An Extensible Software Architecture for Mobile Components

Dag Johansen* Kåre J. Lauvset* Keith Marzullo†

October 6, 2000

Abstract

This paper presents a generic software architecture for large-scale distributed applications where mobile agents are an integral part. We have devised this architecture through completion of a series of mobile agent systems and associated applications over the last 8 years.

1 Introduction

Architecting large-scale distributed applications is difficult. Part of the problem is that while there are many ad hoc application architectures, there is very little agreed upon generic architectures for the analysis and modelling of complex, large-scale applications. This is about to change. Software architectures are now emerging as an important software engineering discipline, where the goal is to carefully devise and approximate frameworks that capture the essence of an application domain. An architecture of this general nature should include aspects such as functional components, their relative control and communication relationships, as well as non-functional aspects such as scalability, security, fault-tolerance, and maintainability.

Mobile agents have potential for being a convenient structuring technique in distributed and Internet applications. The fundamental idea is to move parts of the client computation closer to the server(s). By virtue of moving the computation over the network, several advantages can be achieved [1, 8]. For instance, the computation can exhibit better performance by working locally on data residing at the server. The agent can do a single-hop to execute close to one server, or multiple hops, where it moves among a set of servers.

A number of mobile agent systems are currently under development, others are already put into use. Though, developing applications with mobile agents is still considered to be a complex issue. Hence, we have been interested in specifying an architectural model specific to mobile agent centric applications. The goal is to allow agent techniques to be utilized by non-agent literate developers. Our approach is to derive architectures for this type of computing. In particular, we have derived a generic software architecture where mobile agents are key. We call this the *ACE architecture* [5] (Agent Computing Environment). This architecture has been derived in an iterative manner over the last 8 years to reflect the more specific architecture of real, existing systems. For each iteration, we developed a version of TACOMA, a mobile agent system [7]. Next, we built applications using TACOMA and, then, refined the

*Department of Computer Science, University of Tromsø, Tromsø, Norway. This work was supported by NSF (Norway) grant No. 112578/431 and 126107/431 (Norges Forskningsråd, DITS program).

†Department of Computer Science and Engineering, University of California San Diego, La Jolla 92093-0114, California, USA. In doing this work, Marzullo was supported by NSF (Norway) grant No. 112578/431 (DITS program).

ACE based on real experience from the system developers and the users. Over the years, we converged towards a framework which includes features a distributed mobile agent application should incorporate.

The rest of this chapter is organized as follows. Section 2 presents how mobile agents can be used to build extensible systems. In section 3, we present the ACE architecture. Then, in section 4 we discuss where and how the ACE can be applied in the software engineering process. Section 5 concludes this chapter.

2 Agents and Extensibility

The fundamental problem we have been studying in TACOMA is how to structure distributed applications with mobile agents, or *agents* for short. Initially, we assumed that an itinerant style of agent computing was the rule, where the agent moved about between the TACOMA hosts. In retrospect, we learned that distributed applications, in most cases, are best structured as collections of interacting components [6], some being stationary, others mobile.

Our experience revealed some non-obvious benefits associated with the use of agents for structuring distributed applications. The most practical was that agents provided an excellent mechanism for building flexible and extensible services. For instance, agent-enabled servers can be adapted and extended rapidly to facilitate new or changed requirements. An agent can carry a service request to the server and interact using local inter-process communication mechanisms. Invoking server operations locally is cheap, hence server interfaces can be primitive. Sequences of these operations can be invoked by the agent in order to service any given client request. In effect, the agent implements its own specialized server operations that can be both efficient and well suited for the task at hand.

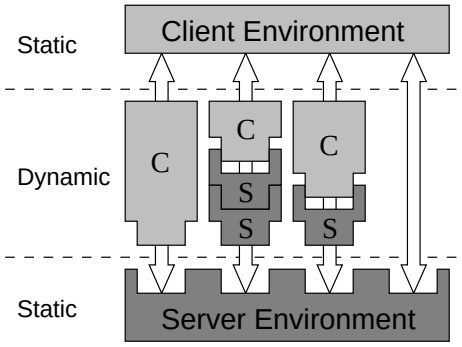

Figure 1: Overall architecture.

Figure 1 illustrates a more overall architecture where agents are used to build extensible systems. Just as in the traditional client/server architecture, the agent architecture consists of static client- and server environments. The *Client Environment* is basically the software local to the user requesting services from the remote Server Environment. The *Server Environment* includes a run-time environment, the TACOMA system, supporting installation and execution of dynamic software components.

Interaction between these two environments can go through standard client/server mechanisms like, for instance, RPC. Alternatively, the interaction can go through an agent (C) that is launched from the Client Environment, transported over the network, and installed close to the Server Environment. The agent is part of an application by implementing some specific functionality of that particular application. The application in the figure consists of three such agents, in addition to the Client Environment.

The overall architecture also contains another type of agents, system agents (S). This type of agent is part of the system infrastructure and can be used to enhance the Server Environment API. Examples of such extensions are specialized search and filtering algorithms. System agents are also installed over the net-

work, but typically by system administrators to extend servers with new functionality.

3 ACE Architecture

We consider the ACE architecture to be a generic architecture. This implies that the architecture contains abstractions derived from a specific domain, and our domain has been one with agent-centric applications and systems. The applications we have built include [4, 6, 9]:

- An MPEG based video-server where agents can visit searching for specific film scenes.
- A satellite-image server where agents can visit searching for images containing certain objects, patterns or colour-values.
- A mobile web robot that can be moved close to web servers for execution (checking of local links). The web robot is based on legacy code wrapped as a mobile agent.
- Alert systems, especially in the Arctic weather domain, where agents can be launched from cellular phones, PDAs or web-browsers and installed at remote web servers for periodic execution.
- Itinerant style computations like, for instance, a workflow management toolkit where a multi-media document is sent through a pipeline of hosts for execution.
- A Web CV crawler, where a dedicated troop of cooperating agents are monitoring a number of databases to set up a team of people matching some specific requirements.
- An agent-based distributed systems management toolkit for installing, monitoring and controlling progress of software components. We generalize the idea of mobile agents in this application and use TACOMA as a deployment methodology for any type of software component that needs to be installed.

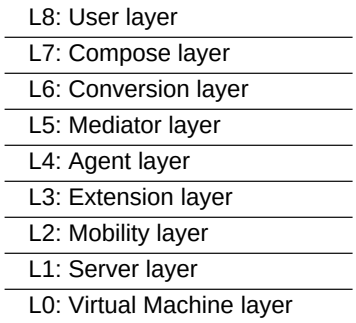

Figure 2: The layered ACE architecture.

The current ACE has been slightly modified from the first version, which had eight layers [4], These layers captured the concept of an itinerant agent moving about, but the concept of server extensibility was not properly covered. To reflect this better, we have added a new layer, the *Server Layer*. Also, we have changed the name of several layers to reflect more generic aspects identified through application construction the last two years. The current architecture is illustrated in figure 2.

3.1 User Layer (L8)

The topmost layer in ACE is basically a GUI for deploying and controlling the agents out in the network, and for receiving status-reports and results from them. In general, this layer should have no real agent computing, but basically function as a front-end to the remote agents. Examples of tasks that must be supported by the user layer are launching remote data queries and searches, updating remote databases and configuring remote triggers (i.e. software that notifies the user when

some threshold values have been reached or some breaking news has occurred).

The GUI requirements of the application will vary, but two specific issues might need to be considered at this layer:

- Connectivity, especially for up-calls, can be vital. Agents are typically used for programming remote alarms with real-time response requirements. This connectivity is important to enable interruption of the user that some user-specific alarm has been triggered.
- The GUI itself might have to be built for users in motion. This implies that speech, simple icons, and power-greedy solutions are used.

3.2 Compose Layer (L7)

Structuring distributed agent-centric applications from components can be difficult. Hence, we need a toolkit aiding the engineering process. We will illustrate the problem by an example scenario. A system architect faces the challenge of building a distributed application by assembling and structuring a set of software components. He uses his architecting software tool to drag-and-drop the components onto a design sheet. The tool allows the architect to configure a number of aspects of the applications like, for instance, information flow and inter-component dependencies, critical components, resource requirements and parallel tasks. Furthermore, he uses wildcards for issues he cannot determine at design time.

Figure 3 shows a graphical view of an example distributed application assembled from software components. The graph could have been the one on the design sheet in the example above, where information flow is shown as arrows. The figure should be read as follows: component *C0* provides input to *C1* and *C2*, which in turn create some output that is taken as input by *C3*. Further, several instances of *C4* are executed in parallel, each instance reading input from *C3*. Finally, the output from all instances of *C4* is correlated by *C5*.

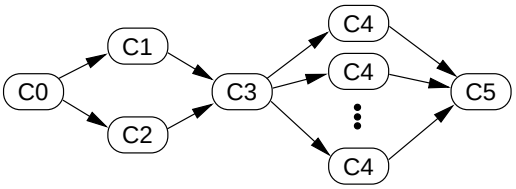

Figure 3: Component dependency graph.

The output produced by the compose layer is a specification describing the structure of the distributed application at stake. This specification must have a well defined and generic format since it defines the interface against lower layers in the architecture.

3.3 Conversion Layer (L6)

The function of the *Conversion layer* is to accept specifications of agents from layers above and convert them to a standard format. Hence, specifications of agents are expanded to standard programming languages or executable binaries. Also, some results computed by agents may have to be converted to a format suitable for client devices with limited capacity. For instance, a cellular phone can launch agents written in a specialized and high-level programming language. This can be a predicate, as in the StormCast alarm service, where an agent is represented as ASCII text in an SMS-mail [3].

```
(temperature < -10.0)
  & (humidity > 70 | windspeed > 12)
  +47 776 44000 & ola.norseman@cs.uit.no
```

Figure 4: StormCast alarm predicate.

The first two lines in figure 4 is the predicate to be evaluated by the server, and the third is how the user should be alerted in case of a posi-

tive predicate evaluation. In this case, notification is sent as a short alarm to a SMS-address, while a message containing much more status information from the server is sent to a specific e-mail address.

3.4 Mediator Layer (L5)

The previous ACE named this layer the Global Management Layer, reflecting that global scheduling decisions took place here. The name change reflects that global scheduling is a subset of mediating clients and services. The *Mediator Layer* provides management policies for deploying and managing agents launched into the TACOMA system. The functionality provided her includes aspects like:

- Global scheduling aiding the agent in the process of where to execute (next). An agent launched into the system can have a pre-specified itinerary, which makes this service obsolete. In other cases, the agent determines itself where to go next. Broker and mediator services in this layer can be consulted for this purpose. The agent can have meta-information associated with it specifying service requirements or preferences.
- Policies controlling the execution of agents. Examples here includes failure-detection policies and policies determining termination and garbage collection of agents.
- Persistent global state collection for auditing and replay purposes. This can include third-party services keeping track of transactions and status logging of roaming agents for debugging purposes.

3.5 Agent Layer (L4)

The *Agent Layer* contains the remote agents implementing basic *functionality* of the agent-centric applications. The application functionality is typically implemented by static components in the User Layer and dynamic components in this layer, while other layers in the ACE are more concerned with non-functional aspects of the application.

3.6 Extension Layer (L3)

The *Extension Layer* is used to customize the individual hosts agents can visit. This is illustrated by the S components in Figure 1, which creates an agent run-time environment at the individual hosts. The S components can be deployed and installed much the same way as an application agent is. Examples of services found in this layer include local accounting mechanisms, monitoring of the local node resources, and error recovery mechanisms for agents at this particular host.

Services provided in this layer are typically closely connected to the Mediator Layer. As a rule of thumb, policies are provided by L5, while associated mechanisms are provided by L3. A L3 component can, for instance, perform load monitoring at the local host and report to a global scheduling component in L5. L5 then uses load information from several hosts to determine where an agent should be executed.

This layer used to be called the Local Management Layer in the previous ACE architecture and was stacked above the Agent Layer. That ordering reflected that the Local Management Layer basically was used in the scheduling and deployment of client agents (C). Now, the architecture also captures the existence of server extensions (S), which belong in this layer. The name change also reflects that local management services are a subset of more general extensions.

3.7 Mobility Layer (L2)

The *Mobility Layer* provides a mechanism for transporting agents between the hosts within the Server Environment. A mobile agent system must be installed on these hosts to function as a configuration backbone for a component-based distributed application.

3.8 Server Layer (L1)

The *Server Layer* is added to this ACE, and it contains the servers that are part of or can be used by applications. The traditional server accessible over the network through standard message passing techniques belongs in this layer. The same does servers who can host roaming agents, whether it is a client agent (C) in L4 or a system extension (S) in L3 extending the server API.

3.9 Virtual Machine Layer (L0)

The *Virtual Machine Layer* consists of the server hosts included in the ACE. This layer provides a basic computing infrastructure of hardware and operating system software. A set of low-end desktop computers and high-end server computers running different operatins systems typically belong in this layer.

4 ACE Applied

A software engineer should not enforce his favourite structuring methodology to the problems he encounter, agents being no exception. It is important that the software engineer is able (and willing) to identify which of several structuring techniques to apply to a certain structuring problem. Important questions he must ask while architecting a distributed application include:

- Should the computation be performed by clients, by servers or both?
- Should the communication be initiated by the clients or servers?
- Should servers have limited or rich APIs?
- Are the communication channels likely to become bottlenecks in the application?
- Is the client computation base a limited resource?

4.1 When to Use Agents

We have developed numerous prototypes to better understand how to structure distributed applications based on the agent paradigm. We had high hopes initially that we would find structuring problems where agent technology was essential in providing a solution. In perspective, we can conclude that mobile agents are not strictly necessary; it is the aggregated advantage that make mobile agents attractive as a structuring toolkit. Though, we have experienced that there are certain classes of problems that lend themselves naturally to agent computing. In particular, there are two types of problems, one related to performance limitations, the other with how to automate the task of Internet users. The performance problem is characterized by being:

1. *Data-intensive*; large data sets need to be processed.
2. *Remotely located data*; the data set is naturally located at the other end of a network connection. This can be due to practical or ownership aspects, where, for instance, the data set is maintained and sold on demand by an organization for profit purposes.
3. *Specialized needs*; the remote server (API) does not support very personalized client needs efficiently.

The other type of problem is one where polling of remote resources typically needs to be done. A concrete example is an Internet user polling remote publishing or news servers to be able to capture the occurrence of some specific event as soon as possible. The agent alternative is to automate this task by deploying one or several agents close to the servers for local polling. The user does not have to be involved again until a notification message is sent him from the remote agent. We can characterize this type of alarm-based applications where agents are useful as a structuring tool by the following criterias:

1. *Remote data/computation source*; all necessary data can not be transferred to the user for local processing. Physical limitations and ownership issues are involved here.
2. *Expensive/intermittent connectivity*; the network is a limited resource, due to either latency or bandwidth, but connectivity can also be problematic.
3. *Personalized*; different alarms (or combinations of predicates) must be executed.
4. *Responsiveness*; (soft) real-time requirements for an alert.

4.2 Where to Use ACE

The IEEE/ANSI 830-1993 standard [2] suggests that the system architecture should be part of the software requirement process. This is also the experience we have gained from developing agent-centric applications over a period of 8 years. In the following, we will demonstrate how we use the ACE as an initial architecture template for an agent-centric application.

We have developed a remote data mining application based on agent technology. The application domain is satellite image processing. We used raw satellite data and images from Tromsø Satellite Station, where Terabytes of image data are stored. The images have been retrieved from polar orbiting satellites, some of which have been operational for almost 30 years. We built an agent-centric application prototype that can parse through this type of raw data searching for specific objects or patterns. The raw data size for a single image is in the order of 40 Megabyte.

In essence, we wanted to demonstrate that users can perform personalized searches on this huge archive over the Internet. In this application, pulling over the entire data storage for local processing was clearly not an option. We showed empirically that an agent-centric solution clearly outperformed a traditional client/server solution [4].

We started the requirement specification process by identifying the overall functional components needed in this application. The ACE can already be used at this stage in the engineering process by parsing through all the layers to see if it captures aspects applicable for the particular application. The functional components we identified for this application were:

- L8: User Layer: User interface (UI) for controlling the progress of the computation and displaying the result.
- L7: Compose Layer: Compose component (CM) for composing the search agent to be moved to the server.
- L6: Conversion layer: Since we programmed the agent in C, there was no need for converting this representation to another format. Hence, we had no need for any functional components in this layer.
- L5: Mediator Layer: Global scheduling (GS) for load-sharing purposes was

needed. To be able to serve many agents simultaneously, we needed to replicate the processing and storage servers containing the satellite image data. The GS was responsible for optimising deployment of new agents within this server infrastructure.

- L4: Agent Layer: Data mining agent (MA) for processing of satellite data.
- L3: Extension Layer: Monitoring extensions (ME) for monitoring the local load and progress of the agent computation. ME reports status to GS.
- L2: Mobility Layer: A middle-ware system supporting movement of agents. We used a Unix version of TACOMA, TACOMA v.1.2.
- L1: Server Layer: A set of extensible Storage Servers (SS) with APIs for searching through satellite image data.
- L0: A set of data processing and storage computers running HP-UX.

Figure 5 illustrates how these components relate to the ACE. The ovals indicate functional components belonging to the application, and the rectangles indicate the non-functional components. The arrows indicate relative control and communication dependencies between the components. In this example, the arrows indicate this flow of control:

1. The user composes the agent (MA) through an interaction with CM.
2. The agent is sent to the global scheduler (GM).
3. The global scheduler (GM) receives status input from the monitoring extensions (ME), and based on this, determines which host in L0 to send MA to.

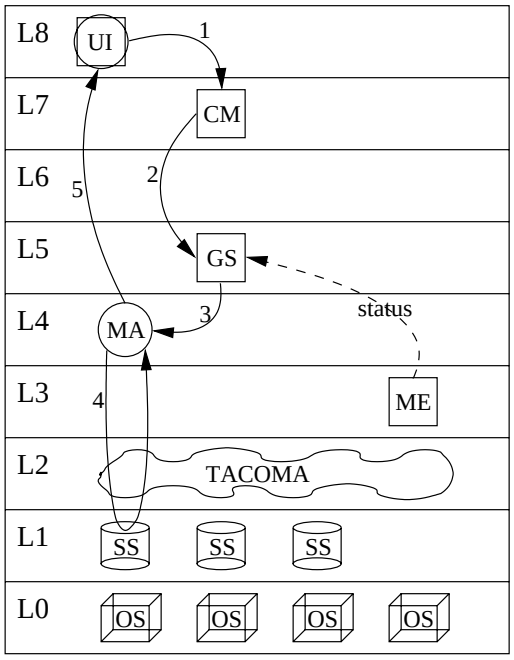

Figure 5: Functional components in an agent-centric data mining application

4. The MA executes on the same host as a storage server (SS) until termination.
5. When MA terminates, it reports the result back to the user (UI).

5 Conclusion

We have derived a generic architecture for agent-centric distributed applications. The ACE has shown to be a convenient structuring template in the early software engineering process. It captures important functional and non-functional components, and the relative control and communication dependencies in this domain.

The ACE represents a step towards understanding how to structure distributed applications with mobile agents, but we do not advocate that this concept should be applied for any distributed application. Hence, we have iden-

tified some general problem characteristics to decide when it is convenient to move an agent close to the server. Typically, the application domain includes aspects as high volume data, where a network can be a bottleneck, and (mobile) users with thin clients.

Acknowledgements

We wish to thank the other members of this joint University of Tromsø, Cornell University and UC San Diego project. In particular, we acknowledge design and implementation work done by Kjetil Jacobsen (alert servers), Nils P. Sudmann (Unix TACOMA), and Michael Susæg (satellite image agents).

References

[1] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical Report RC 19887, IBM T. J. Watson Research Center, 1995.

[2] IEEE. IEEE recommended practice for software requirements specifications. In R. H. Thayer and N. Dorfman, editors, *Software Requirements Engineering*. IEEE Computer Society Press, 1993.

[3] Kjetil Jacobsen and Dag Johansen. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the ACM Symposium on Applied Computing*, pages 399–404, San Antonio, Texas, Feb 1999.

[4] Dag Johansen. Mobile Agent Applicability. In *Proceedings of the Mobile Agents*, pages 80–98, Stuttgart, Germany, Sep 1998.

[5] Dag Johansen, Keith Marzullo, and Kåre J. Lauvset. An Approach towards an Agent Computing Environment. In *The Workshop on Middleware at the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[6] Dag Johansen, Fred B. Schneider, and Robbert van Renesse. What TACOMA Taught Us. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection*. Addison Wesley, 1999.

[7] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, pages 42–45, Orcas Island, Wa, USA, May 1995.

[8] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, Mar 1999.

[9] Nils P. Sudmann and Dag Johansen. TACOMA for UniX (TAX). In *Workshop Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taiwan, 2000.