Embedding Agents in Business Applications using Enterprise Integration Patterns

(Extended Abstract)

Stephen Cranefield
Department of Information Science
University of Otago, Dunedin, New Zealand
scranefield@infoscience.otago.ac.nz

Surangika Ranathunga
Department of Information Science
University of Otago, Dunedin, New Zealand
surangika@infoscience.otago.ac.nz

ABSTRACT

This paper addresses the integration of agents with external resources and services in enterprise computing environments. We propose an approach for interfacing agents and existing message routing and mediation engines based on the *endpoint* concept from the enterprise integration patterns of Hohpe and Woolf.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multiagent systems*

Keywords

Agent integration; Enterprise integration patterns

1. INTRODUCTION

Much of the research in multi-agent systems (MAS) is based on a conceptual model in which the only entities are agents and an abstracted external environment. This is in contrast to modern enterprise computing environments, which may comprise hundreds and possibly thousands of applications, using a variety of communication protocols and interface technologies [1].

The current solutions for integrating agents with external computing infrastructure are: (a) to access these resources and services directly from agent code (if using a conventional programming language), (b) to implement user-defined agent actions or an environment model to encapsulate these interactions, (c) to provide custom support in an agent platform for specific types of external service, or (d) to provide a generic interface for calling external resources and services, either using a platform-specific API [2] or by encapsulating them as agents [3], artifacts [4] or active components [5]. However, none of these approaches are a good solution when agents need to be integrated with a range of technologies. They either require agent developers to learn a variety of APIs, or they assume that agent platform developers or their users will provide wrapper templates for many commonly used technologies.

This paper proposes an alternative approach: the use of a direct bridge between agents and the mainstream industry technology for enterprise application integration: message routing and mediation engines, and in particular, those that support the enterprise integration patterns (EIP) of Hohpe and Woolf [1]. We have implemented this approach by developing a "component"[1] for the lightweight Apache Camel[2] enterprise integration framework, which acts as a bridge between "agent endpoints" in Camel and agents running the Jason BDI interpreter[3]. We illustrate this approach by means of a business process use case requiring the integration of Jason agents with a database management system, a mail server, a message broker and the Apache ZooKeeper coordination server.

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.

2. AN AGENT COMPONENT FOR CAMEL

Apache Camel is based on the EIP concepts of *routes* and *endpoints*. A Camel application comprises a set of *route definitions*. Each route receives messages from a *consumer endpoint*, and performs a sequence of processing steps on each message, such as filtering and transforming messages, before sending the processed messages to one or more *producer endpoints*. Camel has more than 130 *components* that implement endpoints for connecting routes to a variety of external resources, services and protocols. To enable this diversity, any data can be stored in a Camel message as the value of a named header, within its body, or as an attachment. An important feature of Camel is its built-in support for 65 "enterprise integration patterns" (EIPs) that have been identified by Hohpe and Woolf [1].

Using our agent component, Camel routes can include *agent consumer* and *agent producer* endpoints. The former allow messages and action invocations from agents to be translated into Camel messages that are processed by routes. The latter allow Camel messages to be translated into agent messages or percepts to be delivered to agents. Agent endpoint URI parameters and Camel message headers can be used to configure individual endpoints in a route. This endpoint design (detailed in the full version of this paper[4]) is not specific to Jason and Camel and can serve as a pattern for interconnecting any types of agents and message-based middleware.

Each agent and endpoint runs in a separate thread, and agents are run within an *agent container*. After receiving messages and percepts from `agent:message` and `agent:percept` producer endpoints in Camel routes, the container writes messages and percepts to concurrently accessible queues for the appropriate agents. Messages and actions initiated by agents in the container are passed to the agent consumer endpoints for processing by routes.

3. A BUSINESS PROCESS USE CASE

To illustrate our approach we consider the problem of achieving more targeted information flow within an organisation. Our solution, shown in Figure 1, introduces a "to.share" email account, monitored by agents acting on behalf of the users. Users with information

[1] http://github.com/scranefield/camel-agent
[2] http://camel.apache.org
[3] http://jason.sourceforge.net
[4] http://arxiv.org/abs/1302.1937

Listing 1: Camel route for implementing an action as a database query

```
from("agent:action?exchangePattern=InOut&actionName=get_email_accounts&resultHeaderMap=result:1")
  .setBody(constant("select email from users"))
  .to("jdbc:dataSource")
  .setHeader("result").groovy("exchange.in.body.collect{'\"'+it['email']+'\"'}")
```

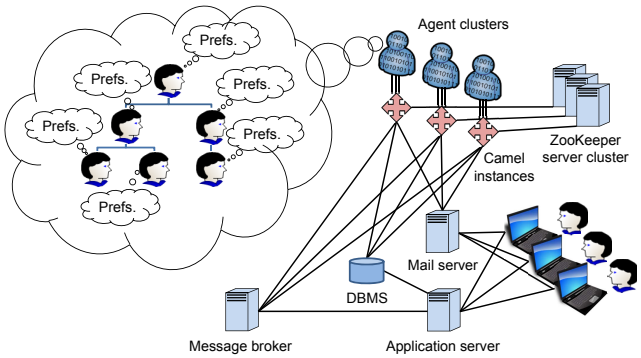

Figure 1: Architecture of our use case

to share can mail it to this account and each agent evaluates the message's relevance to its assigned users, based on knowledge of the users' roles, the organisational structure, and any plans provided by users to encode their goals for receiving information (entered via a Web-based GUI). The agents then forward the message to the relevant users' email accounts. Our design for implementing this business process involves coordinated use of Jason agents, a mail server, a database management system, a message broker and ZooKeeper, with the coordination performed by Camel routes.

One of the Camel routes needed is shown in Listing 1. This is defined using Camel's Java-based domain-specific language. The `from` method creates a consumer endpoint and the `to` method creates a producer endpoint. Endpoints are specified using uniform resource identifiers (URIs), with the first part of the URI (the scheme) identifying the type of the endpoint. The route maps the agent action `get_email_accounts(Accounts)` to an SQL query. The query result is converted to an AgentSpeak list of strings using a Groovy expression, and this is stored in the message's `result` header. The agent endpoint URI has a `resultHeaderMap` parameter specifying that the endpoint should unify the value of the `result` header with the first argument of the action literal.

Other routes, described in the full version of this paper[4], encode the agents' interactions with ZooKeeper (to maintain the list of running agents), a database management system (to retrieve user and role information), a message broker (to receive notifications of changes to user information), and an email server (to retrieve and forward new email messages sent to the "to.share" account). These routes use Camel's existing components for connecting to these types of system. They also use benefit from Camel's built-in operators for enterprise integration patterns, such as *Idempotent Receiver* (to filter out duplicate messages), *Splitter* (to split a message into separate messages), and *Aggregator* (to combine several messages into one, based on a correlation identifier), as well as its recipes for using these to implement larger patterns such as *Scatter-Gather* (sending a request and amalgamating the responses).

These routes have been tested using Jason stubs and the necessary external services, but the full Jason code for this scenario has not yet been developed. However, because the coordination logic is located in the Camel routes, the agent code required will be much simpler than would otherwise be needed without the use of our approach. Most of the agents' behaviour is to react to percepts sent from Camel by performing actions (e.g. to fetch an updated list of email accounts), and to use Jason's internal action library to update the plans used to evaluate the relevance of email messages to users. In response to the goal to evaluate a message, the agents must call the user plans, collect the users for whom these plans succeed, and send these in a message to Camel. The agents must also recompute the allocation of users to agents whenever the list of agents changes or new users are added, which they detect via 'new belief' events.

4. CONCLUSION

In this paper we have proposed a novel approach for integrating agents with external resources and services using the capabilities of existing enterprise integration technology. By using a mainstream technology we can benefit from the competitive market for robust integration tools (or the larger user base for open source software), and have access to a much larger range of pre-built components for connecting to different resource and service types.

We have described an agent component for Camel and illustrated its practical use in a hypothetical business process use case. This demonstrates the benefits of using a specialist coordination tool such as Camel for handling the coordination of distributed agents and services, and leaving the agent code to provide the required core functionality. This division of responsibilities also enables a division of implementation effort: the coordination logic can be developed by business process architects using a programming paradigm that directly supports common enterprise integration patterns, and less development time is needed from (currently scarce) agent programmers. An agent programmer using our framework does not need to learn any APIs for client libraries or protocols—the agent code can be based entirely on the traditional agent concepts of messages, actions and plans. The developer of the message-routing logic does not need to know much about agents except the basic concepts encoded in the agent endpoint design (*message*, *illocutionary force*, *action*, *percept*, etc.) and the syntax of the agent messages to be sent from and received by the routes.

5. REFERENCES

[1] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[2] T. J. Rogers, Robert Ross, and V.S. Subrahmanian. IMPACT: A system for building agent applications. *Journal of Intelligent Information Systems*, 14:95–113, 2000.

[3] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

[4] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[5] Alexander Pokahr, Lars Braubach, and Kai Jander. Unifying agent and component concepts. In *Multiagent System Technologies*, volume 6251 of *LNAI*, pages 100–112. Springer, 2010.