Migrating to Simpler Distributed Applications

Joachim F. Kainz
Wells Fargo Bank
155 5th Street
San Francisco, CA 94103
+1 415 241 1366
kainzjf@wellsfargo.com

ABSTRACT

In 1994 Wells Fargo Bank was the first large financial services company to invest heavily in distributed object-oriented applications for high-volume, mission-critical applications using version 1 of OMG's Common Object Request Broker Architecture. Wells Fargo continued to improve upon its distributed applications technology leadership by launching a Model Driven Architecture [MDA] initiative in 1999. The technology provided a consistent middle-tier for retail applications such as the award-winning Wells Internet Bank and various phone-bank applications.

Between 2001 and 2003 the original CORBA-based solution was completely replaced with new Web-Services-based distributed applications. This report contrasts the past and the present approach to such applications and discusses the lessons learned while transitioning from CORBA-based "Service Oriented Architecture" to one based on Web-Services.

The final section of this report utilizes lessons learned from using CORBA & Web-Services for Intranet applications and proposes the "application transparent distribution" as a potential paradigm shift which may lead to radically faster and more efficient distributed application development.

Categories and Subject Descriptors

[Computer Applications]: Administrative Data Processing – *business, financial*

General Terms

Performance, Design, Reliability

General Terms

Web-Services, Services Oriented Architecture, CORBA, Java, C++

INTRODUCTION

Wells Fargo was the first financial services company to support Internet Banking. The Wells Fargo Internet was implemented as a 3-tier architecture. The Internet is one of several applications implementing the "View" or "presentation tier". Other "presentation tier" implementations include the Wells Fargo Phone-Bank, which provides banking services via an Interactive Voice Response system. Applications implementing the view are called "Channels" in Wells Fargo terminology. Not unlike the evolution of the commercial Internet, the Wells Fargo system has grown from very humble beginnings (2000 remote calls per month) to much more impressive volumes. At peak the systems support tens of thousands of users and processes thousands of requests per second. These volumes need to be supported in an environment that has no tolerance for downtime.

The "database tier" is implemented almost entirely by mainframes and similar types of systems, which remain dominant in the financial services industry. There are also some very large RDBMS systems implementing less significant aspects of the model.

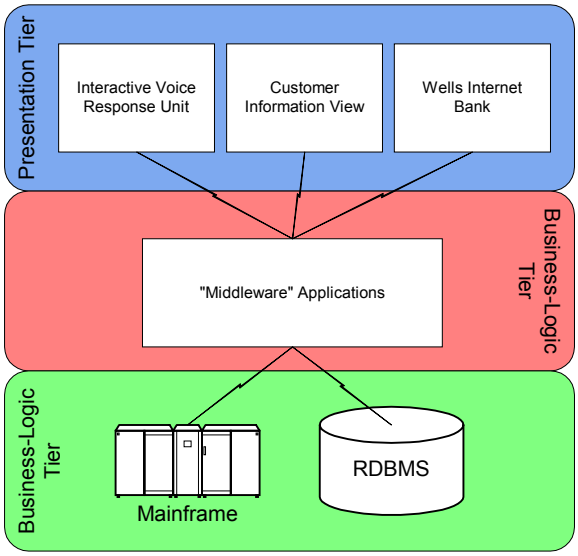

Figure 1 3-Tier Architecture at Well Fargo

To a large extent the business-logic is implemented in the middle-tier. Consequently it can be shared by the different channels of Wells Fargo Bank. The obvious benefit of this design is that business logic does not have to be implemented multiple times for different channels. It is not entirely possible to implement business logic in the middle-tier only once, because of differences in requirements. Business proponents of the different channel applications sometimes represent different interest groups or legal positions and do not always succeed in identifying one set of requirements.

The middle-tier provides a polymorphic view of the data stored by a very diverse set of backend systems. This is one of the most important and successful features of this component. Channel applications can represent account information in a consistent manner, representing many products such as savings accounts,

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

loans (including mortgages), credit-cards, and even stock-portfolios. This feature allows bank customers to view account information from their personalized home-page on the Wells Fargo Internet in addition to performing various self-service features such as money transfers. To achieve this type of polymorphism the middle-tier performs a function best described as "data-scrubbing" (cleansing data from different data sources to improve data-consistency and data-integrity). This function is required because, in addition to using a diverse set of technologies, the different backend-systems also use a very different and partly overloaded business terminology, which results in often incompatible data-dictionaries and data-layouts.

Whenever possible a "concurrent fan-out" is used to retrieve information from the backend-systems in parallel. This drastically reduces the latency of channel requests processed by the middle-tier (if backend_call(i) is the time it takes to make the individual required calls to the backend systems, the systems responds in O(max(backend_call(i))) instead of O(sum(backend_call(i)))). This is important when building the personalized home-page for the customers and in other situations. While building the home-page the channel application calls the middle-tier with a query to return the list of accounts for the customer. The middle-tier first performs a call to a backend-system that indexes all other mainframe and database systems storing account data and then calls the appropriate systems in parallel to retrieve the requested information.

1. THE PAST

The original implementation of the system was based on an early CORBA Object Request Broker, the “ObjectBroker” from DEC.

The technique used to implement this "concurrent fan-out" was based on a feature of the DEC ObjectBroker ORB (OBB) used by the CORBA-based distributed applications, which enables Asynchronous Messaging. In the CORBA environment the middle-tier was built using several different levels of server processes. The lower level of processes would be dedicated to communicate with a particular backend system. The higher-level server, responsible for the fan-out, calls the lower-level CORBA servers asynchronously, which resulted in the desired parallelism.

Another noteworthy feature of the CORBA-based solution was that the middle-tier servers were almost entirely stateless. No customer or session-specific information ever resided on the middle-tier in-between invocations. Instead, all relevant state information was passed to the channel application as part of CORBA object references. In subsequent calls to the middle-tier the client unwittingly passed the object references back to the server and the server was able to re-materialize the object-graphs discovered during the previous invocations from the data cached in the object references. This approach created a highly scalable solution because customer sessions were not tied to a particular middle-tier machine. Calls in a session were routed to any available machine, no matter what where previous calls where processed. This also meant that by adding additional machines the performance of the system scaled almost linearly.

The CORBA-based system was also implemented without using distributed transaction technologies, i.e., no two-phase commit was mandated by the requirements of any of the applications. It turns out that there are very few use-cases in retail banking that justify the relatively high-cost of distributed transactions managers because very few customers simultaneously use Internet Banking, the IVRU and ATMs. This is particularly surprising since much of the literature regarding distributed transaction management is focused on banking environments.

The support for the ORB used by Wells Fargo Bank was about to expire and the ORB vendor was unable to provide a feasible migration strategy for the roughly 5 million lines of existing code. This was the main reason that while the CORBA-based solution was very successful, a rewrite of most of the functionality was unavoidable.

Another problem with the CORBA system was interoperability. The ORB was written before IIOP was defined. This made it very difficult to integrate with other CORBA solutions or use platforms such as Java, which were never supported directly by the ObjectBroker ORB. These problems proved to be a limiting factor in one of the key initiatives at Wells Fargo, known as "Any to Any". This initiative aims to ensure that every system in the bank can communicate with all other systems, if a use-case exists that requires such communication.

Middleware development teams were required to keep experts on staff in order to be able to deal with proprietary nature and the complexity of the underlying technology. This need for experts limited the number of teams that could afford to use this technology on a large scale. The limited number of teams meant that the few teams providing support for distribution risked becoming bottlenecks in the development process.

Versioning was another issue with the CORBA solution. Clients and server code was tightly coupled, which tended to make even the smallest changes very expensive. Minor modifications on the server side often required all clients to be redeployed, which resulted in high costs considering the client software was deployed on thousands of different machines.

2. THE PRESENT

When replacing the CORBA-based distributed application platform the goal was to retain the benefits of the old system, but remove some of the inherent liabilities to create a simpler, faster, cheaper, and more reliable solution. Another goal was to make the development process more cost effective and predictable.

Web-Services in general and SOAP in particular were identified by the Enterprise Architecture team as the desired suite of communication protocols and standards for distributed applications. The rapid and not always predictable evolution of SOAP and Web-Services standards put an additional burden on the development teams. The resulting uncertainties required consideration both from the software architecture perspective as well as from project management.

The use of XML-based protocols immediately enabled larger number of teams and team members to create components and applications that produce and consume XML. Unfortunately this also means that the enterprise ends up with a large number of solutions that may interpret more complex standards, such as XML-schemas or SOAP, very differently. This leads to a new set of interoperability problems that eventually must be resolved.

The new application server-based solution replacing the high-volume, mission-critical CORBA-based application uses a strictly

layered architecture [POSA1]. A "distribution layer" isolates the business logic from being exposed to Java-bindings used by the distribution technology. This encapsulation makes it simpler to react to changes in the standard as well as to support business partners. This flexibility is required because of previously mentioned interoperability problems (business partners occasionally keep insisting on a reading of standards that may not be entirely comply with the mainstream interpretation).

In a similar fashion the business logic is also isolated from the details of components used to communicate with mainframe systems and databases through a persistence layer. Adjustments to the backend systems therefore cause fewer changes to the rest of the application server code. It also enables the middle-tier teams to test the connectivity and functionality of the backend systems in isolation. Additionally, this layer is a very good place to add a "backend-simulator", which helps to decouple the development effort of the backend systems from the middle-tier. Such a simulator is also very useful for functionality and performance testing.

Java replaced C/C++ as the implementation language to take advantage of a more productive, modern platform. The added benefit is that Java also produces more portable applications. Portability, at least in theory, creates more competition among hardware vendors.

The new application server architecture does not use any J2EE functionality other than Servlets. The servers initially were built as J2EE applications using Enterprise Beans but have evolved very quickly beyond this design. Eliminating Enterprise Beans from the servers removes a distribution layer that was not useful, hence simplifying the solution and reducing latency. Like their predecessor systems the new servers still do not need to support any use-cases that would require transaction semantics, interact very little with RDBMS systems, and typically do not use Message Oriented Middleware to communicate with mainframe systems. As result there are few benefits to using Enterprise Beans.

The J2SE-based application-servers are stateless much like the original CORBA-based system. They do not retain any session or other customer-related state in-between servicing remote calls. Because SOAP does not really support objects and does not have the equivalent of an Object Reference, a new technique was implemented by the application development teams. This technique is based on the Memento[GO4] design pattern. Object graphs are marshaled into tokens, which are passed on to the channel applications. In order to reduce latency channel developers are expected to pass these tokens back to the server as optional input parameters for subsequent calls. The use of the Memento tokens is optional. The only penalty for omitting such tokens in calls is additional latency because object-graphs need to be re-materialized by calling the backend systems repeatedly. Since reducing latency is a primary focus of all development projects in the bank there are few channel developers who choose not to use Memento tokens.

The HTTP protocol used by the present generation of application servers represents a simplification over the previous CORBA-based solution. The implementation of the GIOP protocol used by the ObjectBroker ORB creates a significant overhead because tasks, such as discovering and managing available object implementations and servers, are no longer required. Typically, multiple TCP/IP connections had to be made with the ObjectBroker solution to execute a single remote call. The more basic HTTP protocol relies on "well-known ports", which can be effectively managed by even the most basic server load-balancer solutions. Load-balancers and the fact that application-servers are stateless create a very simple, but at the same time scalable and robust solution. Scalability is archived by adding machines behind the load-balancer, which increases the capacity of the overall solution in an (almost perfectly) linear fashion. The solution is robust due to the high number of redundant servers and server instances that are front-ended by the load-balancer. Load-balancers are able to detect if one of the instances fails and will stop routing traffic to the malfunctioning components. In contrast, CORBA implemented scalability and fail-over features in a much more convoluted fashion, which resulted in systems that are more difficult to understand and manage. When using the HTTP protocol only one connection has to be created to access information on a remote server, which also greatly reduces the latency of the distributed call.

The elimination of CORBA calls within the middle-tier seems to be the primary reason that the present Java-based solution outperforms the original C/C++-servers. Remote calls inside the middle-tier are no longer necessary because the "concurrent fan-out" was implemented using multi-threading rather than Asynchronous Messaging. Multiple levels of application servers are no longer required. This change represents another significant simplification. Instead of managing a large number of server processes organized in multiple levels the present J2EE solution relies on a comparatively smaller number of processes that have an identical code-base and configuration.

As a result of the relative simplicity of SOAP and HTTP it is comparatively easy to acquire or even create client and server infrastructures that support these protocols. As a result many more hard- and software platforms can be used with this approach than with CORBA. Consequently the "Any to Any" initiative mentioned before has substantially benefited from the change to simplified protocols. As also previously mentioned the downside of this large number of components that implement the web-services standards is the lack of uniformity between them. These problems are likely to make it increasingly difficult to grow the "Web-Services stack" as aggressively as the proponents of this technology hope. Similar incompatibilities seemed to have limited the propagation of CORBA.

It is apparent that within even this simplified solution the demand for "experts" has not decreased. After all, most of the problems that require management when building large-scale, high-volume, low-latency distributed application do not change when swapping protocols and infrastructure components. The analysis, design, implementation, and diagnostic skills of the "experts" are still invaluable and hiring qualified senior engineers remains a difficult task.

Lastly, "looser coupling" between client and server applications was not realized in the environment described here, even though the solution was based on modern standards that claim to foster loosely coupled architectures. It appears that there is little flexibility to foster "looser coupling" built into standards such as SOAP. A low-latency environment with clients and server using SOAP still depends on a well thought-out and executed versioning strategy. For example, XSLT and similar technologies

do not seem to work in environments where adding 100ms to a call is considered a drastic and unacceptable increase in response time.

3. THE FUTURE

Building high-performance, mission-critical applications for the enterprise remains a complicated undertaking involving costly analysis and design. For most project teams replacing distribution technologies with a more "fashionable" alternative is little more than a distraction from the task at hand. In many cases it is hard to see how the end-users will significantly benefit from this type of change.

Instead of chasing the latest buzzword or industry trend, it seems more productive to use good software design principles to strictly isolate the distribution aspects of an application from its actual core, the business logic. The strictly layered architecture[POSA1] is an invaluable tool for implementing applications in this manner. Using this type of design is initially more expensive, but it helps to isolate many types of changes. It typically only requires a small sub-team to implement a new "distribution layer" when support for another distribution technology is needed.

To promote this approach a new generation of tools that automates the creation and maintenance of "distribution layers" would be extremely beneficial. Instead of utilizing the classic methods of "skeletons" & "stubs", or the interface conventions of technologies like EJBs, these tools should be able to work off existing code. They have to be able to add the bindings required to support distribution technologies without introducing any modifications of existing code. Aspect Oriented Programming[AOP] seems to have demonstrated the feasibility of this type of tool in other areas of software development. These tools would enable "application transparent distribution" and eliminate the need for developing and maintaining "distribution layers", without loosing any of their benefits.

4. REFERENCES

[POSA1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons, 1996

[GO4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Addison-Wesley, 1995

[MDA] Paul Harmon. MDA: An architecture for the e-business era. http://www-106.ibm.com/developerworks/webservices/library/co-omg/, 2001

[AOP] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. Proceedings European Conference on Object-Oriented Programming, 1997