

Mobile Agent Interaction in Heterogeneous Environments

Peter Dömel Anselm Lingnau Oswald Drobnik
E-Mail: {doemel,lingnau,drobnik}@tm.informatik.uni-frankfurt.de

Dept. of Computer Science
J.W. Goethe—Universität Frankfurt/Main

Abstract An open problem in mobile agent research is the interaction between agents in different ‘worlds’. In this paper, we survey various approaches to agent interaction and discuss the interoperation between Telescript and `ffMAIN` agents through the use of the Hypertext Transfer Protocol (HTTP) and the shared information space provided by `ffMAIN`.

Keywords: mobile agents, interaction, HTTP, Telescript, `ffMAIN`, KQML

1 Introduction

Mobile agents—programs that can move around in a computer network on behalf of their users—are attracting more and more attention. Various research efforts take place in this area, for example TACOMA (Tromsø and Cornell, [1]), Ara (Universität Kaiserslautern, [2]), Mole (Universität Stuttgart, [3]), and the projects InAMoS and InAVAS of the DAI Laboratory/TU Berlin [4]. Several interpreted languages are suitable for the programming of mobile code. Among them Java [5], Tcl (with extensions like `incr tcl` or `AgentTcl`) [6], Python [7], and Perl [8]. Commercially available systems include Telescript [9] and CyberAgents [10]. Operating systems like Inferno [11] are also likely to support mobile agents in the future, because much of the necessary infrastructure is already provided.

With so many infrastructures available, an obvious question to ask is how one could try to make different kinds of mobile agents interoperate. Since it is unlikely that a common standard for ‘doing’ mobile agents will emerge anytime soon, it seems worthwhile to examine how different agent systems handle interaction ‘internally’, and how one might establish ‘interfaces’ that let information cross the border between different mobile agent ‘worlds’.

To begin with, a coarse distinction may be drawn between *direct* and *indirect interaction*. With direct interaction, agents can directly call code within other agents—this is usually approached from an object-oriented point of view, with agents advertising *interfaces*, i.e., sets of methods that other agents may access. For example, this style of interaction is used within Telescript. Indirect interaction means that the agent system mediates interaction between agents without letting them contact one another directly—either by having agents send *messages* to others which are forwarded by the agent infrastructure, or by interposing an *information space* where

agents can deposit items. These approaches are surveyed more closely in the first part of the paper.

A protocol which is widely used, albeit not only in the context of mobile agents, is the *Hypertext Transport Protocol* (HTTP, [12]). This is the standard protocol for information access on the World-Wide Web [13]; since many agent systems allow agents to retrieve WWW resources or even to provide such resources themselves, it turns out that the necessary mechanisms for HTTP access are already in place for many mobile agent systems. The second part of the paper examines two such systems—Telescript and the authors' own *ffMAIN* (short for *FrankFurt Mobile Agent INfrastructure*)—in more detail, and in the final part of the paper we argue that HTTP is a useful protocol for enabling information exchange and cooperation across different agent 'worlds'. This is exemplified by an experiment linking Telescript and *ffMAIN* agents in the context of an electronic marketplace.

2 Mobile Agent Interaction: An Overview

2.1 Direct Agent Interaction

In some of the systems, agents are implemented as active objects (objects which represent running threads or processes). The natural way for an agent in such an environment to interact with another agent is by simply calling that agent's methods. To be able to do this, agents need references to one another. By carefully controlling which agent obtains what references to other agents, the privacy of agents can be assured.

To give an example of a system with very tightly coupled object-oriented agent interaction, we look at the Telescript platform. However, it is imaginable that Java-based mobile agent platforms could use a similar way for agents to communicate, because the basic language concepts of Java and Telescript are very similar.

Telescript is a proprietary programming language developed at General Magic [14]. Its goal is to make it possible to program computer networks like Postscript allows to program printers. Mobile agents seem to be ideally suited to provide offline support for small portable devices like PDA's (Personal Digital Assistants), PICs (Personal Intelligent Communicators) and computerized cellular phones.

Telescript can be briefly characterized like this: There are three simple concepts to the language: agents, places and 'go'. Agents 'go' to places, where they interact with other agents to get work done on a user's behalf. Agents are, in fact, mobile programs capable of transporting themselves between places in a Telescript network. The language is implemented through a multitasking interpreter, called the *Telescript engine*.

The use of direct method calls is one of the central agent interaction mechanisms in Telescript. In order to be able to call another agent's methods an agent must first meet the other agent, which is only possible if both agents are visiting the same place. A meeting is initiated by executing the *meet* operation of this place, which returns a reference to the agent to be met. This operation takes a *petition object* as

its argument. An agent can only successfully meet another agent if it possesses a valid petition.

When an agent calls another agent's methods, it usually passes some objects as arguments and often receives one or more objects as the result of the method calls. But what happens when two agents have references to the same objects and one agent decides to go to another place (which might be located on another machine)? This conflict is resolved in Telescript by the *object ownership concept*. The default behaviour is that an object is owned by the thread under whose control it was created. This means that when an agent creates an object, it owns that object. Whenever an agent migrates itself to another place, it takes with it all the objects it owns. To do this, the engine collects all the objects owned by this agent and voids the references other agents or the place may have to these objects. In case an agent does not recognize that its partner has left and tries to use the reference an exception will be thrown.

Other ways for agents to interact asynchronously in Telescript are through thread synchronization mechanisms like events and resource-protection.

A system which extends the direct interaction paradigm is the Common Object Request Broker Architecture (CORBA, [15]). This architecture allows the remote invocation of objects in a distributed environment via an RPC mechanism. An object request broker (ORB) allows the interaction of objects written in different languages but not the mobility of code.

2.2 Indirect Interaction

The ffMAIN Infrastructure Many mobile-agent systems (e. g., Telescript) can be considered closed and proprietary: They prescribe a given language for agent development and use custom protocols for agent mobility and interaction. In contrast, the ffMAIN infrastructure [16] developed by the authors is not tied to any particular programming language and effects agent interaction and movement through the *Hypertext Transport Protocol* (HTTP), an Internet standard [12]. Agents are controlled by *agent servers* (typically one per host) which help agents communicate among themselves and with users as well as move between servers. Users can take advantage of common World-Wide Web browsers to submit agents to the system as well as to interact with the agents or retrieve results; the agent servers function as HTTP servers in order to present information to users via pages and forms using the Hypertext Markup Language (HTML) [17].

With a language-independent system like ffMAIN, a direct-interaction approach such as the one discussed in the previous section is not feasible, since differences in the languages concerned will preclude the direct invocation of one agent's methods by another. (This could be worked around to a certain extent by concepts like CORBA.) Therefore, the abstraction used in ffMAIN is similar to that put forward in LINDA [18]: Each agent server maintains a local *information space* that agents can use to publish interesting pieces of information for perusal and manipulation by other agents. Items in the information space are identified through unique keys and

endowed with a set of *access rights* detailing which agents are allowed what operations. The basic operations defined on information space items include *put* (insert an item), *get* (read an item) and *dget* ('destructive get'—read and remove an item). All these operations are performed atomically; their implementation is based on HTTP with several (small) extensions¹. The information space is also used by the agent server to publish interesting data. This lets agents find out about other agents as well as the services offered by the agent infrastructure on the host in question.—It is possible to develop more refined interaction schemes such as RPC, broadcast, blackboard systems, ... based on these basic operations (see [19]). While agents currently use custom protocols to interoperate—they can find out what other agents support by looking at the agent list published by the server—it would also be possible to implement KQML-based agent interaction (see next section) via the information space.

KQML and its Infrastructure KQML, the *Knowledge Query and Manipulation Language* [20], is a language and protocol for exchanging information. Its purpose is to facilitate the cooperation of very different intelligent agents, by providing a framework to support them in identifying, connecting with and finally exchanging information with each other. It does so by defining an extensible set of message types (*performatives* or *speech acts*) and their meanings. KQML consists of a *communication layer* (dealing with the identity and addresses of agents), a *message layer* (defining the performatives) and a *content layer* (dealing with the actual content); it does not attempt to standardize the format of the information being exchanged, nor does it specify exactly how this information is to be moved.

Figure 1 shows the conceptual architecture used in the KQML specification by the External Interfaces Group, a working group of the ARPA Knowledge Sharing Effort. This architecture serves as another example for indirect agent interaction, whereas the language KQML itself can well be used in other architectures (like `ffMAIN`) as an abstraction for describing and encapsulating agent operation invocations. In contrast to direct object-oriented agent interaction, it does not require mobility of the interacting agents, although agent mobility could improve the efficiency of the system.

Each software agent has a separate router process (K-router), which handles all KQML messages sent and received by its agent. When a K-router is unable to find the recipient for a message, it consults *facilitator agents* which provide network services like content-based routing, brokering between suppliers and consumers, recruiting suppliers to deal directly with consumers, and smart multicasting. The KRIL (KQML Router Interface Library) is an API to the external K-router process which is embedded in the agent application itself. Two common KRIL operations are `send-kqml-message` and `declare-message-handler` for submission and reception of KQML messages.

KQML represents a framework for agent communication rather than something which is directly useful. It can serve as a conceptual *middleware* between agents by

¹ Making extensions to HTTP is explicitly condoned by the protocol definition.

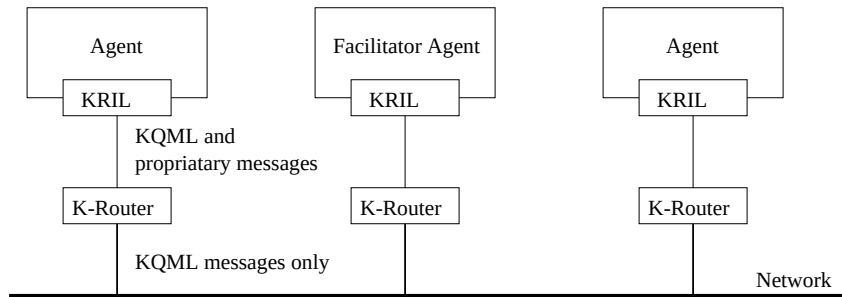


Figure1. Conceptual KQML infrastructure

introducing a common set of operations (the performatives) that various agents can support. KQML by itself cannot guarantee that the agents will actually be able to do something useful with the information they exchange.

3 Interfacing Mobile Agents and the Web

3.1 Telescript Agents and the Web

The Telescript runtime environment was originally designed for the use in big network servers, operated by service providers. A set of tools [21] allows users to create, control and interact with Telescript agents via ordinary Web browsers. Moreover, these tools enable Telescript agents to access the WWW and support them in using its services by filling out forms and parsing the resulting HTML pages.

Using these tools, Agents may be used to build value added Web services. Agents can, e.g., be used to search information in the Internet, monitor changes of Web pages or build meta services². The mobility of agents can help to do this more efficiently by avoiding network traffic whenever great amounts of data have to be processed.

In figure 2 we give a rough overview of how a Telescript engine is connected to the WWW via the *Telescript Web Tools* resp. *TabrizWare*.

Users (and administrators) can create and control their personal agents by selecting special URLs in their Web browsers. Another way to communicate is by filling out HTML forms generated by their agents. Such a URL may, e.g., look like:

`http://deneb:3000/login/telescript/CDShoppingAgent/findCD?title=Equinoxe`

The browser requests the resource bound to this URL from an ordinary Web server running on host deneb using port 3000. This server was configured such, that the login part of the URL causes it to perform HTTP-based authentication. By requiring users to log in, the system can bind identities of agents to identities of users. The URL part *telescript* refers to a CGI process, *webtots*, which takes all the

² Hide several similar services behind one common user interface.

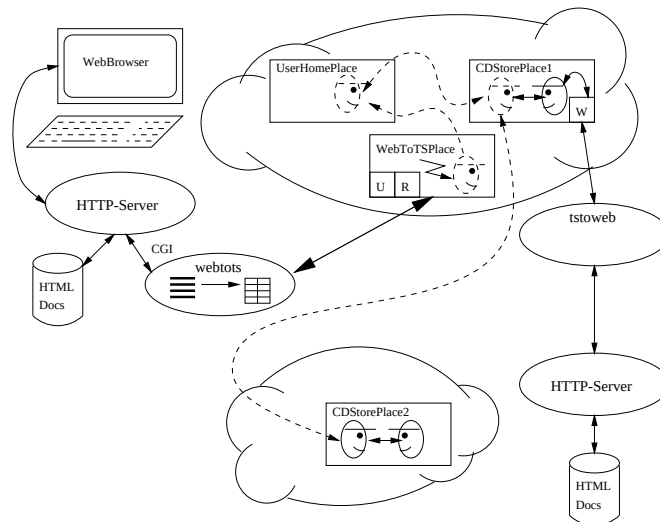


Figure2. Telescript Web Tools (Tabriz)

CGI variables provided by the Web server and the user and creates a Telescript data-structure out of it. Then `webtots` connects to the Telescript engine (represented by a cloud) where the rest of the URL is processed.

Because the user has not created a `CDShoppingAgent` earlier in our example, by default such an agent will be created by the `webtots` place, which also contains a registry of services ('R' in figure 2) and information about registered users ('U' in figure 2). From this service registry the agent gets the addresses (represented by ticket objects) of two places offering CDs as well as a ticket to the `UserHomePlace`. First the agent 'goes' to the `UserHomePlace` where it makes a note (enters its URL) in the user's personal HTML pages, so that the user can contact it, if necessary.

The part `findCD?title=Equinoxe` of the URL invokes the agent's `findCD` method. Because the agent owns the associative array (created by the `webtots` process) which contains all the necessary CGI variables, it knows that the user is interested in CDs with the title *Equinoxe*. First it visits the `CDStorePlace1` where it meets the shop's service agent and asks it about availability and price of CD's with the title *Equinoxe*. This service agent, however, is only a front-end to a normal WWW CD store service. With the help of a Web access API ('W' in figure 2) it communicates with the external process `tstoweb`, which uses the HTTP client library `libwww` [22] to access a HTML form based WWW CD store service. The service agent of the `CDStorePlace1` simply fills out the appropriate forms and parses the resulting Web pages. From this information it generates the result to the `CDShoppingAgent`'s question.

After it got its first result, CDShoppingAgent 'goes' to the CDStorePlace2 which is provided by another Telescript engine on another network server. This engine may also be connected to the Web using the Telescript Web Tools, but it may as well be a standalone Telescript engine, offering pure Telescript services. In our example the service agent of CDStorePlace2 has its own data repository inside the place and no need to access other services in order to operate the 'shop'. However, this agent could as well be an interface agent to an external database system.

This little electronic marketplace scenario covered all the important parts of the Telescript Web Tools, which consist not only of the external processes, the `webtools` place and the so-called mix-in interfaces 'R', 'U' and 'W', but also of an HTML API which on one hand allows agents to create HTML documents as a hierarchy of Telescript objects which are automatically rendered to HTML, and on the other hand is able to parse HTML code and generate the corresponding hierarchy of Telescript objects.

A library of superclasses gives agents, e.g., the ability to interactively 'talk' to users via HTML forms. These superclasses also provide all of the necessary infrastructure to hide HTTP-specific issues from agents and places.

3.2 Controlling Agents in `ffMAIN`

As explained in section 2.2, the `ffMAIN` infrastructure is completely based on HTTP. In principle, a user can create an agent simply by submitting an appropriately formatted file containing its source code and initial state through a WWW browser (or even a TELNET connection). For convenience, this is usually done via an auxiliary 'launching' program. As a matter of fact, this program can be invoked from the Web; this makes it straightforward for information providers to add mobile agents to their offerings. Another advantage of this approach is that agents can be conveniently configured through a HTML form which is constructed dynamically by the launching software from a specification embedded within the agent's source code (see fig. 3).

Once an agent has been submitted, the launching software returns to the user a URL which can be used for subsequent queries about the agent, to fetch its results or to kill or revoke the agent. This URL leads to a HTML page maintained by the agent server the agent was initially submitted to (its *home server*); the agent servers keep track of agents moving among one another in order to keep the information about an agent's whereabouts up-to-date. HTTP 'redirect' messages are used to point an inquiring user's WWW browser to the correct server. Agents, in turn, can send partial results to their home servers to be made available to users as HTML pages.

This focus on HTTP for agent mobility and control allows users to view mobile agents as a seamless extension to their usual World-Wide Web environment. They do not have to get familiar with yet another user interface in order to take advantage of mobile agents.

In `ffMAIN`, agents can also interact with other resources on the World-Wide Web. In order to effect a separation of concerns and to keep generic mobile agents operating on a basis of least-privilege, WWW access would usually be through a

```
#Desc: Find cheapest supplier of product \
# and order it
...
# Product: STR
set product %%PRODUCT%%
# Amount to order: STR
set amount %%AMOUNT%%
# Disposal: ALT [Buy outright|Just reserve]
set disposal %%DISP%%
```

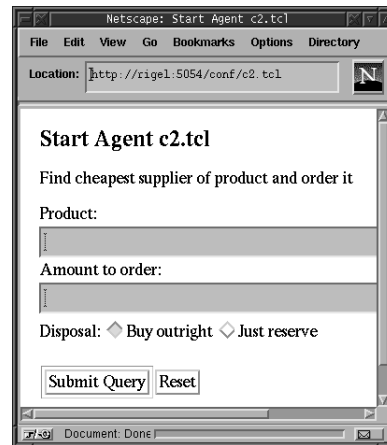


Figure3. Agent configuration in ffMAIN: Specification and presentation

trusted stationary agent accepting URL requests via the information space, possibly vetting these to avoid costly requests (e.g., non-local or international), and returning the data addressed by the URL in question to the requesting agent, again via the information space. While there is no problem in principle with allowing every mobile agent to access the WWW at will, this approach lets the maintainer of an agent server retain maximum control over the site's security and resource usage. ffMAIN does not offer in-built methods for parsing and generating HTML by agents; it is believed that this can be safely left to the relevant support available in the various agent implementation languages. Tcl and Perl, for example, contain sophisticated extensions for HTML management; it appears more worthwhile to offer safe methods of accessing these rather than re-invent the wheel.

4 Interaction Between Agent Systems Through HTTP

4.1 Direct Agent Interaction

If users can control and interact with agents through WWW browsers by using an HTTP-based remote control mechanism (as in section 3.1) and agents can act on behalf of their users by issuing HTTP requests, it follows that agents can interact with one another using HTTP. In the case of direct interaction, HTTP represents a simple RPC mechanism. Every URL corresponds to a remote method invocation; arguments can be wrapped up in the URL, and results are returned just like WWW pages. Thus, agents have the option to decide whether they want to 'go' to another agent, or whether it is cheaper or faster to remotely invoke its methods. This decision is a trade-off between the costs of agent migration (depending on its code and data size and the costs for encryption and encoding of the agent) and the amount of data to be exchanged between service and agent (plus the RPC-overhead).

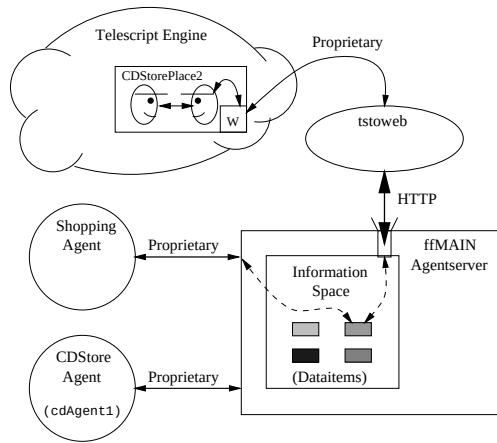


Figure4. Telescript agents access information space via HTTP

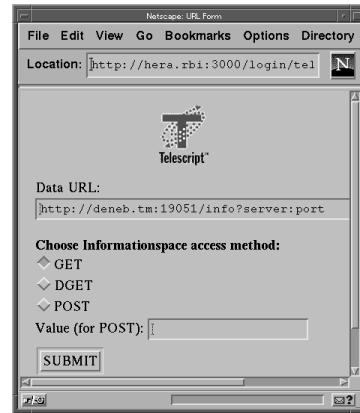


Figure5. Access form generated by a Telescript agent

As already mentioned in section 2.1, CORBA could be considered for language independent remote interaction between agents. But although CORBA allows programs written in different languages to interact, usually different platform/language-dependant APIs are used (e.g. DSOM for IBM UNIX workstations and DCOM / ActiveX for environments based on Microsoft Windows). This does not represent a problem for non-mobile distributed software, but for *mobile* agents this is more of an issue because the CORBA libraries needed for the remote method invocations may differ from server to server. It could be necessary to have the agent platform provide some additional middleware as a common agent CORBA API. This 'glue' code would likely be language-dependant as well as platform dependant. Moreover, the runtime environment needed by CORBA-based applications is relatively heavy-weight and would require service providers to buy and run CORBA support software, whereas HTTP is, by comparison, such a lightweight protocol that every agent can easily carry its own protocol implementation. Also, there is no need to define and compile CORBA IDL specifications in order to remotely invoke methods offered by other agents or services. It is sufficient to create simple URL strings containing the parameters. A more practical reason why we decided to experiment with the HTTP approach is that currently CORBA support by online service providers is not very common, but most services already are HTTP-based.

Another simple example of HTTP usage, this time for agent/user interaction, is shown in figure 5. The HTML form was generated by a Telescript agent which takes the input from the user to access the information space and reads the `prices` value stored by the `cdAgent1`.

Telescript agents may use the existing `WebAccessing` mix-in class to issue HTTP requests. This mix-in also uses a proprietary protocol to communicate with the external `tstown` process which, e.g., translates Telescript objects into C++ ob-

jects. This interaction can be considered similar to the KRIL/K-Router connection mentioned in section 2.2. In this case, the `WebAccessing` mix-in corresponds to the KRIL and the target of the HTTP requests—e. g., an `ffMAIN` information space—to the set of interconnected K-Routers.

Furthermore, as we have seen in section 3.1, it is also straightforward to construct an HTTP interface that will accept requests from an outside source and translate them into the appropriate native calls. Therefore, HTTP can play the role of a ‘common denominator’ for communication between many different platforms.

4.2 Indirect Agent Interaction

The *agent servers* in the `ffMAIN` infrastructure communicate via HTTP. Their HTTP interface also allows authenticated access to their information spaces from outside. Not only does this enable management tools to connect to agent servers, e.g., to display and edit the content of the information space, but it also provides a simple way for other agent systems to read, write and remove data items with ease. Figure 4 shows how, e.g., Telescript agents may interact with `ffMAIN` agents by exchanging data via the information space. Other systems could be connected in the same fashion. (Actually, native `ffMAIN` agents use a compressed protocol based on HTTP to interact with the `ffMAIN` agent servers and to access the information space. This saves overhead and allows for more advanced features – native agents can, e.g., request to be asynchronously notified by the information space, whenever a certain data item has changed. This does not detract from the general accessibility of the information space through the usual HTTP operations.)

Table 1 shows some examples for the protocol data exchanged with an agent server to communicate to the information space. The `GET` example reads the server’s port number (made available by the server itself in `server:port`). The `POST` example creates a new data item `prices` which is owned by the agent `cdAgent1`. The `DGET` example reads and removes the entry `cdAgent1:prices`.

Figure 6 gives a few examples of how an `ffMAIN` agent would interact remotely with a Telescript agent using the information space. Of course, two Telescript agents could also interact this way, which might be useful to save the overhead of two `go` and one `meet` operation when only a few bytes of information have to be exchanged.

In the ‘electronic marketplace’ example, `ffMAIN` and Telescript agents can coexist and work together through the information space. ‘Shops’ based on both `ffMAIN` and Telescript agents can publish catalogues and prices in the information space or accept requests for further information through an HTTP-based remote call mechanism. This means that `ffMAIN` users can take advantage of information available in the Telescript ‘world’ and vice-versa.

4.3 High-Level Interaction

Enabling different agent infrastructures to interact through a mechanism like the `ffMAIN` information space does not solve the problem that agent developers still have to know a lot about the other agents with whom ‘their’ agents are going to

GET /info?server:port HTTP/1.0 Authorization: Infospace-1 wizard xyzzz HTTP/1.0 200 Request fulfilled, document follows Server: ffMAIN/1.0 Perl/5.003 Date: Thu, 20 Nov 1996 14:11:48 GMT Content-Type: application/octet-stream Content-Length: 5 Content-Transfer-Encoding: binary 19051
POST /info?cdAgent1:prices HTTP/1.0 Authorization: Infospace-1 wizard xyzzz Content-Type: application/octet-stream Content-Transfer-Encoding: 8bit Content-Length: 4 7.95
DGET /info?cdAgent1:prices HTTP/1.0 Authorization: Infospace-1 wizard xyzzz

Table1. HTTP protocol data exchanged in information space access

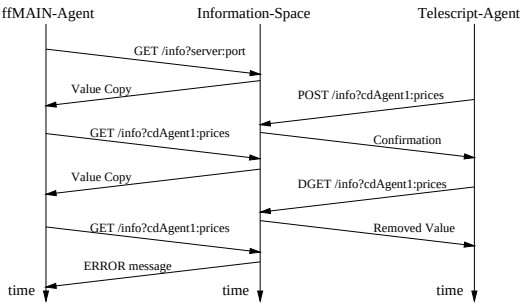


Figure6. Accessing the Information Space.

interact. They need to know the other agent’s interfaces (operation signatures, protocols and data formats) in order to communicate effectively. Moreover, the agents will need support to find other agents which provide a certain service or possess certain information.

By encapsulating the data written into the information space inside KQML performatives, all the higher-level agent support mechanisms offered by KQML—like facilitator agents and other mediators—come within the compass of mobile agents. HTTP to access to the information space plus the information space itself would provide a possible implementation of the communications infrastructure consisting of KRILs and K-Routers shown in section 2.2.

Another advantage of KQML is that it facilitates the integration of mobile agents with stationary agents and other knowledge-based systems. Using KQML, however, requires a translation between KQML performatives and an agent’s internal programming-language-dependant mechanisms (usually methods or functions) which implement the actual service. Mobile agents could contain an KQML parser and operation dispatcher. However, it would probably be sufficient to have one dedicated agent per host and agent platform to provide a KQML translation service. This agent would interact with other agents in its environment by using the interaction mechanism native to this platform (in Telescript, e.g., it would meet other agents and invoke their methods).

5 Conclusion and Further Work

In this paper we demonstrated that it is feasible to interconnect very different agent systems—Telescript and ffMAIN—using HTTP, letting agents within one system communicate with agents inside the other. HTTP is an obvious choice because it is likely to be supported in some way or other by most agent infrastructures.

We have implemented a prototype system resembling the one shown in Figure 4. We plan to proceed in the direction described in the previous section in order to

provide ‘our’ agents with support for a subset of KQML. Of course there are still many problems to be solved, e.g., how to define the right ‘ontologies’ to ensure that different communicating agents actually ‘understand’ one another. However, the same problem also exists in homogeneous systems. This falls in the realm of AI research. The biggest challenge remains, however: Now that we can get information across the boundary between agent worlds, what about the agents themselves?

References

1. Dag Johansen, Robert van Renesse, and Fred B. Schneider. *An Introduction to the TACOMA Distributed System*. Computer Science Technical Report 95-23, Universitet Tromsø, Institute of Mathematical and Physical Sciences, Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway, June 1995.
2. Holger Peine, Torsten Stolpmann, and Jürgen Nehmer. Ara: Agents for Remote Actions. See <http://www.uni-kl.de/AG-Nehmer/Ara>.
3. Fritz Hohl. Konzeption eines einfachen Agentensystems und Implementation eines Prototyps. Diplomarbeit, Universität Stuttgart, August 1995.
4. Sahin Albayrak, Siegfried Ballmann, Indra Gunawan Harijono, and Többen Hermann. InAMoS: Intelligent Agents for Mobile Services; InAVAS: Intelligent Agents for Value Added Services. See <http://dai.cs.tu-berlin.de/e/projekte>.
5. James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. Technical Report, Sun Microsystems, 1995.
6. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
7. Mark Lutz. *Programming Python*. Nutshell Handbook. O’Reilly & Associates, Sebastopol, CA., 1996.
8. Larry Wall and Randal L. Schwartz. *Programming Perl*. O’Reilly & Associates, Sebastopol, CA, 1990.
9. General Magic, Inc. *Telescript Language Reference*. 420 North Mary Avenue, Sunnyvale, CA 94086, 1995.
10. FTP Software. CyberAgents. See <http://www.ftp.com/cyberagents>.
11. Lucent Technologies. Inferno. See <http://www.lucent.com/inferno>.
12. T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol—HTTP/1.0*. RFC 1945, Network Working Group, May 1996.
13. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.
14. James E. White. *Telescript Technology: The Foundation for the Electronic Marketplace*. General Magic White Paper GM-M-TSWP1-1293-V1, General Magic, Inc., 420 North Mary Avenue, Sunnyvale, CA 94086, 1994.
15. Object Management Group. *Common Object Request Broker Architecture and Specifications (Document no. 91.12.1)*, 1991.
16. Anselm Lingnau, Oswald Drobnik, and Peter Dömel. An HTTP-based Infrastructure for Mobile Agents. In *Fourth International World Wide Web Conference Proceedings*, number 1 in World Wide Web Journal, pages 461–471, Boston, MA., December 1995. W3C, O’Reilly and Associates.
17. T. Berners-Lee and D. Connolly. *Hypertext Markup Language—2.0*. RFC 1866, Network Working Group, November 1995.
18. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

19. Anselm Lingnau and Oswald Drobnik. Making Mobile Agents Communicate: A Flexible Approach. In *The First Annual Conference on Emerging Technologies and Applications in Communications (etaCOM'96)*, pages 180–183. IEEE, May 1996.
20. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proc. of the 3rd International Conference on Information and Knowledge Management (CIKM '94)*. ACM Press, November 1994.
21. General Magic, Inc. Tabriz. See <http://www.genmagic.com/Tabriz/>.
22. Henrik Frystyk Nielsen, Tim Berners-Lee, Håkon Lie, Anselm Baird-Smith, José Kahan, and Jean-François Groff. *Libwww—The W3C Reference Library*. World-Wide Web Consortium. Available from <http://www.w3.org/pub/WWW/Library/>.