acmqueue Tackling Architectural Complexity with Modeling

Component models can help diagnose architectural problems in both new and existing systems.

Kevin Montagne

The ever-increasing might of modern computers has made it possible to solve problems once thought too difficult to tackle. Far too often, however, the systems for these functionally complex problem spaces have overly complicated architectures. In this article I use the term *architecture* to refer to the overall macro design of a system rather than the details of how the individual parts are implemented. The system architecture is what is behind the scenes of usable functionality, including internal and external communication mechanisms, component boundaries and coupling, and how the system will make use of any underlying infrastructure (databases, networks, etc.). The architecture is the “right” answer to the question: how does this system work?

The question is, what can be done about the challenge to understand—or better yet, prevent—the complexity in systems? Many development methodologies (e.g., Booch¹) consider nonfunctional aspects, but too often it stops at the diagram stage. The mantra of “we can address [performance, scalability, etc.] later” can be crippling. Individual components (applications) in a system can typically be iterated, but it is often far more difficult to iterate the architecture because of all the interface and infrastructure implications.

MODELING EXISTING SYSTEMS

Later in this article I describe an approach to architectural design when embarking on creating a new system. But what do you if the system already exists in some form? Much of my architecture work has been with existing systems—many times as an “outsider” who is invited (or sent) in to evaluate and improve the state of the system. These assignments can be quite challenging when dealing with complex systems.

One advantage to modeling an existing system is that the general behavior is already in place so you are not starting from a blank state. You also probably do not have to contend with the creation of the functional parts of the system. This comes at a price, however. There is a fair chance the system’s architecture is complex and not well understood. Additionally, many solutions may not be practical because of the high cost of a system overhaul.

With any type of system the goal is to understand the architecture and system behavior as much as possible. When a large system has been around for years this may seem like a monumental effort. Many techniques are available for discovering how a system works and ways it can be improved. You can ask members of the development and maintenance teams. Diagnostic tools (e.g., DTrace) can help make quick work of finding performance or scalability offenders in a system. You can comb through mountains of log files to see what the developers thought worthy of note. In this article I focus on how modeling the various system components can be used to gain a greater understanding and provide a foundation for evaluating possible changes.

This type of modeling is not just a whiteboard or paper exercise. It is the creation of drivers and

components to emulate various aspects of the system. The drivers are used to invoke the various parts of the system to mimic its normal behavior. The idea is to exercise the architecture without the “burden” of ensuring functional correctness. At times these drivers may be scripts written with established tools (e.g., WinRunner, JMeter), but I have often found more value in developing programs specific to the component to be driven. These have allowed me to get the information I needed to make quality decisions. It is important to understand that the model components and the associated drivers are not just simple test programs but are to be used as the basis for exploration and discovery.

The process of modeling the system should start by examining one or two components at a time. The initial targets should be components suspected of negatively impacting the whole system. You can then build independent drivers to interact with the component(s). If a problem component is confirmed, then experimentation with possible changes can begin. These could span from code changes to infrastructure changes to hardware changes. With the right drivers and component modeling, it may become practical to address redesigning some of the components.

Sometimes the functionality contained within a component is so intertwined with the architecture that it's necessary to create a lightweight replica. It is not unusual for some functional aspects of the system to mask the behavior of the underlying technology or infrastructure in responding to the requesting applications. In these cases having a lightweight model can allow the architectural interactions to be explored and better understood. If you discover architectural solutions, then you can move on to the various functional implementations.

MODELING AN EARLY WINDOWS SYSTEM

My first experience with modeling involved creating both drivers and mock-up components to explore a new technology. I was working for a large financial institution in the late 1980s when Microsoft Windows 2.1 was released. A group of developers had created a fairly sophisticated suite of Windows applications for telephone-based customer service representatives. The applications provided the ability to retrieve customer information, balances, etc. from several mainframe-based systems (using the now-ancient concept of “screen scraping” the data intended to be displayed on an IBM 3270 dumb terminal) and then present the data in an aggregated view. It also allowed the customer service representatives to place trades on behalf of the customer.

The suite started as a proof of concept, but the prototype demos went so well it was rushed to production. When I joined the team it was already deployed to about 150 representatives. As the programs began to be used all day, problems began to occur frequently. These were manifested in a variety of forms: memory leaks, access violations, spurious error messages, and machine lock-ups (aka freezes).

Our small team was busy adding functionality to meet the rapidly growing wish list and at the same time addressing the stability issues. We navigated through the source, attacking memory leaks and access violations. We struggled to track down the growing list of newly observed error message. The most challenging task was “freeze patrol,” where we expended a great deal of time hunting down those machine lock-ups. The problem was that we did not have a really good understanding of how Windows worked behind the scenes.

Those familiar with programming with the early Windows SDKs will remember that documentation (not to mention stability) was not well developed. The API functions were pretty low level, and it seemed like there were a bazillion of them. (If it were not for Charles Petzold's

Programming Windows [Microsoft Press, 1988], I am not sure how many Windows applications developed outside of Microsoft would have been completed in the 1980s.) The code base for the applications was already pretty large—at least for applications in those days—and each was implemented slightly differently (they were prototypes, after all). Microsoft offered a few sample programs but nothing close to the complexity of these applications. Therefore, we decided to build components (applications) that imitated the Windows behavior we were trying to achieve.

These components were mostly void of functionality but started off with the basic structure and interface mechanisms similar to the actual applications. The drivers sent fine-grained Windows messages to the model components to simulate key presses and other externally originated actions. They also sent DDE (Dynamic Data Exchange, a primitive way to communicate data between Windows programs) messages throughout the suite of applications. As we matured the model, we began to merge in more of the API calls (e.g., user interface controls) used in the actual programs.

Many of the freezes were tracked down to undocumented idiosyncrasies of Windows GDI (Graphics Device Interface) calls. Examples included sensitivity to the ordering of some API calls, incompatibility between certain calls being made in the same context, and resource exhaustion possibilities. In the early versions of Windows the GDI libraries were tightly interwoven with the kernel libraries. As Windows matured, similar quandaries became error messages, exceptions, or just the offending application locking up.

The result of the modeling was that we gained enough information about this novel Windows technology to morph the programs to where stability was a reasonable expectation. Within 18 months the system was deployed to more than 4,500 workstations and survived well into Windows NT's life.

MODELING A "SLAVE" SYSTEM

Not all of my modeling experiences resulted in such a positive outcome. Several exposed fundamental flaws in the architectural design, and with a few the only option was to abandon the system and start over. These messages were not typically well received by project management.

One of the more notable examples occurred in a system intended to be a "slave" receiving updates from several existing systems and applying them to a new database. The database would be used by other new systems to form the basis to replace the older systems. The systems would be built using a new technology platform. The technologies were so different and the functional breadth so wide that the development team had grown to more than 60 people for the slave system alone.

I joined the project after the basic architecture and much of the functionality had already been designed and developed, but it was still months away from production. My team's assignment was to help get the most out of the infrastructure and optimize how the applications interacted with each other. After just a few weeks we suspected that some bad initial assumptions had impacted the architectural design. (I do not mean to disparage any teams in my examples, but merely to point out the potential problem with too much focus on functionality at the expense of a solid architectural foundation.) Because it looked like performance and scalability were going to be major concerns, the architecture team began working on some model components and drivers to investigate the design.

We did some research around the incoming rate of messages and the mix in the types of transactions. We also sampled timings from the functional "processors" that had already been built. Then, using the same messaging infrastructure as the existing dispatcher, we built a component that would simulate the incoming message dispatcher. Some of the messaging technology was new to the

company. At one end of the dispatcher we had drivers to simulate inbound messages. On the other end we simulated the performance of the FPs (functional processors) using pseudo-random numbers clustered around the sampled timings. By design, there was nothing in the modeled components or drivers related to the functional processing in the system.

Once the model was fully functional, we were able to play with various parameters related to the incoming message rates and simulated FP timings. We then began to weight the FP times according to processing cost variations in the mix of incoming message types. Prior to this modeling effort, the design had (wrongly) assumed that the most important performance aspect was the latency of the individual transactions. Several seconds of latency was acceptable to all concerned. After all, it would be quite some time before this slave would become the system of record and drive transactions the other way.

The modeling results were not encouraging. The latency was going to be a challenge, but the overall throughput requirements were going to bury the system. We started exploring ways to address the performance problems. The system was already targeted for the fastest hardware available for the chosen platform, so that option was out. We delayed looking into improving the performance of the individual functional processors; that was deemed to be more costly because of the number that had already been written. We thought our chances of quick success could increase with a focus on the common infrastructure pieces.

We worked on new dispatching algorithms but that did not result in enough improvement. We looked at optimizing the messaging infrastructure but still fell short. We then began to benchmark some other message formats and infrastructures, and the results were mildly encouraging. We examined the existing programs to see how easy it was going to be to alter the messaging formats and technology. The programs were too dependent on the message structure for it to be altered within a reasonable timeframe.

Given the still poor results, we needed to examine the functional algorithms and the database access. We took a few of the midrange and lengthier running processors and inserted some logging to obtain split times of the various steps. Many of the functional algorithms were relatively expensive because of the required complexity for the mapping and restructuring of the data. The database operations seemed to take longer than we logically thought they should. (Over time an architect should develop a sense for a performance budget based on an abstract view of similar functionality where he or she had previously maximized performance.)

We then examined the logical database model. The design was not a pattern that would be performant for the types of programs in the system. The SQL from a few of the algorithms was extracted and placed in stand-alone model components. The idea was to see which types of performance increases were possible. Some increases came from changing some of the SQL statements, which were taking excessive time because the chosen partitioning scheme meant that reading core tables typically involved scanning all partitions. As our simulated database size grew, this became punitive to scalability. The primary problem, however, was not the extended length of time for individual statements but the sheer number of calls. This was a result of taking normalization too far. There were numerous tables with indexes on columns that changed frequently. Additionally, multicolumn keys were being used instead of artificial keys, sometimes referred to as surrogate keys. The system generates them (typically as integers) to represent the “real” keys. This can improve performance and maintenance when dealing with complex key structures

and/or when the actual key values can change.

We determined that material improvements were possible if we restructured the database design and changed the associated SQL statements. The programs were written in a way that would have made the changes very expensive, however. Our conclusion was that the system would need a major overhaul if it were to be successful. Since the project had already spent well over \$10 million, this recommendation was a hard sell.

After an additional \$5 million, the project was cancelled, and my team's focus was redirected to other efforts. The modeling process had taken only about six weeks. The point to be made here is that it would be possible to use modeling to vet the major architectural decisions before committing large expenditures. It is vastly less expensive to discover that a design will not perform or scale *before* a system is built rather than after it has been placed in production.

MODELING NEW SYSTEMS

It should be standard practice to research the architectural options for new systems—or when making substantial overhauls to existing ones. The experiments should be with lightweight models rather than a full system, but it is vital that these models accurately capture the evolving behavior of the system. Otherwise the value of the modeling process is diminished and may lead to erroneous conclusions.

I typically start by trying to understand the functional problem space in an abstract fashion. Is the primary functionality a user-requested action followed by a system reply (i.e., request/reply)? Is it a request followed by a stream of notifications (e.g., ticking quotes) or bits (e.g., music or video)? Is it to process some input data and send the result to another process or system (i.e., flow-through)? Is it to crunch through a massive dataset in search of information (decision support system)? Is it a combination of these, or something altogether different?

Some may ask: how do I know which portions of the system to model and how much time and effort should be spent in the process? It is a simple case of risk management. The modeling should focus on the areas that would be the most expensive to get wrong. The process should continue until the high-risk decisions can be justified. Make an effort to retest the decisions as often as practical.

One of the most challenging aspects of modeling is finding the right balance between capturing enough of the system behavior and keeping the model from becoming too complex (and expensive) to implement. This is easier with an existing system. As you progress through the modeling iterations, if the observations begin to mimic aspects of the system, then you are probably pretty close. You can begin to alter the modeling drivers and components to explore more of the behavior. For a new system I typically look to model components that can be used as shells for real components. The goal is to provide the responsible developer with a starting point that allows the focus to be on the functionality rather than having to explore the critical nuances of the underlying technology and infrastructure.

There are numerous technical modalities to consider when designing or evaluating architecture: performance, availability, scalability, security, testability, maintainability, ease of development, and operability. The priority ordering of these modalities may differ across systems, but each must be considered. How these modalities are addressed and their corresponding technical considerations may vary by system component. For example, with request/reply and streaming updates, latency is a critical performance factor, whereas throughput may be a better performance factor for flow-through message processing or bulk-request functionality. A perhaps subtle but nonetheless important

message is to avoid mixing different modality implementations within the same component. Failure to adhere to this lesson puts the architecture on a sure path to complexity.

It is far too common to hear the excuse: “The system is [going to be] too large to take the time to model its behavior. We just need to start building it.” If the chore of modeling is considered too onerous, then it will probably be very challenging to achieve predictable performance, scalability, and other desirable technical attributes. Some development projects have a strong focus on unit tests, but in my experience it is rare to find a corresponding focus on testing the system architecture as a whole.

MODELING A SAMPLE COMPONENT

Describing the modeling of a sample component may provide additional insight into the approach I am advocating. Suppose a new system calls for receiving some stream of data items (e.g., stock quotes), enriching the data, and publishing it to end users. An architect may suggest that some type of publisher component be built to perform this core requirement. How can this component be modeled before investing money in building a system around it? Data throughput and latency are probably primary concerns. Ideally, we have some target requirements for these. Scalability and availability are also issues that can be addressed with later iterations of the model but before proceeding with the functional development.

Based on this simple example, the model should contain at least two building blocks distinct from the publisher component. The incoming data feed needs to be simulated. A driver should be built to pump data into the publisher. Additionally, some type of client sink is necessary to validate the flow of messages and enable the measuring of throughput and latency. Figure 1 shows a simplified drawing with drivers and sinks for the proposed publisher.

The publisher model component should be built using the proposed target language. It should use any frameworks, libraries, etc. that may affect the model outcome, though it may not be obvious which of these could have an effect. In that case you should take a risk management approach and include those that are core to the operation of the component. Any new technology whose behavior is not already fully understood should be included as well. Any nonsuspect infrastructure can be added in later iterations. It is important not to get mired in trying to build the functionality too early. As much as possible should be stubbed out.

In some systems a component such as the publisher may present the largest scalability hurdle. In that case we need to know what type of message flow can be handled, what type of latency can be expected, how many clients can be supported, and what type of flow the client applications can handle.

The data-feed driver should accept parameters that allow the message rate to be dialed to arbitrary levels. Any driver should be capable of pushing its target well past any expected high-water mark. The messages do not have to match the intended format, but they should be relatively close in size. Since the driver is tightly coupled with the publisher, it should be written for and run on the same type of platform (language, operating system, etc.). This enables the same developer to build both the component and the driver. (I strongly suggest that each developer responsible for a system-level component also create a distinct driver and a possible sink as a standard practice.) The same holds true for the client sink so all three can be packaged together. This provides a cohesiveness that will allow the model to be reused for other purposes in the future.

As the modeling progresses, another model receiver should be built for the target client platform using its expected frameworks and communication mechanism. The reason for the two different platform receiver/sinks is to allow the publisher model component to be tested without involving another platform (e.g., scalability testing). The client-platform model receiver can be used to determine if the publisher is interacting with the client platform properly. During future troubleshooting sessions these separate receivers would provide a means to isolate the problem area. All of the drivers and sinks should be maintained as part of the development and maintenance of the publisher.

The next step is to evaluate the publisher model in action with the drivers and sinks. To characterize the performance, some type of instrumentation needs to be added to the client sink to calculate throughput. Care must be taken with any type of instrumentation so it does not influence the results of the test. For example, logging every single message received with a timestamp is likely to be punitive to performance. Instead, summary statistics can be kept in memory and written out at periodic intervals or when the test ends.

The data-feed driver should output data at a configurable rate while the client sinks count messages and compute the rate of data received. Another instrumentation method could be used to sample the latency. At specified message count intervals, the data-feed driver could log the message number and the originating timestamp. The client sinks could then log the receive timestamp at the same interval. If logged at an appropriate frequency, the samples could give a good representation of the latency without affecting the overall performance. High-resolution timers may be necessary. Testing across multiple machines with a latency requirement lower than the clock synchronization drift would require more sophisticated timing methods.

This model should be exercised at various message rates, including rates that completely

overwhelm the publisher and its available resources. In addition to observing throughput and latency, the system resource utilization (CPU, memory, network, etc.) should be profiled. This information could be used later to determine if there are possible benefits in exploring infrastructure tuning.

As mentioned earlier, the publisher is required to do some type of data enrichment as the messages pass through. Throughput, latency, and memory consumption are likely to be impacted by this enrichment. This influence should be estimated and incorporated into the model publisher. If realistic estimates are not available, estimate high (or, following the philosophy of this article, build another model and characterize it). If the cost of enrichment varies by message type, then a pseudo-random delay and memory allocation clustered around the expected averages could be inserted into the model publisher.

OTHER USES FOR MODELING

Modeling is an iterative process. It should not be thought of as just some type of performance test. Here is a list of items that could be added to further the evaluation process.

- Use the model to evaluate various infrastructure choices. These could include messaging middleware, operating system and database-tuning parameters, network topology, and storage system options.
- Use the model to create a performance profile for a set of hardware, and use that profile to extrapolate performance on other hardware platforms. Any extrapolation will be more accurate if the model is profiled on more than one hardware platform.
- Use the performance profiles to determine if multiple instances of the publisher (horizontal scaling) are likely to be required as the system grows. If so, this capability should be built into the design and modeled appropriately. Converting components designed to be singletons could be very expensive.
- Use the model to explore the set of possible failure scenarios. Availability is one of the primary attributes of a quality system. Waiting to address it after a system is built can cost an order of magnitude more.

The examples used in this article can be seen in abstractions of many systems. Similar modeling approaches should be undertaken for any material component. When interrelated models have been built and tested they can then be combined for more comprehensive system modeling. The approach of building one model at a time allows the system behavioral knowledge to be gained in steps rather than attempting to understand—not to mention build—one all-encompassing model.

One key element present in almost all systems is some type of data store. Evaluating a database design can be complex. There are a number of steps that are similar to the system modeling already discussed, however. Once a draft of the database model (columns, tables, etc.) is available, it can be populated with enough generated data to enable some performance testing. The effort required to write a data generator for this purpose will give an idea of how easy it will be to work with the database during the development process. If this generator seems too difficult to tackle, that may be a sign the database model is already too complex.

After the tables have been populated, the next step is to create driver(s) that will exercise the queries expected to be most expensive and/or most frequent. These drivers can be used to refine the underlying relational model, storage organization, tuning parameters, etc. Performing this

type of modeling can be priceless. Discovering flaws in the application-level data model after all the queries have been written and the system is running in production is painful. I have worked to improve database performance on dozens of systems. Optimizing queries, storage subsystems, and other database-related items post development can be really challenging. If the system has been in production for some time, then the task is even more difficult. Many times the low-level infrastructure changes could have been determined by early modeling. With the proper design, more standard configurations might have sufficed.

INSTRUMENTATION AND MAINTENANCE

Regardless of the type of driver/component combination, instrumentation is vital to both modeling and the long-lasting health of a system. It is not just a luxury. Flying blind about performance is not advised. Visual flight rules (i.e., without instrumentation) can be used only when the skies are clear. How often is that true for modern systems? The functional and technical complexity typically clouds the ability to see clearly what is happening. System performance can be like floating down the river in a raft. If you do not observe the speed of the water periodically, then you might not notice an upcoming waterfall until the raft is hopelessly plunging over the edge. As mentioned previously, when the volume of instrumentation data is too high, consider using “tracers” and/or statistical sampling.

There are numerous advantages to keeping the drivers and model components up to date as a system evolves:

- They can be used for general regression testing for performance, availability, scalability, etc. when changes are proposed.
- They can be used for capacity planning by extrapolating performance from a smaller set of resources. The only practical way to do this is by fully understanding the resource usage characteristics.
- They can model infrastructure or other large-scale changes that may need to be made to an existing system.
- At times there are factors outside the control of the maintenance/development team (e.g., infrastructure changes). The drivers could be used to test an isolated portion of the system. If any degradation was caused by the outside factors, then the results could provide “defensive” data to have the changes altered or rolled back.
- When some type of performance, availability, scalability, or other infrastructure problem arises, it would be much quicker to pull out the model and drivers than to take on the possibly overwhelming task of updating them while under pressure to troubleshoot a production problem.

Modeling is an extremely powerful method to understand and improve the overall quality of a system. For systems that are expected to last for years, this improvement translates into real monetary savings. Development organizations can then spend their budgetary money on providing functionality. If the models and associated drivers are sustained, then this functional focus can be widely celebrated. ☐

REFERENCES

1. Booch, G. 1993. *Object-oriented Analysis and Design with Applications* (2nd edition). Redwood City: Benjamin Cummings.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KEVIN MONTAGNE has over 25 years of experience in the IT field working with large-scale systems where performance and availability were critical. He spent twenty of those years in the financial industry, including more than a decade as an architect of front-office trading systems.

© 2010 ACM 1542-7730/10/0900 \$10.00