AUCS/TR9803

Reusable Components for Knowledge Base and Database Integration

A D Preece, A J Borrowman and T J Francis

Department of Computing Science
King's College, University of Aberdeen
Aberdeen, Scotland, AB24 3UE
Email: apreece@csd.abdn.ac.uk

May 1998

Abstract

Organisations increasingly need to integrate their database and knowledge-based systems into an enterprise-wide information system. This need applies to both new and legacy database and knowledge-based systems. This paper argues that modern middleware technology, notably Java and CORBA, provides an effective integration medium, particularly when combined with software agent technology. Defining the components of an enterprise information system as software agents provides a degree of uniformity which facilitates integration; Java and CORBA middleware provide a solid platform on which to implement the agent-based architecture. The paper is illustrated with an example of a medical information system prototype, featuring the integration of a number of SQL databases and a CLIPS knowledge-based system, integrated by lightweight and reusable Java/CORBA components.

1 Middleware for Information Integration

There is currently considerable interest in using middleware technology to integrate sources of data and knowledge. Some of these sources are legacy systems (pre-existing databases and expert systems), while others are custom-built services specifically designed to operate as components within distributed information architectures. Some of the common scenarios served by such architectures are:

- extracting data from databases and providing it as the input to knowledge-based systems, which in turn derive new information;
- extracting data from databases and knowledge from knowledge bases, and combining both to compose a new information source;
- extracting and transforming data and knowledge into constraint programs, the solution of which yields new information.

Some example instantiations of these scenarios in actual application domains are:

- distributed engineering design, where data on components is combined with knowledge of how designs are composed, and constraints given by the customer's requirements (for example, configuring a modular computer system);
- medical informatics systems, where patient data is fed into expert systems for therapy recommendation or critiquing (for example, recommending appropriate drugs based on the patient's needs and current drug usage);
- University admissions systems, where appropriate programmes of study can be offered, based on the student's needs and consistent with data on their academic history (for example, knowledge of prerequisites can be checked against the student's database entry from their previous institution).

The "technology push" behind these kinds of integrated application originates in projects such as the Knowledge Sharing Effort (KSE) [5] and the Stanford Mediators work [9] of the early 1990s. In particular, the KSE work has popularised the notion of viewing information sources as software agents, which interoperate using an agent communication protocol such as KQML [1]. KQML supports both the transmission of information (data and knowledge statements) and the co-ordination and control aspects of a distributed information system. The Mediators work has promoted the idea of a three-layer information systems architecture, in which "top-layer" user interface components access the "bottom-layer" information sources through a "middle-layer" of *mediator* components. These mediators are considered to "add value" to the information, for example, by "fusing", filtering, or sorting the source data and knowledge [2].

Additional impetus has been provided by the widespread adoption of internet technology, which has made the implementation of distributed information systems much more straightforward by providing a standard network platform (the TCP/IP protocol stack). More recently, higher-level programming paradigms have emerged to support the development of platform-independent, interoperable software components on TCP/IP internetworks. Termed *middleware*, technologies such as Java and CORBA offer an effective means of "glueing" heterogeneous applications together.

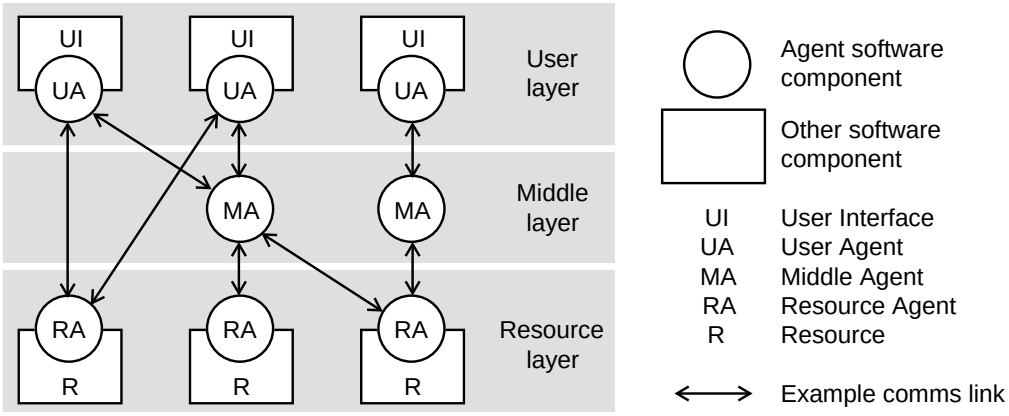

Figure 1: Three-layer agent architecture.

The above technologies are still relatively new, and there is relatively little experience in combining them to meet the requirements of enterprise information integration systems. This paper presents an experience report on efforts to develop reusable components for integrating database and knowledge-based systems, taking a KSE-inspired agent-based approach, implemented using Java and CORBA middleware. An example application is featured, which seeks to integrate a number of "legacy" SQL databases with a "legacy" rule-based expert system developed in CLIPS. Section 2 defines the logical and physical architectures employed; Section 3 describes the agent communication protocol used; Section 4 presents an example application, examining each kind of agent used, and noting the reusable components of each; Section 5 briefly summarises some support tools under development; Section 6 concludes.

2 Logical and Physical Architecture

The architecture employed in this work is a synthesis of ideas from the KSE and Mediators work: conceptually, the architecture is three-layered [9], but any agent at any level is free to communicate with any other agent at any level [1]. The logical architecture is shown in Figure 1. Note that the communication links shown in the figure are only examples of possible interactions between agents: links are formed dynamically at run-time between acquainted agents which need to exchange messages. Messages are transmitted using an agent communication protocol.

The function of each layer of agents is as follows:

User Agents These are agents that interact directly with human users; typically they offer graphical user interfaces; sometimes they are embedded in Web pages in the form of Java applets. Conceivably, however, they could offer some other kind of external interface, such as speech.

Resource Agents These are resources — databases or knowledge-based systems — which have been "agentified" by providing them with an agent front-end. They may be legacy systems, or purpose-built for integration.

Middleware Agents These are the agents that actually provide the database and knowledge base integration services; typically, each performs some useful task, on

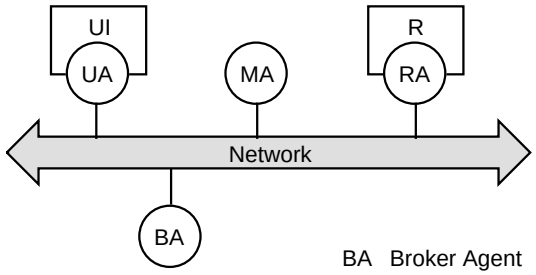

Figure 2: Abstract physical agent architecture.

request from some other agent or on its own "initiative", and calling upon the services of other agents. Some of these will be *mediators* as described in [9].

In a typical usage scenario, a user invokes a request via the user interface of a User Agent. This request is expressed in the agent communication protocol, and is relayed over the underlying network to either a Middle Agent or a Resource Agent (depending upon the type of request). A Middle Agent will typically handle a request by decomposing it into sub-requests which it will relay to other appropriate agents using the agent communication protocol. A Resource Agent will relay a received request to the resource in the local query language of the resource (for example, an SQL query to an SQL database). The Resource Agent then relays the response from the resource back to the originator of the request, again using the agent communication protocol.

Since agent communication is peer-to-peer, the logical layering vanishes in the abstract view of the implementation architecture shown in Figure 2. The three types of agent simply communicate over the underlying network. An additional feature of this architecture becomes visible here: in order for agents to direct their requests to appropriate peers, they must know what services are provided by each agent. The Broker Agent performs this function: it acts as a "yellow pages" directory of available agent services, performing a similar role to the *facilitators* of the KSE work [1] and the Broker Agent of the InfoSleuth architecture [7].

The architecture is implemented using Java and CORBA middleware as shown in Figure 3. All of the agent components share a common communication facility, implemented as a Java class, `Messenger`, which handles conversational operations in the agent communication protocol, using the Java Remote Method Invocation (RMI) mechanism to handle the fundamental messaging functions. Java was chosen because it is platform-independent, and RMI was chosen because it offers a high level of abstraction for distributed programming (the level of distributed object-oriented programming). User Agents benefit from the use of Java, in that their user interfaces may be implemented as applets, runnable from a web browser (the entire User Agent, including its `Messenger`, may reside in an applet, as long as browser security allows RMI calls from the applet to other agents on the network).

Middle Agents are also implemented in Java for maximum portability. Using Java for Resource Agents provides convenient SQL database access via Java Database Connectivity (JDBC). The problem of integrating non-Java components within this architecture is solved using CORBA: Figure 3 shows this method used to incorporate a CLIPS knowledge-based system. CLIPS is implemented in C, and was "wrapped" as a

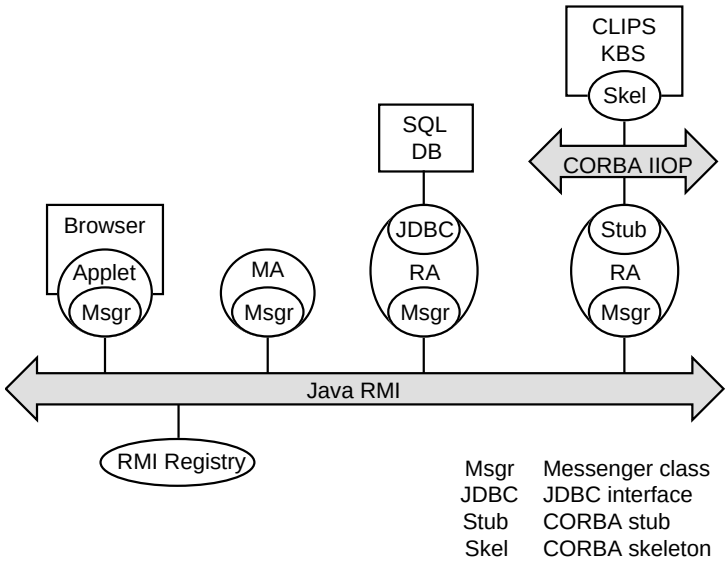

Figure 3: Concrete physical agent architecture.

CORBA object by the provision of a server-side CORBA *skeleton* object implemented in C++. A corresponding client-side CORBA *stub* object implemented in Java allows the Resource Agent to call upon the services of the CLIPS resource. The CORBA linkage between Resource Agent and back-end CLIPS resource runs over the Internet Inter-ORB Protocol (IIOP), and does not use the agent communication protocol.[1]

Brokerage is provided by the KQML Registry service, which allows a Java object to register itself along with a description of the service it provides.

3 Agent Communication Protocol

The previous section described the agent architecture, and showed how Java RMI is used to carry messages between agents. The actual messages are in the form of a subset of the KQML protocol developed by the KSE project [1]. (Part of the intention of the work was to examine the suitability of the KQML specification for this kind of task.) Each KQML message has a *performative* that defines the type of "communication action" that the message is. A KQML message consists of the *performative name* followed by an unordered list of *parameter-value* fields. An example KQML message is:[2]

```
(ask-one
    :sender       Gen-Agent
    :receiver     Gen-DB
    :reply-with   Q42
    :language     Prolog
    :ontology     Geneology
    :content      "mother('Elizabeth',X)")
```

[1] A design decision was taken to use RMI exclusively for inter-agent communications, to allow full exploitation of the close integration of RMI with Java (for example, the passing of arbitrary Java objects across the network by copy, which supports potential mobility of agents).

[2] This message is shown in the standard KQML LISP syntax, in which parameter names begin with a colon.

Agent Requirement	*KQML Performatives*
Knowledge and data Interchange	
Ask queries	`ask-if`, `ask-one`, `ask-all`
Tell data or knowledge to a peer; reply to queries from a peer	`tell`, `deny`
Advertise capabilities to a peer	`advertise`
Invoke "side effect" operations on a peer	`achieve`
Susbcribe to services of a peer	`subscribe`
Networking	
Register their existence with a peer	`register`
Locate a peer who can provide some service	`recommend-one`, `recommend-all`
Broadcast a message to all peers	`broadcast`
Forward a message to a peer	`forward`
Error-handling	
Indicate that a message is invalid	`error`
Indicate that no response can be provided	`sorry`

Table 1: KQML performatives to meet agent communication needs.

Here, `ask-one` is the performative; the communication action of this message is that the sender, `Gen-Agent`, is asking the receiver, `Gen-DB`, for a response to the query contained in the message `:content`. The `:language` field indicates that the `:content` is expressed in Prolog, and the `:ontology` field informs the receiver how to interpret the terms in the `:content` (for example, the predicate `mother`). The `:reply-with` field carries a query identifier (`Q42`) that the recipient should use when issuing a reply. A reply to this message will include the field "`:in-reply-to Q42`".

Table 1 lists the communication requirements of agents, and cross-references these with KQML peformatives that meet the needs. The performatives are as defined in the 1996 KQML specification [3].

The KQML specification defines valid *conversations* (sequences of messages). A common conversation instance involves a "customer" agent using the `recommend-one` performative to ask a Broker Agent to put the customer in touch with a "supplier" agent. The broker obliges, and then "drops out" of the transaction. The full sequence of messages in this conversation is summarised below, and illustrated in Figure 4:[3]

(i) Agent A `advertise`s to broker B that it can handle `ask-one` queries about knowledge K.

(ii) Originator O requests B to `recommend-one` agent that can handle `ask-one`s about K.

(iii) B `forward`s O the `advertise`ment that A can handle `ask-one`s about K.

(iv) O sends A an `ask-one` query Q about K.

(v) A `tell`s O some information I in response to the `ask-one` Q.

[3]Figure 4 is one valid conversational sequence composed of `advertise`, `recommend-one`, `forward`, `ask-one`, and `tell`.

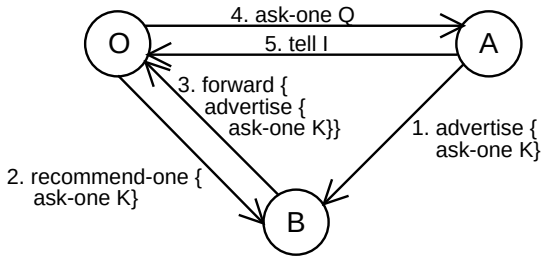

Figure 4: Example KQML conversation using `recommend-one`.

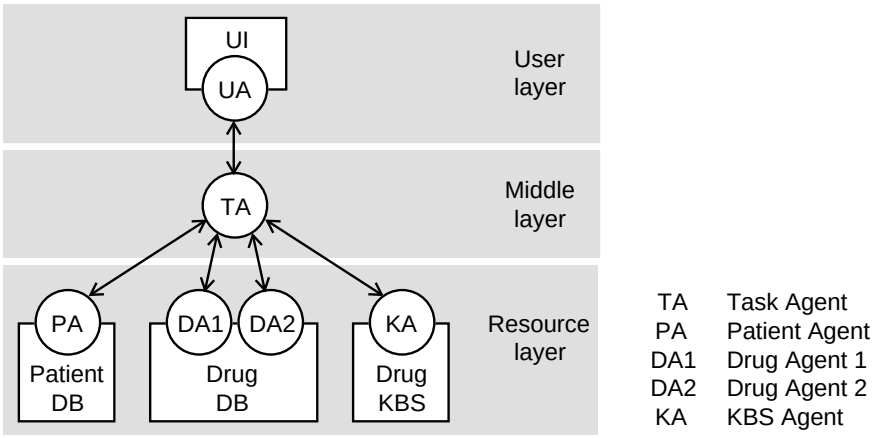

Figure 5: Example Medical Information System architecture.

4 Example Medical Information System Application

The Medical Information System shown in Figure 5 illustrates the application of the architecture described in the previous sections. This system is designed to recommend drug therapies for patients, based on:

- the patient's therapy requirements;
- the patient's existing drug regimes;
- information on available drugs;
- knowledge of undesirable drug interactions.

The application makes use of three sources of information:

Patient DB SQL database containing patient records.

Drug DB SQL database containing information on drugs and their uses.

Drug KBS CLIPS knowledge-based system providing advice on undesirable drug interactions.

Each of these sources is "agentified" with appropriate Resource Agents, described below.

Patient Agent

The Patient Agent provides access to the patient database. Upon start-up, it sends an `advertise` message to the Broker Agent, indicating that it can supply instances in the form of the following relation:

```
Patient(Name: PatientName, CurrentDrugs: List of Drug)
```

This relation is carried in the content of the advertisement message, and stored as a string in the RMI Registry, serving as the Broker Agent. It is worth noting that the terms used in the relation are defined in an ontology for the application domain. In the current architecture, this ontology is not stored within the system, and cannot be queried or manipulated. Other related work is investigating the provision of on-line ontology information [2].

It should also be noted that the patient database schema was developed independently of its use in the integrated application shown in Figure 5; the Patient Agent can be considered to be a *wrapper* for the SQL database. In this case, the wrapper essentially serves up a view on the database. In more complex cases, the wrapper will provide more than just a view: it can perform some pre-processing on the data, to transform it in some non-trivial way. The internal architecture of the Patient Agent is as shown for the database-accessing Resource Agent in Figure 3: a JDBC client component is used to access the SQL Patient Database.

Drug Agents 1 and 2

The Drug Database is provided with two independent wrappers, each of which serve up database information in different ways. The relation advertised by Drug Agent 1 is of the following form:

```
TherapyDrug(Therapy: TherapyName, Drugs: List of Drug)
```

This agent provides information on what drugs are suitable for what kinds of therapy. The relation advertised by Drug Agent 2 is:

```
Drug(Name: DrugName, Group: DrugGroup, SubGroup: DrugSubGroup)
```

This agent provides information on individual drugs: drugs are organised into various groupings.

Wrapping the Drug Database in different ways for different uses supports reuse of the information contained therein. Representation of the underlying data may change (for example, in time, the single Drug Database may be replaced with more than one database, or some combination of database and knowledge-base) without affecting the agent-level services.

The internal structure of the two Drug Agents, and the Patient Agent, is relatively simple and highly-reusable. They can be considered "lightweight wrappers", readily deployed to provide new services in time. As for the Patient Agent, the Drug Agents employ JDBC client components to access the SQL Drug Database.

KBS Agent

In contrast to the other three Resource Agents, the KBS Agent provides the services of a CLIPS knowledge-based system rather than SQL databases. The knowledge-based system is a standalone rules-based expert system that provides advice on acceptable and undesirable interactions of drug therapies. The relation `advertised` by the KBS Agent is:

```
SafeDrugs(CurrentDrugs: List of Drug,
          ProposedDrugs: List of Drug,
          SafeDrugs: List of SafeDrug)
```

`SafeDrugs` is the subset of `ProposedDrugs` that won't interact undesirably with the drugs listed in `CurrentDrugs`. `SafeDrug` is the following relation, between a drug, a drug with which that drug interacts, and the type of interaction:

```
SafeDrug(Name: DrugName,
         InteractsWith: DrugName,
         Interaction: InteractionName)
```

The internal architecture of the KBS Agent is as shown for the KBS-accessing Resource Agent in Figure 3: a CORBA stub object is used to access the CORBA skeleton object which "wraps" the CLIPS runtime system (into which is loaded the drug interactions rules base). It is perhaps worth noting that the CORBA stub and skeleton are highly reusable, and in fact *were* reused from an earlier project which provided network access to a CLIPS rules-based system.

Task Agent

The sole Middle Agent in the example application is the Task Agent responsible for running the distributed information integration task.[4] Like the Resource Agents, the Task Agent `advertises` its service in the form of a relation to the Broker Agent (RMI Registry):

```
ProposeDrugs(Name: PatientName,
             Therapy: TherapyName,
             ProposedDrugs: List of SafeDrug)
```

Given a named patient and desired therapy, the Task Agent finds a list of suitable drugs for that theraphy, given the patient's drug regime. Upon receipt of a request matching this relation (presumably from the User Agent), the Task Agent performs the following sequence of sub-tasks:

(i) issue a request to Drug Agent 1 to find a list of suitable drugs for the therapy identified in the incoming request;

(ii) issue a request to the Patient Agent to get the current drug regimes for the patient identified in the incoming request;

(iii) for each drug in the reply from the Patient Agent, request from Drug Agent 2 its grouping information (needed by the KBS Agent);

[4] The term "task agent" in this context was adopted from the InfoSleuth project [7].

(iv) send the combination of proposed suitable drugs and current drug regimes to the KBS Agent, requesting the list of those suitable drugs that will not interact undesirably with the current drugs;

(v) respond to the sender of the original request (typically the User Agent), passing back the list of acceptable drugs.

Note that the Task Agent can perform sub-tasks 1 and 2 concurrently. Note also that, before contacting each new agent, the Task Agent will issue a `recommend-one` request to the Broker Agent to find an agent that can provide the service it needs at each step. For example, when it needs patient data in step 2, it will issue a `recommend-one` request to the Broker Agent, specifying the information relation it needs, and receiving in response a message identifying the Patient Agent (as in the generic conversion of Figure 4). Once it has identified a service-providing agent for each new information relation, it can cache that information for future use.

User Agent

Rather than provide a fixed interface to allow the user to invoke the service provided by the Task Agent, the User Agent instead queries the Broker Agent to determine what services are available on the network as a whole. Using the information relations thus obtained, the User Agent dynamically creates Java windowing toolkit panels to allow the user to send queries to *any* of the available agents on demand.[5] In this way, the user is not restricted to directing queries only to the Task Agent, but can obtain information directly from the Patient Agent, either Drug Agent, or the KBS Agent. This "generic" User Agent is therefore a highly reusable component.

An example panel generated from the relation obtained from the Task Agent is shown in Figure 6.

Reusable Components

To summarise the reusable components of the example application:

- an agent communication `Messenger` Java class, using a subset of KQML, running over Java RMI;
- lightweight wrappers for "agentifying" SQL databases, using JDBC;
- a CORBA object wrapper for CLIPS knowledge-based systems;
- a generic User Agent, that builds (simple-but-usable) graphical user interface displays on demand.

It is worth noting that the `Messenger` class implements the basic sequencing rules that ensure valid KQML onversations. The rules needed for the KQML subset detailed in Section 3 are:

(i) `ask-one` and `ask-all` messages must be preceded by corresponding `advertise` messages (an agent will not allow unsolicited requests);

[5] This appears similar to the way Java applets are employed to provide customised user interfaces by the InfoSleuth project [7].

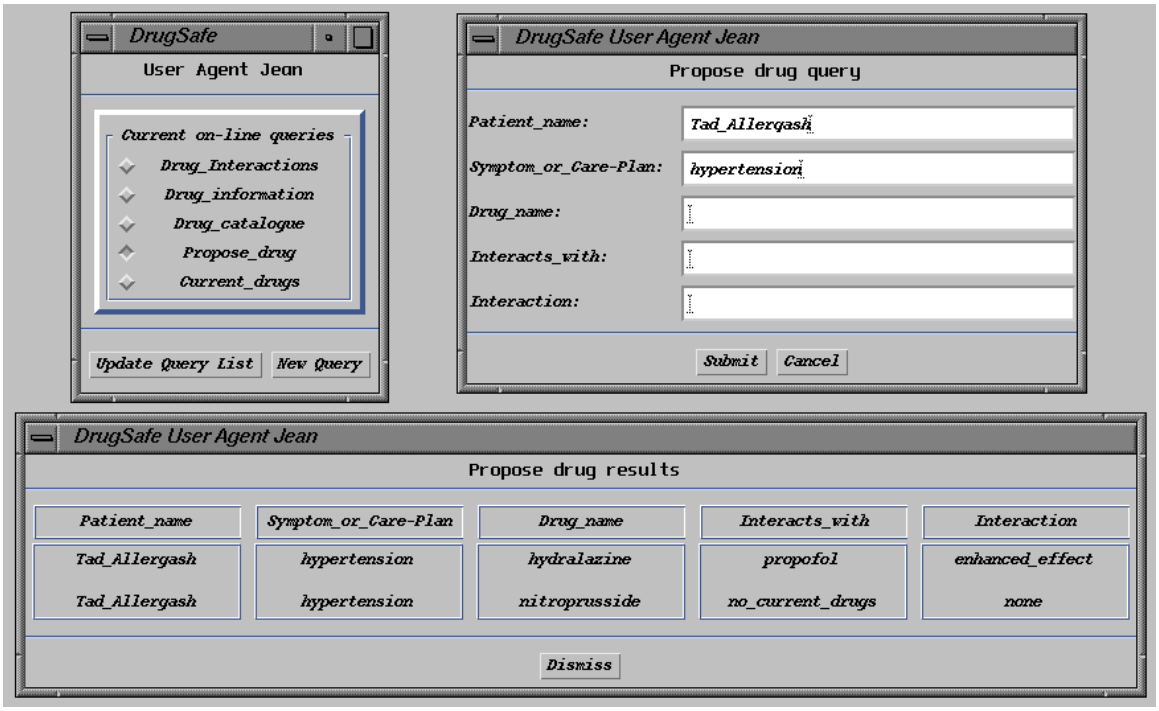

Figure 6: Example User Agent screenshot, for the Task Agent service.

(ii) `tell` messages must be preceded by corresponding `ask-one` or `ask-all` messages (an agent will not accept unsolicited information);

(iii) `advertise` and `recommend-one`/`recommend-all` messages can only be handled by Broker Agents.

The Resource Agents and Task Agents reuse common components which allow them to handle multiple requests, employing Java's multithreading capabilities. Multithreading is also employed when an agent issues multiple requests: a new thread is spawned for each new request, indexed by a hashtable using the KQML `reply-with` tag as key; this makes it easy for the agent to reconcile incoming messages with KQML `in-reply-to` fields with the original requests.

5 Support Tools

Construction of integrated knowledge-based and database systems using the resuable components described in the previous section is eased by a number of support tools.

Monitoring and Visualisation

Tools are available for monitoring and visualising the interaction of agents, for debugging and demonstration purposes. A Monitor Agent receives echoes of each message sent between agents, and displays the interactions graphically. The Monitor makes use of a generic `message` Java class, of which KQML messages are a subclass. A sample screenshot from the Monitor Agent is shown in Figure 7. Here, an agent called `shopper`

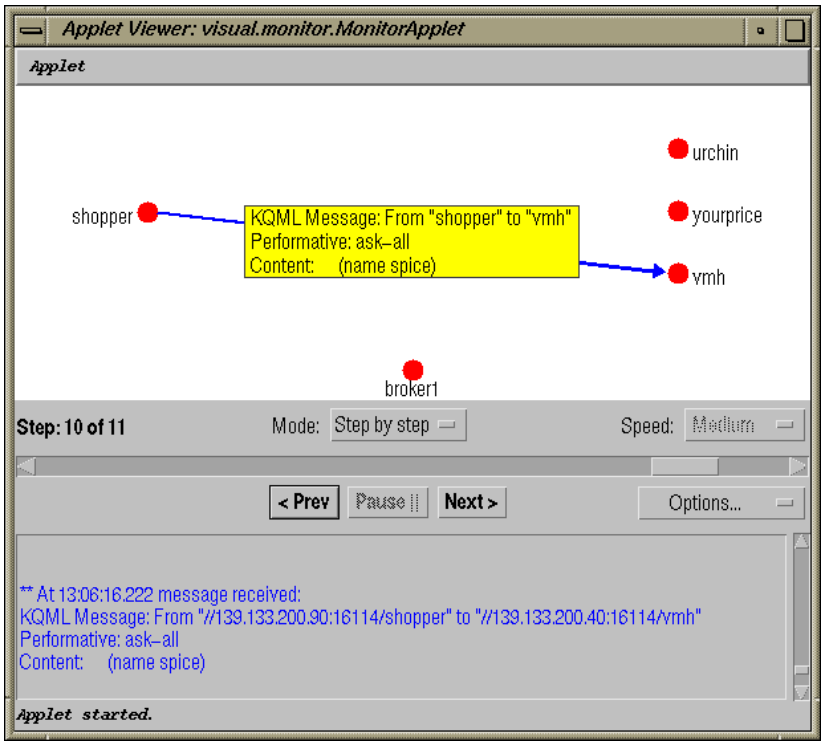

Figure 7: Example Monitor Agent screenshot.

is shown sending a KQML `ask-all` message to an agent called `vmh`. More information on the agents and the message can be obtained by clicking on their visual representations. Exchanges of messages are stored and can be replayed and stepped-through.

Verification and Validation

Following earlier work in the verification and validation of standalone knowledge-based systems [6], tools are under development to assist in the design and validation of distributed knowledge base and database systems. DISCOVER and COVERAGE are tools that support the construction of multiple-agent systems. DISCOVER verifies that "agentified" knowledge bases and databases conform to a shared ontology; this is a pre-requisite for sharing knowledge in an integrated system [8]. COVERAGE verifies the well-formedness of "teams" of agents: it establishes the closure of task interdependencies, by checking that agents are able to meet their advertised commitments [4].

6 Conclusion

This paper has presented an experience report of using a combination of agent-based and middleware technology to construct integrated database and knowledge-based systems. While these technologies promise a great deal, there is relatively little concrete experience in their use to date. The starting point for the architecture was a combination of ideas from Knowledge Sharing and Mediators work [5, 9]. The agent-based conceptual architecure was implemented on top of a layer of Java and CORBA middleware, which

was found to be an effective combination.

Looking at related work, the InfoSleuth project [7] in noteworthy in that it also makes extensive use of Java to provide its infrastructure, including the dynamic creation of customised applets for User Agent interfaces, and use of both KQML and RMI within the architecture. However, InfoSleuth does not appear to use RMI to deliver KQML messages; nor do its User Agents appear to use KQML to contact Task Agents; in effect, KQML and RMI are used as independent peer protocols for different parts of the InfoSleuth infrastructure.[6]

Experience gained from the work described in this paper shows that the combination of KQML as the message "content" protocol, with RMI as the "carrier" protocol, has proven convenient and effective. Having all agents communicate uniformly with KQML provides homogeneity at the inter-agent level, and facilitates integration. Another advantage of running KQML over RMI is that an arbitrary Java object can be carried in the `content` ("payload") field of the KQML message, supporting potential moblity of agent components.

Another finding from this work has been that the use of *lightweight* resource wrappers, each offering fine-grained information relations, is an effective and flexible way to make legacy data and knowledge available to a network. In the past, the dominant approach has been to hide each back-end data sources behind a single "coarse-grained" server with a large set of interface operations. In contrast, the approach taken here was to provide access to the back-end resource via a number of "fine-grained" Resource Agents, each offering a relatively simple interface. A good design motto seems to be: "wrappers are cheap".

Future work involves supporting the *extraction* of knowledge from legecy knowledge-bases, and enriching the kinds of knowledge integration and transformation performed by Middle Agents. Improved Broker Agent services are also planned. Much of this work will be done in the context of the KRAFT project[7], which aims to create a generic architecture for sharing knowledge in the form of *constraints*. Constraint knowledge will be extracted on demand from databases and knowledge-bases, transformed to a shared ontology, and delivered to an appropriate constraint solver. Middle Agents will locate and transform the constraints, and Broker Agents will provide rich directory and brokerage services. KRAFT is specifically intended to support distributed engineering design applications, and a prototype application is currently under construction. An overview of the project has been published in [2].

[6] InfoSleuth is also different from the work described here in that it focusses upon *database* integration, rather than *database and knowledge-based system* integration.

[7] KRAFT = "Knowledge Reuse And Fusion/Transformation"; the project is funded jointly by BT and the UK EPSRC.

References

[1] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of Third International Conference on Informatio n and Knowledge Management (CIKM'94)*. ACM Press, 1994.

[2] P.M.D. Gray, A. Preece, N.J. Fiddian, W.A. Gray, T.J.M. Bench-Capon, M.J.R. Shave, N. Azarmi, and M. Wiegand. KRAFT: Knowledge fusion from distributed databases and know ledge bases. In R.R. Wagner, editor, *Eighth International Workshop on Database and Expert System Applications (DEXA-97)*, pages 682–691. IEEE Press, 1997.

[3] Yannis Labrou. *Semantics for an Agent Communication Language*. PhD Thesis, Baltimore, Maryland, 1996.

[4] Neil Lamb and Alun Preece. Verification of multi-agent knowledge-based systems. In *ECAI'96 Workshop on Validation, Verification and Refinement of Knowledge-Based Systems*, pages 114–119, Budapest, Hungary, 1996. ECCAI/NJSZT.

[5] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senatir, and W.R. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[6] Alun D. Preece, Rajjan Shinghal, and Aïda Batarekh. Principles and practice in verifying rule-based systems. *Knowledge Engineering Review*, 7(2):115–141, 1992.

[7] R. Bayardo Jr. et al. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. URL `http://www.mcc.com/projects/infosleuth`.

[8] A. Waterson and A. Preece. Knowledge reuse and knowledge validation. In *Verification and Validation of Knowledge-Based Systems: Papers from the 1997 AAAI Workshop (Technical Report WS-97-01)*, Menlo Park, CA, 1997. AAAI Press.

[9] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.