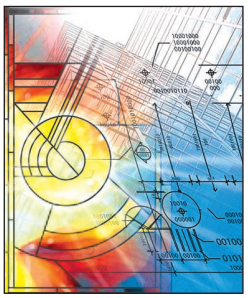
Agents, Grids, and Middleware



Craig W. Thompson • *University of Arkansas*

Agents, grids, and middleware are more closely related than you might think. I am involved in developing the software architecture for two software projects: one involves infrastructure for distributed, pervasive computing; the other involves partitioning huge data sets across data grids consisting of thousands of PCs. On the surface, these projects seem very different, but there are similarities and lessons we can learn from comparing them that have implications for agents, grids, Web services, pervasive computing, and middleware.

Everything is Alive Agent System

The Everything is Alive (EiA) project at the University of Arkansas is developing an agent system for pervasive computing — extending Internet and Web-based agents to communicate with everyday things that, in turn, communicate with each other. In this world, toys play together, pets converse with their owners, vehicles talk to road signs, refrigerators know when items inside expire, and backpacks let you know if you forgot your socks. The EiA thesis is that sensors, actuators, processors, memories, and communications are becoming cheaper and smaller and will soon be everywhere. Indeed, there is much evidence that this is happening already.[1] The EiA project is targeting a lightweight, evolvable system architecture that could potentially be standardized.

EiA Architecture

From the start of the project, we adopted an agent architecture to model large numbers of distributed, autonomous entities. Initially, we prototyped a collection of interesting agents that could interact with people and each other. A homeland security scenario included agents representing rangers, platoon leads, headquarters, vehicles, and various sensors. All agents communicated in XML using a schema based on the standard Foundation for Intelligent Physical Agents (FIPA) Agent Communication Language, which specifies publish and subscribe messages that let agents request and receive periodic updates from other agents. The prototype also included more abstract agents that represented data sources and messaging systems. Implemented as proxy agents, these specialized agents translated messages back and forth between EiA's interagent XML language and legacy interfaces of preexisting systems, so that they appeared as agents to the rest of the EiA system.

In the architecture, EiA messaging supports blind-carbon-copy messages to message-logging agents. Later, the message log could be replayed to simulate agent communication or provide after-action analysis. Finally, a "world agent" simulated the environment and sent agents messages that were ostensibly from sensors or external stimuli. In the initial prototype, agents began life as generic agents, and acquired their roles (for example, as a ranger or sensor), maps, and address books by receiving messages.

Maximal Agents

One problem with our design was that every agent's codebase was maximal — each contained all the code to take on any role, and also contained all the system services (messaging, a GUI to let humans interface with them, filters to control incoming and outgoing messages, and so on). This approach clearly wouldn't scale to hundreds of agent types, nor could the system evolve so that agents could receive additional capabilities at runtime. We had observed a similar problem earlier when trying to characterize agent systems — some agents were mobile, some were intelligent, some needed interfaces to human operators, and the list goes on. Which elements from this long list of

 1089-7801/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

properties are central to making something an agent?

Agents and Plug-ins

We needed some way to provision a baseline agent with capabilities that could be added or removed dynamically. We considered the idea of plug-ins (code modules that could be dynamically loaded) and located a promising approach in the Eclipse project (www.eclipse.org), which is successfully developing editors and integrated development environments. Eclipse plug-ins can define extension points for additional plug-ins. This provided a way to dynamically load code (such as Java .jar files). We developed our own variant of this architecture that used XML as the interface extension definition language. On reflection, we realized we should also make the system compatible with the Web Services Description Language (WSDL).

The restructured architecture now consists of a generic container with a bootstrap communication and interpretation module that can receive plug-ins via messages. We are currently developing the following plug-in services:

- a messaging service that depends, in turn, on sockets, email, and other message-transport plug-in services;
- a GUI consisting of a collection of XForm panels;
- a natural language interface;
- a digital rights management service that defines security and privacy limitations to constrain what agents can say to one another; and
- a licensing service that monitors service usage and micro payments.

We can now develop many other agent capabilities in a modular way. By becoming compatible with WSDL, EiA can now use any Web service defined by any universal description, discovery, and integration (UDDI) registry. We can also choose whether a service should be local (dynamically loaded) or remote. Furthermore, we plan to extend the plug-in architecture to make it possible to define "before" and "after" plug-ins, which will provide implementations for adding new aspects to the existing EiA architecture. If successful, we will remain compatible with the Web services world (including the SOAP, WDSL, and UDDI standards) while generically adding dynamic loading, digital rights, licensing, and other services that agents need.

Data Grids

The second project involves developing an architecture for data grids. Generally, grids involve some kind of sharing, and much of the effort over the past several years has involved sharing computation among large numbers of commodity machines. Much of the grid research community bypassed Interface Description Language (IDL) and Java and went straight to adopting XML as an IDL, recently retargeting WSDL. Meanwhile, some in the database community discovered that commodity PC platforms can be used to store massive data sets. Oracle and Microsoft are developing grid-based relational database-management systems (DBMSs). Our second project involves operating on giant flat files, spread across hundreds of PCs, processed with custom operator algebras. At one level, the architecture provides a pool of grid nodes and mechanisms for allocating a collection of these to a higher-level application; the next-level architecture builds indexing structures using grid nodes; and at a still higher level, operator workflows function in parallel on records to transform, augment, or fuse information sources. Some of the subproblems we're working on involve:

- *Hotspot management*. Grid data nodes report their resource utilization to manager nodes. Automating hotspot detection is desirable for providing automatic recovery.
- *Index creation*. Huge indices are distributed across grid nodes so that a query can access many grid nodes in parallel to return results, and batched streams of records can be inserted in parallel across many grid nodes.
- *Fault tolerance*. Applications need hot backup nodes when grid nodes fail.
- *Workflow automation*. Currently, workflows are manually specified, but we're looking for ways to treat them as goals that a problem solver can refine into optimizable plans.

The current implementation uses a mix of Corba and XML. From the outside, we could view the emergent system as a massive and monolithic database machine, but, from an internal perspective, we can view it as an open, extensible collection of middleware design patterns arranged in the form of a service-oriented architecture, including familiar services such as name services, persistence services, metadata registries, network management services, and security services.

Similarities

At first glance, agent and data grid architectures are very different. But can we learn from one to improve the other?

Consider the data grid — its data and management nodes (as well as other nodes not described) could be viewed as types of agents. If we take this view, we might discover that we could use XML everywhere as the IDL. Some of the services and plug-ins we defined for agents might also work for data grids; for instance, the digital rights service could ensure that customer data in one part of the grid can be combined with another customer's data only if both agree and if the composition does not violate a privacy constraint.

How would the agent system ben-

efit from the data grid? Perhaps it could reuse the fault-tolerant scheme for creating and managing replicas to make individual agents more likely to survive. Additionally, the system could use the management service for a similar purpose, having it identify when agents die or are overloaded. It currently appears plausible that both might use the same workflow plug-in.

Both systems might benefit from common use of XML, SOAP, WSDL, and UDDI, along with a principled use of "aspects" that extend the SOAP family framework.[2] Such aspects could include security, logging, licensing, micro payments, and other services. Though it is the subject for another column, both architectures might benefit from common ways of exposing metadata about the agents and modules so that higher-level policy management engines could operate to control both the aspects and the systems' emergent behaviors.

Lessons Learned

Can we generalize from the observation that we can compare and learn from each architecture and transfer results to the other? We might speculate:

- Agent architectures and data grid architectures appear to be constructed from middleware primitives; perhaps these provide unification for several other architectures as well. If so, we can expect the design patterns that the Object Management Group and other communities discovered to resurface and provide inspiration for the Web services' plumbing toolset. By mining older architectures, this observation provides a quick route to discover the missing capabilities of the current SOAP, WSDL, and UDDI family — demand loading, aspects, policy management, autolicensing, and others.
- The agent, grid, database, and other communities might be missing the opportunity to learn from one another. If the agent community never realizes that their work can be mapped to data grid architectures, we must question whether the agent community is doing a good job of transferring its results to others, for example. Their results could be locked inside the presumption that agents are special and separate from low-level object middleware patterns. If we can instead piggyback agent services onto today's massive deployed infrastructures (such as the Web, Google, or email), we might start seeing scalable agent solutions become massively deployed, rather than being locked inside idiosyncratic agent systems. In some ways, the agent community is approaching a similar kind of hurdle that hypermedia systems were able to top only after the invention and widespread adoption of HTML, HTTP, and browsers. We must overcome this obstacle to scale agent technology to the Web.

At first look, agent and data grid architectures are very different. But can we learn from one to improve the other?

How does this all relate to the Internet and World Wide Web? We can view SOAP, WSDL, and UDDI as key building blocks of the Semantic Web: these emerging standards provide ways for programs, rather than just people, to connect to other programs in a platform-neutral manner to acquire information and perform tasks, independent of human operators. More research will be required (on ontologies and metadata, for instance) for the Web to be semantically comprehensible, but we can predict, based on our prior experiences with OMG and Java middleware, that many additional plumbing services will be added. We also predict that agents and grids will be among the staple capabilities in a future Semantic Web.

Conclusion

Developing applications for the "Internet of things" will not be entirely different than for today's distributed middleware and grid system. It seems straightforward that thermostats, appliances, toys, and vehicles will soon come not only with conventional instructions but also with RFID tags, a WSDL interface, wireless connectivity, and a PDA/PC-compatible GUI controller, plus aspects like security, usage monitoring, policy management, and trouble-shooting. In short, we will see many of the same middleware patterns in both agent-based pervasive computing and large-scale grid system development.

References

1. C. Thompson, "Everything is Alive," *IEEE Internet Computing*, vol. 8, no. 1, Jan/Feb 2004, pp. 83–86.
2. *Aspect-Oriented Software Development*, R.E. Filman et al., eds., Prentice Hall, to appear.

Craig W. Thompson is professor and Acxiom Database Chair in Engineering at the University of Arkansas and president of Object Services and Consulting. His research interests include data engineering, software architectures, middleware, and agent technology. He received his PhD in computer science from the University of Texas at Austin. He is a senior member of the IEEE. Contact him at cwt@uark.edu.