Author

Roshni Malani's interest in software engineering research began in the classroom. From this initial impetus, she pursued her project with Professor Richard Taylor and Eric Dashofy, one of his graduate students. Roshni's experience with undergraduate research confirmed her desire to become a university professor. She is currently a student at University of California, San Diego working towards a doctorate degree in Computer Science. For those students who are considering an undergraduate research project, Roshni advises beginning their projects early and making them a priority. When she is not analyzing and interpreting data from engineering research, Roshni can be found playing tennis, skiing, and shopping.

Key Terms

- Architectural Frameworks
- Java Message Service
- Middleware
- Software Architectural Styles

The Benefits of a Java Message Service Implementation of the C2 Framework

Roshni Malani
Information and Computer Science

Abstract

Software architectural styles are the key to designing and implementing large software systems. Architectural styles impose rules and constraints on software architectures to elicit beneficial properties from those architectures. C2 is a component- and message-based style that promotes the development of loosely-coupled applications that are distributable, dynamic and adaptable. However, the constraints of this style are not well supported by modern object-oriented programming languages. The Java Message Service (JMS) standard, implemented by many middleware vendors, provides a number of capabilities needed by C2 applications. This study describes a novel software architecture framework for building C2 applications based on a JMS implementation. An empirical comparison of this framework with a hand-coded, non-middleware-based framework was performed. The results of this comparison demonstrated that although both frameworks exhibited a linear degradation in performance as the number of components was increased, the non-middleware-based framework performed better. However, the JMS-based framework has the potential to provide many other beneficial features.

Faculty Mentor

An architectural style for significantly-sized software applications must be carefully chosen to ensure the final system will exhibit all the qualities the designers seek. Once chosen, appropriate implementation techniques must also be selected. It is commonly assumed that generic, commercially-produced infrastructure will be superior to open-source technologies. Roshni Malani's project put this assumption to the test—and discovered that, in the domain she studied, the conventional wisdom was wrong. Commercial middleware can impose heavyweight constraints, leading to sub-par performance as compared to more limited, judicious designs. Projects such as this confirm again that the best approach a researcher can take is to question assumptions and follow the data!

Richard Taylor
Donald Bren School of Information and Computer Sciences

Introduction

The demand is increasing for large software systems that are reliable, fast, distributable, and scalable. These system qualities are achieved using well-designed software architectures based on well-established architectural styles.

The C2 architectural style is a component- and message-based architectural style, developed at the Institute for Software Research at the University of California, Irvine, that facilitates the construction of distributed, event-based, loosely-coupled software architectures. C2 architectures must be decomposed into autonomous components, connected together by explicit connectors, and communicated through asynchronous messages. Since object-oriented programming languages do not directly support the constraints of the C2 style, a framework was needed to provide development support for implementing C2 architectures. A framework for the Java programming language, called c2.fw, and a set of sample C2 applications were previously developed.

Java Message Service (JMS) is the industry standard for distributed application development using Message-Oriented Middleware (MOM). Several vendors implement the JMS Application Programming Interface (API) to allow Java components to communicate via the reliable exchange of asynchronous messages.

The constraints of the C2 architectural style and the concepts embodied by the JMS API have many similarities, especially the message-based, asynchronous communication of distributed, loosely-coupled software components. Thus, the research goal was to construct a JMS-based framework that supported the C2 style and compared empirically with the pure Java-based framework. The JMS-based framework, called c2.fw.jms, would employ JMS topics in the publish/subscribe messaging paradigm to deliver messages between components and connectors. The performance of the frameworks was tested by developing a suite of C2 architectures and measuring the average message processing time on each framework.

After successfully building the JMS-based C2 framework and running a series of experiments, the message processing time of both the Java-based and JMS-based frameworks was empirically shown to increase linearly with the increase in number of components. Even though c2.fw.jms had a greater initial overhead and took longer to process each message, both frameworks exhibited a linear degradation in performance as the number of components was increased. We believe the JMS implementation performed poorly in comparison to the pure Java implementation because it was not optimized for local message delivery and because it provided additional features, such as the reliable, guaranteed delivery of messages.

Background

Software Architectural Styles

Software architecture is the high-level structure of a software system. The structural elements of architectures are components (the loci of computation), connectors (the loci of communication), interfaces (that define how components and connectors interact), and links (that define the topology). Software architecture provides an abstraction for the design of large-scale software systems that is meaningful, valuable and visual. The importance of software architecture in software engineering is analogous to architectural blueprints in civil engineering. The blueprints provide everyone involved in the project a common understanding of the system and define a plan for converting the concept into a functional building.

Software architecture is an expression of a particular software architectural style—a set of rules that define or constrain the architectures that conform to the style. These rules and constraints are intended to make it easier to elicit desirable properties of the system (Di Nitto, 1999). Returning to the civil engineering analogy, architectural styles, such as Gothic or Baroque styles, define a set of rules for construction of specific types of buildings that have certain desirable properties. For example, Gothic building features, styles such as buttresses and stone construction, result in structures that can support large open spaces and survive for many centuries. Likewise, many software architectural styles have been developed over the years, including: the pipe-and-filter style used in the UNIX operating system, the blackboard style used in the artificial intelligence (AI) domain, the implicit invocation style used in event-based applications, the peer-to-peer style used in file-sharing networks, and the client-server architecture used throughout the Internet. These styles have few constraints and, as such, elicit few desirable software qualities. For example, in the pipe-and-filter style, communication is restricted to byte streams, but the style does not constrain the contents of these streams. Because of this, pipe-and-filter can guarantee communication but not meaningful interoperability. Highly complex architectural styles specify more rules, require more constraints, and elicit qualities that are more valuable.

The C2 Architectural Style

The C2 architectural style is a component- and message-based architectural style. A component-based style is one in which systems are modeled through the explicit use of components and connectors. A message-based style employs asynchronous messages to facilitate communication throughout the system. Constraints placed by the C2 style include: 1) each component and connector have only two interfaces, explicitly referred to as the top and bottom interfaces; 2) the top of a component may only be connected to the bottom of a single connector; 3) the bottom of a component may only be connected to the top of a single connector; 4) many components and connectors may be connected to the top or bottom of a single connector; 5) communication between components and connectors occurs only through asynchronous messages and through no other means; and 6) components may assume an independent thread of control (Medvidovic, 1997). A simple example of an application built using the C2 style is the client-server chat system shown in Figure 1. This application functions as a simple Instant Messaging system with three fixed clients who can send short text messages that will be broadcast by the chat server.

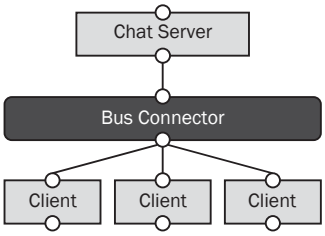

Figure 1
The Client-Server Chat System

The constraints of the C2 architectural style induce many beneficial properties from the system. Since each component and connector is assumed to run in its own thread and in its own address space, the system is easy to distribute across multiple processes and machines. The components are loosely-coupled and independent, with a limited view of the rest of the system. Since connectors facilitate the dynamic transmission of messages, the system can be reconfigured at runtime by adding and removing components and connectors (Taylor, 1996).

A more involved example of a C2 application is the KLAX video game, shown in Figure 2. Colored tiles fall down chutes, are caught by the palette, and are dropped down wells. The objective is to eliminate tiles by collecting three of the same color in a given well. This architecture can be modified during runtime to implement a different game called SpellingKLAX, in which letters fall down chutes and are eliminated only if they spell a valid word. Because of C2's requirements of loosely-coupled components and explicit connectors, the three components involved in determining the next tile, in matching the tiles, and in displaying the tiles in this architecture can be replaced easily during runtime with three components that determine the next letter, match the spelling, and display the letters (Medvidovic, 1996).

Object-oriented languages like Java do not directly support the C2 style constraints. Computation in C2 architectures is achieved by components with two interfaces, whereas Java provides generic objects. Communication in C2 architectures is achieved by asynchronous messaging, whereas Java only supports synchronous, parameterized method calls. Thus, a software framework, called c2.fw, was developed to ease the implementation of C2 architectures in the Java programming language. The framework provides abstract base

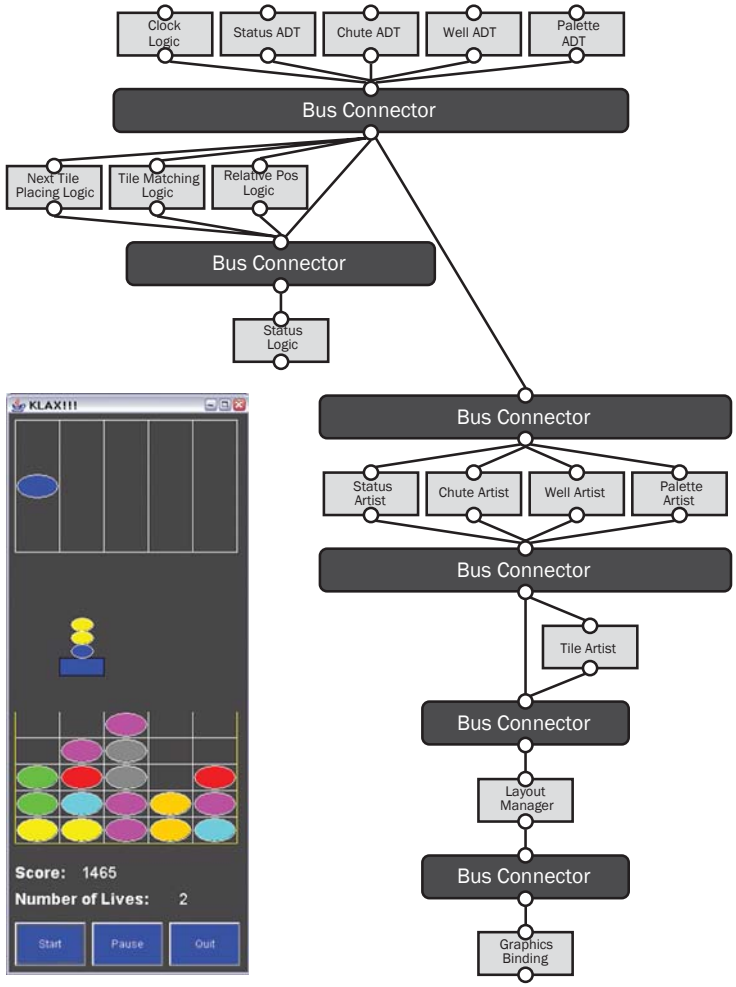

Figure 2
KLAX Architecture and Screen Shot

classes for C2 entities such as components, connectors, interfaces, and messages, and allows the user to control the behavior or architectures via pluggable threading policies, message queuing policies, and architecture managers that govern topology.

Middleware

Large, distributed systems are often plagued with problems of heterogeneity and interoperability. Heterogeneity issues may arise from the use of different programming languages, hardware platforms, operating systems, and data representations. Interoperability denotes the ability of heterogeneous systems to communicate meaningfully and to exchange data or services. With the introduction of middleware, heterogeneity can be alleviated and interoperability can be achieved.

Middleware is a layer of software between the distributed application and the operating system that consists of a set of standard interfaces that help the application use networked resources and services. There are several different categories of middleware: transaction-oriented middleware, such as IBM CICS of Tuxedo, which supports the integration of transactions across distributed databases; Remote Procedure Call (RPC), which supports the invocation of procedures across machine boundaries; and object-oriented middleware, such as CORBA, Remote Method Invocation (RMI), and Microsoft's COM, which facilitate the synchronous communication between objects. Another middleware category is MOM, which ensures that asynchronous messages are transmitted reliably among software components. Messages are self-contained packages of autonomous data sent to inform an application of some event. MOMs provide fault tolerance, load balancing, scalability, and transactional support (Emmerich, 2000).

Java Message Service

JMS is a Java API implemented by several MOM vendors to allow distributed Java components developed in the Java 2 Platform, Enterprise Edition (J2EE) to communicate via asynchronous messages. JMS applications consist of a provider, clients, messages, and administrated objects, and are developed in either the publish/subscribe paradigm or the point-to-point paradigm. A JMS provider is a messaging server that supports the creation of connections (multi-threaded virtual links to the provider) and of sessions (single-threaded contexts for producing and consuming messages). JMS clients are Java programs that produce or consume messages. JMS messages are objects that communicate information between JMS clients, and are composed of a header, some optional properties, and an optional body. Administered objects are preconfigured JMS objects, such as a connection factory (object a client uses to create a connection to a provider) and a destination (object a client uses to specify the target of its messages) (Armstrong, 2003).

The publish/subscribe messaging paradigm, as illustrated in Figure 3, is built on the concept of a topic, which functions somewhat like a bulletin board. Consumers subscribe to receive messages from a given topic and publishers address messages to a topic. The JMS provider distributes the messages arriving from a topic's multiple publishers to its multiple subscribers. Thus, publishers and subscribers remain relatively anonymous. However, there exists a timing dependency, in that a client who subscribes to a topic can only consume messages published the subscription has been created. On the other hand, the point-to-point messaging paradigm, as illustrated in Figure 4, is built on the

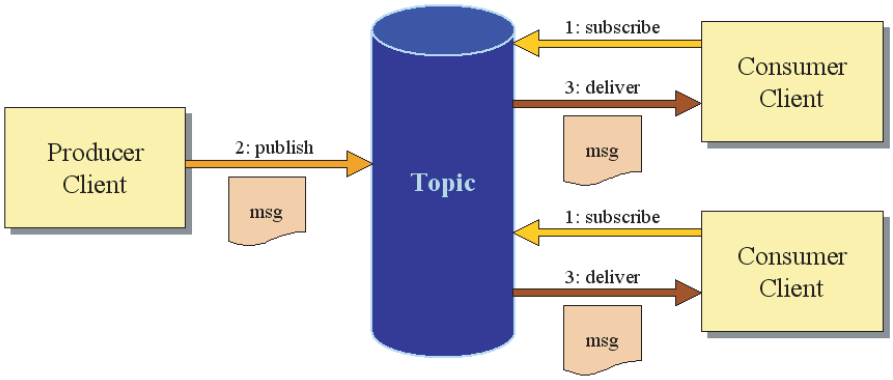

Figure 3
Publish/Subscribe Messaging Paradigm

Figure 4
Point-to-Point Messaging Paradigm

concept of a queue. Senders address each message to a specific queue and receivers extract messages from the queues established to hold their messages. Unlike the publish/subscribe model, messages in the point-to-point domain may only be consumed by one receiver. There exists no timing dependency in the point-to-point paradigm because a receiver can extract its messages whether or not it is running when the client sends the message (Armstrong, 2003).

Thus, the key features of JMS applications are loose coupling, message-based communication, asynchrony, distributability, reliability, and resilience. JMS applications are loosely coupled, in that producers and consumers are not explicitly aware of each other, but rather only the JMS provider. Since communication is achieved solely through the exchange of messages and since the JMS provider delivers the messages, the applications are event based and asynchronous. Since all clients are autonomous and since the provider handles the delivery of messages, the use of JMS eases communication over process and machine boundaries, resulting in a distributed application. The JMS API supports the use of guaranteed message delivery as well as the optional use of fault tolerance and load sharing, thus allowing JMS applications to be reliable and resilient.

Motivation

The concepts at the heart of the JMS are similar to those embodied by the constraints of the C2 architectural style. Both the JMS API and the C2 style require asynchronous communication via messages. Both promote the development of distributed, loosely-coupled software components interacting through explicit connectors. In the C2 style, connectors are first-class objects like components; in the JMS API, topics and queues are explicitly modeled in the architecture. This indicates that it should be possible to use JMS to create a lightweight C2 framework that leverages JMS' strengths.

Why create a JMS-based framework that supports the constraints of the C2 style? One reason is to determine how off-the-shelf message-oriented middleware can simplify the development of a framework. Another goal is to discover how and if the performance of C2 applications improves by using a middleware solution. An additional purpose is to study the feasibility of leveraging other capabilities of a middleware solution, such as fault tolerance and support for distributed applications, to make the current C2 framework more powerful. These three motivations guided our development of a new framework for the C2 style using JMS.

Approach

JMS-Based Framework for C2

One of the most important design goals of the JMS-based framework (c2.fw.jms) was that it be compatible with the current Java-based framework (c2.fw). The primary reason for compatibility was so an empirical comparison of the two frameworks could be directly evaluated, without having to reimplement the components or connectors. Another reason was to provide developers with a simple way to migrate their applications to the new framework.

The c2.fw framework provided base classes for the many different C2 concepts, such as components, connectors, links, and messages. Some boilerplate connectors and generic message types were also provided. Three different objects control C2 architecture: the architecture manager, message handler, and architecture engine. The architecture manager is responsible for the topology of the architecture, and manages what components and connectors are in the architecture and which interfaces are linked to which other interfaces. The message handler provides a queuing policy for messages by either a single message queue for the entire architecture or one queue for each interface. The architecture engine manages the threading policy—either a static pool of threads for the entire architecture or one thread for each brick. These three objects together control any C2 architecture. In order to create an interface-compatible JMS-based framework, JMS was employed to implement the functionality of these three objects.

Creating a JMS-based framework to support the constraints of the C2 architectural style involved several important design decisions. The first important decision was the choice of the JMS programming paradigm: publish/subscribe or point-to-point. In the publish/subscribe paradigm, components and connectors can simply maintain a topic for the top interface and another topic for the bottom interface. When sending a message, the component or connector can publish to one topic, without explicit awareness of the recipients of the message. However, in the point-to-point paradigm, each link between a component and a connector must have a queue at both ends. Therefore, if a connector is linked to six different objects on its bottom interface, then it will need to maintain six different queues for its bottom interface, thus making it explicitly aware of the recipients of its messages. In addition, a queue retains its messages while the receiver is not running and delivers them when the receiver becomes active. This lack of timing dependency between the production and consumption of messages is not characteristic of the C2 style. Thus, for

greater simplicity, for more decoupling of components and connectors, and for a timing dependency, c2.fw.jms was built on the publish/subscribe model.

The next design decision in the process of developing a JMS-based framework was how many topics to use throughout the architecture. The first idea was to replace each connector with a JMS topic, since both perform the same function. However, since connectors may perform functions other than those supported directly by JMS, the framework should retain the notion of connectors and not replace them in their entirety. The second idea was to create one topic for each component and connector, thus treating both as first-class modeling objects. However, the difficulty with this idea was the lack of explicit support for the C2 architectural style constraint of two independent interfaces. Thus, the third idea was to create one topic per interface on each component and connector. Even though the first two ideas required a smaller number of topics, the third best reflected the constraints of the C2 style.

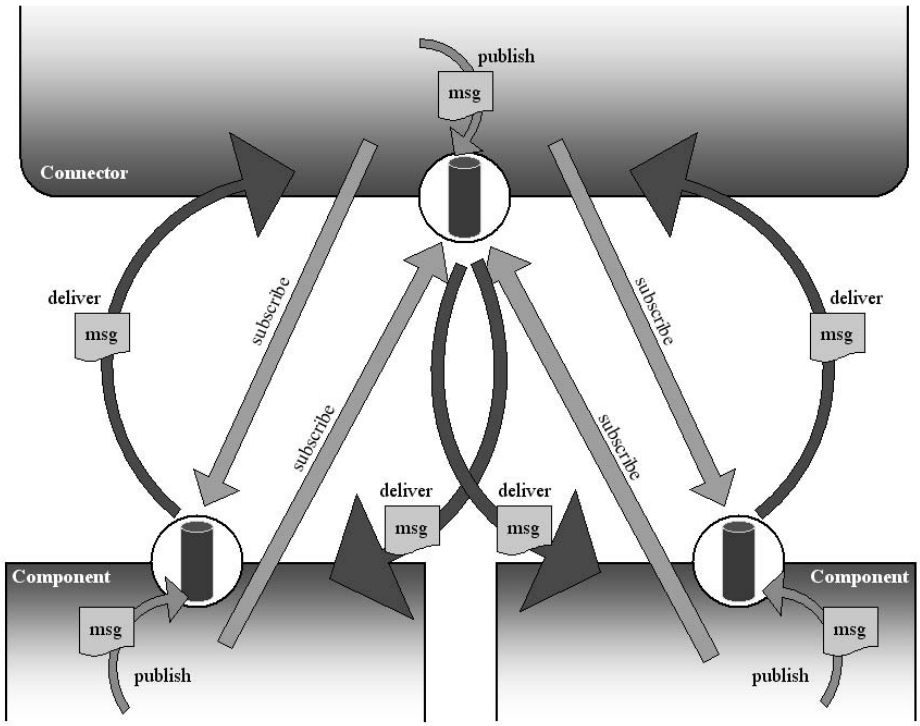

Figure 5
JMS-Based C2 Framework Design

Another design decision regarding JMS was whether to use administratively-created topics or programmatically-created temporary topics. The advantage of defining topics administratively is that they are created when the JMS provider is started, rather than when the application is started, thus resulting in a faster startup time. However, the administrative topics are fixed and cannot be changed dynamically by the program during runtime. Therefore, despite their greater overhead, temporary topics were better suited to the constraints of the C2 style.

Because of these design decisions, construction of the JMS-based framework for C2 applications was fairly straightforward. When creating components or connectors, the framework created two temporary topics, one for each interface. When creating a link between two interfaces, the framework created two subscriptions: one subscription from the first brick (component or connector) to the second brick's interface, and another subscription from the second brick to the first brick's interface. When a component or connector received a request, it published to the topic on its top interface. When a component or connector was sent a notification, it published to the topic on its bottom interface. If the message was sent to multiple components or connectors, the message needed to be published only once to the appropriate topic. After publishing to a topic, JMS took care of routing the message to the appropriate subscribers automatically. The sequence of steps required to send and receive messages is demonstrated in Figure 5.

In contrast to c2.fw, c2.fw.jms was a much simpler implementation. Of the three objects that control the C2 architecture in c2.fw, the functionality of the message handler and architecture engine became almost obsolete, whereas the functionality of the architecture manager remained nearly the same. Since JMS implementations were responsible for allocating threads among the topics, c2.fw.jms did not need to do this explicitly in the architecture engine. In addition, a message handler was not needed because the JMS delivered messages to the appropriate component or connector. Even though these objects were not needed anymore, they were retained as empty skeletons in order to maintain compatibility with c2.fw. Only the architecture

manager was modified to handle the creation of topics in addition to the creation of components, connectors and links. Thus, c2.fw.jms resulted in a simpler implementation.

The final design decision questioned which JMS implementation to use in building c2.fw.jms. There are at least eight valid implementations of JMS available: some are expensive, others are free; some only support the functionality of the API, others provide more features; and some are bundled with other software products, others are standalone products. Of the eight, the two most commonly used implementations in industry are BEA WebLogic and IBM WebSphere, both of which are large, heavyweight products. The decision was rather arbitrary, since c2.fw.jms simply requires a valid implementation. Since the BEA WebLogic Platform has a free one-year development license, it was used in constructing c2.fw.jms.

Framework Performance Measurement

One of the purposes of developing another framework for the C2 architectural style was to determine the relative performance of both frameworks. In order to compare empirically the performance of both the Java-based and c2.fw.jms, a set of different C2 architectures was developed that stressed the framework in different ways. The architectures were created in each framework and messages were sent through the architecture. The performance of each framework was measured and compared.

The set of C2 architectures that tested the frameworks was developed based on the observation that most C2 architectures grow in two directions. As demonstrated by the KLAX architecture in Figure 2, and as derived from experience with many additional C2 applications, most C2 architectures tend to grow vertically as more layers of components and connectors are added and horizontally as more components are added in a given layer. A vertical increase in components and connectors tests the overhead of managing more components, connectors, topics, and links. A horizontal increase in components tests the multicasting ability of the connector. Even though most architectures grow in both directions, the performance of each direction was investigated separately. The reason for analyzing each direction independently was to observe whether JMS might perform better in the horizontal direction due to its inherent support for multicast publish and subscribe. A set of test cases was developed, and a file format for describing the architecture was designed. An automated bootstrapper read a file describing an arbitrary C2 architecture and instantiated the given architecture. Two different types of performance were measured: the time required to instantiate the given architecture and the time needed to process all the messages sent through the architecture. The tests were run and the data was collected, organized and analyzed.

Evaluation

Since C2 architectures are two-dimensional, they grow in horizontal and vertical directions. Architectures grow horizontally as more components are linked to the same connector. They grow vertically as more layers of components and connectors are linked on top of each other. Each direction was evaluated independently in a series of tests that stressed the foundations of both the Java-based and JMS-based frameworks.

Figure 6 demonstrates the results of increasing the components in a vertical direction. The x-axis displays the number of components layered vertically between connectors, and the y-axis measures the number of milliseconds it took on average to send 100 messages from the top of the architecture to the bottom. The message processing time began after the first message was received and ended after the last expected message was received. Since the two sets of data points seem to form a straight line, a linear regression line and equation is shown with each. The performance degraded linearly as the number of components to be traversed increased linearly, as expected. The slope of the lines indicates the increase in the amount of time each framework takes as a layer of components is added to the architecture. As the graph demonstrates, c2.fw requires approximately 10 milliseconds per additional component vertical layer, whereas c2.fw.jms requires approximately 60 milliseconds. However, since both frameworks demonstrate a linear degradation in performance, they have the same asymptotic order in the number of components.

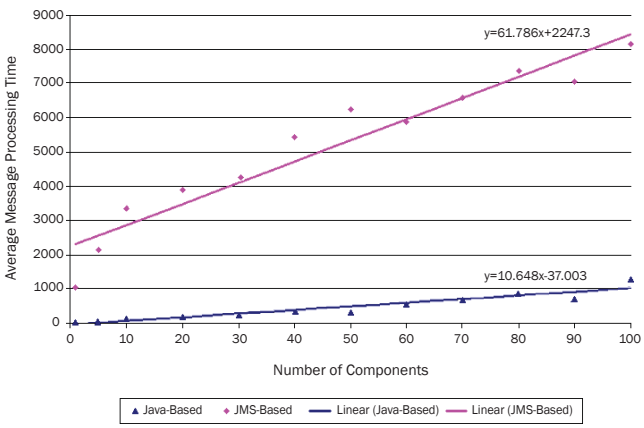

Figure 6
Vertical Increase of Components

Figure 7 is similar to Figure 6, except that it shows the horizontal increase in components. Again, the two sets of data points seem to form a linear line. As the linear regression equation indicates, c2.fw requires approximately 6 milliseconds per additional component in the horizontal layer, whereas c2.fw.jms requires approximately 42 milliseconds.

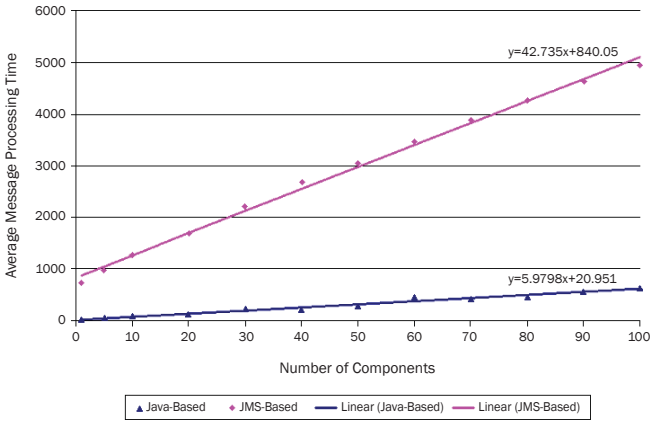

Figure 7
Horizontal Increase of Components

Thus, the data provided in the two graphs indicate that both c2.fw and c2.fw.jms exhibit approximate linear message processing time in the number of components. On average, c2.fw.jms required approximately six to seven times more processing time per message than the c2.fw. Further, the vertical increase in components required a longer message processing time per additional component than the horizontal increase in both frameworks.

In addition to measuring the average message processing time for each of these test cases, the average initialization time for the architecture was measured separately. The architecture initialization time started at the beginning of the architecture constructor and ended after each of the components and connectors were started but before they began sending messages. The average initialization time depended on the number of components being instantiated. For the tests run in this experiment, the approximate initialization time for c2.fw was hundreds of milliseconds, whereas for c2.fw.jms, it was thousands of milliseconds. Thus, c2.fw.jms required approximately ten times as much initial overhead time.

Hence, as the number of components increased, c2.fw.jms performed slightly worse than c2.fw, both in terms of the initial preprocessing as well as the processing of each message. The initialization involved creating the architecture manager, the message handler, and the architecture engine, adding components and connectors, and beginning all the architectural components. The c2.fw framework employed native Java constructs, such as vectors and hash tables, to store all the data. On the other hand, the c2.fw.jms framework also contacted the JMS server, created a connection, established a session, instantiated several temporary topics, and specified the publishers and subscribers of the topics. The processing of messages involved the transfer of messages from one component or connector interface to another connector interface, or vice versa. In both frameworks, the message handler ensured that the message was sent. The c2.fw framework employed a queue of messages associated with each interface, and sent a message by adding the message to an appropriate queue. When a message was added to a queue, the component or connector was notified and handled it accordingly. On the other hand, the c2.fw.jms framework published the message to the temporary topic associated with the appropriate interface. When a message was published to a topic, the JMS provider ensured that all JMS clients that were subscribed to the topic received that message. The subscribers in our framework were actively listening for messages and handled them accordingly.

All of the tasks of the JMS-based framework, such as creating connections, sessions, topics, publishers, and subscribers, as well as publishing messages, required interaction with the JMS provider. The JMS implementation supplied this provider. Even though we could not see how the provider was implemented, we knew that our heavyweight implementation executed more code, utilized more resources, and performed slower than the highly optimized c2.fw framework. However, this constant factor of performance degradation provided many benefits, such as guaranteed once-and-only-once delivery of messages and the persistent storage of messages for crash recovery.

Related Work

Other researchers are exploring the concept of using middleware solutions to provide distributed communication in software architectures.

Eric Dashofy (1999) has compared the benefits of four different middleware technologies, including RPC and RMI, used to implement software connectors in the C2 style. His work is similar to our research for two reasons. First, JMS is simply another messaging middleware technology used to implement connectors, and provides some alternative benefits to those technologies studied by Dashofy. Second, his work highlights the importance of encapsulating the functionality of the middleware within software connectors so that the implementation-dependent factors are separated

from the architectural style. This separation relieves the architect of the burden of examining the properties of different middleware technologies, yet allows the implementor to leverage the different benefits of the different middleware technologies and to choose the technology most appropriate for the given implementation.

Nenad Medvidovic (2002) approached bridging heterogeneous middleware by using architectural constructs. His work addressed the interoperability of software across different middleware technologies and presented software connectors as a possible solution to unify two different technologies. However, Medvidovic's work is limited to pairwise solutions, and multiple technologies cannot be accommodated yet. JMS is an emerging middleware technology that is not explored by his paper, yet general inter-middleware connectors should be able to interoperate with JMS systems as well.

Middleware connectors not only provide interoperability, they also provide quality of service aspects, such as security, scalability, distribution, and concurrency. Model-Integrated Computing can be systematically synthesized and integrated into middleware components at six different points to guarantee quality aspects of an application (Gokhale, 2002). Further, software architecture is being introduced into middleware platforms to offer integrity and configurability. Blair *et al.* (2000) argue that systematic synthesis of middleware configurations can be modeled with architecture descriptions and that such models can then adapt to unanticipated, dynamic changes. The ability of a JMS-based framework to provide quality of service and dynamic reconfigurability are interesting areas of open research.

Conclusion

The constraints of the C2 architectural style and the concepts embodied by the JMS are congruous. The purpose of JMS is to provide commercial applications with a standard for asynchronous message production, distribution, and delivery to allow for a loosely-coupled interaction between new Java components and existing legacy systems capable of messaging. However, the JMS API is flexible enough to allow the exchange of any data messages, including C2 style requests and notifications. Thus, the JMS-based framework that facilitated message exchange between C2 components and connectors using a commercial JMS implementation was successful.

Even though both the Java and JMS-based frameworks exhibited a linear degradation in performance as the number of components was increased, c2.fw.jms was slower. The data shows that both frameworks demonstrated linear performance degradation, which indicates that the messaging policy is implemented in a similar manner. c2.fw was highly optimized for local messages, that is, those messages that are not sent across process or machine boundaries. On the contrary, c2.fw.jms was not optimized for local messages, but rather provided support for the reliable exchange of distributed messages. In addition, this framework also provided guaranteed once-and-only-once delivery of messages as well as the persistent storage of messages for crash recovery. Further, many more objects must be instantiated and many more methods must be invoked in c2.fw.jms as compared to c2.fw, thus resulting in a greater initial overhead time. Thus, c2.fw performed better for local C2 architectures, but did not have the many benefits of JMS.

Future Work

The successful creation of a JMS-based framework for the C2 architectural style paves the road for further research in improving the performance of C2 applications. The benefits of alternative design decisions, such as replacing C2 connectors entirely with JMS topics or using queues instead of topics, need to be determined. The possibility of further performance improvements may be achieved through administrative tuning features available with WebLogic JMS. Such tuning features include deferring JMS acknowledgements and commits, changing the server-side and client-side thread pool size, and disabling synchronous writes and enabling direct file writes. Another area of further research is leveraging additional features of JMS to implement more powerful C2 connectors. Such additional features include distributability across process and machine boundaries, guaranteed delivery of messages, and persistent message storage for auditing. In addition, the performance benefits of employing JMS implementations from different vendors, such as WebSphere, SonicMQ, TIBCO and OpenJMS, need to be explored. The performance benefits of creating a C2 framework with alternative middleware solutions for distributed systems, such as CORBA, COM, ILU and TCP Sockets warrant future exploration.

Acknowledgements

I would like to thank Professor Richard Taylor for undertaking this research project and for consistently providing support and encouragement. I am very grateful to Eric Dashofy for hosting regularly scheduled meetings, for helping every step of the way, and for providing invaluable feedback.

Works Cited

Armstrong, E., J. Ball, S. Bodoff, D.B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock. "The J2EE™ 1.4 Tutorial. Chapter 33: The Java Message Service API." November, 2003. [http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html].

Blair, G.S., L. Blair, V. Issarny, P. Tuma, and A. Zarras. "The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms." ACM International Conference on Distributed Systems Platforms. p. 164-184, New York, NY, April 2000.

Dashofy, E.M., N. Medvidovic, and R.N. Taylor. "Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures." Proceedings of the 21st International Conference on Software Engineering (ICSE'99). p. 3-12, Los Angeles, CA, May 16-22, 1999.

Di Nitto, E. and D.S. Rosenblum. "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures." Proceedings of the 21st International Conference on Software Engineering. p. 13-22, IEEE Computer Society. Los Angeles, CA, May, 1999.

Emmerich, W. "Engineering Distributed Objects." Chichester, England: John Wiley & Sons, Ltd., 2000.

Gokhale, A., D.C. Schmidt, B. Natarajan, and N. Wang. "Developing and Integrating Enterprise Components and Services: Applying Model-Integrated Computing to Component Middleware and Enterprise Applications." Communications of the ACM. p. 65-70, ACM Press. New York, NY, October 2002.

Medvidovic, N. "Component-Based Software Engineering: On the Role of Middleware in Architecture-based Software Development." Proceedings of the 14th International Conference on Software Engineering and Knowledge. p. 299-306, ACM Press. New York, NY, July 2002.

Medvidovic, N. and R.N. Taylor. "Exploiting Architectural Style to Develop a Family of Applications." IEEE Proceedings - Software Engineering. 144 (5/6, October/December), p. 237-248, 1997.

Medvidovic, N., P. Oreizy, J.E. Robbins, and R.N. Taylor. "Using Object-Oriented Typing to Support Architectural Design in the C2 Style." Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering. p. 24-32, ACM SIGSOFT. San Francisco, CA, October, 1996.

Taylor, R.N., N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy and D.L. Dubrow. "A Component- and Message-Based Architectural Style for GUI Software." IEEE Transactions on Software Engineering. 22 (6), p. 390-406, June, 1996.