

# Dynamism Gloom and Doom?

T. Gamble D. Flagg R. Baird R. Gamble<sup>1</sup>

Software Engineering and Architecture Team  
Department of Mathematical and Computer Sciences  
The University of Tulsa  
600 South College Avenue  
Tulsa, OK 74104 USA  
{gamble}@utulsa.edu

**Abstract.** In the areas of application integration, mobility, and agent technology, dynamism is an increasingly desirable quality. However, deploying dynamism is problematic due to constraints on and limitations of the applications and middleware used in such distributed systems. In order to realize dynamism in these contexts, it must be well defined and it must work with available features in middleware. The application components, objects, and agents involved must themselves be equipped to perform dynamically. In this paper, we outline issues for incorporating dynamism into application integrations that rely on middleware.

## 1 Introduction

Dynamism is a key aspect of application integration design and development. When COTS products are integrated, commercially-supported middleware generally forms the backbone of the system. With respect to the dynamic manipulation of components, developers are limited to not only the features the middleware provides, but also to those features that translate into functionality that is understandable, usable, and reliable within the context of the integration. Conversely, when expecting COTS products to exhibit dynamic behavior, the component is likewise constrained by its communication protocols, service descriptions, management policies, and data types. These problems are not lesser issues by any means. In fact, these constraints may prohibit the component from interacting dynamically without augmentation.

Thus, the middleware must be adaptive (something that commercial vendors have not yet delivered), the component must be adaptive (difficult for COTS components), or the connector providing the interaction must provide the capability (a more likely scenario). For true dynamism, connector properties should not just be adaptable, but automatically adaptable. However, in this evolving area there is, as yet, no set of best practices.

For this paper, we focus on the pertinent features of middleware that COTS systems must accommodate. We examine three commercially-supported frameworks within the middleware “genres” of CORBA, Message Queuing, and the Grid. Specifically, we examine an implementation of each of these frameworks, all of which are widely used: Borland VisiBroker [14], IBM Websphere MQ [13], and the Globus Toolkit [17].

---

<sup>1</sup> This work is supported in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320).

## **2 Where Dynamism is Found**

Dynamism represents system change, mainly during run-time. The term “dynamic” can imply changes to the network topology, processor mobility, participating client processes, processor and network failures, and timing variations [1]. Key aspects of dynamism are how much change can occur, how much is known before the change, and both where and how the change is controlled. Researchers approach the accommodation of change by restricting it, planning ahead, and/or implementing predetermined change management solutions wherever possible. True dynamism, though, should allow for limited stored and predefined information by the component or the middleware.

When one or more of the components involved in an integration activity changes, it affects the architecture of the system, causing a dynamic reconfiguration. Many solutions include describing a configuration manager, which assures that no change occurs that is not already specified. Using a predetermined set of allowable state changes eases the task of assuring that the system remains in an architecturally permissible state [2, 3, 4]. These predefined constraints on the types of change provide for the dynamic creation and deletion of processes and clients [5].

Connectors have been designed to facilitate reconfiguration. By specializing components and providing a way for them to manage the changes to incoming and outgoing information through a connector, components and connectors can be swapped to accommodate those changes. The connectors are context-aware so they can provide the interaction functionality the components need as they change [6, 7].

For web services, dynamism often incorporates aspects of discovery, binding, composition, and configuration. User requirements may be leveraged to predetermine service inclusion. Abstractions can be inserted into a predefined configuration to be bound at runtime to a service in order to satisfy the constraints [8, 9]. Here, the components have some context-awareness to provide services dynamically [10]. It is recognized that some form of adaptive ability is needed within the component, its connector, or the middleware to support dynamic web services [11]. One goal is that service-oriented integration would permit the automatic composition of services given a client’s description of the needed tasks and available components to provide the services [12].

## **3 Is Middleware Prepared to be Dynamic?**

Modern middleware systems span many technologies and programming environments. Some, such as IBM’s WebSphere MQ message queuing system, have been on the market for many years and have matured through a number of development iterations. Others, like VisiBroker, are a blend of technologies from middleware standards such as CORBA [15] and Sun’s J2EE framework [16]. We now see a class of newer middleware for the Internet to support grid computing. The most prominent example is the Globus Toolkit. In its current incarnation (version 3 of the toolkit), Globus has merged a number of

characteristics from earlier Globus versions that focused on distributed high performance computing with the emerging standards for Web Services [18]. Having examined these middleware products, we outline certain fundamental features that impact dynamism (Table 1). These are registration, data typing, asynchrony, and security. It is not a comprehensive set of features, but it shows that COTS components in dynamic integrations have a responsibility for connection and further operation. This is especially true if the component participates in multiple middleware frameworks simultaneously.

*Registration* is the act of the component and the middleware becoming aware of each other. Registration is needed for interacting components to reliably identify one another, generally in the form of establishing a representation of a remote computer with a local context for later reference. While most middleware frameworks focus this activity on the component (whether a client or a service) establishing itself with the middleware, it can be a bilateral process whereby the component also retains information identifying the middleware. This is usually required prior to further interaction between the two entities. It may involve inclusion in a directory service, establishment of a programmatic reference, publication of an interface specification, or another mechanism that binds identifying information for later use. To facilitate dynamic behavior, registration must support binding and rebinding at runtime.

There are a number of implications for dynamism based on how, or if, registration is performed and its scope or depth. For example, IBM's WebSphere MQ<sup>2</sup> requires a large amount of information about the communication channels between servers, the queues residing on those channels, and the queue managers handling message delivery across queues. Applications reading or writing queues need only identify themselves in a more limited fashion. When using the MQ publish-subscribe interface, components subscribe to topics of interest and then receive messages only when other components publish on those topics. Components also specify how they should be alerted when a new event appears on the topic "channel" to which they subscribed. In this case, registration is a two step process involving subscription and event management. This behavior is actually similar to the other frameworks when using asynchronous interfaces. For example, in the current incarnation of Globus, grid services are registered by creating and transmitting a Web Services Description Language (WSDL) document specifying their service definition.

We define *data typing* as the collection of capabilities for type definition. It is required for enforcement of type safety across component interactions and is particularly important when interactions are only loosely defined, if at all, prior to runtime, as is the case with all dynamic systems. Facilitating this must be a function of the middleware or must occur at the integration endpoints, either in the component itself or in a connector to the middleware. Thus, the particular realization of these capabilities depends largely on the programming model that is assumed within the component-level interfaces to middleware. VisiBroker uses CORBA's interface definition language (IDL) to register types for interfaces. Delayed binding of typing is supported via the "ANY" type in IDL. The

---

<sup>2</sup> For brevity, we henceforth refer to this product as simply MQ.

Globus Toolkit is increasingly dependent on Java and J2EE run-time features as well as Web Services standards for interface specification. The use of WSDL as a definition mechanism is becoming the predominate method for interface definition. These type definitions must be submitted when services register in Globus. MQ has few data typing facilities and treats data payloads in messages as simple strings. There is limited matching done between publish-subscribe channels based on topic names.

*Asynchrony* is a middleware feature that is based on the context of interactions between components. This style of interaction must allow the components to communicate but operate independently, in a loosely coupled fashion. Loose coupling reduces dependencies between components and makes changes due to dynamic behavior easier to manage. This normally means a component sends a message to another component and expects a reply, but doesn't know when that reply will come and, more importantly, it does not wait for it. Most middleware frameworks that allow asynchronous behavior provide mechanisms for message transmissions to be reliable, requiring an implementation of persistence in the form of queues. If the model of message delivery is event-based, then the receiving components must be prepared to process the incoming events.

Within MQ, asynchronous behavior is inherent. There are no specific mechanisms for clients and servers to rendezvous. For VisiBroker, components may implement the CORBA Asynchronous Method Invocations (AMI) to support polling and callback of services by clients. This requires the *a priori* agreement of both client stubs and server skeletons. In Globus, asynchronous behavior is largely centered on mechanisms for batch processing, which reflect Globus' heritage in scientific data processing. Clients asynchronously submit batch *jobs* and may be alerted via *notifications* when batch jobs complete. In all cases, it is important to note that the components' interaction with the asynchronous features of middleware must provide appropriate behavior for that programming model. If they cannot, then some form of explicit mediation is required.

When looking at *security* in integration, there are many considerations. For the most part, these involve authentication, confidentiality, integrity, and non-repudiation. Dynamism results in change, which normally translates into added risks for secure systems. We focus on confidentiality and integrity, where the middleware provides them at either the transport level or the message level. For transport level security, the middleware need only assure that the overall connection and all data that travels over it be secured. In many cases, this means encrypting data along a communications channel between instances of the middleware on multiple operating systems.

The use of secured channels between middleware server processes is a common behavior across the middleware products in Table 1. MQ does transport encryption in the processes that implement channel endpoints. VisiBroker uses the broker implementations deployed to the operating systems where CORBA clients and/or servers reside. Globus has a "gatekeeper" process that is spawned in the local OS (operating system) to handle connection management for components. In each case, there is actually a physical process instance associated with one or more components running within the context of the same OS. This process represents a connector provided by the middleware that the component

**Table 1.** Middleware Features Impacting Dynamism

<b>Feature</b>	<b>IBM MQ</b>	<b>Borland VisiBroker</b>	<b>Globus Grid Toolkit</b>
<b>Registration</b>	Identification of queues and pub/sub topics. Establishment of callbacks or polling mechanisms for event processing.	Components must specifically state what is transferred and which services they offer or require, respectively. Registration requests are processed by local instances of broker servers.	A service component submits a WSDL document that describes its services and what is transferred, while clients submit a WSDL document in order to submit jobs.
<b>Data types</b>	There is no explicit checking of data types, as message payloads consist of raw bytes.	Registration includes checks to make sure the information to be transferred between components matches predefined data types.	Registration checks to make sure the information transferred between components matches predefined data types.
<b>Asynchrony</b>	Communication is inherently asynchronous.	Communication is inherently synchronous. Components may use supplied connectors to support polling and callback of services.	Clients can submit and be asynchronously notified by the services when processing has completed.
<b>Security (transport level)</b>	Communication channels between queue managers may be encrypted.	SSL is used between ORB instances on separate host environments.	A private protocol (httpg) is implemented between Globus containers (aka servers) on remote machines.
<b>Security (message level)</b>	No inherent capability.	Encryption is at the transport level, but message level integrity may be use checksums.	The message level security is based on WS-Security, XML Encryption and XML Signature standards using either a pre-established security or credentials supplied inside the calling parameters.

must use. The connector is context-aware in the sense that it supports the programming model(s) available on the OS. Communication between the component and the middleware through that connector is often not considered to be within the scope of transport concerns. Thus, it is not explicitly secured by middleware features and any dynamic connectors could cause a security breach.

Rather than protecting all interactions across a secure channel, the middleware may allow each component to component interaction to be selectively secured. For MQ, message level security isn't included. User programs may do their own encryption if they have consistent logic with peer programs. VisiBroker provides for limited message level security in the form of integrity checking via secure checksums. For encryption, only transport level is supported. Globus tends toward a potpourri of implementation choices,

and the security features it provides are no exception. Message level security in Globus is similarly influenced by Web Services standards and is based on WS-Security, XML Encryption and XML Signature standards.

## 4 Conclusions

Commercially-supported middleware still has many difficulties with dynamism. Features central to allowing basic interaction are often at odds with dynamism. Much *a priori* knowledge is needed to support interconnection. Also, middleware sets expectations, (e.g., code customization and interface interoperability), which are impractical for most dynamic COTS applications. More research is required to define automated, adaptable connectors and marry them with the architectural analysis to assure correct run-time behavior.

## References

1. N. Lynch, A. Shvartsman, Communication and Data Sharing for Dynamic Distributed Systems, *Future Directions in Distributed Computing*, LNCS, Vol. 2584, 2003.
2. R. Allen, R. Douence, D. Garlan, Specifying Dynamism in Software Architectures, *Proceedings of Foundations of Component-Based Systems Workshop* (Sept. 97).
3. E. L. White, "General strategies for dynamic reconfiguration," *ACM SIGSOFT Software Engineering Notes*, Volume 25, Issue 1 (January 2000) pg. 93.
4. M. Wermelingerl, J. L. Fiadeiro, Algebraic Software Architecture Reconfiguration, *FSE'99*.
5. R. Allen, R. Douence, D. Garlan, Specifying Dynamism in Software Architectures, *Proceedings of Foundations of Component-Based Systems Workshop* (Sept. 97).
6. M. Mikic-Rakic, N. Medvidovic, Adaptable Architectural Middleware for Programming-in-the-Small-and-Many, *ACM/IFIP/USENIX Int'l Middleware Conference*, June 2003.
7. M. Mikic-Rakic N. Medvidovic, A Connector-Aware Middleware for Distributed Deployment and Mobility, *ICDCS Workshop on Mobile Computing Middleware* (MCM03). May 2003.
8. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Z. Sheng, Quality Driven Web Services Composition, *WWW2003*, May 20–24, 2003.
9. C. Ferris, J. Farrell, What Are Web Services? *CACM*. June 2003/Vol. 46, No. 6.
10. A. Arsanjani, B. Hailpern, J. Martin, P. Tarr, Web Services: Promises and Compromises, *Queue*, March 2003.
11. G. Piccinelli, W. Emmerich, C. Zirpins, K. Schütt, Web Service Interfaces for Interorganisational Business Processes, (EDOC2002).
12. M. Turner, D. Budgen, P. Brereton, Turning Software into Service, *IEEE Computer*, 2003.
13. <http://www.ibm.com/software/integration/mqfamily/library/manualsa/>.
14. <http://www.borland.com/besVisiBroker/>.
15. [http://www.omg.org/technology/documents/formal/corba\\_2.htm](http://www.omg.org/technology/documents/formal/corba_2.htm).
16. <http://java.sun.com/j2ee/docs.html>.
17. <http://www-unix.globus.org/toolkit/documentation.html>.