Making Distributed Systems Stable

Alfred J. Barchi
ajb@ajbinc.net
http://www.ajbinc.net/

Abstract

The author recently had a conversation with a self-styled "system architect" about how to achieve stability in distributed systems and distributed applications. The author asserted that in order to create a stable system, you have to do stability testing once it is built. The system architect stated that stability testing is expensive, many software development houses lack the resources to do it properly, and it's more efficient to do adequate modelling of the system before it is built. The author certainly agrees with the idea that up-front modelling efforts are needed to avoid costly design errors.

However, upon further discussion, it turned out that the system architect had the curious notion that modelling is not only necessary, but that it is sufficient to eliminate all stability problems in a system. Everything in the author's 30+ years of experience in building distributed systems and applications told him that this is simply wrong. Unfortunately, the author hadn't thought much about why this idea is wrong, since the author had never encountered it before. Since it appears that the system architect is not alone in his heresy, however, the author wrote this paper in reply. It explains why modelling isn't sufficient by itself, and discusses a strategy that does produce effective results.

Models are only as good as the assumptions on which they are based.

This may seem profoundly obvious. But, in order for modelling to be sufficient in and of itself to provide system stability, the assumptions on which the model is based must all be completely accurate, and nothing significant may be overlooked.

It should also seem obvious that a model is never as complete and detailed as the thing itself, or else it becomes the thing itself, or at least becomes as complex and costly to build as the thing itself. In fact, the point of modelling is to produce a simplified representation of the thing to be developed, in order to allow it to be manipulated and design concepts validated at a fraction of the cost of building the actual system. The idea is to achieve an optimal strategy where the cost savings of the design flaws detected and avoided through modelling outweighs the cost of the modelling activity itself.

This concept seems to have been lost on those who used to advocate the use of such tools as "pseudo-code", i.e., PDL (pseudo-design language) to model complex applications before actually building them. An engineering process maven once showed the author a presentation he had made in which he asserted that the proper ratio of PDL to actual code

ought to be 10-to-1. Certainly, if you can adequately represent your system with a model that is only 10% of the size and complexity of the actual system, this is a good thing, and a worthwhile up-front design exercise to avoid costly design errors. The process maven was quite upset, however, when the author pointed out to him that in the example that the maven had used, the actual ratio was only 2-to-1.

Nowadays, we have more sophisticated tools for doing modelling – visual tools, fourth generation languages (4GL's), etc., which not only support modelling, but also provide for the automated generation of working code directly from the model. But, while it is important to do high level modelling of transactions, data flows, the structure of data, objects and their interactions, etc., it is also important to recognize that the 50% to 90% of the thing itself that we leave out of the model isn't just extraneous window dressing. And if we don't model that 50% to 90%, then our model won't catch any problems that occur in that 50% to 90%.

Another thing to consider is that in some instances, such as the development of mathematical models, the model itself can become too complex to manipulate effectively. Many years ago, an initiative was started that was known as "provably correct software." This approach attempts to mathematically model software and apply mathematical proofs to either discover errors or verify correctness.

Early attempts to do this modeled individual computer operations mathematically. Examples that were cited in research papers were typically no more than 20 to 50 lines-of-code long. It was found to be impractical to apply this approach to a complex application consisting of several hundred thousand lines of code written in a high-level language.

Later on came the "formal specification." The idea is that the specification of the product is expressed in an abstract, mathematical form, i.e., a formal specification, which is then tested for mathematical correctness. There are two approaches to formal specifications: the property-based approach and the model-based approach.

In the model-based approach, one uses the tools of set theory, function theory and logic to construct an abstract model of a system or application. This model is *high-level, idealized and free of implementation bias*. Special languages (similar in many ways to PDL) are used to construct this model, such as the "Z" specification Language or the VDM (Vienna Development method) specification language. There are several tools that have been developed for building and testing Z specifications, e.g., CADIZ, FUZZ and ZED.

Supposedly, IBM used the Z language to develop a formal specification for CICS. Whether this was done before or after CICS was implemented, whether it was done in lieu of or in addition to other types of specification, and to what extent it was useful in eliminating stability errors in CICS is unclear. It's also unclear whether this was a sincere attempt by IBM to minimize the cost of developing CICS, or whether it was part of a research project to explore alternative methodologies of software development.

No doubt there are tools of more recent vintage than Z specifications, such as UML or XML-based approaches. But all such tools address the same problem; they would all share similar economics, and they would all, of necessity, use a fundamentally similar approach, i.e., they would all implement models that are high-level, idealized and free of implementation bias. Perhaps this is what the system architect was thinking of in his discussion with the author.

In any event, actual systems that have been implemented are not high-level, idealized and free of implementation bias. Use of formal specifications does not eliminate the need for testing. And, in fact, IBM allocates a considerable portion of their development budget to testing.

Software is written by humans…

And, to err is human. Even automatic code generators are written by humans. The code that is output is seldom as streamlined and efficient as it could be if it were written directly by human programmers since it is designed for the general case and requires an underlying structure to be applied in order for the elements of the generated code to work together – a structure that is necessary for the tool to work properly but which has nothing to do with the generated application itself. Code generators are large, complex and sophisticated applications. They provide the ability to improve productivity by leveraging other people's work, i.e., the coding of that mundane 50% to 90% that we leave out of our models. But that also means that we are relying on an unknown third party to produce perfect code for us.

You simply can't eliminate human error in the development process by modeling, even when you generate code automatically from the model itself. You can certainly reduce human error this way; specifically, you can reduce errors in the design, but you can never eliminate them.

You can produce a perfect design and you can implement it perfectly, and your system may still be unstable.

How is this possible? The author has personally found errors in the compilers used to build the code, the binaries that get linked into the developed software by the compiler at compile-time, the operating system on which the application runs, the channels over which the application communicates, the hardware on which the application runs (including such things as the design of the CPU board and the interrupt controller chip), and of course, errors in the systems and devices with which the application communicates.

A few examples seem appropriate here. Others will be mentioned later:

- The author once discovered an error in the way that an interrupt controller chip worked. The manufacturer verified that it was, in fact, an error in their product.

When asked what they were going to do about it, the manufacturer replied that they were going to modify their documentation to advise their customers that the chip couldn't be used in that mode. This necessitated a major design change to the system being developed by the author's team.

- On the same project, the author discovered that when the CPU pre-fetched a certain sequence of bytes, it caused a massive voltage spike across the entire CPU card that resulted in a reset of the CPU. Using an in-circuit emulator, the author was able to demonstrate that the anomaly occurred during the pre-fetch – the CPU never actually executed the instructions being fetched. The author was also able to demonstrate that the anomaly was independent of the location from which the sequence was pre-fetched, and independent of the instructions that were executed immediately prior to the pre-fetch. The result was that the CPU board was re-designed with a new layout.

- The author once discovered an error in the implementation of the TCP/IP stack of a vendor. The author was able to narrow down the problem to the point where a one-line test program was developed that reliably produced the error. Even with this, and the assistance of a team of the author's engineers, the vendor was unable to locate and correct the problem. This resulted in the re-design of the author's application to avoid the situation that caused the error.

- The author recently implemented an application that communicated with several remote devices. Transaction modelling showed that there was no way that the application and the remote devices could get out of synch with each other. Nevertheless, they did anyway. It was discovered that the remote devices were randomly dropping messages. The author 'tuned' the application so that the situation under which this problem occurred didn't happen.

None of these problems was avoidable through modelling. None of these specific problems could even be anticipated, although it is completely reasonable to anticipate that there would be some kind of problems.

Classes of stability errors

This paper is about making distributed systems stable, and it seems appropriate at this point to define what the author means by a "distributed system." So:

- A distributed system is any application of set of applications that consist of more than one process or thread, running autonomously on one or more hosts, such that the processes or threads communicate with one another, either directly or indirectly, and contend with each other for resources.

This can include anything from a multi-threaded Java application running on a single host, to a huge, globally distributed information management system, where processes

that generate information and store it in a database are essentially communicating indirectly and over time with other processes that use or consume the information.

In addition to third-party anomalies, there are four classes of stability problems that tend to occur in distributed systems:

- **Race conditions** – These occur when a shared resource is not adequately protected by a semaphore. They also occur as a result of 'hysteresis', i.e., the lagging of an effect behind its cause. Complex networked systems are particularly prone to this phenomenon, due to the delays in transmitting information, including state information, over a network. In practice, it's extremely difficult to completely avoid these errors.

- **Boundary/Limit problems** – These include the ubiquitous "buffer overrun" that is so often exploited by hackers to launch malicious code. But it occurs in other situations as well. Computers have limited resources, including memory, disk space, I/O channels, etc. When these resources are exhausted, the software is forced to execute an error path through its code. These error paths are frequently not well thought out and not well tested. Unpredictable results can occur when adequate attention has not been paid to the design of these error paths. This happens all too often, because it is difficult for developers to conceive of running out of resources on a modern computer, so they tend to focus their attention in other areas. But it does happen, particularly when there are other applications running on the same machine at the same time, or using the same network resources at the same time. It's also possible to cause third-party applications to crash and bring down the entire host, even when your own software contains no errors at all. It's interesting to note that buffer overruns are still detected and reported with high frequency by computer security organizations, such as the Computer Emergency Response Team (CERT), despite the use of modern compilers designed to prevent this. It seems like not a week goes by that the author receives another CERT advisory containing the phrase, "… multiple buffer overruns detected in …"

- **Resource Leaks** – These occur when a resource is used and then not freed when processing has been completed. Languages such as Java that do automatic garbage collection help to minimize this. However, the author has found that it is still possible to create resource leaks even with Java. And garbage collection introduces random delays in processing that tend to exacerbate the problem of race conditions.

- **Deadlocks** – These occur when two or more processes become blocked while waiting for each other to release a resource. None of the processes involved can release the resource because they are blocked. This can happen with any kind of resource.

  It's also possible to cause deadlocks in third-party applications. The author once

discovered a deadlock problem in which the operating system went into a tight loop where a page kept getting swapped into and out of memory. The operating system could never exit the loop because the correct page was never in memory when the test-and-exit operation was performed. Does deadlocking the operating system count as a deadlock in your application? If your application creates the conditions under which this occurs, then you bet it does. And you can't eliminate it by modelling.

Attitude

Is it legitimate to say something like, "Gee, Mr. Customer, we're sorry that the system you paid us all that money to develop doesn't work. We have no idea why. But we can show you conclusive proof that it's not our fault!" Can you say, "Lawsuit?" The author used to tease certain members of a particular development team about this attitude, suggesting that to listen to them, the company's slogan ought to be, "*It's not our fault!"*

The team in question had done the appropriate modelling up front. They had also built prototypes to validate their models (of course, the modelling and prototyping was done with a later version of the operating system from the one on which the modelling and prototyping had been performed, and the application had been built with a later version of the compiler than the one used in prototyping). They had conducted testing of their middleware product prior to delivering it to the rest of the project. They couldn't conceive of how their product could possibly be unstable. Nevertheless, their product was so unstable (it never managed to run for more than 20 minutes at a time) that the project that relied on it (several million lines of code and hundreds of millions of dollars in budget) was unable to keep going long enough to complete a single test during the system-level test phase. This caused the project to slip by a rate of about 3 man-months per day, as over 125 project personnel sat around, unable to make any progress. A crash would occur, the development team members would race down to the test area to diagnose the problem, and they would discover that there was no evidence left to indicate what might have happened. Quel dommage!

Needless to say, nobody was pleased with this situation. The development team had no idea how to address this problem, so they simply threw up their hands. This attitude didn't go over real well with the customer, whose people kept having visions of their careers circling the drain.

The correct response was to acknowledge the stability problems existed and to formulate and implement a strategy for detecting and correcting them. This should have been done up-front, during the planning phase of the project, when resources could have been set aside and allocated to this activity. But it wasn't. The program management team and the development team shared the attitude that modelling and prototyping would prevent the situation from ever occurring. They were wrong. They performed the stability testing activity anyway, but as a cost overrun. It is interesting to note however, that saving a mere three days of schedule slip in the project was adequate to pay for the cost of the whole activity.

There is another attitude that deserves mentioning. At one company where the author used to work long ago, the CEO of the company pulled the author aside one day and explained that while our company had very limited resources, our customers had a couple of orders of magnitude more people than we had developers, that they worked with our product all day long, that they would find our problems a lot more quickly than we would, and they wouldn't be shy about calling us up to tell us about it. This person was also very adept at getting our customers to pay us to fix our mistakes. Needless to say, this company didn't get a lot of repeat business. And our developers spend an inordinate amount of time traveling to far away places to diagnose and fix problems that we could have corrected a lot more cheaply and efficiently at home, had we been allowed to do adequate testing.

Lastly, it is naïve to assume that using a third party solution such as J2EE's App Server, IBM's Websphere, Microsoft products, etc., will provide any insulation from stability problems. These are large, complex applications that provide a great boost to overall productivity, but it is naïve and unreasonable to assume that they come free of bugs.

Detecting and Correcting Stability Problems

So, how does one detect and correct stability problems? Well, by doing stability testing, of course. Let's say for example, that you have a messaging subsystem that has the characteristic that it randomly drops messages at a frequency of about 1 in 10,000. Depending on the nature of your system, this might manifest itself as a failure of the applications using the messaging subsystem maybe once every few hours, or every few days, or every few weeks. This is not an easy problem to debug. We need to increase the frequency of errors to seconds, or at most minutes, so we can effectively diagnose what is happening.

Since, in our example, we are talking about a messaging subsystem, it only makes sense to design a test application that sends messages through the subsystem – lots of messages. If, for example, the subsystem is specced to handle 10 messages a second, we design a test application that will try to send messages at a rate of 100 per second. This not only drives the messaging system at its maximum possible rate, it also creates a backlog that that consumes all available resources, and eventually drives the subsystem to execute its error paths. This allows us to detect boundary limit problems, resource leaks and deadlocks of the type that occur from improperly designed queues.

Since race conditions and certain types of deadlocks occur as a result of a random confluence of circumstances, this approach is also effective in dramatically improving the probability of detecting them.

How many messages do we need to send through our system in order to have, say, a 95% chance of encountering one or more errors? This is not hard to calculate. Assuming each message represents a separate, discrete, independent event, then in our hypothetical subsystem-under-test, there is a probability of 0.9999 that the message will be handled

correctly. There is a $(0.9999)^2$ probability of two messages in a row being handled successfully, and a $(0.9999)^X$ probability of 'X' messages in a row being handled correctly. The probability of 1 or more errors occurring in 'X' messages is then equal to 1 minus the probability that no errors will occur, or $1 - (0.9999)^X$. For a 95% coverage rate, we set: $0.95 = 1 - (0.9999)^X$ and solve for 'X'. This is the equivalent of solving: $0.05 = (0.9999)^X$. Allowing for round-off, this works out to 29,956 messages. If you want to have a higher probability of detecting and error, you have to send more messages. If you have a higher frequency of error, you will have a higher probability of detection with the same number of messages.

Usually, you want to design software to be robust enough to handle certain kinds of random errors and keep on running. Not so with test software. You want to design your test software to stop as soon as it detects and error, and report as much information about the error as possible in order to aid in debugging. The developers of the software-under-test can add monitoring and debugging code to their software which can be triggered by the test software as soon as an error is detected in order to save state information about the software-under-test. Using this information, the developers can then hone in on the cause of the problem.

A distributed system or a distributed application may have any number of functional capabilities that are susceptible to race conditions, boundary/limit problems, resource leaks and deadlocks. These should all be identified and stability tests designed of each. Messaging was just one example. Typically, any operation that involves interaction with another process or thread, or which utilizes shared resources is a candidate.

Porting Distributed Applications to New Platforms

Assume for the moment that one could perfectly model a distributed application, the operating system and environment (other applications running concurrently) under which it runs, the hardware on which it runs, and the systems and devices that it communicates with. What happens when one then attempts to port the application to a new platform? The answer should be obvious. How many readers have experienced the thrill of having their applications break or become unstable when they upgraded to a new version of Microsoft Windows, for example.

Even when you use a portable programming language (i.e., "write once, run anywhere") such as Java, you have to be aware that on a new host, the implementation of the virtual machine will necessarily be somewhat different, the timing of operations will be different, your application may rely on some obscure, undocumented feature of the language that no longer works in the new environment, etc.

How do we deal with this problem? Do we painstakingly measure and collect data on all of the operating characteristics of the new environment, factor these into our model, and then fix all of the stability problems that the model then predicts? Or, do we re-run the stability tests? Certainly, some basic level of modelling may seem appropriate before investing in the new environment, or paying the costs associated with porting to the new

environment. But, once you have the new environment, your ported application and your suite of stability tests, what's to be gained?

Modelling is itself an expensive process, and there comes a point in the development life cycle when further modelling or greater fidelity in the model becomes cost-prohibitive. This is obviously true in the porting situation, but it also applies to the initial development situation, where the cost of design errors is significantly higher.

Conclusion

Modelling is a necessary activity in order to avoid making costly design mistakes. But it is not sufficient by itself to prevent all stability problems. Real-world systems are built by fallible humans who make mistakes in design, modelling, implementation and integration. Models are also dependent on the validity of the assumptions on which they are based as well as the robustness and fidelity of the model. In addition, there is always the potential for unanticipated problems caused by third-party products upon which the system being designed depends, or which run concurrently on the same hosts as the system being designed.

Real "system architects" are people who design, develop and field real systems. They understand that real systems can be incredibly complex and are essentially non-deterministic in nature. They are comfortable with the concept of feedback control loops, and their application to the development process, including the use of design reviews, unit testing, beta testing, stability testing, independent verification and validation, etc.

It's necessary to adopt a realistic and pragmatic attitude toward system stability in order to achieve it. Realistically, system stability can only be adequately accomplished through stability testing. This activity must be planned and budgeted just like every other activity on a project. It's going to happen anyway, whether you plan for it or not. The only difference is how well you manage it.