TWO-TIER VS. N-TIER CLIENT/SERVER ARCHITECTURES: TOWARD SELECTION HEURISTICS

Jennifer Sovik and Thomas Sandman***

** First Health*
750 Riverpoint Dr, West Sacramento, CA 95605
(916) 374-3813
jennifersovik@firsthealth.com

***College of Business, CSUS*
6000 J Street, Sacramento, CA 95819-6088
(916) 278-6536
sandmant@csus.edu

ABSTRACT

There are many alternative architectures available for use in information systems application development. One alternative that is growing in popularity is n-tier client/server. As its name implies, an n-tier client/server application is distributed into several logical and possibly physical tiers. The middle tier of an n-tier application typically contains the business logic, an approach that provides many benefits but also adds complexity to the application. This paper examines the benefits and costs of n-tier client/server and provides criteria for choosing an n-tier architecture versus a two-tier architecture.

LIMITATIONS OF TWO-TIER CLIENT/SERVER

A traditional client/server application consists of two tiers: the client tier and the server tier. The client provides the user interface and typically runs on PCs and other desktop machines that are connected to a LAN. The server stores shared data and can run on anything from a PC to a mainframe. The two-tier client/server approach has two options: place the business logic on the client (fat client) or place the business logic on the database server (fat server). Both of these options have potential drawbacks. When business logic is placed on the client in the form of application code, the code will have to be changed each time the business logic changes. The revised application code must then be recompiled and distributed to all of the workstations; this can be time consuming and inefficient. Another disadvantage of fat clients is the increased potential for poor response time. Structured Query Language (SQL) commands must be used to query, insert, update, or delete records in a relational database. Placing the business logic on the client requires sending large blocks of SQL to the server for processing; the increased network traffic can result in poor performance – especially if the database is located remotely and the client must send SQL over the Wide Area Network. Another drawback of having SQL code embedded in the client application is that the front-end developers must have intimate knowledge of the structure and contents of the relational database. In a typical organization, there are several core databases, each having thousands of tables; it often takes a new developer a few months to learn the tables and columns appropriate to use for different applications. To remedy some of these problems, the business logic can be placed on the database server in the form of triggers and stored procedures. When the business logic is located on the database server, the SQL is physically located in the same place as the data it manipulates, thus preventing the need to send large blocks of SQL over the network. The client application calls a database procedure, passing it any necessary parameters, and the result is then passed back to the client application. A benefit of fat server is that it helps centralize business logic. If several client applications access the same business logic through database procedures, efficiencies can be gained by modifying the logic in a single place, thus reducing the time and effort needed for maintenance. In addition, if a database procedure already exists, a developer need only know the name of the database procedure to call from the client application, thus shielding the developer from some of the complexity of the database structure. Placing the business logic on the database server may seem to be an obvious choice, but it is important to note that there are drawbacks of using database procedures for business logic. Procedures are written in proprietary SQL that has only a few additional procedure constructs in addition to standard ANSI SQL. Therefore, regardless of the database, the programming language used to write stored procedures is typically limited and cannot do nearly as much as other programming languages. Another disadvantage of stored procedures is that the business logic becomes coupled with the database vendor in use at the time. If the database vendor is changed, then all of the triggers and procedures must be rewritten in the new vendor's proprietary language. Neither fat client nor fat server provides a perfect solution. In fact, the two-tier client/server approach has serious limitations regardless of where you place your business logic. The limitations of two-tier client/server applications were revealed when attempts were made to develop large-scale two-tier applications. Two-tier client/server applications do not scale well to support thousands of users. Linthicum states "...the database servers simply can't manage thousands of client connections. The host operating system tends to thrash and die when you scale to high user loads. This is because of the one-client, one-connection requirement of two-tier client/server" [1]. In two-tier client/server, each client application running has its own thread to the database that uses host operating system resources on the server. Thousands of client connections would require high levels of system resources. Connectivity above a certain threshold will result in poor performance and perhaps crash the system.

BENEFITS OF N-TIER CLIENT/SERVER

Due to the limitations of two-tier client/server, many organizations are turning to n-tier client/server architectures. An n-tier client/server architecture logically separates the user interface, database, and procedural code (business logic) into at least three separate tiers. The business logic resides in the middle tier and may run on one or more servers. The user interface interacts with the middle-tier by remotely invoking procedures and receiving results. The concept is similar to the use of database procedures except the middle tier procedures are not limited to the proprietary SQL of an RDBMS. In fact, many middle-tier procedures are written in languages such as C, C++, and Java. The middle-tier procedural code is managed and deployed separately from the user interface and database. This design makes it possible to modify one of the tiers without affecting the others. In addition, the interface, database, and business logic are often physically distributed to run on many different machines. When an n-tier application is distributed effectively, system resources are more efficiently used and applications have better performance. N-tier client/server is able to scale to thousands of users because it does not have the one-client, one-connection requirement of two-tier client/server. N-tier client/server applications use middleware to funnel client connections to the database server. Middleware can be thought of as the glue that holds an n-tier application together. Middleware products, such as Bea's Tuxedo, Bea's M3, and Microsoft's MSMQ offer a wide range of features. By providing the middle-tier processes with a connection to the database instead of the clients, middleware enables many client applications to share the same connection to the database. Many clients are able to access the same middle-tier processes. Usually the n-tier application is configured so that many redundant middle-tier processes are loaded to memory and are available to be used. The middleware funnels the client requests for a process to the next available open process. The n-tier application can be configured so that if all processes are currently in use, a new process will be dynamically created (or spawned) on demand. This is known as load-balancing. The middleware of the n-tier application balances the client load among many servers. Some middleware products even provide the ability to process client requests based on a pre-assigned priority. Small requests that will be processed quickly can be set with a higher priority so that they will be processed before a large request that will take longer to run.

N-tier client/server provides many benefits when working with relational databases. Clients connect to the middle tier instead of the database; thus shielding the client application developer from the complexity of the database structure. In addition, by placing the SQL code inside server processes, the client application is shielded from database changes. When changes are made to the database, the middle-tier server process must be modified, but the client applications can remain unaffected. If several client applications share the same server process, the change only needs to be made in one place.

N-tier/client server has been used to create complex applications that would be difficult, if not impossible, to create using a two-tier approach. Certain middleware products, such as Bea's Tuxedo, can handle transactions across multiple heterogeneous databases. N-tier client/server has also made it possible to integrate legacy mainframe applications into a distributed environment. The mainframe legacy application becomes the "back-end" of the system, and middle-tier processes are created to access the data on the mainframe.

COMPARISON OF ALTERNATIVES FOR TODAY'S SYSTEMS

Deciding which architecture is best for an application is not an easy task. There are many options to choose from, many of which will work for any given application. There are many factors to consider and trade-offs to weigh. In general, a two-tier application is cheaper and faster to build but it does not provide as many robust features as an n-tier application. The following heuristics should be examined when making the choice between two-tier and n-tier client/server.

- If the number of users exceeds 100 or the number of users is expected to grow to more than 100 during the life of the application, scalability issues will arise with a two-tier application.
- If the application uses business logic that is common to other applications in the organization, it may be worth the investment to place the business logic in middle-tier processes so that it can be accessed by multiple applications.
- According to the Gartner Group, if an application is complex, involving more than 50 distinct business processes, n-tier should be used. As applications become more complex, two-tier applications become exponentially more difficult to develop [2].
- If the application is needed for a specific purpose for a short amount of time, it is not worth the investment of time and money to create an n-tier application. However, if the life of the application is expected to be more than three years, an n-tier approach should be used [2].
- If there is a plan to change to a different database in the future, it may be worth the investment to create an n-tier application. Modifying an n-tier application to a new database vendor is easier than modifying a two-tier application.
- A mission-critical application needs to be continuously available and thus requires a comprehensive disaster recovery plan. Disaster recovery is much more complicated in a distributed n-tier environment.
- An n-tier application can access multiple heterogeneous data sources and even provide two-phase commit controls across them. A two-tier application cannot support heterogeneous databases. If the application needs to access heterogeneous databases, n-tier is the only option.
- If the application needs to be deployed urgently, a two-tier architecture will get the job done faster. N-tier applications are more complex and take longer to create. An n-tier application can take up to twice as long to create compared with a two-tier approach [1].
- N-tier applications can be configured to have better response time than two-tier client/server applications. When databases are located remotely and the WAN time is significant, the n-tier application can be much faster than a two-tier application.
- Applications with more than 50,000 transactions per day should be developed using an n-tier approach [2].
- The power of the middleware comes with a price. Usually, a consultant that specializes in the kind of middleware you want to implement will be needed for a few months to get things going. Employing an on-site consultant can be very expensive. The licenses for the middleware products are also expensive.

These heuristics will, in all likelihood, conflict. However, they provide a starting point for the investigation of selecting an appropriate client/server architecture.

REFERENCES

[1] Linthicum, D. S. Moving to n-tier RAD. *DBMS*, 1997, 10(3), 3-8.

[2] Edwards, J. Three tier client server at work. New York, NY: Wiley Computer Publishing, 1999.