COMPONENT ARCHITECTURE
THE SOFTWARE ARCHITECTURE FOR MISSION REQUIREMENTS

Thomas Huang

Space Science Data Systems Section
Jet Propulsion Laboratory, NASA
Pasadena, CA 91109, USA
Thomas.Huang@jpl.nasa.gov

ABSTRACT

Software reuse is a common strategy in developing complex systems and has proven successful in reducing labor and maintenance costs. However, simply reusing modules will not produce a system that is adaptable to a variety of mission requirements. Because of this, projects often involve development of similar software systems from scratch in order to satisfy requirements. The end result is a system that can only operate in a specific environment and be used only in a specific way, with consequentially higher costs for maintenance and user training.

Component architecture consists of a framework that defines the standard interactions between components and standard interfaces for useful components to attach to the framework and interact with other components. Modern object-oriented programming languages are very good in their support for static interfaces, but need additional work in the area of dynamic interfaces. Reflection, which is available in some OO languages, should be considered in developing model component systems to enable dynamic discovery of service components at runtime. This enables software systems to be assembled at deployment time and provide users the ability to customize the software system with respect to their operating environment.

Our File Exchange Interface (FEI) is af ile transaction service that offers portable, high performance, database-driven file management and transfer service. Unlike the common File Transfer Protocol (FTP), FEI provides file integrity verification on the fly, user authentication and authorization support, and database transaction management. FEI played a major role in file archiving and delivery service in flight missions such as Galileo, Mars Pathfinder, Deep Space 1, Cassini, and Space Infrared Telescope Facility. The new FEI version 5, code named Komodo, is a component-based service to enable pluggable support for various mission security requirements, database repositories, communication protocols, concurrency model, and file systems.

This paper presents the challenges in developing a dynamic service such as FEI to support various mission requirements while still being able to reduce cost on maintenance without sacrificing reliability and performance.

Keywords: Component, Component Configurator, Software Product Lines, Design Patterns, Reflection, Framework, Database.

1. INTRODUCTION

Despite dramatic increases in network and desktop computer performance, it remains difficult to design, implement, and reuse communication software for complex distributed systems. As the world's eyes and ears to the unknown frontier, the Multimission Image Processing System (MIPS) at JPL is expected to be able to accurately process all live science data gathered by spacecraft and distribute the processed data products to the science communities with respect to stringent quality-of-service (QoS) requirements. The image-processing framework, shown in Figure 1, consists of intelligent business components that perform acquisition and processing of telemetry data, cataloging of data products and onboard instrument states, visual verification and monitoring, science data processing, and distribution to subscribing science communities. While the framework defined the system's core services, each mission has its own set of requirements. These requirements may specify the method of telemetry data acquisition, visualization interface (if any), where and when data product distribution occurs, and most importantly of all the QoS

requirements. To satisfy these mission-specific requirements, software engineers customize each service component within the framework while maintaining interoperability with the reset of the services. The image-processing framework is really a Software Product Line (SPL) [14] where mission data is being processing with software systems that are built from the core business components with added mission-specific characteristics. SPL, a concept first formalized by the Software Engineering Institute (SEI), defines a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [14].

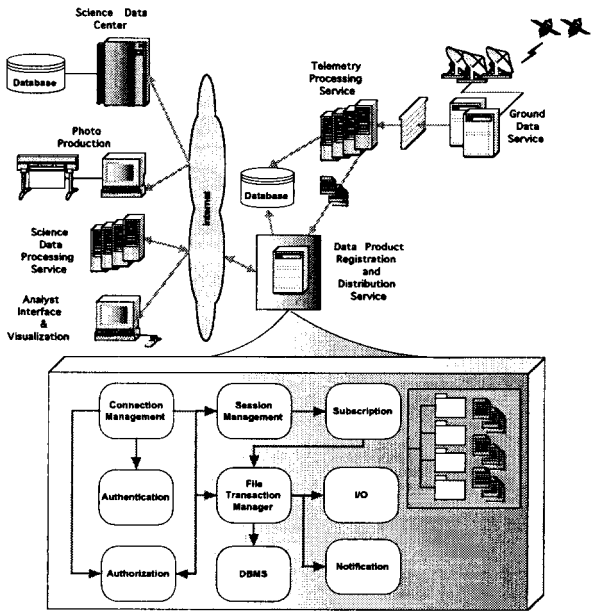

Figure 1. Image processing framework.

While each business component is critical to the quality of the end data products, the most visible components to the science community are the data product distribution and registration services. They serve as the live data communication channel for the science communities. Both services must be scalable, reliable, and adaptable to various mission requirements. The services must be scalable to the volume of data they manage and the number of users they serves. They must be reliable to preserve the integrity of the data they manage and distribute. And finally and most importantly of all, they must be adaptable to their operating environment and the mission-scientists' needs. Component architecture is the key to satisfy performance and the QoS requirements.

2. MOTIVATION

File Exchange Interface (FEI) is a data product distribution and registration service that is designed as a service component to MIPS. The service organizes data products through user defined file types and offers secure, high-performance file transaction and distribution capabilities that conventional file distribution services lack. As a core communication component in the image-processing framework, there are inherent and development-induced complexities in the design of FEI. The inherent complexities stem from various QoS mission requirements and fundamental challenges of developing any networked transaction services that include detection and recovery of network and host failures, minimizing the impact of communication latency, and determining the optimal transaction processing model to minimize lock contingencies. The development-induced complexities stem from the limitations of tools and techniques used in developing scalable transaction-oriented data streaming services.

2.1 PREVIOUS VERSIONS

Former implementations of FEI had adopted the conventional object-oriented paradigm for promoting abstraction, encapsulation, inheritance, and reuse. The idea of building a program by the composition of modules, or objects [8] really simplified the design and development process. One of the limitations of the conventional OO development paradigm is the requirement of having static interfaces for each object so that they can communicate by invoking each other's methods [18]. Another limitation in the conventional OO development paradigm is that it encouraged extensibility through inheritance. Software maintenance was done by stuffing an existing factory method [6] with additional conditions in order to instantiate an ewly implemented subclass of an existing abstract base class. The resulting software application is like a Swiss Army Knife that has a set of generalized objects for each set of specialized conditions. The long-term effect of such an approach is gradual degradation in software performance because of the added conditional branches and complex inheritance hierarchies. The increase in its complexity also always translates into costly maintenances.

3. OVERVIEW OF KOMODO

Frameworks are an object-oriented reuse technique [16]. They are built from reusable components with design patterns as the micro-architectural elements. Some of the well known distributed component frameworks including

Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) [4] offered flexible solutions to large-scale client/server systems. Komodo, also known as FEI5, has adopted the idea of reusable interface designs and components to offer an approach that is different from the conventional OO development paradigm and is much more flexible in handling various mission requirements without a performance tradeoff. The goal is to design and develop a component-base framework that dependents on a set of virtual components [1] to enable pluggable configuration for mission specific business requirements. These virtual components are to be specified and loaded into the Komodo core during deployment time to enable deploy-time assembling of mission specific service.

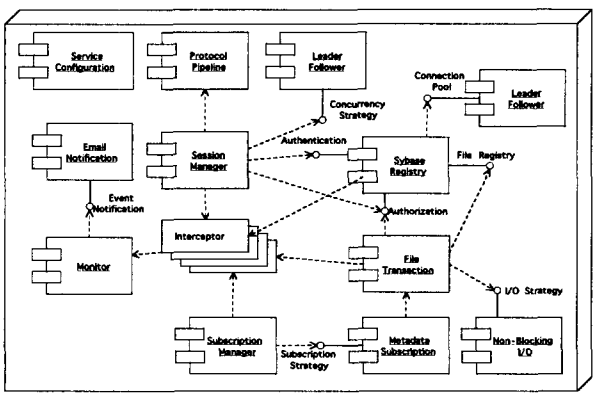

Figure 2. Example Komodo deployment diagram.

Komodo is the latest incarnation of FEI. It has incorporated some fundamental ideas from SPL by combining core service components into its framework. These core service components, illustrated in Figure 2, include file service configurator, file transaction manager, session manager, protocol pipeline [12], session manager, subscription manager, and monitor service. The deployment diagram also identified a few virtual components [1], depicted with a circle followed by a solid line attached to a physical component. A virtual component consists of an abstract interface that is required by the core framework. The actual implementation of the component is to be determined during deployment time. In the illustration, the circle denotes the abstract interface that the Komodo core requires and the physical components are components that are identified during deployment time. The new design also recognizes the importance of having a standardized component management solution [11], which enables unified service management for all instances of the service components. Each component in Komodo is a managed component, because they each represent a resource that is required by the service.

The virtual components identified by the framework are considered to be mission-specific service components. Since they are virtual components, the concrete implementation of each of these components will be loaded, configured, and bound to the Komodo core. Dynamic service configuration and deployment offers maximum administration flexibility without taking a huge toll in service performance. It depends on the component loading strategy chosen for the service instance; the performance tradeoff may take place during service startup or when the specific component service is first utilized. The remainder of this section discusses key components and interface requirements for Komodo.

3.1 SERVICE CONFIGURATION AND RECONFIGURATION

The ability to load, configure, bind, and unload components is the key to any component based systems. Komodo has adopted the virtual component design pattern [1] as its method of handing various component loading strategies and dynamic configuration and reconfiguration of service components.

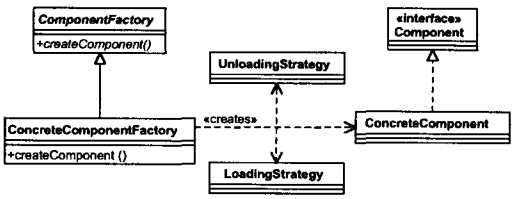

Figure 3 Virtual component.

The component loading strategy defines how and when components get bound to the Komodo core. Komodo uses the eager static [1] scheme to load core service components. This scheme requires the concrete components to be loaded immediately when the program initializes. Komodo uses the eager dynamic [1] scheme to load pluggable components where the concrete components get loaded when their factory is instructed to resolve the components at run-time. Both schemes can be easily implemented with Java using its dynamic class loader [17]. Component configurator design pattern [5], shown in Figure 4, should be used when implementing the eager dynamic scheme.

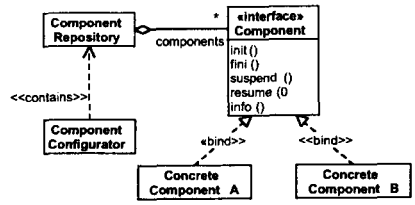

Figure 4. Component configurator design pattern.

3.2 SECURITY STRATEGIES

Security includes authentication, authorization, and accounting. This is different from network communication security strategies. Secure Socket Layer (SSL) [7] is the *de facto* standard for secure network communications and Komodo uses SSL by default. We are also considering support for other secure communication schemes by using the strategy design pattern [6, 13] as the abstract interface to various message encryption algorithms. The authentication schemes could include one-way encrypting of user password and private keys such as Kerberos [2]. The authorization schemes determine the user and file type role associations within the Komodo service. These roles could include operational user, principle investigator, administrator, etc. Each user in Komodo is associated with one or more roles, which determine their access and privileges to the file types managed by Komodo, see Figure 5. The pluggable authorization interface allows for retrieval of external user role information from an existing directory service such as Lightweight Directory Access Protocol (LDAP). The Accounting scheme keeps track of each user operations to facilitate future auditing of data modification history.

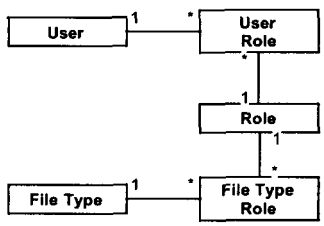

Figure 5 The Komodo authorization class relation

The security component only gets loaded at service initialization time for obvious reasons. Its implementation defines how users will be authenticated, the source for user roles, and how accounting will be handled.

3.3 FILE REGISTRY

A file registry serves as a file lookup and registration component for all the files and their types managed by Komodo. Since this is a registration service component, it must also support transactional updates to the registry. This is the key component in Komodo that distinguishes it from a simple FTP server. While it may sound a bit extreme, it is what makes Komodo special. File transactions such as adding a file to the server will require locks to prevent other users from accessing that particular file until the registration is complete. The registration processing includes receiving the entire file from the contributor, performing a file integrity checks, and recording the file metadata to the registry.

The simplest type of file registry implementation could be the use of a relational database system (RDBMS), since it provides optimal query processing mechanisms and has a standardized logical data organization structure. The inherent complexity in using a RDBMS results from the fact that all RDBMS do not provide the same functions and capabilities. The Java Database Connectivity (JDBC) API and Open Database Connectivity (ODBC) API have offered a unified query interface API, but it does not address other portability issues such as SQL statement and return data set. It is a known fact that every RDBMS has its own implementation of a subset of the SQL specification [3]; therefore SQL is not portable among RDBMS. Features that are RDBMS specific include the mechanisms used in enforcement of referential integrity, subqueries, views, stored procedures, SQLJ support, result set cursor (implicit/explicit) and many more.

Another non-portability issue of RDBMS is their support for standard SQL data types. Again, the SQL specification has specified a list of standard data types but each RDBMS vendor has its own favorite set of standard types. This non-uniform support of standard SQL types has an impact on how queries can be issued from the application level and how the return data are mapped back to application-level as abstract data values.

There have been many publications in the area of application-level query abstractions that range from implementing a simple jump table to inventing a whole new object language engine. All current solutions have performance penalties. The Komodo framework made the file registry component a virtual component and it is up to the registry implementer to design and implement the necessary operations to interact with the targeted file registry with minimal overhead.

3.4 SUBSCRIPTION AND NOTIFICATION STRATEGIES

File subscription service is a unique feature in FEI. It is a mechanism to enable automatic delivery of data products, see Figure 6. This feature plays a key role during mission operations where remote scientists can have the latest processed data products delivered and with optional triggering of an additional processing chain at the remote site. A subscription is best described using the observer design pattern [6, 13], where events occurring within an object (subject) cause dispatching of others (observers). The similar model can be applied to general event notification mechanisms. Significant event notification is important in order to page, email, or to interact with other enterprise services.

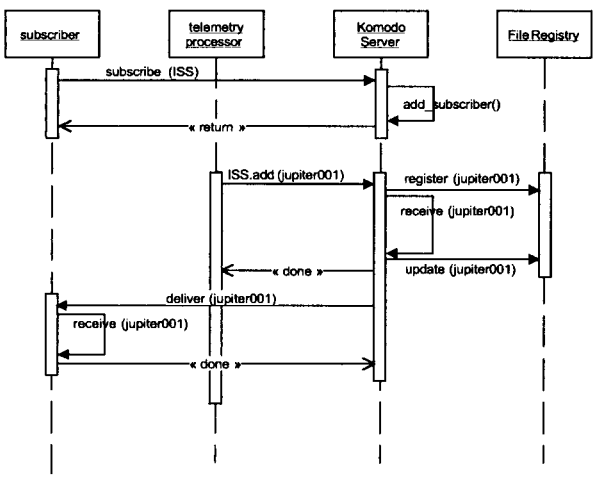

Figure 6. A simple subscription sequence diagram.

Implementations of event dispatch mechanisms [10] for multi-threaded environments are faced with several challenges.

General network communication failure: The service is unable to dispatch an event to a subscriber due to a communication failure or because the subscriber is no longer connected. In this case, the service should not continue to attempt to deliver any more messages to the non-existent subscriber.

Non-responding subscriber: The subscriber is connected but iti s not accepting the dispatched message. This is usually caused by a resource consumption problem at the subscriber host that is prohibiting the subscriber from handling any messages from the server. The schemes for handing such situations can vary among implementations. One option is to schedule for future re-dispatch of events. The consequence of such an approach is that the implementation must decide when to give up, that is to remove the non-responding subscriber, while resources are being consumed by the server's subscription cache. Another approach is to simply disconnect any non-responding subscriber and require the subscriber to re-subscribe when its local resources are available. The consequence of such approach is that it requires the subscribing client to implement the reconnection strategy and also query for any missed events.

Frequency of significant events: In the significant event notification case, the method of event dispatch is always asynchronous and a non-responding subscriber will not receive any further significant event notifications. One criterion that the administrator should specify is the frequency of dispatching of the same significant event from the same source. For example, a storage threshold has been reached and a significant event was dispatched to notify the administrator. Successive file registration will also trigger the same significant event until the administrator creates more storage space. The notification strategy should not overwhelm the administrator on the same significant event generated by the same source. The ability to specify the frequency of dispatching of the same event created from the same source allows the administrator to specify how often he/she wishes to be reminded.

General multi-threading issues: Given the multi-threading environment, there are many opportunities for deadlock, reduced concurrency, and priority inversion due to recursive calls [10] in the subscription component while it is busy handling dispatching requests.

There can be many variations in implementations of the subscription strategy due to the varying mission needs and the size of the data the service must handle. For example, if a mission only generates many small data products and all processing teams are operated in a high-speed LAN with sufficient resource, then it can choose to have the actual data products as part of the message being delivered to minimize excessive disk I/O. For missions that have huge data products and/or operate under diverse network configurations, the message being delivered should only contain metadata for the new data product.

3.5 I/O STRATEGY

There have been many publications regarding optimizing software I/O, since it is the number one performance bottleneck in any large-scale client/server application. The optimal I/O strategy can vary depending on the operating system and the type of file system under which the software must operate. The most portable kind of I/O strategy is blocking I/O and it is the default for most client/server applications. However, it is inefficient. I/O performance can be improved by varying the binary data buffering strategy and minimizing synchronization.

3.6 MANAGEMENT INTERFACE

Service management is always important for any large-scale client/server systems. The ability to perform health monitoring and dynamic reconfiguration on service resources is essential. Traditionally the management and monitoring consists of ad hoc implementations that are scattered around the software. A unified method in developing management interfaces [11] will simplify development of service management applications, as shown in Figure 7.

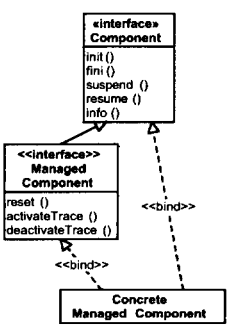

Figure 7. Service management class diagram.

Monitoring is often done by having a Singleton [6, 13] object that provides accounting services. This requires all critical operations to be reported by this object, which has many performance implications, because multiple worker threads are competing to report their states to the single monitor object. Gathering of statistical data on an active server should only introduce minimal overhead otherwise the method of gathering affects the actual statistics. One simpler approach is to apply the interceptor architectural pattern [5] to transparently introduce statistic-gathering modules without affecting the overall processing flow of the active service.

4. CONCLUDING REMARKS

Software maintenance and support are large investments [15] for any space missions. It is commonly held that reuse can reduce the cost in software development and increase the quality of the product being produced. While module reuse can be beneficial to as ingle mission, an architectural reuse can be beneficial to all missions. Our experience from working on Komodo has produced an alternative approach to the conventional OO paradigms. Component architecture offers a much more flexible approach to the handling of diverse project or mission requirements by promoting separation of concerns in developing a software framework against well-defined interface to virtual components. What makes such an approach different from the conventional OO architecture is its ability to rely on factory objects to bind concrete component implementations to well-defined interfaces at runtime. Developers can benefit from such an approach by stubbing out all non-essential components to reduce the development complexities. Space missions can benefit from software developed using components by reusing a well-tested framework that enables them to specify the concrete components that meet their QoS requirements.

5. ACKNOWLEDGEMENTS

Thanks tom y MIPL Data Management System development team for making Komodo a reality. Thanks also to Larry Preheim and Dr. H. Norton Riley for helpful comments on the paper.

6. REFERENCES

[1] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual Component: A Design Pattern for Memory-Constrained Embedded Applications," *Proceedings of the 9th Annual Conference on the Pattern Languages of Programs*, Sept. 2002.

[2] B. Tung, *Kerberos: AN etwork Authentication System.* Addison Wesley, 1999. ISBN 0-201-37924-4.

[3] C. J. Date, *An Introduction to Database Systems, Seventh Edition.* Addison-Wesley, 2000. ISBN 0-201-38590-2.

[4] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," *Proceedings of the 6th USENIX C++ Technical Conference*, Apr. 1994.

[5] D. C. Schmidt, M. Stal, H. Rohner, and F. Bushmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects,*

Volume 2. Wiley & Sons, 2000. ISBN 0-471-60695-2.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Programming*, Addison-Wesley, 1995. ISBN 0-201-63361-2.

[7] E. Rescorla, SSL and TLS: Designing and Building Secure Systems, Addison Wesley, 2000. ISBN 0-201-61598-3.

[8] F. P. Brooks, *The Mythical Man-Month.* Addison-Wesley, 1995. ISBN 0-201-83595-9.

[9] G. Wang, L. Ungar, and D. Klawitter, "Component Assembly for OO Distributed Systems," *IEEE Computer*, vol. 32, no. 7, pp. 71-78, Jul. 1999.

[10] I. Pyarali, C. O'Ryan, and D. C. Schmidt, "A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware," Proceedings of the IEEE/IFIP International Symposium on Object-oriented Real-time Distributed Computing, Mar. 2000.

[11] J. S. Perry, *Java Management Extensions*, O'Riley & Associates, Inc., 2002. ISBN 0-596-00245-9.

[12] M. E. Fayad, R. E. Johnson, Editors, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, 2000. ISBN 0-471-33280-1.

[13] M. Grand, *Patterns in Java™, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition*, John Wiley & Sons, 1998. ISBN 0-471-22729-3.

[14] P. Clements, and L. Northrop, Software Product Lines: Practices and Patterns. Addison-Wesley, 2002. ISBN 0-201-70332-7.

[15] R. B. Grady, Practical Software Metrics for Project Management and Process Improvement. P. T. R. Prentice-Hall, 1992. ISBN 0-13-720384-5.

[16] R. E. Johnson, "Frameworks = (Components + Patterns)," *Communications of ACM*, vol. 40, no. 10, pp. 39-42, Oct. 1997.

[17] S. D. Halloway, *Component Development for the Java™ Platform*, Pearson Education, Inc., 2002. ISBN 0-201-75306-5.

[18] T. J. Brown, I. Spence, P. Kilpatrick, and D. Cookes, "Adaptable Components for Software Product Line Engineering," *Proceedings of the Second International Conference, SPLC2*, pp. 154-175, Aug. 2002.