CSSE 377 – Software Architecture & Design II

Software Architecture in Banking

A Comparative Paper on the Effectiveness of Different Software Architectures Within an Financial Banking System

JD Hill
Andrew Robby Kruth
Joe Salisbury
Sam Varga

11/9/2010

Introduction

Software architecture design is an important aspect of our daily lives whether we know it or not. There are so many different systems that we interact with on a daily basis that we do not recognize are closely tied to software architecture decisions. These systems handle things from stoplights, electricity, televisions, communications, and many other things. Another example of a software system involved with our life is a system that powers banks and financial networks across the globe. When implementing such a system, there are several things that need to be considered.

The design of any software system needs to be thought out and have certain aspects considered from the beginning. By choosing specific quality attributes to be designed into the architecture, there is a greater chance that the system will be successful. Specifically for a bank system, there will be several important attributes to focus on. First, the performance of the system needs to be high quality. Bank workers, people at ATMs, and bank administrators will all be interacting with other systems and so the new system will need to perform fast enough to allow everyone to complete the necessary tasks. Also, the reliability and security of the system are two of the most important thing to focus on. The system needs to be reliable such that it does not crash and has a very high uptime. The security is important because banks hold a large amount of private information. If this private information became available to people who should not have access, the company which made the system could face a lot of legal issues due to the breach of privacy. A bank system needs to make sure that people who are authorized to get in the system have the ability to do what they need to, and more importantly keep unauthorized users out of the system and unable to attack it. This system needs to be able to withstand many different types of attacks. Financial systems are often attacked for various reasons. Hackers envision getting access to bank systems and becoming billionaires in a second. Others could access personal information to aid in identity theft. It is evident that security is a major concern when developing bank systems.

This particular bank system will have to incorporate aspects into the architecture design to support many different features. Bank workers and customers of the bank will need to be able to complete different types of transactions using the system. Also, the system needs to be able to be accessed from different environments and different devices. For example, the computers at the actual bank and other organizations need to interact with the system and the data that is being stored. Also, customers will likely be accessing information about their bank statements from their own personal computers. An additional feature that may be needed is the ability to have a mobile application for the bank system. In the current day, mobile apps are very popular and many customers want to have access to their information wherever and whenever they can. The system should be designed with several spots that will make modifications and adding features to the system as easy as possible in the future.

Software Architecture Comparison Model

In this paper, we will look at several different software architecture strategies and compare and contrast different elements of them. Each different strategy has its own benefits, but which one to go with really depends on what the specific system is going to be used for. To go about analyzing the different methods we will use the Software Architecture Comparison Analysis Model. This is a model that is sometimes used to look over and determine which software architecture style to go with. The first step is preparation. During this step you need to analyze what business goals you are trying to achieve. Second is criteria collation. Here you take the business goals you decided upon and translate those into quality attributes of the system. Third you need to determine the extraction directives. This means determining what architectural views, tactics, styles and patterns you are looking for based on the previous step. Fourth, view and indicate extraction. This step extracts the architectural views for each candidate according to the specific pieces chosen in the previous step. Fifth, you want to score the ability of each candidate to meet the criteria you have chosen. Finally, you want to summarize the results and provide a recommendation on which process to choose from.

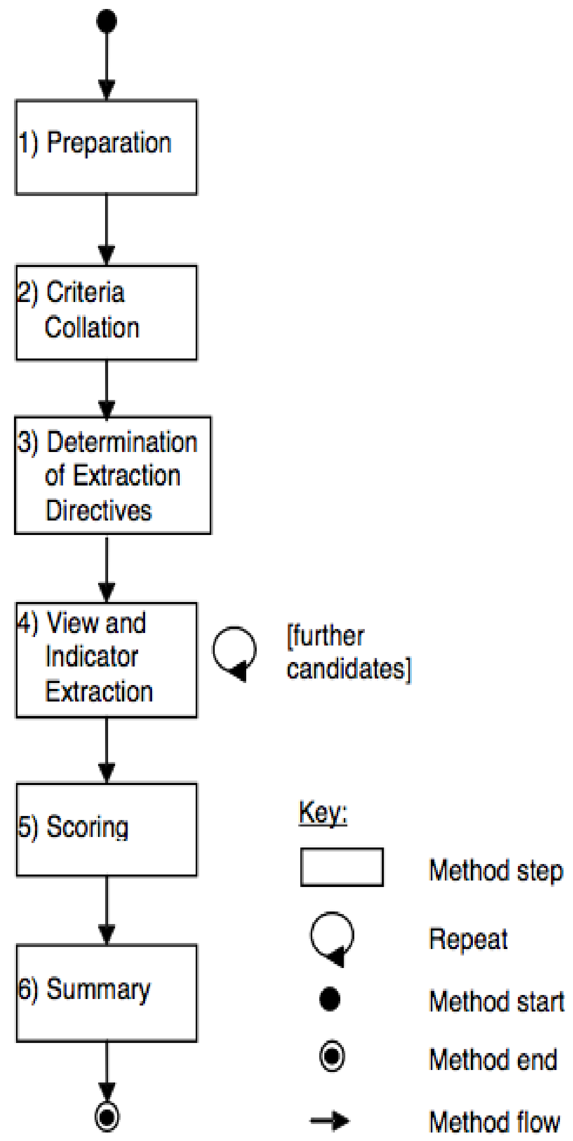

Figure 1: Diagram of the Software Architecture Comparison Analysis Method

Key Business Goals & Quality Attributes

We have already decided that the key business goals of the system will be to handle a wide variety of customer needs. They cover things from all spectrums. One is the ability for the customers to have ease of mind that their data is safe when they use the system. Another aspect that is important is the ability for the system to be accessed on a variety of devices. These devices can range from desktop systems at a bank, personal computers, and smartphones. All the different systems will provide different business values because they all fit a specific niche of customer or stakeholder. Due to these different interactions and use cases for the system, it needs to be easy to develop on and make several different user interfaces. These ideas were then transferred into the different quality attributes described earlier.

These different quality attributes will bring out different parts of architecture documentation to meet the system's needs. With security being a big issue, we will want to make sure that the architecture document shows the necessary aspects that make the system secure. This could be encryption algorithms or specially designed components that make security breaches less likely. For the modifiability of the system, we will want to have a component catalog that goes through each aspect of the system and outlines what its functionality is. Within this document, we will also want to see the types of feature that could possibly be added to the system and where they would recommend additional components be built on.

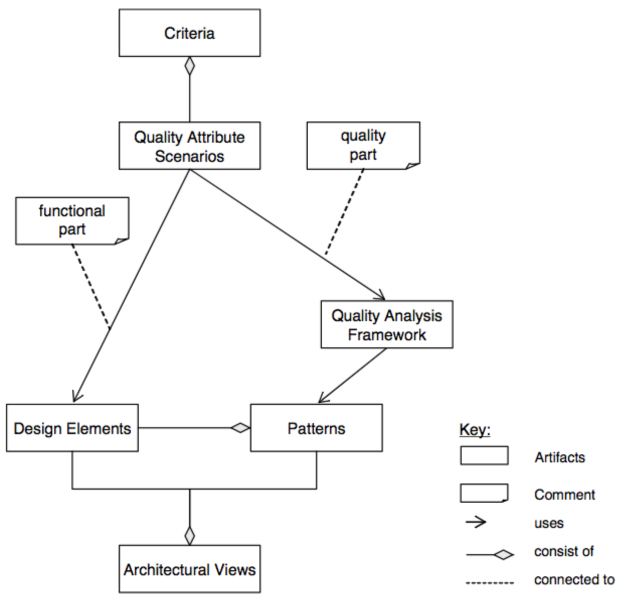

Figure 2: A Flow Chart of how criteria turns into architectural decisions and documentation

Database-Driven Architecture

One of the most prominent software architecture design strategies is a repository-based system. There are two different parts of repository based that we will look at: Blackboard architecture and Database-driven. Both of these styles have some common aspects, but also differ somewhat. The main component of each system is based on a central data structure. This data structure can interact with other aspects of the system. Also, the composition of these systems allows there to be independent computational elements. This can allow several different processes to run at a time and use data from the central repository. The ability to have data accessed to a wide variety of processes at a given time is one of the most important aspects of the repository strategy.

Database-driven architecture relies on a central database to build architecture on and support the rest of the system. Another important aspect of this strategy is that the input into the database selects which process to run from there. Within database-driven design, there are several different paths that one can go down.

Relational Database Management Design

First is what most people think of right away when they hear database-driven architecture. This type of architecture uses a relational database management system. Throughout the years there have been a lot of development put into Database Management Software and many developers and organizations are becoming dependent on these tools. This style of architecture also allows for rapid product development due to some of the existing tools that are already out there. There is a wide user base already available and some are highly skilled at developing new features for custom systems. From a security aspect, the Database Management Software can provide a lot of options and levels of security. There are several large companies that use this sort of architecture and have been able to implement the necessary parts into making the system very secure. The software can provide different levels of authorizations and checks on the table to hold up against several different attacks. In addition to that, encryption is one thing that allows these systems to be so secure. If a hacker were able to break through and get access to some tables, they would be encrypted using some algorithm. This would add another level of security to the system, which would be very important to a bank. The architecture documents for this strategy would be closely tied with the database management software. It would involve user manuals and other aspects that describe how the systems fit in together and works.

Stored Procedure-Driven Design

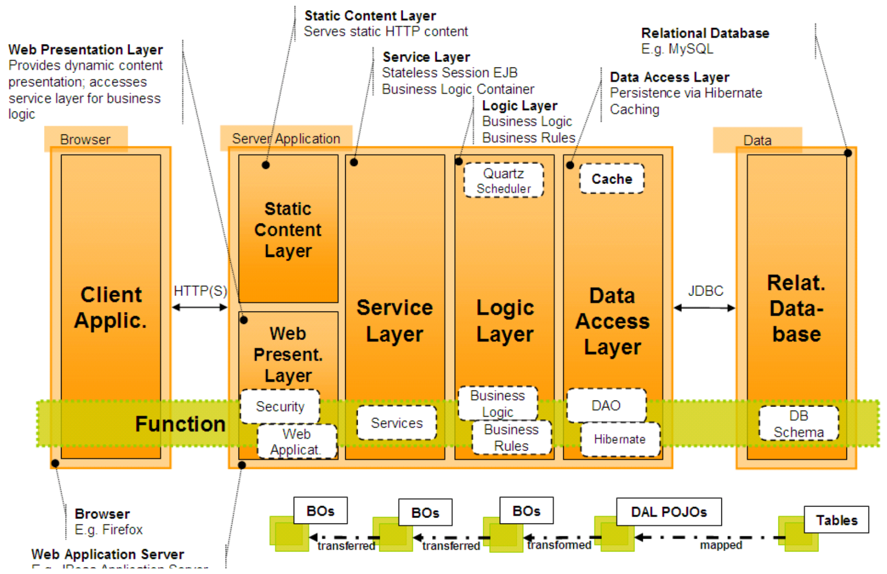

Figure 3: An example of database-centric architecture. Everything is based on access to the Database

Another way a system could be used is to have a database on some servers that has stored procedures which run certain processes. There is some debate as to whether business logic that is stored in these procedures should be run on the back end versus another level of the architecture. However, some situations make use of this technique very effectively. Some security issues could arise with this method as well. For example, if a hacker were able to get into the back end system they could modify some of the stored procedures. Since these procedures are run behind the scenes, intrusions could take some time to make themselves evident to the users. This could potentially cause big problems for a bank system. By the times changes to stored procedures are noticed, a hacker could have had access to large sums of money or to confidential data. Also, for this setup the technical department at the bank would need to have some experience with databases and stored procedures to really make full use of the systems capabilities. Writing and using stored procedures can become fairly complex. There would also need to be heavy documentation on how to use the actual system and what the backend procedures are capable of.

Dynamic Table Logic Design

Using the database as a dynamic, table driven logic is another variant of database-centered design. The extent of this really depends on what types of logic the bank would want to put within the database. It could be very complex or fairly simple. This allows for a large amount of flexibility for the uses of the system. This concept also opens up for features of dynamic programming languages and control tables. These tables are normally embedded within programs as data structures but can now be moved to the central database. The architecture documentation for this would vary depending on how complex the system becomes. Most likely the technical staff at the bank will not be familiar with this architecture so that could pose some problems for them.

Use of Communications Between Processes

A final database centric approach is to use the central database as the main method of communication between different processes in applications. This capability allows for the benefit of distributed applications that simplify some of the database management software. These systems can work better with transaction processing and indexing of activities. This allows for high reliability, performance, and the ability to handle a large volume of traffic. These benefits would be very important to a bank system. The reliability and security of the system have been detailed earlier in this paper. It is one of the most important parts of the system and therefore this could be a major reason to choose this architecture style.

Blackboard (Repository) Architecture

Blackboard architecture is another variation of a repository-based system. The main difference between blackboard and other database designs is that the central data structure selects what processes to run based on the inputs. The decisions that the central database makes are based on the data that is shared with the blackboard. Therefore the blackboard style is similar to a data warehouse in that is houses and shares all the data among different things. It uses this knowledge to go about deciding what actions to take due to various inputs. This is similar to implicit invocation. The users of the system do not call specific actions from the system but they are giving some sort of input and expect the system to determine the correct output. The control is driven solely by the blackboard. Due to this, the context of the inputs is very

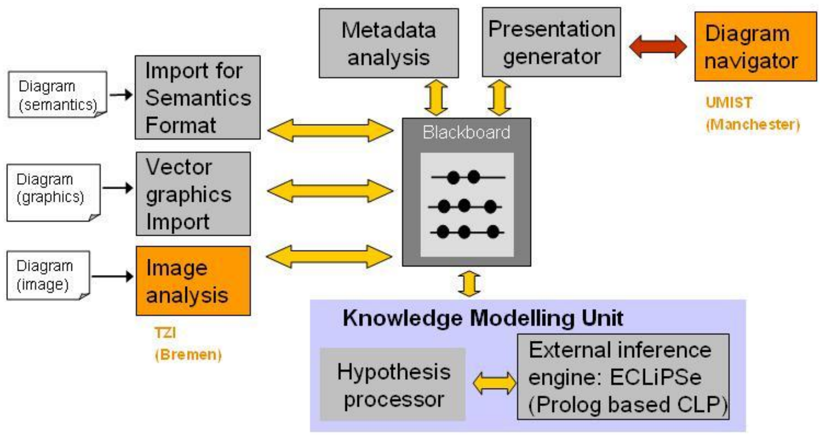

Figure 4: A sample of the architecture design of a Blackboard system

important to the reasoning and processes the blackboard chooses. In most cases that use blackboard architecture, there is complex interpretation of data.

A common example of a blackboard system is a speech recognition system. The users could speak something into the system. The system could then interpret the input and output different things to the user based on the different systems it is interacting with. This could be things such as grammar corrections, language, dialect, or even the name of the individual speaking, Blackboard systems can be very complex but they can achieve some pretty amazing results as well.

This type of architecture is also common with implementations of artificial intelligence. In cases of AI, the blackboard is updated by groups of specialists that start with problem specifications and end with solutions. Following various iterations of this accumulation of knowledge, the system has a wide range of data and problem solving tactics to pull from. Therefore when it is presented with some sort of new situation, it can go through complex algorithms to intertwine problem solving tactics from the various situations and come up with its own solution to the problem.

Using an architecture pattern such as this for a common bank system may be a bit over the top. While there are definitely some interesting directions the bank could go in by using this system, it would most likely not be the most effective way to go about meeting their business goals. In addition to this, it is an uncommon design pattern for this industry and the technology managers at the bank would likely not have prior experience with a system of this type. This would complicate maintenance and modifications to the system. In addition to that, the architecture documentation would be very complex for this system. It would be difficult to make effective documentation that would suit the target audience that would need the documentation.

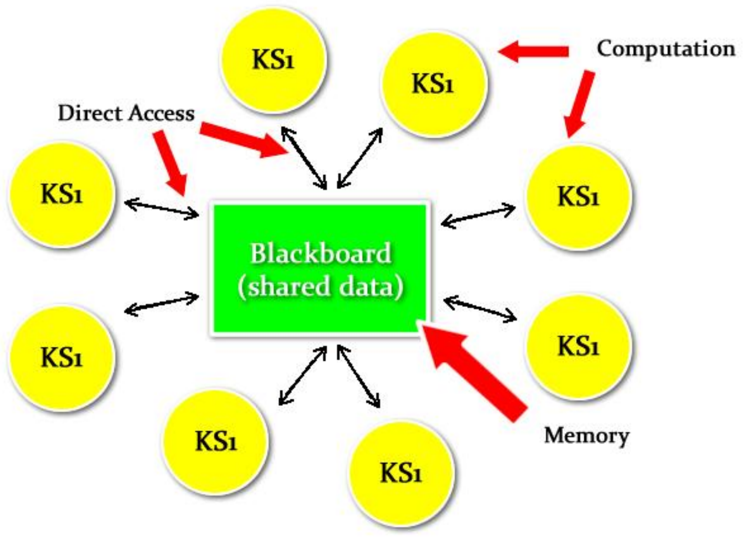

Figure 5: Depicts the ability of a Blackboard system to accumulate knowledge from various sources into one output.

Client-Server Architecture

A banking software system depends on several individual components to completely work. All of these components must be linked together. Each connection must be independent of the other connections in the system. All of the individual connections however must share the same information in a real-time environment. With today’s electronic banking standards, data must be processed nearly instantaneously and simultaneously across the world. Anytime a system is designed that involves such a distributed network, there will always be communication between the nodes of that network. While most designs can be adapted to fit this model, only one architectural model can boast the true standards of these aforementioned design specifications. That architecture is *Client-Server.*

Client-Server architecture presents a number of benefits for any software development team. These benefits can be seen from the beginning of design all the way through maintenance of a system. Client-Server offers a number of potential contributing factors that propels software applications to the enterprise level of distributed networks.

Client-Server Basics

This portion of paper will present the details of the Client-Server architecture in general terms. The particulars associated with the Client-Server model will be discussed to get a grasp of the advantages and disadvantages associated with this type of architecture. These principles will then be applied specifically the design of a banking system.

The Client-Server architectural model takes advantage of its distributed application structure. This structure divides the actions of the software into two key groups: providers, known as servers, and requesters, known as clients. This separation of application tasks allows for a more powerful application. In most cases the server machine(s) are usually more powerful and can perform much more advanced tasks that the individual clients may not be able to do. As shown in figure 6, the Client-Server model in its most basic form is built around a single server that provides information to clients that are all the same. The centralized server runs the portion of the application that can do any combination of data hosting, performing calculations, tracking data, or various other application processing tasks.

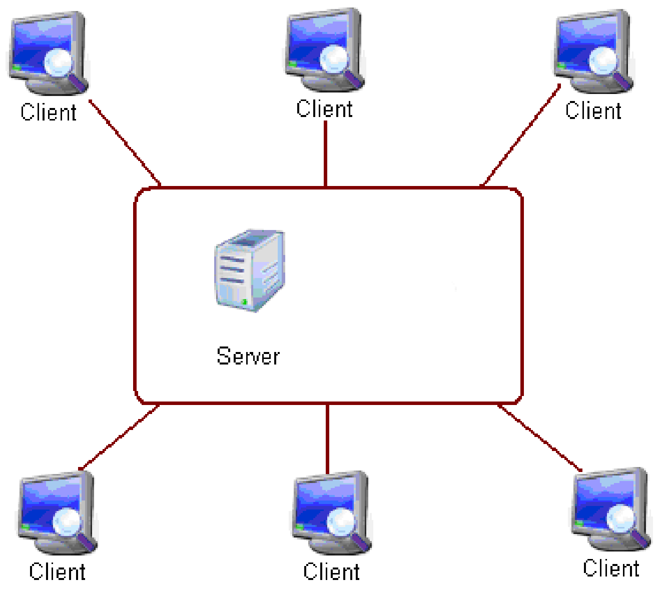

Figure 6: An example of the Client-Server architecture model in its most basic form.

The client(s) and server(s) communicate across some sort of network. The Client-Server architecture's basic principles that define the separation of processing tasks into the client and server nodes allow for a communication layer in-between that can vary drastically based on the type of software system desired. For instance, the Client-Server model can be seen in today's video game consoles. The wireless controllers can sync with the console using either proprietary radio wave communication or even the standard Bluetooth protocol. This allows the controllers to send inputs to the console over the air and sometimes even responses sent back for vibrations to the controller from the console. In addition, Client-Server models use standard TCP/IP connections for the network communications. No better example can be seen than the World Wide Web. Millions of webpages are hosted on dedicated webservers and even more client machines can access webpages from all over the world simultaneously. With the huge scope of the World Wide Web in mind, the Client-Server model's extensibility comes into play.

As the Client-Server model is extended, utilizing different devices with varying hardware and capabilities comes into play. Client devices can vary in the implementation of software or the utilization of hardware. This allows for different types of tasks to be performed on each individual device. Again, a great example of this extended architecture can be seen from the World Wide Web. The web provides a strict set of protocols for displaying information in the form of webpages. These protocols, such as HTML, can be interpreted by following a set of rules. With this in mind, the webpages that are hosted on a server can be retrieved by any client that can successfully request this information. This allows all of the clients to run independently from the server. Thus, virtually any hardware solution with internet access can communicate with the web servers. This hardware independence allows for web browsers to be implemented on almost anything. In today's world we see cell phones, televisions, DVD players, video game consoles, computers, and many more items with access to the web. The core principles of the Client-Server model are behind all of this. Without the principle of steady providers, all of the individual nodes could not access the available information. With these factors in mind, is where the model can be fully extended into a multiple server system. Multiple servers can be available that perform different tasks or the mirrored tasks to further increase usability.

The basic principles of the Client-Server model provide a few key advantages as well as some noticeable disadvantages that must be mitigated. Advantages can be seen a number of locations. Most of these advantages can be tied to quality attributes that affect systems that follow this model. An obvious advantage can be seen in performance. Hosting data or performing processing on a more powerful server machine can speed up an applications response time and distribute the workload much more effectively. Also, improvements can be seen in reliability and maintainability. When the client systems are not dependent on one another the clients can be taken offline for updates or anything else without affecting the server or other clients. This allows for greater uptime and a much more effective application. Another advantage of the Client-Server model can be seen in the area of security. Centralized servers are often easier to secure. If the data needs to be protected from unauthorized access, having it stored in a specific location allows for more effective protection measures to be implemented. Security patches and updates only need to be pushed out to the servers instead of all the clients.

With the Client-Server architecture, there are indeed some disadvantages associated with the design. Anytime a Client-Server system becomes so large and numerous clients are trying to access the same server for the same tasks or set of information, there can be a bottleneck created. Servers must be built with adequate processing power and the network that connects the server to the clients must have sufficient bandwidth to support the communications. Steps must be taken to reduce this problem. The most common way is to have multiple servers implemented to break up the processing or the data storage for different components of the system. Finally, the most obvious disadvantage of the architecture is the key dependence from the clients on the server(s). Since all of the client nodes depend on the server to perform tasks or store information, if the server is down the entire system cannot function completely or maybe at all. With this in mind, caution must be taken to insure catastrophic failures to not take place.

Client-Server Architecture in the Banking Industry

The Client-Server architectural model presents designers with several key benefits and a few disadvantages that can be managed and reduced. When the Client-Server model is looked at from the banking industry's perspective, several contributing factors can be seen that allow for the model to have a great positive impact in the domain. The banking industry as a whole has moved towards almost an entirely digital locale. As this digital revolution has taken place, banks have been scrambling to keep up with ever changing customer demands. Consumers are constantly looking for more access to their information, access at faster speeds, and access from virtually any device they own. Client-Server architecture has been at the heart of this ever changing software landscape and allows for some key components of these systems to work effectively.

In the early stages of digital transformation, Banks were able to keep centralized records of account information that was accessible at either a banking center by a teller or a remote automated teller machine. At any of these two locations, account holders could check balances, withdraw funds, and often deposit funds. The Client-Server architecture was clearly utilized in the earlier stages of digital banking to implement this style of system. Teller systems (automated or personal) were the client nodes in the model and the centralized data stores were the servers. This basic foundation was in place in almost all commercial banks. This laid the groundwork for larger more extendable banking software to come. Without the banking industry's roots of this distributed system, the advanced Client-Server layouts we see today could not have been accomplished.

As technology has improved and access to the internet has grown rapidly, the banking industry has begun to expand the software and systems that supports the everyday management of all sorts of transactions. Customers now have more access than ever to their money and all of the fringe benefits that banks offer to their account holders. "Online banking" has become a part of everyday life. Customers can now login and see account information and perform all sorts of tasks. Funds transfers can now be made over the internet and not in person. People can move money from once account to another account within the same bank or even transfer funds across accounts from different banks. Customers can also view their past account history for better budgeting and export their records to third party financial software to merge personal and business expenditures.

Customers can do all of these things from a number of different portals. People can perform online banking from personal computers, smart phones, televisions, and even music players. At the heart of these technological breakthroughs, the Client-Server model can be seen. The hardware independent structure of the architecture provides software designers with ultimate flexibility in the implementation of new interfaces. The vast array of hardware devices that contain software applications that provide financial interfacing to the banking industry have been all designed around the ability to access the data and issue tasks to and from a centralized server. Through the utilization of an application programming interface (API), developers can implement all kinds of clients to access the same network.

As the architecture is expanded, more and more servers are added to the center of the Client-Server model to provide additional functionality to the banking customers. The concept of a centralized cloud allows for the clients to access these servers without actually knowing how many or which servers they are communicating with. This allows for the processing to be broken up into even further task specific nodes. The layout of this concept is seen in figure 7. The banking industry takes full advantage of the model to distribute its application task and processing to insure performance, reliability, and security. To keep up with the ever changing technology software developers must remain ahead of the curve.

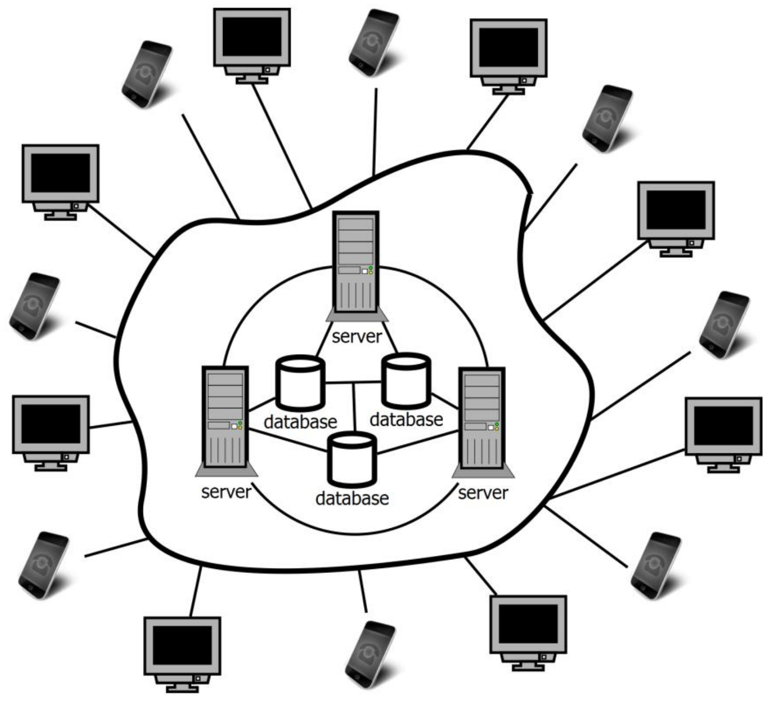

Figure 7: Client-Server Architectural Model in today's Banking Industry

Relaying on strong server standards, help the core functionality of the Client-Server design to remain static for as long as possible. This allows the model to be extended into future devices that have never been used before to access the mobile banking applications. As the industry continues to mature and expand, developers will be able to keep up with the demands of the customers for more functionality, higher performance, better security, and increased flexibility from the banking software. Even as technology changes and new architecture patterns are introduced into components of the system, at the core, the Client-Server model will still be involved due to the nature of hardware variability of accessing financial information. In short, Client-Server architecture has allowed the banking industry to make a giant leap into the digital age of computing.

Three-Tiered Architecture

As all of the components of the client-server architecture need to be connected over a network from server to server, performance problems arrive with bottlenecks on the server side of things. This problem could cause a system to be a failure or not be able to adapt well over time. A natural solution to this problem would be to try and remove those bottlenecks by splitting the server side of things into several servers, which all do different things with the data. This approach is called the multi-tiered architecture. The three-tiered architecture is the most common of these, and will be evaluated for its applicability to a banking software system.

The Tiers

The three "tiers" of the three-tiered system are the presentation layer, the domain layer, and the technical services layer. In a good three-tiered architecture, the layers will be coupled to each other as little as possible by using the Controller and Adapter software patterns. Because of this, the layers are very modular in form, which means that they should be able to easily swap out multiple user interfaces and technical services components (i.e. data storage structures or authentication services).

The presentation layer represents how the system interacts with the users. This is basically the user interface(s) that the system will contain. A system can have a variety of presentation layers, ranging from a command line to a touchscreen to the holographic interfaces seen in a lot of sci-fi movies. The main quality attribute inherently associated with user interfaces is usability. The different types of presentations available to a user allow flexibility, but all of the interface implementations must be usable for a successful presentation layer.

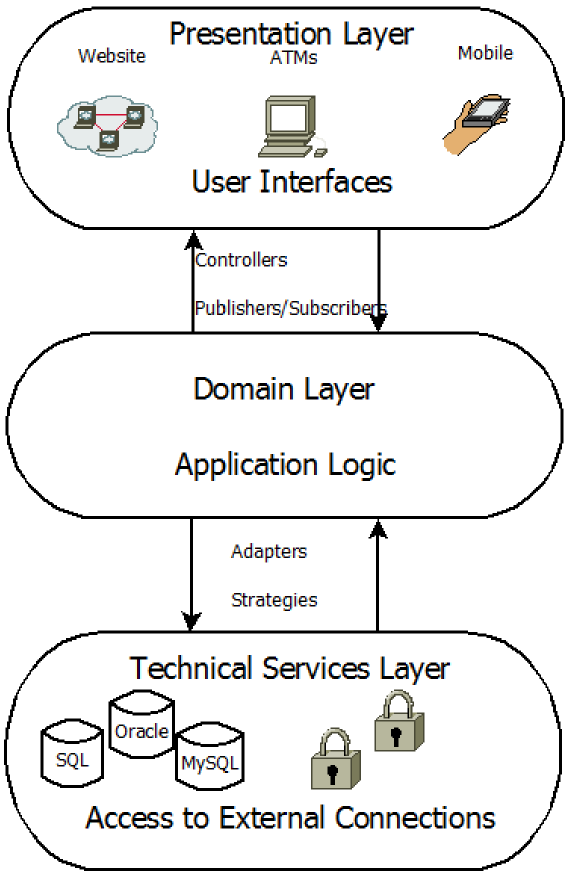

Figure 8: The three-tiered architecture as it would be applied to a banking system. The data flow between the layers are labeled with the patterns used as connectors

The domain layer houses all of the application business logic, which means that this is where all of the processing of data takes place. The domain layer is the heart of the system; the other layers are periphery to the domain layer. This is the layer in which a product line can be developed. This is what the banks are really paying for when they ask for an accurate system; other components of the system are usually outside functional requirements and main business drivers for the project, and are easily contracted out. The domain layer also contains Controllers and Adapters which connect to the other layers in the proper way, which is what makes a three-tiered system so thin (i.e. makes it very modular and easily modifiable). It is also a channel through which data from the technical services layer reaches the presentation layer.

The technical services layer contains all of the parts of the system that the domain layer relies on for interactions that are very low-level from a systems standpoint. This would include network protocols and data-persisting mechanisms within the system. Although these components of the system are very important and can be difficult to implement, there are many existing solutions that the development team could probably just write Adapters for within the domain layer in order to save time (and quite possibly money). The technical services layer in a good three-tiered architecture should be completely invisible to the user; he or she should just know that the system is getting its data correctly and it is communicating appropriately to the external connections.

Currently, three-tiered architectures are very popular in the software architecture world. It is very similar to the client-server architecture, except that extra application layer really gives better fit for adaptability to different technologies on the technical services level of things. One downside to the three-tiered architecture as compared to the client-server is that since the business logic is separated from the technical services layer, the communications are a little harder to master between the layers. This trend increase substantially with each layer added to a system, and must be weighed in comparison with the adaptability and performance benefits which can be gained by adding that extra layer.

Three-Tiered Architecture in the Banking Industry

As noted above, the divisions of layers for the three-tiered architecture make it a great option from a flexibility standpoint. For instance, if a bank used a legacy system to store their data and wanted to migrate all of their data to a more up-to-date system, developers would just have to add a class that implements an Adapter to communicate with the data storage system.

In a similar vein, the presentation layer sits on top of the application layer and receives information through controllers. This means that the application layer should be able to take user input from any medium and process it without any problem. With smartphones as the one of the primary means of communication and data transfer, portability is a very important aspect of software systems that a bank would have to take into account. The three-tiered architecture allows mobile apps to be developed very easily provided that the developers have a well-documented architecture of the domain layer and know by which channels to input the data.

This fact also makes the system very scalable, which is a rather important quality attribute for a bank. Since the technical services (especially the data storage facility) reside on their own server, none of the rest of the system would have to be touched for the data store to be increased to support more customers (except for a couple of Adapters within the domain layer possibly). This is also positive from a reliability standpoint—Adapters can be written for new technologies or a better network provider, and then the switchover within the technical services layer would create as small a downtime as possible for the entire system.

The three-tiered architecture also makes a great business decision. Since the domain layer is very thin, it can easily be developed as a product line. Multiple components from both the presentation layer and technical services layer and simply be plugged in to the application layer. Suddenly several banks are using the same product, but all of the different system have different feels due to the custom implementations of the user interfaces. The application layer being so thin would also mean that the development team could focus more on the business requirements rather than periphery aspects of the system. For instance, the storage of the data could be kept in whatever form the banks currently has as long as the domain layer has the drivers and security authorization to access it. This would mean that the architects could deliver a product faster and for less money to the client.

This is also good news from a security standpoint, because whoever makes the design decisions could outsource the security for the system and just implement an Adapter that could negotiate authentication with the security system. Of course, if the security were outsourced to a big security company many people are familiar with, the potential to open the system up to hackers increases. It can also be argued that this security will be tighter than, say, security measures put in place by the company developing the entire system, since it is less likely that they can account for all of the possible attacks the large security-specific firm has surely dealt with over time.

One of the biggest problems with a three-tiered architecture is the performance. A three-tiered architecture will be object-oriented, which means the data flow through the system will likely face many layers of indirection within even one of the larger architectural layers. Considering that the data flow already needs to come from some external data store, then through the various channels within the domain layer, which may take some time-intensive business calculations and algorithms, and finally be presented to user may leave them wondering why the system is taking so long to respond. Not only would this happen for data travelling through the system, but the security aspects of the technical services layer would also take longer to authenticate users. Although performance wasn't one of the main quality criteria to consider the architecture appropriate for the bank system, it is nonetheless important to customers and can make a big difference to the satisfaction to users of the bank system.

However, compared to the client-server architecture the three-tiered architecture can improve performance. Although the data and technical services layers are physically close in a client-server architecture (thusly increasing performance), for large server loads the architecture tends to perform poorly because of bottlenecks due to database calls and business logic all happening at once. The three-tiered architecture gets rid of this problem, allowing higher throughput. It is more likely that a bank will serve a large amount of customers, so the having data go through another layer instead of keeping all of the domain aspects on the database side of things will end up being a performance saver in the longs run.

One advantage to the three-tier system from a project communications standpoint is that the three-tiered architectural is relatively straightforward to nontechnical people. It would only take a couple minutes to explain how the system works and why a three-tiered architecture is an advantageous model to use. In fact, of the architectures analyzed in this paper, the only more straightforward model would be the client-server architecture, which many people are already familiar with in their exposure to technology on the internet.

On the flipside of this, if the client trusted the development team for the system, and the three-tiered architecture were explained in a way that would make him or her think that only the main business logic would be developed by the team, with many other components outsourced, the client might be wary of the architecture. For instance, he might believe that the other components won't be as well-documented as the domain layer being developed by such the stellar design team. Trust is a very important aspect of software management, and, depending on a customer's openness to working with even more companies in developing this system, the three-tiered architecture might diminish some of this trust.

One more criterion to note with the three-tiered structure is the ability to handle database transactions, since that is probably the largest aspect of a banking system. People all over all the time will need to perform transactions and change data in tables within the database. The three-tiered architecture certainly isn't the worst type of architecture for this requirement, although it obviously isn't the best. As noted above, some of the database-driven architectures are specially designed to handle this large load of concurrent database connections. However, the three-tiered architecture will still outperform the client-server architecture for the same reasons listed in the performance section.

Another advantage to the three-tiered structure is that the amount of space required on the client (ATM, mobile phone, website, etc.) is minimal. This is because only the UI logic is stored on these machines, whereas an entirely separate server is used for the application logic. This is an advantage especially for mobile phones which do not have a lot of space for data. A light application would mean that more people would be willing to use the app, allowing the bank to attract more potential customers and flourish from a business standpoint. Newly-produced ATMs could benefit from this small space needed for the UI layer because the bank could spend less money on the hard drive for the ATM and more money for either performance or the development of a highly usable interface design, which would again lead to reduced cost or more happy customers.

Space Based Architecture

Space based architecture (SBA) is an architecture system built off of self-sufficient, independent parts called processing units. SBA falls under the R.E.S.T. style of architecture and is based off of the tuple-space paradigm and uses Object space to help with transactions. This system is great for scalable systems because more independent processing units can just be added to the system to increase functionality. In this same way your system becomes more manageable when the code is already broken up into self-sustaining pieces. It is also very fast, since partitions of the system do specific transaction code, you don't have to pass through the whole system to get things done. Availability is made easy when module based systems like this are employed. In terms of our banking software this is a fair choice of architecture, and supports a ground up approach at development.

Representational State Transfer: R.E.S.T.

Representational State Transfer, or R.E.S.T., is based on clients and servers. Clients can post request to servers and servers do work and return appropriate responses to the client. Requests can be any coherent representation of a resource that can be used to change the state. This is where the PU's come in; they perform actions on the state and then push them out again. PU's act as resources for the server to perform actions with. Responses are the representation of the state or next state to be had. At any point in time clients can either be transitioning their states, or be at "rest". While at rest the client can interact with its user but has no server memory or load time. At any time the client

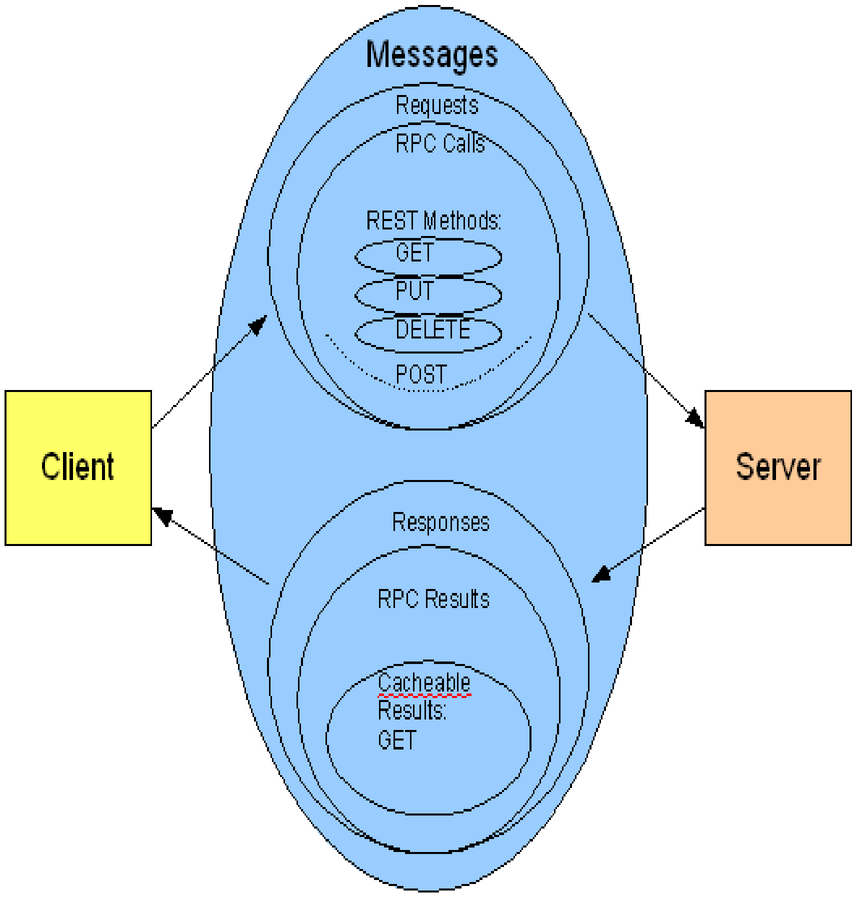

Figure 9: Diagram showing the basic flow of REST as applied to a Client-Server environment.

can send a request to the server to change or transition to a new state based on user response. The diagram on the right shows how a client becomes at rest and send requests to the server with actions to perform. The server performs those actions and returns an appropriate response. The cycle moves on from there.

Object Space

Object Space is a service oriented strategy providing a distributed object exchange and coordination mechanism for objects. It is used to store the distributed system state and implement distributed algorithms. In an Object Space, all communication partners, or peers, communicate and coordinate by sharing a state. Object spaces use the Master-Worker strategy. The Master system pushes objects or actions out to the "space". Then the associated workers read these objects and take them in. When the work that needs to be done is processed it is pushed back out to the space with a new set of actions that need to be performed or location to be transferred to. In this system the PU's of the system are treated as resources that the workers use to process and change the object in the space. There are really only four actions performed in an object space:

- *put* – Put an object into the space
- *Notify* – Notifies the workers when there is something in the space that concerns them, or that they can work on.
- *Get* – Get something out of the space, when this happens nothing else can get that particular object until the worker is done.
- *Read* – Reads in a copy of the object in the space to perform actions on but leaves the original in the space for other workers to use.

With just these four actions we can control the object space and effectively run the Master-Worker strategy.

Processing Units and Partitioning

In terms of scalability SBA is great for allowing a linear software size increase strategy to your system. SBA is made up of many Processing Units (PU's). A PU is an individual unit in the system that has a specific job and can do that job on its own. Each PU is completely independent, this helps in

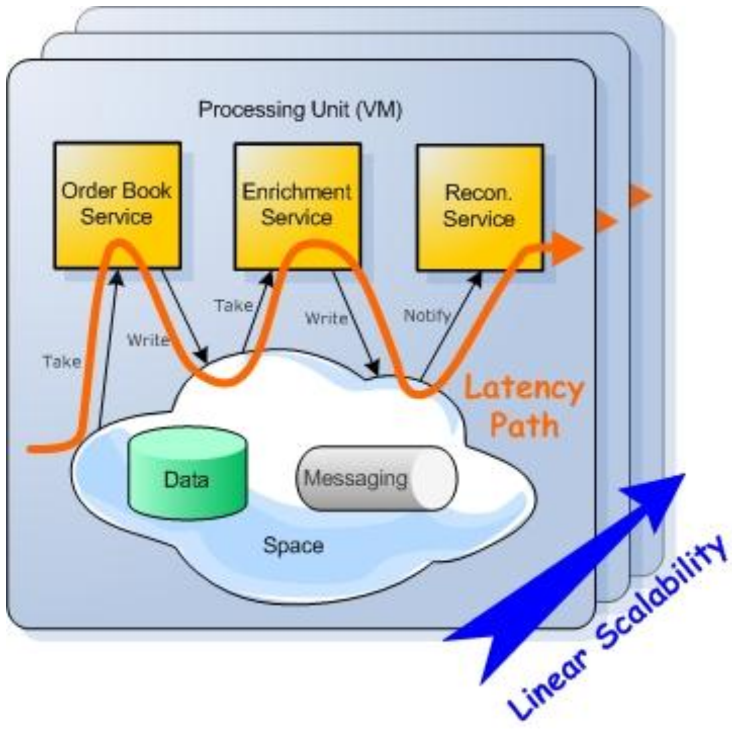

Figure 10: Processing units provide a modularized structure that is easily scalable

many ways. First off, with each PU being completely independent the coupling and cohesion of your system becomes extremely manageable. Second since PU's are independent it is increasingly easy to add new PU's to the system without having to alter and manage existing code. This is the primary reason the scalability is very easy with this architecture system. Lastly since the system is already so modular a few strategies can be employed to ensure availability fo the system. As depicted below, you can simply make an number of backups to each partitioned PU that can cover for any downed PU's. Secondly, in a more reactionary approach, with proper action out put you can pinpoint the PU that cuased the problem very quickly. This allows for quick fixes and short downtime. Either one of these solutions makes this architecture strategy extremely available and maintainable.

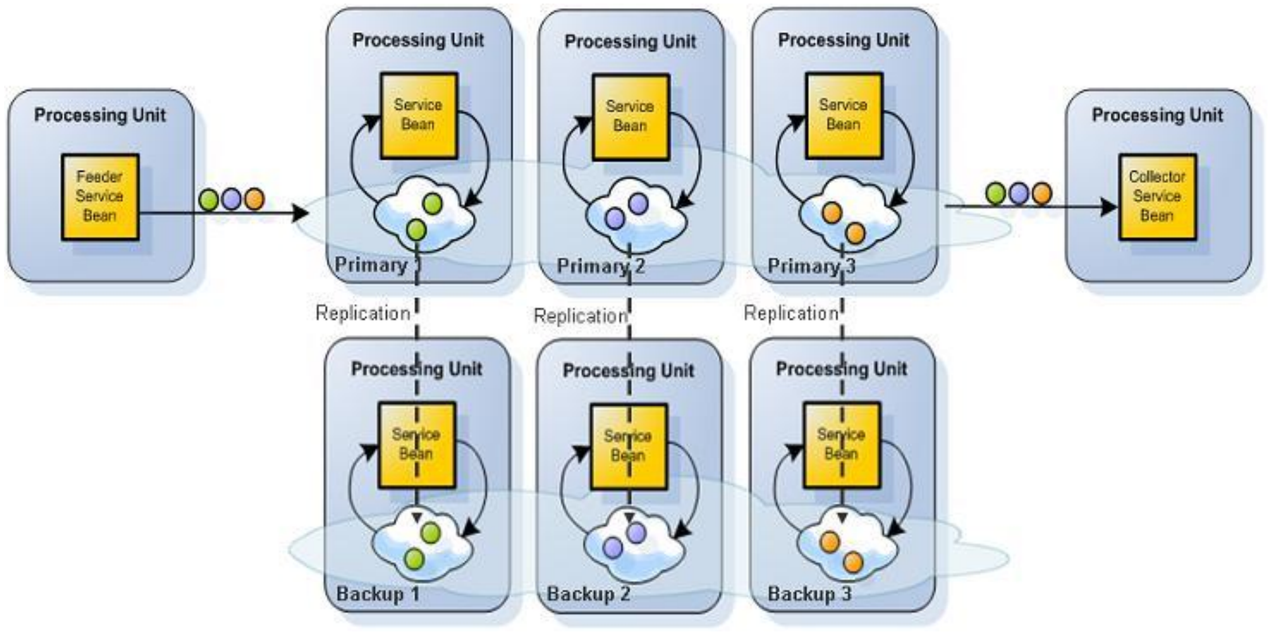

Figure 11: Processing Units can have any number of backups to provide better availability

Performance Gains using the "Space"

The performance gains of this type of architecture are three fold. First off transaction based systems benefit gratefully from this architecture because actions are performed in one place and don't have to pass into many places of the system to finish. This saves time on specific transactions if you design each PU to house specific code for each transaction type. By doing this, code completion on a transaction bases becomes very fast. Secondly, since we can do these small transactions very quickly, we can process many transactions relatively fast. Real time analytics such as that needed to do credit score calculations and insurance information processing are now fast. Lastly, the modular nature of this architecture system is a perfect environment for parallel processing for high end applications.

Space Based Banking Software

In terms of banking software development we would start small. With just a few basic features like accounts, withdraws, deposits, etc. each of these features would gain its own partition and Processing Unit in the system. As the system builds new features would be proposed and developed for the system. Mobile access, ATM functionality, online check deposits, insurances claims, 401K's and the like. All these banking features would secure a Processing unit and then added in along with all the others. These processing units would have a certain input case, like the withdraw partition would look for a withdraw instruction in objects in the space. In the far back end of course we would need some sort of database to hold all of the customers personal and account information. The space would be just a place for transactions in process to stay while being processed. From the end users point of view, you would come to a teller or an ATM and try to make a transaction on an account. All this information, account number security password, dollar amounts and so on would be packaged into a transaction request. This is the object that goes into the space. For example, here is a scenario:

I walk up to an ATM and enter a password for my account, the fact that I want to make a withdrawal and the amount.

Then the architecture system would handle it like this:

1. The three pieces of information would be packaged into a transaction object
2. This object would be put into the space.
3. The PU responsible for validating passwords would then either validate or invalidate the password, then put the response along with the object back in the space.
4. If the password was invalid, a PU would return an error message to the user.
5. But if it was valid a separate worker in charge of withdraws would take up the request and process it with the dollar amount provided.

With the example you can see the processing power of space based architecture. Actions are done easily, effectively and only when they are required to. The system is also very flexible there aren't a bunch of statements guiding the process through the system. Each worker is responsible for grabbing the object that it can work on and then returns a response. The system is also very performance minded because each transaction is handled specifically and quickly because only the required Processing Units required to do the work are actually accessed and used. This strategy cuts out the extra processing time that it would have taken to skip through all the other processing units like you would have had to do in some other Architecture systems.

Scoring of Architectures

Now that the systems have been described in detail and the benefits and drawbacks have been analyzed in relation to a banking system, the next step in the Software Architecture Comparison Analysis Model is to score the different architectures for their suitability in the system at hand. To do this, a decision matrix was formed based on the key aspects determined to deem the project a success. It is appropriate to weigh each of the criteria depending on the value of that attribute to the rest of the system.

	Support Transactions	**Usability**	**Scalability**	**Security**	**Performance**	**Modifiability**	**Dev. Effort**	**Total Score**
Weighting	***4***	***3***	***3.5***	***5***	***4***	***2***	***2.5***	
Relational DB	3	4	3	4	3	2	3	**78**
Stored Procedure	2	2	3	1	4	2	4	**59.5**
Dynamic Table Logic	2	2	2	2	2	2	2	**48**
Comm. Between Processes	4	3	4	3	4	3	4	**86**
Blackboard	2	2	3	3	3	2	2	**60.5**
Client-Server	4	3	3	4	3	4	4	**85.5**
Three-Tiered	3	4	5	4	4	4	2	**90.75**
Space-Based	4	4	5	3	4	4	3	**92**

Table 1: The final decision matrix. The weighting of the criteria in the matrix is based on the attributes of the system that are likely to be perceived as the most important to customers. For each of the eight types of architecture detail in the body of the paper, scores were given based on their tendency to succeed for each of the given criteria (with 5 points being the maximum).

As can be seen in the decision matrix, none of the database-centric architectures performed that well. The architectural approach of using the database as a basis for central form on which communications performed would be a reasonable approach for the banking system, but outperforms the client-server architecture only by a little. According to the table above, the architecture suitable for most big bank systems would likely be based on a space-based or three-tiered architecture. These are also two of the most object-oriented architectures investigated in this paper, which point to yet another reason that object-oriented development is very important to system architecture today.

Of course, other considerations will have to be taken into account before agreeing upon the architecture. Collaborating with the bank to determine their specific requirements for a project is one of the biggest driving factors for this decision. For instance, smaller banks who aren't really expecting much growth could probably get away with a simpler architecture such as client-server, depending of course on their other needs. It is also important to realize that although these architectures were given numbered scores, some of the ideas from an some architecture can be applied to other architectures, making a sort of hybrid with the good qualities of the architectures involved and as few bad qualities as possible.

Summary & Conclusion

In general, any architecture chosen for a big financial-sector system like a banking system will need to incorporate many quality attributes as mentioned throughout this paper, such as security, reliability, availability, etc. Some may be more functionality-driven, but the decision matrix above with a little tweaking will be a suitable model for choosing system architecture according to the specific criteria in most cases. With this assumption, it can be said that for newly developed financial systems of great magnitude, the architecture should generally stack up against each other as shown in the decision matrix.

However, the answer to the question, "*Which software architecture is **best** suitable for a banking system?"* is not always going to be the same answer for other types of systems. There are many systems out there for which a blackboard approach would be much more appropriate than any other architecture. Judgment of software architecture cannot be made by looking at one specific domain. Differing architectural styles are present in the software industry because there are benefits and disadvantages that each model introduces. With that said, each architectural model must be evaluated for the specific circumstances at hand.

The most important takeaway from this term paper is not what would be the best architecture for a banking system (although the research done may prove to be useful if, in the future, one of us has to work on such a system), but that there are many different architectures, and each has a proper use. In the course of the paper the team discovered a lot about the various architectures researched. The space-based architecture and some of the database-driven architectures were entirely new to the team, and allowed them to pick up some emerging ideas within the software architecture world. While specifically choosing only a few architectures to cover in the paper, the team encountered far more architectures in the initial stage of looking for potential candidates of a sensible banking system, allowing them to broaden their knowledge of different types of architectures in an even large scope than illustrated in the paper.

<u>Bibliography</u>

"Base One - Database-centric Grid and Cluster Computing." Base One - .NET Database Programming Tools. Web. 05 Nov. 2010. <http://www.boic.com/dbgrid.htm>.

Daniel D. Corkill. Collaborating Software: Blackboard and Multi-Agent Systems & the Future. In Proceedings of the International Lisp Conference, New York, New York, October 2003.

Dué, Richard T. "Client/Server Feasibility." Information Systems Management 11.3 (1994): 79-82. Academic Search Premiere. Web. 05 Nov. 2010.

Gallaugher, John, and Suresh Ramanathan. "Choosing a Client/Server Architecture." Information Systems Management 13.2 (1996): 7-13. Academic Search Premiere. Web. 05 Nov. 2010.

"Introduction to 3-Tier Architecture." DotNetSlackers: ASP.NET News and Articles For Lazy Developers. Web. 05 Nov. 2010. <http://dotnetslackers.com/articles/net/IntroductionTo3TierArchitecture.aspx>.

Michalarias, I., A. Omelchenko, and H. Lenz. "FCLOS: A Client–server Architecture for Mobile OLAP." Data & Knowledge Engineering 68.2 (2009): 192-220. Academic Search Premiere. Web. 05 Nov. 2010.

"Owen Taylor on 'Space Based Architecture' - TheServerSide.com." TheServerSide.com: Your Java Community Discussing Server Side Development. Web. 05 Nov. 2010. <http://www.theserverside.com/news/thread.tss?thread_id=42928>.

"Space-Based Architectural Thinking." CQRS, DDD, and NServiceBus Video. Web. 05 Nov. 2010. <http://www.udidahan.com/category/space-based-architecture/>.

"Space-Based Programming - O'Reilly Media." ONJava.com: The Independent Source for Enterprise Java -- Java Development, Open and Emerging Enterp. Web. 05 Nov. 2010. <http://onjava.com/pub/a/onjava/2003/03/19/java_spaces.html>.

Tyree, J., and A. Akerman. "Architecture Decisions: Demystifying Architecture." IEEE Software 22.2 (2005): 19-27. Academic Search Premiere. Web. 05 Nov. 2010.

Wikipedia, the Free Encyclopedia. Web. 05 Nov. 2010. <http://en.wikipedia.org/wiki/Main_Page>. (Used multiple articles as base line for some information gathering and to seek out other sources)