Message Domains as an Architectural Solution to Complexity in Distributed Message-Oriented Systems

Aurélio Akira Mello Matsui and Hitoshi Aida

***Abstract*—Effectiveness of message passing as the means to provide communication in distributed computing motivates the use of such strategy in highly distributed service environments such as grid and cloud computing. Grids and clouds share some important features such as multiplicity of administrative domains and complexity to create applications to be executed over these platforms. In this paper we argue that message domains can provide architectural simplicity to split distributed applications into layers. Interdependence between layers is controlled by visibility policies in each message domain. This architecture allows for the creation of infrastructures of distributed agents in which applications can be modular.**

***Index Terms*—Distributed system, architecture, message-oriented system, component pluggability**

I. Introduction

Highly distributed computing models such as grid and cloud computing are becoming popular as companies and academic institutions start to realize the benefits of computational resources virtualization, platform as a service, and systems as a service.

But computation as a commodity comes at a price. As systems scale, so the complexity to keep them working and evolving increases. Popular systems tend to have a large number of customers, therefore, developers of those systems are expected to receive constant requests for enhancements.

Although message passing is largely used to create distributed applications in a more controlled scenario, the principles of message passing can be also applied to grids and clouds. Therefore, we need to adapt system distribution to address the new questions introduced by the global scale.

When computational resources are dynamically provisioned, the exchanged messages are not only about data and execution control, but may be also about resource finding, reservation, and payment. When nodes are separated by continents, multicasting must be used wisely. If public networks are utilized to transport the messages, extra care should be taken with security. If more than one organization is exchanging messages in the same grid, message exchange within nodes that belong to the same organization should be isolated from messages that provide communication between organizations. As customer expectations increase due to a high offer of remote services, so does the struggle of development teams to keep the costs of software development and software change under control.

To address these problems, we advocate that global scale distributed systems should be based on reliable multicast domains. Those domains should be flexible to allow for application designers to create custom configurations that will support decisions over module partitioning, user roles, secrecy of data, and division of concerns into independent layers.

The next section will present our argument for the adoption of multicast domains to address those issues. Section III presents an example of message domains. Section IV presents some details of our experience to implement the message domains using JMS. Section V presents related research. Finally, we present our conclusions in Section VI.

II. Message Domains

We propose that message multicasting should be partitioned into independent message domains, and each domain should have its own secutiry policies. Within a message domain, all connected software agents are able to send and receive both multicast and point to point messages to and from all other agents. Multicast is necessary to avoid central points of failure and performance bottlenecks. On the other hand, multicast coverage should be controlled in grids and clouds since we can expect messages to cross administrative domains, we may have multiple service implementations, and service instances.

If, on one hand, limiting messages to specific domains may force the existence of bridges between domains, on the other hand, this separation makes it possible for system architects to partition distributed systems horizontally, into stacked layers, and vertically within each layer, into subdivisions that represent distinct roles of software components.

Table I shows our proposed model. Message domains are table rows and roles are table columns. The table cell that is the crossing between a row X and a column Y will contain the read and write permissions for the agent of column Y in the message domain of row X.

Table cells are grouped in accordance with the function. Cells with the same function belong to the same region in the table. Regions are identified with Greek the letters α, β, γ, δ, and ε. Function γ is subdivided into γ_1, γ_2, ..., γ_N. We will explain this subdivision later.

Message domains are separated into two large groups. The upper part is the application layers and the lower part is the infrastructure layers.

A. Application Layers

Each of the *N* application layers App.1, App.2, ..., *App.N* represents a single distributed application that can be executed by the distributed resources of the grid or cloud.

Manuscript received August 15, 2012; revised October 12, 2012.

The authors are with the Electrical Engineering and Information Systems, University of Tokyo, Japan (e-mail: akira@aida.t.utokyo.ac.jp, aida@ee.t.utokyo.ac.jp)

Each application layer *App.i* is subdivided into M_i message domains *A.i.1, A.i.2*, ..., *A.i.*M_i. An application may be partitioned into more than one message domain to organize its agents. For instance, certain agents may interact with clients, while other agents may be hidden from clients and perform internal tasks to process requests.

TABLE I: ARCHITECTURE TO MESSAGE EXCHANGE COORDINATION. (α) ENTRY POINT; (β) REQUIREMENTS; (γ) APPLICATION ORGANIZATION; (δ) SUPPORTING INFRASTRUCTURE ORGANIZATION; (ε) INFRASTRUCTURE SERVICES

				Infra. agents			Application-specific agents						
	Layer	domainMessage	client				App.1			...	App.N		
				i_1	...	i_R	$a_{1,1}$	...	$a_{1,S1}$	...	$a_{N,1}$	…	$a_{N,SN}$
Application Layers	App.1	A.1.1	α	β			γ_1						
		A.1.2											
		⋮											
		A.1.M_1											
	⋮	⋮											
	App.N	A.N.1									γ_N		
		A.N.2											
		⋮											
		A.N.M_N											
Infrastructure Layers	Infra.1	I.1.1		δ			ε						
		I.1.2											
		⋮											
		I.1.Q_1											
	⋮	⋮											
	Infra.P	I.P.1											
		I.P.2											
		⋮											
		I.P.Q_p											

B. Infrastructure layers

Similarly, each of the P infrastructure layers Infra.1, Infra.2, ..., Infra.P is subdivided into message domains. An infrastructure layer Infra.i is subdivided into QP message domains I.i.1, I.i.2, …, I.i.QP.

C. Roles

Columns are of three different kinds of roles: clients, infrastructure agents, and application agents.

A client requests services and can be an application connected to the distributed services infrastructure, or an agent that belongs to one of the applications of the grid or cloud. For instance, an agent that belongs to a virtual shop application may play the role of client of a billing application.

Infrastructure agents provide base services to the applications. Such services are similar to what middlewares provide. Logging, service directory, authorization, and certification are examples of services provided by the infrastructure agents.

Finally, application agents are components of the applications. For several reasons (software organization, modularity, security, secrecy, among others), an application may be composed of several kinds of agents.

D. Functions

1) Region α - entry point

Represents the client-service exchange function. Message domains in this partition are interfaces with service clients.

Infrastructure agents can be used by distributed applications as a distributed middleware container if the application does not allow clients to connect in *α* and if the application delegates the interface with clients to the infrastructure layers.

2) Region β - requirements

Represents what the infrastructure demands from the applications. The supporting infrastructure can use this partition to force applications to comply with a certain set of requirements. For example, it may be required that all message domains in the application layers should allow a monitoring agent to listen to the multicast messages. Obviously, this requirement does not guarantee that the monitoring agent will be able to understand the traffic in the application layers. Therefore, requirements declared in this partition should be considered only communication capability requirements, not semantic requirements.

3) Region γ - application organization

Applications may require the existence of private message buses to work. Message domains in this partition aim to provide this sort of intra-application message exchange. Direct communication between applications in the *γ* region is considered a design flaw, since it creates internal dependencies between applications. If one application needs to utilize another as a service, this communication should occur in an entry point *α*.

As the internal functioning of applications should be not exposed to neither other applications nor to the infrastructure, each application A.i has its own private sub-partition γ_i inside of *γ*, and sub-partitions do not overlap.

4) Region δ - supporting infrastructure organization

Represents the internal organization of the infrastructure agents. It is the similar to the *γ* region for the infrastructure.

The infrastructure should provide a platform to the applications, but while doing so, the infrastructure should also hide its own complexity from the applications.

It is common that infrastructure layers have intrinsic relations with other infrastructure layers. This is the case, for example, of the dependencies between a resource brokerage agent and a monitoring agent. Resource brokers need to utilize data from monitoring agents to select resources.

In contrast with γ, infrastructure layers are allowed to directly communicate (and therefore have dependencies) in δ. Modularity in the infrastructure layers can be harder to achieve than modularity in the application layers. This is because the component elements of the infrastructure layers need to work in coordination, whereas applications can be independent. Agents in the infrastructure layer have no underlying layer and therefore two infrastructure agents cannot utilize bridges outside of the infrastructure layer to communicate or to coordinate their work. In other words, we cannot organize the δ function utilizing a structure like the γ function. The result would be no coordination in the distributed infrastructure.

Therefore, the problem we face is at the same time ensuring that the infrastructure agents can work in cooperation and provide means to create pluggable infrastructure layers.

Our proposal is the creation of a coordination infrastructure layer, as shown in Table II. The regions δ_1, δ_2, …, δ_P isolate the infrastructure layers Infra.1, Infra.2, …, Infra.P the same way the application layers are isolated in the γ region. In the architecture shown in Table II, coordination is performed by the agents $i_{C,1}$, $i_{C,1}$, ..., $i_{C,R}$. Coordination is performed utilizing the δ_C region.

TABLE II: Alternative Organization of the δ Region to Allow for Infrastructure layers Infra.1, Infra.2,... , Infra.P to be Independent. the δ_C Region is Utilized to the Coordination of the Infrastructure Layers.

			i1,1	i1,2	...	i1,R1	...	i2,1	i2,2	...	i2,R2	iC,1	iC,2	...	iC,Rc
Infrasructure layers	Infra.1	I.1.1	δ_1												
		I.1.2													
		I.1.Q_1													
	⋮	⋮													
	Infra.P	I.P.1					δ_P								
		I.P.2													
		...													
		I.P.Q_P													
	Infra. Coord	I.C.1	δ_C												
		I.C.2													
		...													
		I.C.Q_C													

Then, a modular infrastructure layer and an application layer may look the same if we observe only that they have private message exchange areas and communicate with a lower layer. A modular infrastructure layer differs from an application layer in that an infrastructure layer may have requirements to application layers and not the other way around. Another difference is that the structure of the infrastructure is established to create the framework over which the applications will be developed.

5) *Region ε - Infrastructure services*

Represents a partition for the supporting infrastructure to provide services to the applications. Designers of application layers are free to utilize those services or not.

III. An Example of Architecture Using Message Domains

Table III shows an example of a message domain partition to support two distributed applications (a data mining application and a sensor data collection application) and some accessory services (service market, resource location, monitoring, and authentication). This partition does not define service cardinality, resource location, neither whether the services are written, provided, or maintained by a single or by several organizations.

TABLE III: Example of a message domain structure.

Layer	Section	a	b	c	d	e	f	g
Data mining app.	client	rw		r		rw		
	Data bus			r		rw	rw	
	Data sync			r			rw	
Sensor data app.	Collection	rw		r				rw
	Data sync			r				rw
Service market			rw	r		rw		rw
Resource brokerage	Location	rw	rw	r				
	Availability		rw	rw				
Monitorin g			w	r	w	w	w	
Authentica tion	consumer	w				w		w
	provider	r				r		r

Lines are message domains and columns are software agents. The agents are: (a) service consumer; (b) resource broker; (c) monitoring agent; (d) certificate server; (e) data mining service; (f) database; (g) sensor. Access to message domains are marked as r for read, and w for write. The symbol means that the message domain does not provide access to the agent since access in these cases are considered an architectural error.

Each line represents a message domain. Participants of the same message domain should exchange messages freely, therefore ideally they should share the same protocol. Conversely, different domains could adopt different protocols and translation may be necessary.

Each layer is supposed to offer a specific service to the stack. To organize its internal structure and to protect message buses that should be private to its components, layers may also be divided into sections. For example, in Table III, service consumers (a) cannot access the availability section, since this section should be used only for resource brokers and monitoring agents to communicate.

The resource brokerage layer is used to support for replicas of the same service and dynamic service location. Resource brokers are commonly utilized to optimize access to resources in grids [1]. If, for the sake of high availability or to optimize performance, a set of resource brokers is used (as in [1]), these brokers need to keep a database of resources synchronized among all of them. As resource availability in a grid is supposed to be dynamic and unpredictable, this resource database is also supposed to change frequently.

Message traffic used to keep synchronization of the distributed resource database should be kept away from non-resource broker agents to minimize network traffic and to simplify development of agents. Agents that do not perform resource brokerage will not need to reject message brokerage messages, as they simply do not arrive.

Also, data about resources should be kept among authorized hosts only, as paid service providers could use access to those messages to cheat in the competition for service consumers.

In this example, the design decision was that distributed databases do not need to verify the identity of consumers, since databases are only reachable through the data bus section. Also, it was decided that it is not a threat to secrecy or security if all data mining service instances receive data from databases. These assumptions reflect into simpler source code for the database agents.

Structure of processes can influence the design of those domains. For instance, participating in service bidding may be a complex task as it involves several steps until the negotiation is done. Service consumers should not have to know how to participate in service bidding since having to do so would create a dependency between consumers and resource providers. So instead of having direct access to the service market layer, a service consumer needs to indirectly utilize the service market via communication with a resource broker. A resource broker will then intermediate the negotiation and use the service market message domain to exchange messages with the data mining services and the sensors.

From the view point of the data mining service (e), it is necessary to participate on six message domains. If data received from one of the message domains needs to be fed into another message domain, and the two message domains do not use the same protocol, translation may be necessary.

In this example, the infrastructure layers are too connected to allow for the layers to be independent, as in Table II.

IV. Implementation

We created a test implementation for this architecture and found that it fits well to the Java Messaging System (JMS) model. JMS is not a message exchange specification, but an API specification. Nevertheless, the JMS API requires implementations to follow a standard that allowed us to implement our model easily. We utilized JMS message topics, which deliver messages in real time, in contrast with message queues which is an asynchronous publish/subscribe method.

JMS topics control read and write capabilities of each message client independently, which fits to our model. Also, message delivery optimization can be done by configuring how JMS will route and filter messages.

We tested our implementation utilizing the ActiveMQ [2] message exchange tool, version 5.2.0. The reason for us to utilize ActiveMQ was because it allows for the use of several topologies according with configuration both in the client and in the server side. One can build a message exchange system utilizing a centralized messaging server, a set of non-centralized message servers working in cooperation, or even reliable multicast between clients, which avoids the use of any messaging server. Unit testing can also be performed in a single instance of a JVM utilizing an in-memory message exchange driver.

The main problem with ActiveMQ is the lack of automated visibility control in real time. Our infrastructure layers could not create new users or domains in real time, with ActiveMQ. To solve this problem we had to implement our own login module. As we tried to model authorization as a service of the infrastructure (to allow the login server to be located anywhere and have several replicas of the service for high availability), our login module had to communicate with an agent connected to the JMS server. The best implementation we could find had the login module connecting to the JMS itself as a client and forwarding the authentication request to an agent.

This implementation works for our tests, but is far from ideal, as it causes the login module to be called twice per session: once for the client to connect to the JMS server, and another time for the login module to connect to the JMS server and request the actual authorization to the authorization agent.

We expect the same problem of message exchange management to occur in other implementations that are based on JMS. This is because JMS iftself does not specify how the topology and authorizations of the message topics should be managed.

JMS allows clients to declare message filters when they connect to a JMS server. After connection, the server will utilize the filter provided by the client to avoid sending messages that the client does not want to receive. The limitation of this model is that the server cannot create the filters itself. Filter management in the server side could have helped us implement the message domains utilizing a single message topic.

Also, safe point-to-point message delivery was not supported by the API specification. All agents connected to the same message domain where in practice sending all the messages as multicast. Single destination for messages must be implemented utilizing public key cryptography on top of JMS, for instance.

V. Related Works

In global-scale environments, it is expected that message senders and message receivers may be two pieces of software developed and/or maintained by different teams. Service-orientation is used as a solution to decouple dependent software components, but wrapping message passing in service calls adds an abstraction whose consequence is the impoverishment of distributed algorithms.

Although in the past some versions of MPI to grid computing were created [3]–[5], the objective of these libraries is to make message passing programming style available to grid computing, which allows for the construction of distributed parallel virtual machines, instead of an infrastructure of services.

Here, we assume that a library for message passing is already available. Instead of a version of a message passing interface to grid, we propose an additional control over the messages that are transferred by the grid infrastructure. This control can be used to enhance component organization in development using an underlying message passing mechanism.

In a previous paper [6] we investigated some methods to translate local object-oriented programming (OOP), into grid-enabled local applications. During translation, OOP statements are translated into routines that utilize message passing to data transfer and execution control. We expect to apply message domain partitioning to define visibility between software components in OOP. For instance, a service that is not visible to application layers should be represented by OOP classes that are not accessible by application classes. This cohesion between message exchange framework and programming constructs can be ensured by applying special pre-processors in a general purpose OOP language or by using a domain-specific programming language created for this purpose.

The configuration for message exchange we presented should work in cooperation with, or be part of a infrastructure for governance, as discussed in [7]. For example, policies that control access to resources or services can be reinforced by creating classes of clients. Each class could be represented by a column in the entry point partition. The model in [7] also suggests that tools should be used to retrieve meta-information about service infrastructures, which can be implemented using directory services such as UDDI [8]. In addition, the model we propose can also benefit the development of services similar to those currently in use in grid computing, such as the grid resource allocation management (GRAM) [9].

The idea to structurally decouple clients from grid complexity was discussed in [10]. Their approach is based on client source code and transformations that can connect or disconnect it to a grid environment. We believe that the visibility control provided by the message domains can be utilized to generate local stubs for this sort of pluggability. While the application layers offer remote service classes, the infrastructure layer will define non-functional requirements for grid clients.

VI. Conclusions

The structure of grid and cloud computing requires complex message deliver systems in which message recipients should be carefully selected to provide scalability and protection against data stealing, abusive behavior, and attacks. Nevertheless, message passing can be utilized in highly distributed systems, if correct adaptations are made to ensure that it supports dynamic and unpredictable server availability, long latency, and cross-administrative domain issues.

We propose a system organization principle in which first the outline of a distributed system architecture is created using message domains (e.g. by a consortium that decides how the grid will be managed). Next, infrastructure layers can be designed and implemented to support applications. Finally, this structure will be utilized to free applications from having to implement base services.

From the perspective of network usage, message domain partitioning can control complexity and can be used to minimize network traffic. Also, partitioning multicast message domains can simplify the design of service-oriented applications since they can be used to guide data flow. For these reasons, we argue that message passing partitioning should be a fundamental building block of grid and cloud computing environments.

References

[1] C. Dumitrescu, I. Raicu, and I. Foster, "A distributed approach to grid resource brokering," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pp. 38, Washington, DC, USA, 2005. IEEE Computer Society, 2005

[2] Active MQ. [Online]. Available: http://activemq.apache.org.

[3] I. Foster and N. T. Karonis, "A grid-enabled MPI: Message passing in heterogeneous distributed computing systems," ACM Press, 1998.

[4] C. Clauss, S. Lankes, and T. Bemmerl, "Design and implementation of a service-integrated session layer for efficient message passing in grid computing environments," *Parallel and Distributed Computing, International Symposium on*, pp. 393–400, 2008.

[5] Grid MPI. [Online]. Available: http://uddi.xml.org/

[6] A. A. M. Matsui and H. Aida, "Refinements to task control in the aspect-oriented programming model for computing grids," in *Proceedings IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2009.

[7] P. Derler and R. Weinreich, "Models and tools for SOA governance," *TEAA,* pp. 112–126, 2006.

[8] UDDI. [Online]. Available: http://www.gridmpi.org/

[9] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-allocation," in *Proceedings of the International Workshop on Quality of Service*, pp. 27–36, 1999.

[10] J. L. Sobral. "Pluggable grid services," in *Proceedings of the 8th IEEE/ACM international Conference on Grid Computing (September 19 - 21, 2007). International Conference on Grid Computing. IEEE Computer Society,* Washington, 2007.