light lines show monitoring information. A line with a double arrows represents a synchronous exchange while one with a single arrow indicates asynchronous dataflow. (See the text for details.)

an exception; this exception is trapped by the *JGram* framework and the *delegator* is notified. Similarly, faces located in the image are automatically forwarded to the *recognizer* agent, a face recognition system. When a face is identified as a known visitor, all *notifier* agents watching for that visitor's arrival are alerted. The *notifier* interface agents pop up a window on each user's machine as shown in Figure 2. If the user provides feedback (such as correcting an erroneous visitor identification), the *JGram* framework automatically forwards the correction to the *recognizer*. The remaining agents supplement this main loop: the *reclassifier* is an interface agent that allows users to provide offline training signals to the face recognition system; the *monitor* agent (shown here as communicating to the *delegator*) queries agents in the ARGUS system to provide centralized status information.

As discussed above, ARGUS is constructed from several components, distributed over several machines operating on different platforms; some consisting of legacy software and others built completely in *JGram*. ARGUS has been running at Just Research since January 1999 and has processed several thousand visitor and employee arrivals (ARGUS operates 24 hours a day, 7 days a week). User interface agents were able to re-establish connections with their peers as various parts of the system were upgraded. The remainder of this paper describes the features that made the *JGram* agent framework well suited for our task.

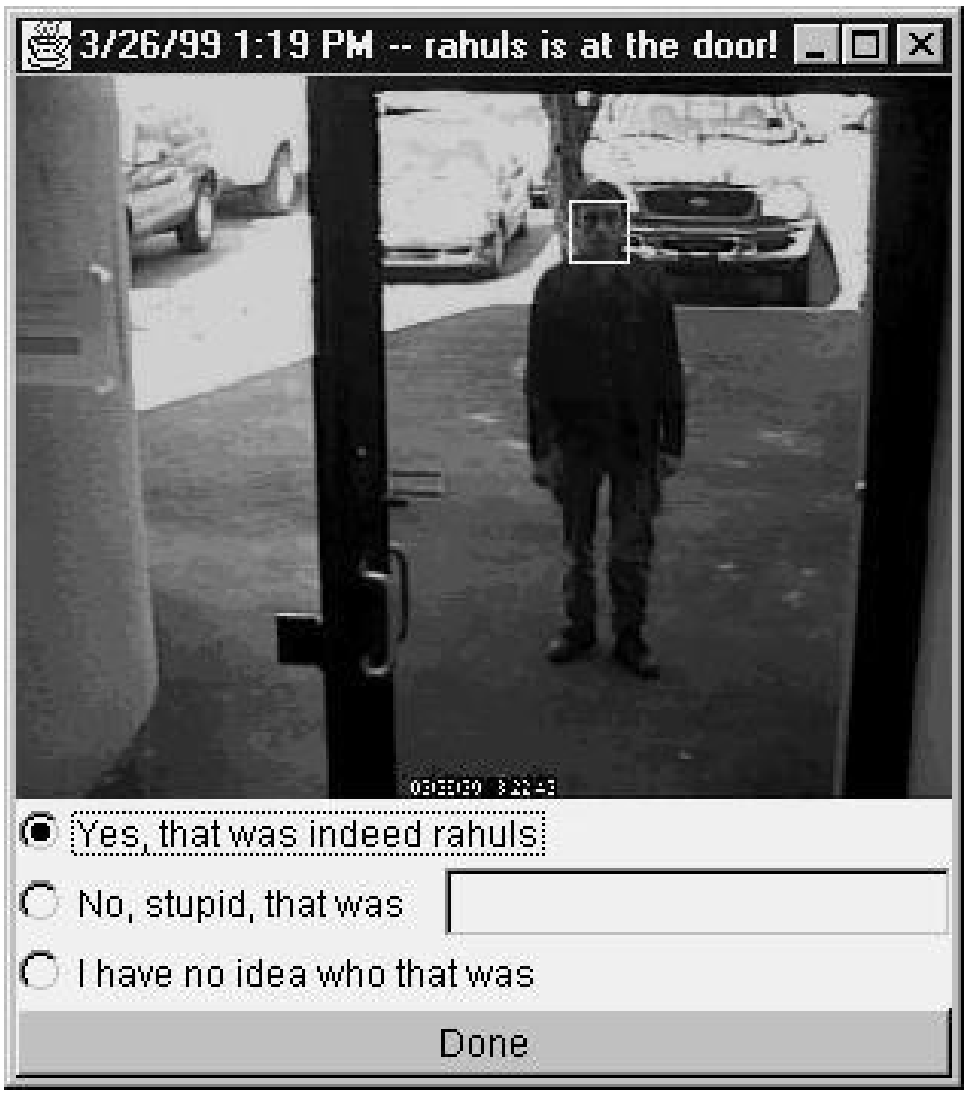

Figure 2. The user-interface agent displays an image captured by the security camera along with a box surrounding the face of the visitor and a tentative identification.

3: The *JGram* Agent Framework

What distinguishes the *JGram* agent framework from similar Java-based agent development systems? *JGram* is not designed to be a general agent framework; it is designed specifically to solve two important problems:

Rapid development and integration:

Agent developers often spend precious time reimplementing solutions to the same basic tasks: network communications, multi-threading to handle incoming requests, creation of name servers to locate external entities, or wrapping (non-agent) legacy code into the agent system. The *JGram* framework enables developers to provide concise high-level agent specifications, and source code for these tasks is automatically generated. New source code added by the developer is automatically merged with machine-generated code as agent specifications evolve over the lifetime of the project.

Rudimentary agent cooperation:

As discussed in the introduction, complicated real-world tasks can often be decomposed into hierarchies of simpler, sequential tasks. Agents in such a system attack these tasks by dynamically delegating sub-tasks to other agents and chaining these results. Ideally, the design is elegant since each agent is responsible only for a small aspect of the problem, and can remain blissfully ignorant of the true complexity of the task. Unfortunately, error handling in such a design is not necessarily simple, particularly when the agents are unable to resolve problems at the local level. The *JGram* framework enables agents to create dynamic, hierarchical "pipelines", with transparent propagation of results. The notion of exceptions is generalized so that the responsible agents are automatically notified when errors in a *JGram* pipeline arise.

These issues are discussed in greater detail in the following sections.

3.1: Communications Infrastructure

The communications infrastructure lies at the heart of any multi-agent system. In the *JGram*[1] framework, all agent interactions occur through the transmission and receipt of objects known as JGram slates. A JGram slate (see Figure 3) consists of two parts: a header specifying addressing information and delivery instructions; and a body containing a set of typed, named, complex entities (e.g., the list of JPEG images named "recent visitors", or a Date named "current time"). A JGram slate is passed from agent to agent, such that the current holder of the slate can read, modify or delete entries. Since a slate may record arbitrary objects, it serves as a rich channel for inter-agent communication. Naturally, a disorganized version of slate-passing could rapidly become unmanageable. To counteract this, the *JGram* framework makes it easy for agents to declare how their services will interact with incoming JGram slates, in a manner analogous to function declarations in programming languages. In exchange, the *JGram* framework provides benefits such as parameter checking, thread management, authentication, agent name service, and error handling. Thus, developers can focus on creating and using agent services while the *JGram* framework manages the low-level details.

JGram slates are sent from agent to agent using Java Remote Method Invocation (RMI) [17]. The *JGram* framework multiplexes all remote communications through a single remote method call, so

[1] "*JGram*", like "telegram", is intended to be a message that travels rapidly between agents.

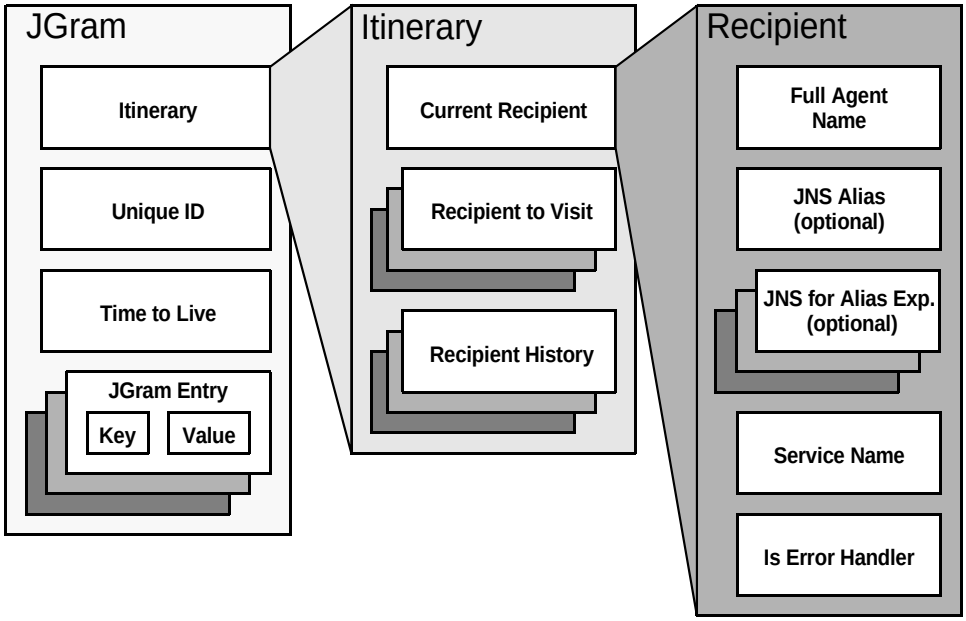

Figure 3. The internal structure of a JGram slate the basic communications object. Agent data consists of a list of typed, named objects. The *itinerary* provides the routing information, where recipients are specified either by a physical address, or by an alias (registered with a nameserver).

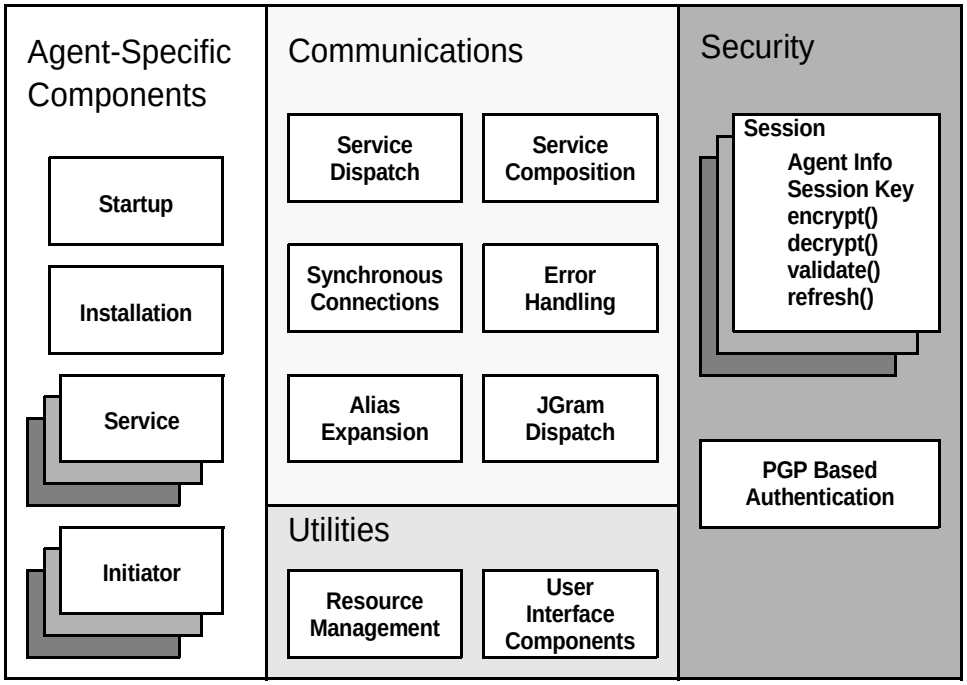

Figure 4. An overview of the internals of a *JGram* agent. All of the components in the *Communications, Security* and *Utilities* sections are provided the *JGram* framework and Java source code skeletons for the *Agent-Specific Components* are automatically generated from the high-level agent specification file.

that *JGram* agents can provide reconfigurable interfaces while all sharing the same static Java RMI stub file. The multiplexing also allows the framework to perform extra processing before and after each communication. For instance, the framework can verify that the incoming JGram slate satisfies all of a service's constraints (a generalization of type checking), and can return an error before the service is initiated. Additionally, the framework can automatically start each service request in its own thread, if the agent designer has indicated that the services are thread-safe. This enables an implementation of synchronous and asynchronous service requests in a manner that is transparent to the server agent.

Figure 4 depicts the internals of a *JGram* agent. The *JGram* framework implements the components shown in the the *Communications*, *Security* and *Utilities* sections. Skeletons for the *Agent Specific Components* are automatically generated from the *JDL* specification file, but the details must be fleshed out by the human developer. The interactions between these components is best illustrated by an example.

Consider the *Delegator*, *Detector* and *Recognizer* agents from Figure 1. Upon acquiring an image from the camera, the *Delegator* creates a JGram slate containing the image and places the *Detector* and *Recognizer* agents on the slate's pipeline itinerary. The *JGram* framework now takes over. First, the *JGram dispatch* component in the *Delegator* agent establishes a connection with the *Detector* agent's *Service Dispatch* component (*Alias Expansion* or authentication may also need to be performed first). Once the JGram slate has arrived at the *Detector* agent, the *Service Dispatch* component verifies that the JGram slate satisfies the input requirements for the selected service,

`findFace`. After these checks have been completed, the agent framework for the *Detector* agent executes the code implementing the `findFace` service (in this case, a JNI invocation). The `findFace` service sets, clears or modifies the desired entries in the JGram slate and exits. Once again, the agent framework assumes control and automatically forwards the JGram slate onto its next recipient, *Recognizer*. Any problems encountered during this pipeline that are not handled by the agent itself are automatically trapped by the *JGram* framework and processed as described in the next section.

3.2: Exception Handling for *JGram* Pipelines

The *JGram* framework provides a cross-agent exception handling capability with semantics that should seem natural to people familiar with the exception handling mechanisms in modern languages such as C++ and Java.

In the *JGram* framework, agents are responsible for handling errors that may occur in the pipelines that they initiate. These agents are known as "managers", and errors in a section of a pipeline are sent to the appropriate manager. Agents are allowed to modify the itinerary of the slate by prepending a section of pipeline, known as a "detour". Detours enable agents to delegate tasks to other agents[2]. An agent that adds a detour becomes responsible for exceptions generated during the detour (since the initiator of a detour is typically the one most likely to understand the reasons motivating the detour). Exceptions can arise in two ways: (1) unhandled Java exceptions occuring within an agent service; (2) explicit throws of an exception by an agent. By throwing an exception, an agent service can abort the execution of the pipeline. The exception is caught by the *JGram* framework, which forwards the offending JGram slate to the manager responsible for the current section of the pipeline. If this agent cannot solve the problem, the exception gets re-thrown. Thus, if an error occurs deep within a nested pipeline, it will percolate up one pipeline-level at a time, until it is either addressed successfully, or until the top-level pipeline is aborted. Manager agents may re-submit the failed subtask (since failures need not be deterministic), correct the JGram slate prior to resubmission, or abort the subtask in a controlled manner.

The semantics of *JGram* exception handling become clearer when the analogy to existing exception schemes is made explicit. The notion of flagging agents as managers is identical to entering a `try/catch` block. The forwarding of JGram slate to the closest manager corresponds to throwing an exception that is caught by the closest `catch` statement. Similarly, if a given manager ignores the error, it is automatically forwarded to the next manager in the hierarchy, exactly in the manner that uncaught exceptions are propagated to the next `try/catch` block. The manager is removed from the itinerary only after all recipients (and their detours!) have successfully completed. This is analogous to a normal exit from a `try/catch` block.

3.3: The *JGram* Description Language

The *JGram Description Language* (*JDL*) enables agent designers to specify the high-level behavior of their agents, in the form of *services* and *requests*. Consider the face detector component in ARGUS: a Windows NT implementation of the Rowley-Baluja-Kanade [13] neural-network, developed at Carnegie Mellon University. From ARGUS' perspective, the face tracker is an agent that performs a single service: given an image, it outputs the location of a human face in that image, if one exists. This is expressed in *JDL* as shown in Figure 5. The `agent` keyword is followed by the

[2]Note that, by including itself in the detour list multiple times, an agent can process intermediate results from these subtasks.

```
agent DetectFace            : "DetectFace" {
  handles findFace              : "Looks for a face in a given image." (
    byte[] image                : "GIF or JPG that might contain a face.",
    nullable Rectangle clip     : "Only search this part of the image.",
    Long timeStamp              : "A unique identification of this image",
    nullable out Rectangle face : "The location of the face, if any."
  );
}
```

Figure 5. This is the high-level agent specification file for the face detector agent from ARGUS. From this *JDL* description, *JGram* automatically generates an agent wrapper for the neural-network face tracker (a Windows NT application written in C).

agent's name, "DetectFace", a colon and a human-readable documentation string in quotes. The same syntax is used to document the other major aspects of the specification file. The documentation strings are automatically incorporated into this agent's *interface description* and peer agents may obtain this description to aid composition of agent services. The rest of the file defines *service declarations* (beginning with the `handles` keyword) and remote *service requests* (beginning with the `initiates` keyword, not applicable here). Both service declarations and service requests accept a list of entries (Java objects of arbitrary type) in a similar manner to that used in a method call. For detailed specifications of the *JGram Description Language*, the reader is directed to the *JGram* documentation.

From this description file, the *JGram* development system generates Java source code that implements communication and type-checking, and creates a skeleton for the human developer where the service can be detailed. In this example, the service is simple: it just invokes the neural-network executable using Java's Native Interface and converts the program's output into the appropriate format.

3.4: Security

As multi-agent systems become widely used in corporate applications, the need for security becomes increasingly important. The security component of the *JGram* framework enables agents to communicate over encrypted communications channels and to authenticate peer agents. When two *JGram* agents interact, they first use public-key cryptography [6] to confirm identities. Since asymmetric key protocols are slow, the agents also negotiate a shared *session key* [14], so that subsequent interactions can use a symmetric protocol. The current implementation of the *JGram* framework uses the Cryptix cryptographic libraries [2], but we are in the process of upgrading the *JGram* security components to use the cryptographic extensions provided by Java2.

4: Related Work

Multi-agent systems are reviewed in [7, 8, 18]. Most multi-agent systems still employ socket-based communications with text messages in agent languages such as FIPA [5] or KQML [4]. This allows large communities of interoperable agents to be built, but complicates the task of building systems like ARGUS, where images and complex data structures would need to be explicitly serialized by each agent. JAFMAS [1] is a Java-based framework that, like *JGram*, uses RMI protocol as a communications substrate providing object transport between agents. However, JAFMAS focuses on supporting structured "conversational models" between agents (derived from speech-act theory) and is not well suited for our applications. Unlike systems such as RETSINA [3], the *JGram*

framework provides no higher-level abstractions for agent planning or cooperation, aside from the relatively low-level pipelining and exception handling mechanisms discussed above. Stanford's JATLite [9] is an agent toolkit with goals that are seem similar to the *JGram* framework: quickly creating new software agents that communicate over the network. JATLite provides templates (Java classes) which can be used by developers to write their own agents. Unfortunately, JATLite has no notion of itineraries: messages cannot easily be sent to a sequence of recipients. Also, since JATLite parameters are simple untyped strings, there is no support for parameter checking (neither existence nor type), and agents must implement their own parameter marshalling. IBM's Aglets [10], MEITCA's Concordia [11] and Fujitsu's Kafka [12] are Java-based agent architectures primarily directed towards building mobile agents (agents that move from machine to machine during execution). While *JGram* agents may send executable code to each other for remote execution, this is not explicitly supported by the current framework. Sun's JINI [19] promises several exciting enhancements, particularly in the area of hardware interfaces; we hope to exploit some of these features in future versions of the *JGram* framework. In summary, the *JGram* framework is specialized towards the needs of particular multi-agent systems (those easily expressed using pipelines).

5: Conclusions and Future Work

The *JGram* framework enables rapid development of pipelined multi-agent systems by automatically generating Java source code from high-level agent specifications. The *JGram* infrastructure also provides:

- dynamic composition of agent services using *JGram* pipelines;
- hierarchical, cross-agent exception handling;
- automatic checking of *JGram* service requirements;
- limited security: authentication, encryption, key management;
- transparent multithreading for agent services;
- sync/async communication independent of service implementation;

JGram has been used at Just Research and Carnegie Mellon University to build several multi-agent applications, such as an intranet messaging system and the ARGUS visitor identification system described above. *JGram* may be freely obtained by members of the agent community for non-commercial, research purposes by contacting the authors.

Several extensions to the *JGram* system are planned. First, a generalized mechanism for remote event notification will be added: agents may register an interest in an event, and receive notification when the remote event occurs. For instance, some of the explicit management of remote events in the ARGUS system could be handled by the framework. Second, more support will be added for agents with intermittent network connections. A subset of the *JGram* framework is already supported on the 3Com PalmPilot PDA. Third, since the *JGram* framework does not provide any support for "traditional" agent activities, such as planning, we hope to create gateways that will enable *JGram* pipelines to interoperate (to some degree) with their more "intelligent" counterparts.

Acknowledgments

The ARENA face recognition system was developed in collaboration with Terence Sim, Shumeet Baluja and Matthew Mullin. The authors would like to thank Gita Sukthankar for valuable assistance with this paper and Michael Witbrock for proofreading drafts of an earlier version. Numerous researchers at Just Research and Carnegie Mellon have tested the *JGram* agent framework and provided useful feedback.

References

[1] D. Chauhan and A. Baker. JAFMAS: A multiagent application development system. In *Proceedings of Autonomous Agents*, 1998.

[2] Cryptix Development Team. The Cryptix encryption library, 1998. <http://www.systemics.com/-software/cryptix-java/>.

[3] K. Decker, A. Pannu, K. Sycara, and M. Williamson. Designing behaviors for information agents. In *Proceedings of Autonomous Agents*, 1997.

[4] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, 1997. <http://www.cs.umbc.edu/agents/introduction/kqmlacl.ps>.

[5] FIPA: Foundation for Intelligent Physical Agents. FIPA home page, 1999. <http://www.fipa.org/>.

[6] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly and Associates, 1995.

[7] L. Gasser. An overview of DAI. In N. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*. Kluwer Academic, 1992.

[8] M. Huhns and M. Singh. *Readings in Agents*. Morgan Kaufmann, 1997.

[9] H. Jeon. An introduction to JATLite. Technical report, CDR, Stanford University, 1998. <http://-java.stanford.edu/java_agent/html/>.

[10] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Computer and Engineering Publishing Group, 1998. <http://www.trl.ibm.co.jp/aglets/>.

[11] Mitsubishi Electric Information Technology Center America. White paper: Mobile agent computing. Technical report, MEITCA, 1998. <http://www.meitca.com/HSL/Projects/Concordia/>.

[12] T. Nishigaya. Design of multi-agent programming libraries for Java. Technical report, Fujitsu Laboratories, Ltd., 1997. <http://www.fujitsu.co.jp/hypertext/free/kafka/paper/>.

[13] H. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1), 1998.

[14] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.

[15] T. Sim, R. Sukthankar, M. Mullin, and S. Baluja. High-performance memory-based face recognition for visitor identification, 1999. Submitted for publication. An expanded version is available as Just Research TR-1999-001-1.

[16] R. Sukthankar and R. Stockton. Argus: An automated multi-agent visitor identification system. In *Proceedings of AAAI-99*, 1999.

[17] Sun Microsystems. Java Remote Method Invocation — distributed computing for Java. Technical report, Sun Microsystems, 1998. <http://java.sun.com/marketing/collateral/javarmi.html>.

[18] K. Sycara. Multiagent systems. *AAAI AI Magazine*, 19(2), 1998.

[19] Waldo98. Jini architecture overview. Technical report, Sun Microsystems, July 1998. <http://-java.sun.com/products/jini/whitepapers/architectureoverview.pdf>.