the Availability Digest

# Can 10,000 Chickens Replace Your Tractor?

December 2006

## Commodity Hardware and Open Source Software is Cheap

The abundance of commodity hardware and open source software has spurred several ventures aimed at replacing proprietary mainframe and fault-tolerant systems with large complexes of industry-standard servers. The arguments for such a move are lower cost and higher availability.

Costs should be lower since commodity hardware is so inexpensive compared to the big iron. Perhaps even more important is that the cost of software licenses can approach zero if open source software is used.[1]

Furthermore, since large complexes of industry servers can be used, they can be arranged in redundant architectures. After all, it is redundancy that provides high availability, right?

Well, perhaps neither argument is "right," as has been experienced by several organizations attempting to make this move. The experience is often that, though the initial cost of the system may be less, its ongoing costs more than offset the initial savings. Furthermore, the shear magnitude of managing such a large and complex aggregate of systems, often dozens and sometimes hundreds, leads to operational errors that can severely compromise the availability of the system. The result may well be a system with a higher total cost of ownership (TCO) and a lower availability of services to the users.

## Moving from the Mainframe to Commodity Solutions

Some architectures lend themselves nicely to commodity products. Primary among these are large banks of context-free servers. Context-free means that every request can be satisfied by any server with no knowledge of prior activity. Banks of Web servers that respond with requested pages are perhaps today's best examples of context-free servers. Any request can be sent to any server, which will respond with the desired page. No knowledge of previous requests is required. The total amount of information required to satisfy the request is contained in the request itself.

Should a server in such a context-free architecture fail, it is simply taken out of service. The surviving servers handle all requests thereafter. It is easy to add any level of redundant sparing to

[1] Open source software is also known as free software. However, "free" in this case does not mean free of cost, It means "freedom," as in the freedom to use, modify, and distribute software. Though open source software can be obtained via free downloads from the Internet, there may be other significant costs associated with it. See our review of Martin Fink's book, The Business and Economics of Linux and Open Source, in this issue of the Availability Digest.

such an architecture by simply adding additional servers over the number required to handle the anticipated load. An architecture such as this can indeed show a lower total cost and a higher availability than can be provided by a mainframe or a fault-tolerant system.

However, once the server has to use the context of previous requests, the playing field has changed. Consider an online store, for instance. Initial browsing by a customer can certainly be supported by a context-free bank of servers providing web pages on request. But once the customer wants to buy one or more items, a series of interactions occur, each based on previous interactions.

The customer asks to add a product to his shopping cart. The system must check inventory to ensure that the product can be delivered. If not, it must ask the customer if backordering is permissible. If the product is available, it must be added to the shopping cart. At checkout time, payment information and delivery options must be obtained, the credit card verified, the customer notified that the order has been accepted, a confirming email sent, and a shipment system notified of the order.

This is a very complex series of interactions depending greatly on context – the previous interactions that have led to the current interaction. Of course, the context of the transactions could be maintained on disk for access by different, otherwise context-free servers. Alternatively, the entire context for each interaction could be contained in the request message. However, both of these techniques are awkward and inefficient. It is much more efficient (and simpler) to route all interactions for a given transaction to a single server, which maintains the context for the entire duration of the transaction.

Now one must worry about what happens to the customer's transaction should the server that is handling it fail. Is the transaction simply aborted so that the customer must start all over again? Certainly not if the store wants to keep its customers. Let us do a little availability analysis here. If each server is an industry-standard server with three 9s availability and has a four-hour repair time, its average time between failures is about six months. If there are 50 servers in the server farm, there will be about 100 server failures per year. If each server is handling 100 transactions at any given time, each failure will abort about 100 transactions on the average. This represents 10,000 unhappy customers per year – not a good level of service by any measure.

Of course, redundant servers are the answer, which is accomplished by using clusters. But now the application network has only become more complex.

Online stores are only one example. Many of these attempts have been or are being made by financial institutions, manufacturers, and other large enterprises with significantly more complex applications. Can they do it?

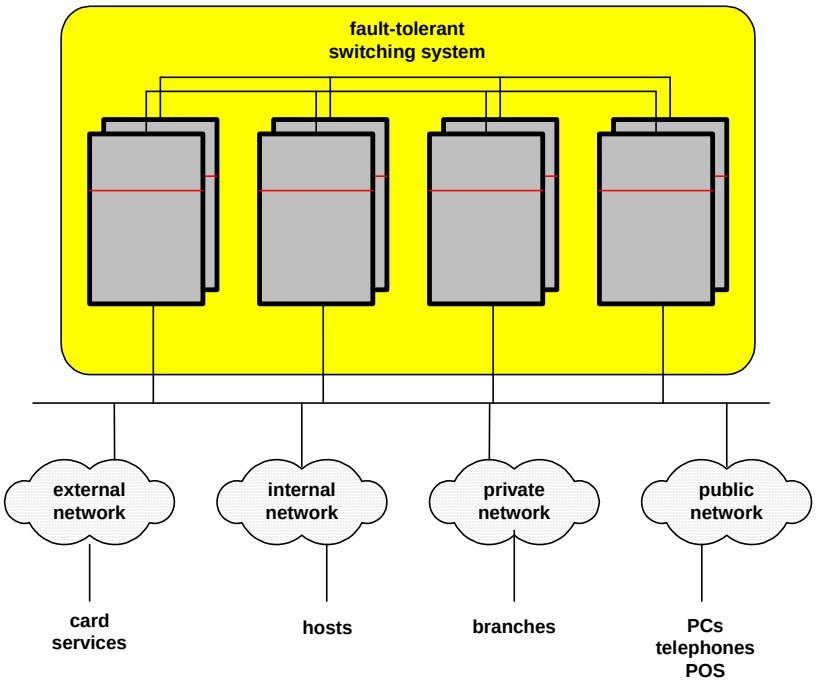
## A Case Study

One international financial institution recently went through this exercise. It wanted to take a major subsystem that was running on a (perceived very expensive) fault-tolerant system and replace it with (perceived less expensive) UNIX high-availability clusters.

### *The Fault-Tolerant System to be Replaced*

The subsystem in question provided switching services between the institutions's various networks. An internal network connected to other mainframe hosts operated by the institution. An external network connected to various card services (Visa, MasterCard, etc.). A private network connected the institution's branches to the institution's host services. Customers accessed the institution's services via the public network.

The switching subsystem comprised four pairs of fault-tolerant systems, each configured as an active/backup pair for very high reliability. The backup system in a pair was kept in synchronism with its active partner via asynchronous replication. If one system in a pair failed, the other could take over quickly.



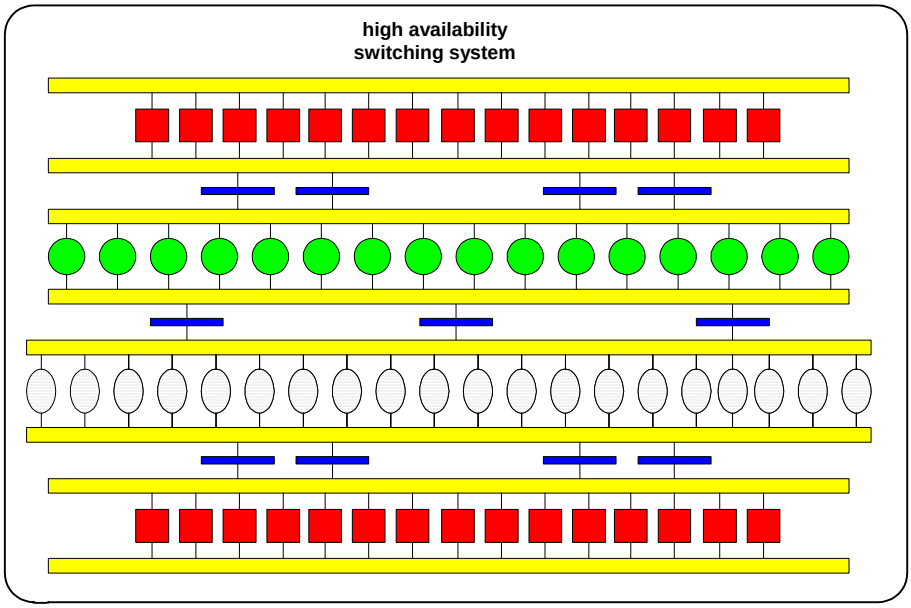

This system had fault tolerance fundamentally built into the box. Networking was integrated with the system. Failover times were virtually instant. It came completely configured with system management software with full automation support.

***The Clustered Replacement***

In contrast, the high-availability clustered system used SMP (shared memory processors). All servers were autonomous with network-based coordination among them. All networking between the systems was external, and the system behavior was highly dependent on network performance. Cluster failover times were much longer than the failover times in the fault-tolerant systems. Commercial network management software was used with varying levels of component support.

Because of the size of the clustered network, particular attention was paid to failover. First, faults had to be detected. Then each fault had to be localized (i.e., the failed component identified). The fault would have to be isolated to ensure that it would not contaminate the system or bring it down. The system would have to be restored to its expected behavior. Finally, the failed component would have to be repaired and returned to service. All of this was provided inherently with the fault-tolerant systems.

The clustered system was designed to have the same transaction throughput of the fault-tolerant system. The resulting clustered network included over one hundred separate components, including servers, routers, and raid arrays. All of this had to be managed with a variety of system management tools.




***The Result***

To say that the resulting system was a disappointment is a gross understatement. While the fault-tolerant system had higher acquisition costs, it had substantially lower operating and maintenance costs. While the staff had to be more experienced, it required fewer staff..

The clustered system required the management of tens of thousands of parameters with differing semantics across products. This compared to a much more finite set of parameters with full integrity checking for the fault-tolerant system.

Extensive security measures were included in the fault-tolerant system. Security in the clustered environment was difficult to determine. It was only as good as the weakest link.

The fault-tolerant system encompassed only a small number of software suppliers, and all software tasks were highly automated. The clustered system involved a large number of suppliers with limited automation and many functional gaps.

The bottom line was clearly stated:

| | **Fault-Tolerant Solution** | **Clustered Solution** |
|---|---|---|
| transaction throughput | 128 tps[2] | 120 tps |
| operating cost | $20,000,000 | $40,000,000 |
| availability | 99.99% | 99.5% |

Not only was the total cost of ownership of the clustered system twice that of the fault-tolerant system, but its probability of failure was fifty times greater (.0050 vs. .0001). Fortunately, the

[2] tps – transactions per second

original system had not been decommissioned, and, needless to say, the fault-tolerant systems are still in service.

## What's Next?

This is not to say that moving from fault-tolerant or mainframe systems to a clustered environment will never work. Some applications with the appropriate characteristics have been successfully moved. But as this case study shows, there must be a lot of analysis and study before undertaking such a project, especially for large, complex mission-critical systems.

One computer manufacturer was interested in doing just this for the systems which it used in its manufacturing process but after careful study wisely backed off. However, there are some significant projects now being contemplated or under way. The London Stock Exchange is in the process of just such a move for its trading system. Amtrak is in the midst of a multiyear study to do the same thing for its train control systems along the Northeast Corridor. It will be interesting to watch these efforts to see how they do.