Agents, services, and electronic markets: how do they integrate?

M. Merz, K. Müller-Jones, W. Lamersdorf

University of Hamburg, Department of Computer Science

Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany

[merz | kmueller | lamersd] @ informatik.uni-hamburg.de

Abstract

Agent-oriented programming (AOP) is a client/server paradigm that gains increasing attention. Recently emerging *AOP platforms* claim to provide suitable technical support for the implementation of *electronic market* (EM) systems. This paper first argues that an *open system infrastructure* to support the 'right' programming paradigm is a necessary but not sufficient condition for dynamic proliferation of EMs. Another important requirement is *organizational openness* of client/server cooperations. The *COSM* (*Common Open Service Market*) infrastructure aims at generic system support for such applications. It was designed and implemented to realize flexible means for accessing remote services dynamically in evolving electronic markets. The paper then shows how the COSM infrastructure can be extended by AOP concepts in order to provide an adequate open systems platform to support highly flexible service offer and access.

Keywords

Mobile agents, agent oriented programming, electronic markets, client/server communication, open distributed systems

1 INTRODUCTION

An electronic market (EM) can be defined as a computer supported medium that allows both demanders and suppliers of goods to settle economic contracts at any time and without any spatial constraints (Schmid, 1993). Accordingly, the underlying technical system platform (the electronic market system) has to satisfy specific requirements that are also crucial for the coordination of demand and supply in real economic markets: e.g. possibilities to enter the market freely with new (innovative) services, to change product features autonomously and to add value by enriching, combining, or customizing existing services (Merz, 1995).

Agent-oriented programming (AOP) is a paradigm that gains increasing attention. Some emerging AOP platforms (e.g. White (1994), Wayner (1995)) claim to provide

suitable techniques for the implementation of electronic market systems that allow both demanders and suppliers of services and goods to exchange them freely based on electronic contracts and clearing services.

Mobile or *itinerant agents* are metaphors used within a wide range of computer science research fields. As far as this paper is concerned, however, only their programming and communication aspects are examined in greater detail.

The paper first argues that satisfying EM requirements does not necessarily correlate with a specific distribution platform or programming paradigm. The suitability of *AOP platforms* to match the specific EM requirements for flexible service utilization is then examined. A corresponding realization based on the existing *COSM* infrastructure (Merz 1994a) is finally proposed and described.

1.1 Cooperation in open networks

Open communication networks serve distributed software applications as well as human users as a common environment for the mediation and utilization of remote services. Beyond the scope of *communication* matters (like, e.g., message delivery or protocol standardization), distributed applications also realize *cooperation* in the sense of activity coordination in order to achieve a semantically higher goal than pure message exchange. Within the scope of this paper, the term 'cooperation' is understood as (and restricted to) the client/server principle, i.e. a model which assigns to one of the cooperating applications a 'demander role', and a 'supplier role' to the other. Additionally, cooperation in open networks can usually be related in this context to one of the two following application domains:

In the first domain of cooperation, software applications always act as requesters, i.e. play the demander role, by remote procedure invocation on the supplier side. In this context, the underlying communication mechanisms (RPC, message exchange, function shipping (ISO, 1993), or global tuple space approaches (Carriero, 1992) of this cooperation are irrelevant. What matters is the fact that in every case both client and server have to cohere semantically in order to engage in a meaningful cooperation in order to constitute a distributed application. It is important to illustrate the context in which the cooperation partners have to 'cohere semantically': The corresponding (client/server) software components do not necessarily have to be developed by a common team; they have, however, to rely on a common convention - as, for example, pragmatic agreements across the desk, mutual specifications, conformance to some kind of standard, or programming according to a common software documentation. In any case, in this application domain both developed components adhere to something in common - consciously or not, they form a cross-company *domain of conformance*. In the remaining part of this paper, this class of applications is called (organizationally) *closed applications of classified services*.

On the other hand, the second domain of cooperation always involves the *human user*. He is able to relax conformance requirements for a given kind of service. Here, cognition and interpretation of 'fuzzy' application descriptions makes dedicated development of client software components obsolete. The World-Wide-Web (WWW) may serve as an example for this application domain: Interactive servers are used correctly in most cases if the provided service semantics is *obvious* to its users. As will be shown later, the representation of communicated data does not matter either in this case: It may, e.g., be exchanged as an HTML document (WWW) by a simple message transfer, or by a remote procedure call. However in contrast to classified services, as addressed above, the message content is *not* standardized here.

An interpretation of protocol data units does *not* take place after data transfer, therefore data is frequently represented in a plain text format. In the rest of this paper, this class of interactively accessed applications is called (*organizationally*) *open applications* of *unclassified services*.

Of course, there is a trade-off between closed and open applications: In the first case, communication can be handled more efficiently, type-safe, and semantically coherent; in the second case, a service can be provided within a much shorter *time-to-market*, client applications do not need service-specific modules (stubs), and service providers are able to develop individual (therefore more competitive) functionality. In this context, early, innovative movers are better off. There is no general reason, however, to favour one approach over the other - both have their specific merits: classified services are suitable for low-level, well established application domains like FTP, NFS, or the transmission of video streams - interactive services, on the other hand, fit well to user-level applications like ticket reservations or 'kiosk services'.

In analogy to these application domains, there is also a choice of different infrastructure concepts: Which aspects of the application domain should be incorporated into the (standardized) platform design and which should be left to individual applications? In other words: Where should a reasonable borderline between the standardized (the *middleware*) and the 'chaotic/open' part (the specific *application*) of a distributed system be drawn? Evolution of commercial information systems shows that the more relevant a system component is for service competitiveness, the more potential for individualization should be allowed by the platform design: At the *application presentation* level, e.g., home banking on-line services provide user interfaces and dialogue control that adhere more to *individual* corporate identity objectives than to *standardized* layout rules. Service types vary accordingly. On the other hand, there are strict conformance requirements to *communication services*.

As shown below, AOP provides a technology that allows development of both closed and open applications. However, before AOP extensions to the COSM open systems platform are introduced, a closer look is taken at system infrastructure aspects which can either be provided by distribution support or are implemented individually by the application itself.

1.2. Open systems platforms

System platforms supporting open client/server cooperation can be classified according to the standardization degree of various functional elements (i.e. by the level of their respective standardization borderline)*:

- *server behaviour* and application semantics which are usually not a subject of standardization in most infrastructures. However, an *implicit* standardization is given if client and server applications are developed by an organizationally closed team (compare to closed applications like, e.g., FTP clients and server). Also CORBA *object services* show, for example, a well-defined, *standardized* behaviour.
- *service definitions* are in most cases represented as an IDL (interface definition language) script. The scope of specifiable service characteristics is therefore restricted to a standardized grammar. IDLs are usually not capable to describe service semantics and EM

* In this context, 'standardization' refers to and is limited to application-specific conventions. A 'print' function, for example, is standardized in this sense if distributed application components (client and server code) adhere to a distinct function signature and semantics.

relevant aspects such as, e.g., billing information. Only CORBA allows to represent interface definitions as first-class items. No platform supports extensible IDL grammars or dynamic evolution of the repository schema in order to allow applications to specify additional service characteristics.

- *quality of service characteristics.* In *ODP trading* (ISO, 1993), *service attributes* that characterize server capabilities have to be agreed upon before any server can be accessed on this basis. PAPER SIZE or QUEUELENGTH have to be well-typed and well-known to the print server, client, and the trader itself resulting into well-standardized applications.
- The *user interface design* is usually left to individual application developers. On the other hand, *representation* and *behaviour* of user interface components, such as buttons or data controls is standardized in systems like WWW or Compuserve.
- Finally, conventions on *communication protocols* are the least common denominator to any distributed application standard. Each open distributed platform trivially relies at least on a common *communication standard*.

Reality shows various approaches which can be distinguished according to their respective different *standardized functions* (Figure 1). This set of choices raises several questions for realizing highly *generic* system platforms for open distributed applications, for example: Which level of conformance is flexible enough to combine an individual application design with efficient infrastructure standards? User interface layout is surely *not* something to be standardized. As argued in Merz (1995), application semantics must remain individual within the context of an electronic service market as well.. Service attribute and service type standardization in the sense of ODP trading (ISO, 1993) is even contra-productive if it is an essential requirement for service accessibility in a competitive scenario. However, as an add-on feature, trading supports service selection efficiently.

Figure 1 Cooperation platforms and their standardized functions.

What remains is to allow providers to offer also *individual, unique* application services and to extend existing distribution platforms accordingly by means of access support to this kind of applications. For designers of electronic market platforms, a general question therefore addresses the gap between requirements of market dynamics and the capabilities of technical

communication systems. This paper argues that the *agent-oriented programming* (AOP) approach provides means to bridge this gap if correctly applied.

The rest of this paper is organized as follows: Section 2 introduces some practically relevant approaches to AOP and analyzes how these satisfy the requirements of an electronic service market. Section 3 shows how adequate AOP concepts can be integrated into the COSM platform and which extensions to the COSM service representation are necessary. A closer look at the structure and behaviour of COSM agents is given in Section 4. After presenting an AOP example and its implementation based on the COSM platform, Section 5 finally focusses on open issues which are addressed in the ongoing COSM design and prototype development.

2 BASICS ON AGENT-ORIENTED PROGRAMMING

Agent-oriented programming as a computer science research field reaches from distributed AI (e.g., Shoham, 1993), distributed programming (e.g., Tsichritzis (1987), Harrison (1995)) to the field of computer communications (e.g., Tschudin, 1993). This paper focusses on client/server cooperation aspects of AOP. In this context, AOP approaches from research as well as from commercial products relevant to open service markets are examined.

In the following, we define a *mobile agent* as an encapsulation of *code*, *data*, and *execution context* that is able to *migrate autonomously* and *purposefully* within computer networks *during execution*. The *agent system* provides an algorithmically complete programming language. Therefore, an agent is able to react sophistically on external events. An agent may be persistent in the sense that it can suspend execution and keep local data in stable storage. After resuming activity, an agent's execution is continued, not necessarily at the same location.

According to the definition, a migrated *process* is not an agent unless the process itself influences the migration target. A remotely executed *script* (see, e.g., SUN's Java language extension to HTML) is not an agent either unless execution state is transferred and full execution autonomy is given. Agents only cease to exist when they are explicitly deleted. (Self-) deletion may be carried out by an agent program. Agents act on behalf of a (human) *principal*, who - directly or indirectly - defines the goal of an agent's activity. The given definition restricts the possible variety of agent implementation approaches to a quite small set of possibilities: each case requires a local evaluator - the *engine* - in order to execute the agent program.

The most obvious agent application domain seems to be *information gathering*. Agents may, for example, perform database queries on behalf of their principals. This may result into a local full-text query or, e.g., a price inquiry for a specific good at each engine involved. Implementing such information gathering agents requires specific knowledge about server semantics - therefore client and server form a *closed application* as defined in Section 1.

Another rationale for agent activity is the provision of added-value: existing - maybe heterogeneous - resources, like booking services, are accessible via an agent through a customized user interface. In this case, a (commercial) third party provides the agent as a *facilitator service* in order to support access to 'awkward' services. A *facilitator agent* can then be transferred to the client's site and be bound to the local user interface library. This enables users to access the facilitator service without using dedicated client software. Here, AOP bridges the gap between third party requirements of individualization (agent program) and infrastructure standardization (user interface components).

Agents may also be developed to provide *notification services*. Here, messages are sent or activities triggered after a distinct event has occurred. This may happen, for example, if an application state changes at the agent's site.

2.2 AOP models

AOP models have various aspects: the *programming* model provides the conceptual basis, but in the context of open distributed applications also agent *migration* and *cooperation* aspects are of interest.

AOP *programming models* are frequently Smalltalk-like in combination with an abstract machine (Wayner, 95). Such mechanisms provide the required homogeneity for allowing seamless transfer of evaluation contexts between heterogeneous hardware and operating system platforms. Compared with communication mechanisms like RPC, agents carry their own process state around. This fosters failure recoverability if the agent state can be preserved by a persistency service. If an agent thus encapsulates client application state, the accessing application may remain stateless - and therefore easy to recover. However, Harrison (1995) argues that even such client/agent cooperations require a specific protocol in order to access the results, i.e., to relate the returning agent or response to the original request. The client application must therefore at least preserve an application protocol state. This restriction applies in any situation where service-specific semantics is obeyed by client implementations - thus leading back to the world of closed applications again.

Another approach to *migration* is to encapsulate evaluation state as far as sufficient by a persistent storage manager (object store) and to allow accessing applications to perform computations on this data in a coordinated way. This approach does not necessarily require an agent representation 'substrate' that circulates through the distributed system. Just a notification message may pass control from a previous evaluator to the next. Heterogeneity is bridged by the underlying communication mechanism like RPC. Although such an approach was considered as well in the COSM project, this paper focusses on the 'substrate' approach.

Agent *cooperation* requires coordination of purposeful activities of several partners in order to achieve a common goal as, e.g., carrying out a purchase. Here, two variants can be distinguished: *direct* interaction between cooperating (client/server) agents ('direct interpretation') or involving a *third party* that supports a purposeful 'match' between the interacting agents ('3rd-party-matching').

2.3 AOP in an open world

This paper argues that an electronic service market emerges most dynamically if an infrastructure that fosters proliferation of open applications is provided. However, our examples have shown that AOP can be interpreted as programming local to the client site. This view is based on a 'closed application' paradigm, since, in this view, service providers and agent programmers constitute a joint domain of conformance as described in the first section.

AOP is thus best applicable if the domain of conformance is restricted to the service providing organization. The only possible way to decouple client and server in such a scenario is to install a server-created agent at the clients site. The respective process may be executed either by the server as a 'remote installation' or by the client by 'agent acquisition'. In either case, such installation is only possible with the client's benevolence. The agent encapsulates all information and code required to allow human users to interact reasonably with the agent itself or the remote service. This principle will be called the *reverse agent principle* (Figure 2).

For supporting electronic market applications, AOP offers certain characteristics that make this approach more favourable than, e.g., RPC: Agents may directly support the application-level protocol required to invoke specific server operations. This enables vendors to allow server access not via specific client applications but rather by general access information to remote resources - encapsulated by a respective agent (Tschudin, 1993). Today's service distribution mechanisms are, in most cases, still based on delivery of RPC or message-passing oriented client applications with all the inherent disadvantages as, e.g.: the lack of persistent application state, arbitrary user interface styles, the burden of handling a large range of software applications with individual installation procedures on client systems, etc. Individual and spontaneous client/server cooperation is only possible if this burden is minimized by, e.g., dedicated facilitator agents that allow for the reverse agent principle.

Finally, powerful *user interfaces* are required for a meaningful interaction between human users and agents. Therefore, an additional client component is needed to support service agent administration and agent interaction. In the case of MagicCap™ (White, 1994), e.g., a city-metaphor is used for a browsing directory to support user access to installed agents resp. services. Here, each service is represented by an individual building; user/agent interactions can take place after a building has been 'opened'.

Figure 2 Agents within the contexts of classified and unclassified services.

Section 3 introduces the *COSM infrastructure* as a complementary platform for the reverse agent principle. Designing and realizing an adequate system support for the access to remote services in open environments is the main goal of the COSM approach. This leads to a system software platform for realistic electronic service markets which are dynamically evolving and comprise great varieties of individual service offers and many service requesting clients.

3 THE NEXT STEP: INVOLVING THE HUMAN USER

As argued above, closed applications have several disadvantages in the electronic service market field. Most importantly: they require dedicated client components which have to be individually installed. Therefore, the burden of system maintenance is shifted to the client users.

On the other hand, current *open* applications, such as interactive WWW services, neither allow to store service descriptions (HTML documents) in a well-structured form nor do they provide sophisticated call-interfaces that conform to existing middleware platforms like CORBA DII (Dynamic Invocation Interface). Combining the advantages of both approaches - decoupled client and server development on the one hand, call interfaces with well-structured and well-processable interface descriptions on the other - leads to the development of the COSM platform (Merz, 1994a). The result is a flexible service implementation and access mechanism based on dedicated service representations.

3.1 Support for a Common Open Service Market

The *COSM system platform* was designed and built in an ongoing project at the University of Hamburg in order to combine and support the advantages of both closed and open applications. The result is a flexible service implementation and access mechanism based on the concept of self-contained, generic service representations. A COSM service representation (SR) contains several descriptonal components defining, e.g., the operational service interface, the *Generic Client* user interface layout, the interrelation between user interface and remote procedure invocation, human-understandable descriptions of service functionality, billing information on the charge of procedure invocations, a definition of legal orders of invocation sequences, etc. The COSM platform prototype consists of the following components:

- A *Generic Client* (GC) component that supports users in service discovery, access and binding, and in the inspection of service descriptions at run-time.
- A common *Service Representation* (SR) which contains several descriptonal components (defining the service interface, the GC user interface layout, the interrelation between user interface and RPC invocation, a definition of the legal order of invocations, etc.).
- *Service providers* which implement dedicated functionalities and are accessible on-line in the COSM network. (They supply their respective SRs to potential users - either directly or indirectly via GC or mediation services).
- The *Object Request Broker* (ORB) which is used as middleware platform for DII-based (therefore dynamically typed) parameter transfer, and server activation. The SR is used as input for a stub generator only at the server's site.
- *SR browsers and repositories* which support users to inspect and store SRs.
- *Service traders* which identify appropriate service suppliers by an automated search for the best possible services based on given service description criteria, and, finally,
- *mediation services* provide application specific tasks like, e.g., collecting news articles for their customers.

In this environment, the SRs correspond to the reverse-agent principle of the AOP context as defined above: The COSM agent (resp. the SR) is developed and provided by the remote server. The user selects an appropriate agent from a distribution service like a repository. Agent and service are implemented within the same domain of conformance - whilst the client remains generic in order to allow its user to access any COSM server. After both client and GC have received a server's SR, the corresponding user interface can be generated automatically at the client site. The user can then inspect the information provided by the SR and familiarize

with the described service functionality. So, a direct client binding to the server can finally be established (or released if this information does not describe the kind of service the user was looking for). The user interface representation of a remote services site is standardized at every GC; service layout and content, however, may vary from service to service. This interface enables users to execute remote operation calls just by filling out forms and pressing corresponding buttons automatically.

Based on this agent/SR analogy, the COSM platform can be extended smoothly with AOP concepts by simply augmenting COSM SRs: A COSM SR could thus contain both code and control flow specifications. Before focussing in more detail on this task, the following section first elaborates a bit more on COSM SRs. Section 4 then presents SR extensions for control flow specifications based on Petri Net representations.

3.2 COSM service representation

In COSM, the crucial data object to contain and transfer arbitrary and location independent service descriptions is the *service representation* (SR). At the most generic level, the SR is a container for data structures of arbitrary run-time types. After receiving an SR, an *SR-interpreter* (a component of the COSM Generic Client) seeks description components it is able to interpret. In the case of the Generic Client there are the following components:

- A specification of operation descriptions, containing operation names and parameter descriptions. Parameter descriptions refer to data objects which contain actual values. These components resemble roughly IDL clauses known from DCE or CORBA.
- A specification of the *user interface* to be generated for human users by the GC. It contains specifications for dialog boxes, data editors and push buttons. Data editors manipulate data objects,
- a specification of the service interface protocol, i.e. which operations are enabled to be invoked at a given state. Currently, this protocol description is based on a *finite state machine* model and comprises a set of application states and transitions between them which refer to operation descriptions. State changes are effected by user-level events and result into RPCs,
- informal description components as, e.g., help texts which directly support human users,
- ‘Price tags’ that inform users on operation invocation costs, and finally
- any local data values used by the GC in order to represent the client state.

Since the COSM Generic Client is not restricted to a distinct set of applications, specific service state information, such as window positions or counter variables, can also be captured by the service representation. Furthermore, it is also possible to *store* the SR *persistently* and to *suspend* interactions with the remote server temporarily and resume them later on based on the stored SR status information or from another network site.

Compared with HTML documents, the SR approach therefore facilitates storage and query processing for service descriptions: SRs can be persistently stored in dedicated repositories. Type information on SR components are used to extend the repository database schema accordingly in order to allow service providers to extend SR description by individual data. This

allows users to place queries against the repository like: “Return all servers that provide an operation ‘Book’, or: ‘Return all parameters of the operation ‘Book’ in SR ‘CarRental’”.

3.3 AOP on the COSM system platform

SR-based agent specifications do not directly invoke remote procedures; they are rather used to guarantee type-safe remote service invocation initiated by the generic client on behalf of a human user. Taken together as a unit, user, generic client, and server form an *engine*, that interprets SRs and performs operations on their local state. Service users trigger invocations, and the server implements the respective service (operation) semantics. If, e.g., a facilitator agent is to be developed, the following extension is required for the COSM architecture: Remote services need *engine capabilities*, i.e. abilities to transfer and receive SRs and to evaluate locally those basic operations of the agent language that are common to all agents. In such an environment, e.g., the operation *MoveTo* <hostname> transfers the service representation including the current evaluation state to the receiving host and *End* terminates the agent, by letting the engine delete the service representation. Every other operation is not carried out by the COSM engine itself but is invoked remotely at the server.

The integration of agents with the COSM generic client user interface also requires a graphical representation of agents that are installed but currently not used. For that purpose, a directory as being used by Telescript (represented by a city metaphor) is externalized into a browsing service in COSM. Agents ready to be used can be obtained from the remote browser.

4 INTEGRATING COSM AND AOP

This Section first focusses on Petri Net representations of agent-based control flow specifications and then on the way agent migration is actually carried out in COSM. An example application of the current COSM prototype implementation illustrates how parts of the COSM infrastructure interact in order to support agent circulation in open networks.

4.1 Embedding control flow into COSM SRs

The base model of COSM did not provide means to incorporate control flow specifications into SRs. This motivates the corresponding SR extension by Petri Net representations as presented below.

In the COSM context, the basic Petri Net model is semantically extended by the concepts of *split* and *join transitions*: a split transition separates a single SR into two or more independently executable SR net representations. In contrast to the standard definition, split transitions do not mark each outgoing place with a token; they rather generate a copy of the net representation after tokens have been withdrawn from all incoming places and occupy only one single outgoing place per net instance instead. In COSM terms, additional *sub-SRs* (resp. subagents) are thus created out of the *master instance*. These instances can be processed independently until a *join* transition is reached. The join transition is not enabled unless all required instances have arrived at the input place. All net instances, except for the master, are removed when the join transition fires. To carry out a join, the required SRs have to be co-located at the same engine.

Each transition definition is associated with one operation description within the SR. If the transition predicate evaluates to *TRUE* transition firing results in an invocation of the referred operation.

Pairs of split and join transitions may be *nested* in net definitions. Therefore, a dedicated naming schema is required to assure correct identification of the agent instances to be merged. These agent instances form a common *task* that is set-up by an initial agent. Tasks belong to a *task type* and tasks of various types may be carried out within the engine network at the same time. A task type is defined by individual agent implementors. The resulting naming schema appears as follows:

1. *Task type identifiers* are coined by agent implementors. However, to enforce uniqueness, a type identifier comprises a universally unique identifier (UUID).
2. At each task creation (i.e. when an initially marked agent is created), a globally unique *task identifier* is appended to the type identifier.
3. Each split transition, in turn, appends a locally unique *instance identifier* to the task identifier. The join operation is therefore required to strip off this identifier and to test the remaining part on equality. Of course, instance identifier may be appended consecutively at each split. This schema assures a process, that merges the least recently created agent instances first.

4.2 Go for it!

In summary, the basic components of the COSM based *AOP engine architecture* are

- the *service representation* as an encapsulation of the agent's local state and control flow,
- the *engine* as an agent evaluator: The engine carries out operation invocations and passes agents over to other engines. Application-specific operation calls are invoked at remote servers which conform to the agent's service description. Every engine performs operation invocations at the same server.
- *COSM servers* provide application services at remote sites. Their operations are invoked by engines. Servers are not specific to the AOP approach, they can still be invoked the 'traditional way' from generic or specific clients. Therefore AOP as a distributed application concept remains compatible with the existing RPC mechanism of COSM.

Agent operations are distinguished into *well-known commands* and *application-specific* ones: Well-known commands can be locally executed by every COSM engine whilst the latter are passed on to the engine-specific COSM server. Such a service invocation is executed as if done by a generic client: First, the current Petri Net state is determined by the engine. If the predicate of the currently enabled transition evaluates to *TRUE*, the transition is fired. By means of the SR operation definition, a named value list is created and transferred to the COSM server through a CORBA DII invocation. Each invocation modifies the local state of an agent due to the delivery of application specific data.

Agent splitting and joining is carried out by the evaluating engine. A split transition definition within the SR is associated with an address list of target engines. When firing, the engine creates subagent instances - one for each element of the address list - which are transmitted to the respective target sites.

Concurrent agents are merged into a single instance if all of them belong to the same task and have arrived at the same engine. These instances are stored locally to the engines site unless all of them have arrived. In this case, all except for the master instance are simply deleted.

Concurrently migrating agents that belong to the same task change their local state individually due to different operation invocations at different engines. Usually the join transition is carried out at the agent implementor's site, since agent and server belong to the same domain of conformance. In principle, any application specific aggregation function may be applied to the individual results. Consistency is therefore achieved by application-specific code that is invoked at the server of the joining engine by each agent instance.

4.3 An example

The following example illustrates an agent's task which aims at obtaining price information from a list of services: At the user's *Home Server*, an agent instance is created and supplied with initial data by the human user. It is then transferred to the *Address Server* by executing an according *MoveTo* command. After an address list with servers to be visited is obtained the agent splits into three instances: the master agent, and one subagent per address in order to perform the *GetPrice()* operation at the respective server(s). Each server still acts as an ordinary COSM server that can be utilized immediately by the generic client as shown in Section 3. After all agent instances have collected their respective price informations, a *Join()* is carried out on the fifth host - a host that belongs to the agent implementor's domain and that is therefore able to perform a task-specific collate function of individual results. For the example task given here, resulting prices are simply aggregated. Finally, the remaining master agent is redirected to the user in order to present the result via the generic client's user interface (Figure 2).

Figure 2 Migration path for the sample application

5 OUTLOOK AND SUMMARY

The current prototype of the COSM AOP system platform was developed in C++ for an IBM RS/6000 workstation with AIX. A second implementation exists for OS/2 with the option to run engines on both platforms and to exchange agents across platforms. The necessary heterogeneity transparency is achieved by a representation-independent implementation of SRs.

This prototype for a COSM AOP infrastructure could be implemented based on several already existing COSM concepts and components - like the generic client, the service representation, the repository servers etc. Embedding additional functions - e.g. security mechanisms like a *notary services*, *electronic currency*, or inter-organizational *workflow management* based on COSM agents - is a current subject of research. In addition, more sophisticated tools for the comfortable and safe development of agents and servers are currently under construction:

- *Workflow management* extension: The agent is extended by role information in order to interpret the embedded control flow definition as a workflow that allows only authorized users to perform server invocations. By this means, agents coordinate user accesses to COSM servers.
- In the current infrastructure, there are no means of securing local sites (engines) against hostile agents or fraud. Any function call is executed by an engine that was written by the application programmer - therefore it is the programmer's responsibility to disallow critical calls. But if an unauthorized principal lets such agent circulate, elaborated *security mechanisms* are required in order to acquire a certification for the agent's principal.
- In realistic electronic market contexts, additional *electronic payment services* are required in order to allow a secure exchange of electronic currency (Merz, 1994b).

Agent oriented programming in combination with the reverse agent principle is a promising approach to implement a platform for electronic service markets. However, standardization of user interfaces, service attributes, etc. becomes a sensitive design decision in this context. Application conventions are reduced to the interpretation of SRs and GUI components in COSM. If the platform imposes over-standardization to its applications there is no possibility left for service providers to keep the application open to differentiate from competitors by embedding innovative functionality into the own service. It was shown that conventions imposed by closed applications can be considered as standards with all shortcomings they imply. Therefore, one rationale for the agent-oriented approach presented in this paper is to identify an organizational model that fosters electronic market proliferation.

The approach based on the COSM architecture aims to add-on AOP facilities to the basic CORBA DII based client/server infrastructure. This keeps AOP as an option which the application implementor may utilize or not. Thus, the service provider's design autonomy is always preserved. Embedding Petri Nets for control flow specifications and to coordinate computations on arbitrarily distributed services encourages third-party vendors to provide added-value services easily based on facilitating agents. This second rationale for the chosen AOP approach emphasizes the extensibility aspect of open distribution platforms. The COSM infrastructure purposefully extends existing concepts of de-facto middleware standards (like CORBA) in order to demonstrate the principal feasibility of future product developments along such a line.

6 REFERENCES

Carriero, N., Gelernter, D. (1992) Linda in Context. CACM, 32(4):444-58.

- Doorn, M. v., Eliens, A. (1995) Integrating applications and the World-Wide-Web. *Computer Networks and ISDN Systems*, **27**, 1105-10.
- Harrison, C., Chess, D., Kershenbaum, A. (1995) Mobile Agents: Are they a good idea?. IBM Research Report RC 19887.
- ISO/IEC JTC1 SC21 WG7 (1993) Basic Reference Model of Open Distributed Processing. Working Documents N7053 ('RM ODP') and N7047 ('Trading').
- ISO/IEC JTC1/SC21 (1993) Remote Database Access (RDA) - Service and Protocol. OSI International Standard 9579-2.
- Merz, M. , Müller, K. , Lamersdorf, W. (1994a) Service Trading and Mediation in Distributed Computing Systems. *ICDCS* pp. 450-7.
- Merz, M. , Müller, K. , Lamersdorf, W. (1994b) Trusted Third-Party Services in COSM. *Electronic Markets*, 4(12).
- Merz, M. , Müller, K. , Lamersdorf, W. (1995) Electronic Market Support for the Tourism Industry: Requirements and Architectures, in *Proc. ENTER95*, 220-9.
- OMG (1991) The Object Request Broker: Architecture and Specification. OMG Document No. 91. 12. 1.
- Schmid, B. (1993) Electronic Markets. *Electronic Markets*, **3** (9/10), 3-4.
- Shoham, Y. (1993) Agent-oriented programming. *Artificial Intelligence*, **60**, 51-92.
- Tschudin, C. F. (1993) On the Structuring of Computer Communications. PhD thesis, University of Geneva, 1993
- Tsichritzis, D., Fiume, E., Gibbs, S. , Nierstrasz, O. (1987) KNOs: Knowledge Acquisition, Dissemination, and Manipulation Objects, *ACM TOIS*, **5**(1), 96-112.
- Wayner, P. (1995) Free Agents. *Byte*, **20**(3) 105-14.
- White, J.E. (1994) Telescript Technology: The Foundation for the Electronic Marketplace. White Paper, General Magic, Inc.

7 BIOGRAPHY

Michael Merz studied Computer Science and Business Science at Hamburg University. His PhD thesis is dedicated to the interdisciplinary study of 'Electronic Service Markets' - the integration of coordination mechanisms from economics into IS communication infrastructures. Beside research assistant tasks he works as a IS consultant and prefers to sail catamarans in the spare time.

Kay Müller-Jones studied Computer Science at Hamburg University. His PhD thesis is dedicated to trading and coordination in the context of ODP standardization. This field includes activity management based on distributed transactions. Kay prefers windsurfing.

Winfried Lamersdorf studied computer science at the universities of Munich and Hamburg. After graduating with a doctorate from the University of Hamburg he spent a post-graduate year at the University of Maryland. In 1983 he was invited as a research scientist to the IBM Scientific Centre and the European Networking Centre in Heidelberg. Since 1986 he also participated in formulating international communication standards for remote database access. In 1991, he became Professor of Computer Science at the University of Hamburg. The emphasis of his current research lies in the field of open distributed systems.