Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

SEI Monographs on the Use of Commercial Software in Government Systems

Case Study: Correcting System Failure in a COTS Information System

Scott Hissam

September 1997

About this Series

Government policies on the acquisition of software-intensive systems have recently undergone a significant shift in emphasis toward the use of existing commercial products. Some Requests for Proposals (RFPs) now include a mandate concerning the amount of COTS (commercial off-the-shelf) products that must be included. This interest in COTS products is based on a number of factors, not least of which is the spiraling cost of software. Given the current state of shrinking budgets and growing need, it is obvious that appropriate use of commercially available products is one of the remedies that might enable the government to acquire needed capabilities in a cost-effective manner. In systems where the use of existing commercial components is both possible and feasible, it is no longer acceptable for the government to specify, build, and maintain a large array of comparable proprietary products.

However, like any solution to any problem, there are drawbacks and benefits: significant tradeoffs exist when embracing a commercial basis for the government's software systems. Thus, the policies that favor COTS use must be implemented with an understanding of the complex set of impacts that stem from use of commercial products. Those implementing COTS products must also recognize the associated issues—system distribution, interface standards, legacy system reengineering, and so forth—with which a COTS-based approach must be integrated and balanced.

In response to this need, a set of monographs is being prepared that addresses the use of COTS software in government systems. Each monograph will focus on a particular topic, for example: the types of systems that will most benefit from a COTS approach; guidelines about the hard tradeoffs made when incorporating COTS products into systems; recommended processes and procedures for integrating multiple commercial products; upgrade strategies for multiple vendors' systems; recommendations about when not to use a commercial approach. Since these issues have an impact on a broad community in DoD and other government agencies, and range from high-level policy questions to detailed technical questions, we have chosen this modular approach; an individual monograph can be brief and focused, yet still provide sufficient detail to be valuable.

About this Monograph

This monograph provides an in-depth technical study about a COTS-based information system made up of several commercial components. In particular, this monograph provides details of the activities needed to make the system functionally useful. While all readers may find value in this report, it is expressly aimed at a technical audience.

The organization that performed the work described in this monograph is not named, nor are the specific commercial products. All of them, however, are common products, and the work is very typical of the type of effort entailed in integrating and tuning a COTS-based information system.

Adequately describing the conceptual foundation of this work would consume too much space, given the intended scope of a single monograph; we have therefore provided only a brief overview here. A full description of the high-level concepts related to debugging COTS-based systems can be found in a monograph devoted to overall debugging issues for COTS components, *Isolating Faults in Complex COTS-Based Systems*.

Case Study: Correcting System Failure in a COTS Information System

1 Introduction: The Ongoing Need for Expertise

There is a common belief that use of commercial off-the-shelf (COTS) components implies a lessened need for expertise as compared to 'ground-up' system development. In the sense that the individual components themselves are simply being purchased, are not being designed or engineered, and can be used with limited technical understanding, there is some justification for this assumption. However, from the perspective of a system integrator charged with the assembly, integration, and maintenance of several COTS components into a multi-component system, the belief that using COTS components will make it unnecessary to have extensive software expertise often proves to be false.

The major complicating factor of using COTS products is the extent of diversity in the system. As the number of components to be integrated, and the number of possible combinations increase, so does the likelihood that failures will occur in

- an individual component
- an interaction between pairs of components
- the entire system itself

And when the system does fail, fault isolation becomes a very difficult matter: Is the failure caused by one component? If so, which one? Is failure caused by an interaction of components? If so, which pair? To the integrator of this system, the components are black boxes, and the difficulty of isolating the fault stems from both ascertaining *how* the components work and then finding out *why* they do not work.

The source of the difficulty is obvious: the integrator has no visibility into the components and no control of their workings. The available information is nearly always incomplete and often inconsistent. The integrator must therefore have keen diagnostic skills and good general knowledge of several underlying technologies to gain an "inside the black box" perspective of the components he/she is attempting to assemble and debug.

The case study described in this paper illustrates the kinds of techniques that integrators need to gain an objective view inside the "black boxes" of off-the-shelf components. The study demonstrates that for a commercially-based system of even moderate complexity, a high level of engineering skill may be required to make the system perform as its users expect. This study also highlights how the application of these techniques can aid communication between an integrator and COTS vendors.

2 A Method for Debugging a COTS-Based System

When a system composed of several COTS components fails, a first recourse is to hope that the vendors of the components might provide some assistance; unfortunately, this is a remote

possibility. It is more likely that each vendor will suggest that the fault lies not with his/her component but with someone else's. This leads to a great deal of finger-pointing, but little else. The only solution is for someone—typically, the integrator who put the system together—to debug the system. This requires a very different notion of "debugging" than when source code is available.

2.1 Hypotheses and Experimentation

A useful method for understanding and correcting system failure in a COTS-based system derives from the classic scientific method of observation, hypothesis, and prediction. This method defines a systematic approach for observing cause and effect (see Figure 1). A hypothesis is formed based on one or more observations. To test that hypothesis, a prediction is made relative to some stimulus and its anticipated effect. An experiment is then designed to test the prediction. Results from the experiment will either support or contradict the hypothesis. If the hypothesis is not supported, new observations, refinement of the hypothesis, or subsequent experimentation are needed.

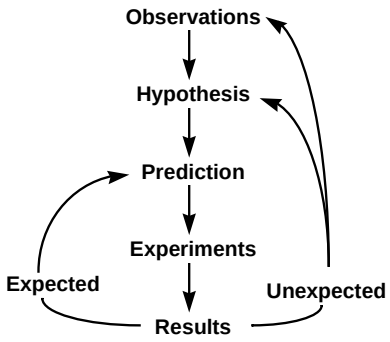

Figure 1: The Scientific Method

The assumption of the scientific method is that there is something to observe. However, in dealing with off-the-shelf components, their opaqueness means that while their external *behavior* can be observed, their inner workings cannot, and we have little or no insight into the components' underlying built-in assumptions or design constraints.

For example, assume there is a problem report about a sorting routine in a system. If the routine is in a system that has been created in-house, the first recourse is to examine the source code. But with an off-the-shelf component purchased from a commercial vendor, debugging in the traditional sense is impossible. Instead, it becomes necessary to isolate the problem by observing the system's behavior in one or more contexts. Thus, a likely hypothesis might be that the sort routine is adversely affected by scarce memory resources. A corollary prediction is that by starving the system of available memory resources, the sort component will fail even when the number of elements in the list remains below that stated in the problem report. An experiment can then be built to consume available memory resources, in which the sort component is re-run with

a controlled data set; if the component fails as expected, then the hypothesis (that the sort component is affected by low memory conditions) is supported.

2.2 The Need for Visibility into COTS Components

The above example illustrates the need for some level of visibility into COTS components. Techniques used to gain such visibility are based on both the nature of the component and the suspected failure. If the component itself is suspect, the integrator uses tools to peer inside the component's outer boundary. If the failure is suspected to be between two components, then the integrator uses tools to look at the interaction of the suspected components. Lastly, if the general system or subsystem of components is in failure, the integrator can use tools and methods to gauge a group of components working in unison. Such techniques, or "observation posts," can be grouped into three areas:

- *Intra-component:* These observe the behavior of a COTS component to understand the component developer's assumptions and intended usage of the component.
- *Inter-component:* These observe the behavior of two or more COTS components to understand potential mismatches between components.
- *Extra-component:* These observe a system of cooperating components, to understand macro-level issues dealing with performance, maintenance, misfits, etc.

The first group, for intra-component visibility, includes techniques, tools, and mechanisms that obtain static or dynamic visibility into parent/child relationships, thread performance, resource utilization, signal and event disposition, system calls (including parameters and return codes), user stack, and open files. An example of such a tool would be `crash` under Unix or `ProcessViewer` under Windows NT.

The second group, for inter-component visibility, includes techniques that provide both static and dynamic visibility into logical and physical protocol streams, state information, procedure calls, and data exchange. Such snooping is applicable to software component interfaces and hardware (physical) interfaces. Snooping can occur at any point where data and control are extended past the boundary of a component's environment. This can include inter-component communication across process (and also processor) boundaries through parent/child communication, remote procedure calls, client/server communication, and dynamic data exchange. Likewise, such snooping is not limited to network traffic (e.g., Ethernet, FDDI, etc.), but can be used on other physical transport layers such as RS-232 or SCSI. An example of such a tool would be `etherfind` (or `snoop`) under Unix or `DDESpy` under Windows NT.

The third group, for extra-component visibility, includes objective measurement techniques for performance, audit logs, system throughput, resource utilization, response time, and similar system-level behavior. This kind of data provides for early detection, causal analysis, and intervention when a component or subsystem of components goes into failure, repair, or upgrade. These data are not typically provided by individual tools, but are gathered from log files, transaction audits, and so forth.

By using techniques such as these, a third-party integrator can determine the point of failure in a COTS-based system. The present case study relied heavily on some methods and techniques from

the intra- and inter-component categories. However, all of these techniques are valuable, and some are indispensable.

3 The Case Study: A Server-Based Query System

This case study illustrates the application of the method (described in Section 2) to gain insight into a system based on a number of commercial off-the-shelf components. The system in question is illustrated in Figure 2.

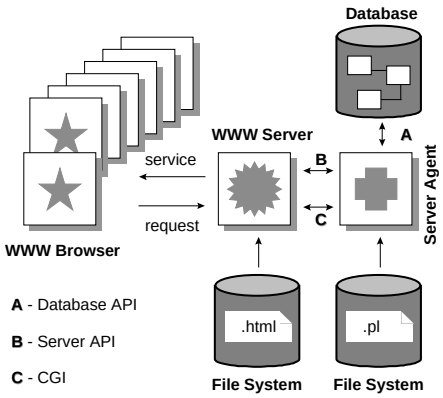

Figure 2 Web Server/Database Diagram

This system provides dynamic access through the World Wide Web (WWW) to data and information contained in a commercial database. The system is primarily composed of a COTS WWW server and a COTS relational database management system (RDBMS) operating on a Unix platform. It was necessary to communicate between the Web server and the RDBMS, and it would have been possible for us to create the needed infrastructure to do this. However, the database vendor also sold a product (labeled as the "server agent" in Figure 2) that provided this capability. This COTS product made use of both the proprietary interface (labeled A) to the RDBMS, and a proprietary interface (labeled B) to the WWW server. Additionally, the server agent supported the publicly defined Common Gateway Interface (CGI, interface C). For our system, we chose Interface B since it provided faster execution and a reduction in system resources in comparison to the CGI protocol.

The system was hosted on a multi-processor hardware platform with 1 Gigabyte of primary system memory and 32 Gigabytes of secondary storage. The system was further serviced by two redundant T-1 lines to ensure connectivity and adequate throughput. Operationally, the system was intended to behave in the following manner:

1. A customer with a WWW browser connects to the WWW server and is presented with a query form.
2. The customer enters data in the form in the search fields of interest and submits the modified form to the WWW server for processing.
3. The WWW server invokes the server agent and passes it the form data.

4. The server agent connects to the RDBMS, issues the query, and collects the requested data set.

5. The requested data set is formatted and sent back to the customer through the channel established by the WWW server, thereby completing the transaction.

This operational behavior was expected to take place simultaneously for several dozen customers, over any number of iterations, without failure. At its initial use, however, there were numerous complaints and problem reports that the system was either not responding or was simply too slow. There could be several possible causes of this behavior. Without source code, however, it was necessary to isolate the different behavioral elements of the system to find the actual fault.

In correcting the problems in this system, we followed the iterative method described earlier. We made four iterations through the "observation-hypothesis-prediction-experiment" cycle, and successfully repaired the system.

3.1 Diagnostic Iteration 1

The observation that began the first iteration was the problem report that the system was too slow. Given the system's intended operational behavior, it seemed reasonable to surmise that the database queries were the guilty party and were probably slowing access to the WWW server. If this was indeed the case, then a prediction could be made that the execution of *any* query to the WWW server would block access to the site for the amount of time needed to perform the query.

To test this hypothesis and prediction, a test harness for the WWW server was created (see Figure 3). In this harness, a simple C program simulates one or more customers connecting to the WWW server. These "browser skeletons" do not fill in any fields of the query form, but simply perform two tasks:

- connect to the WWW server and make an `http` request for the home page
- report the amount of time it takes to perform this task using the Unix `ftime()` call

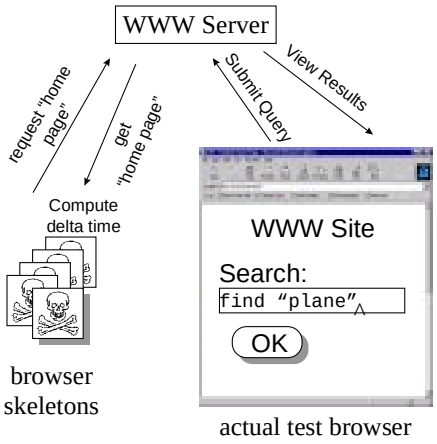

Figure 3: WWW Server Test Harness

The browser skeletons were launched to begin recording the performance of the WWW server. After a few minutes of operation, the skeletons exhibited an average response time of approximately 0.175 seconds.

Next, an actual (i.e., human-operated) WWW browser was launched and connected to the WWW server. The operator filled in the query form and submitted it to the WWW server. While the operator performed this step and waited for the query response to return, the performance of the browser skeletons was observed. Now, all the browser skeletons paused much longer than the previously recorded average response time before resuming. This test was repeated; in all cases, the browser skeletons would pause for approximately 19.0 seconds (two orders of magnitude longer) and then return to the average response time.

The RDBMS's SQL interpreter was then launched to execute the same query against the RDBMS (away from the WWW server environment), to determine the actual length of query execution time. The query execution time correlated to the 19.0 second pause in the browser skeleton.

After repeating this experiment a number of times with different queries submitted to the WWW server, the observed latency of the browser skeletons in the test harness correlated to the time it took to execute a submitted query, thereby blocking the WWW server. The experiment thus supported the hypothesis that queries to the database were slowing access to the WWW server.

3.2 Diagnostic Iteration 2

There was now first-hand evidence that, during the execution of a database query from the WWW server, the system was no longer responsive to other customer connections to the system and was serializing customer requests. The next hypothesis, based on this observation, was that the WWW server was failing to spawn child processes (as would be expected) to handle simultaneous inbound requests. This hypothesis led to a prediction that during the servicing of customers, there would be no child processes spawned by the server.

We reused the test harness to perform this experiment. While the browser skeletons were keeping the WWW server busy, we probed with the native Unix `ps` command (report process status) to observe the state of all processes. If our prediction about child processes were true, then the number of processes (specifically related to the WWW server) should remain the same and no child processes should be spawned. However, upon execution of this experiment, this prediction was not borne out, since several other processes were actually observed. The output from the `ps` tool is shown in Figure 4. There were four child processes running associated with the WWW server (parent process with PID equal to 11379).

```
$ ps -ef
UID    PID   PPID   C   STIME  TTY      TIME COMD
               :
nobody 11379     1   3   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11380 11379 47   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11381 11379 20   Jun 24 ?        0:20 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11382 11379 20   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11383 11379 32   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
               :
```

Figure 4 Output from ps

There were two key observations that resulted. First, we noted that the command line argument to the WWW server, `/export/home2/home/httpd-jc/config`, was a directory containing installation configuration settings for the WWW server. One of these configuration settings was a tunable system parameter used at startup of the WWW server for the number of child processes to spawn; in this case that parameter was set to four. Thus, there should be, and were, four child processes spawned. The second observation was more interesting: we noted that one of the children (PID #11381) had accumulated twenty seconds of CPU time while the other children and the parent had accumulated none.

The hypothesis that no child processes were being spawned to handle customer requests was proven false by this experiment. But this diagnostic iteration did provide valuable information and insight into an aspect of the inner workings of this WWW server:

- confirmation that the WWW server did use child processes to manage inbound requests, and the existence of a tunable parameter currently set to four
- one of the child processes was accumulating CPU time and the others were not

3.3 Diagnostic Iteration 3

Based on the observations from the last diagnostic iteration, the previous hypothesis was modified: it might be that, of the child processes spawned, not all were handling inbound requests. If this was so, then it should follow that some of the children would be idle, while other children would be busy handling requests. Further, given the observation that only one child was accumulating CPU time, then it was reasonable to predict that all but one of the child processes would appear idle.

To test this hypothesis and prediction, the experiment from iteration two was revisited, but instead of using `ps`, the Unix command `truss` was used to trace system calls. This would give insight into what each child was doing when the system was stressed.

On execution of this experiment, the parent was observed as suspended pending receipt of a signal (see Figure 5). Examination of the first child process showed a healthy child handling inbound requests to the WWW server (see Figure 6). As requests entered the system, the healthy child would unblock from the `poll` system call, `read` the request, `write` the response, and continue the cycle. If no requests were inbound, the child remained idle waiting for `poll` to unblock.

```
# truss -p 11379
pause()                         (sleeping...)
```

Figure 5 WWW Server Parent State

However, the rest of the child processes (PIDs 11380, 11382, and 11383 shown in Figure 4) were not idle, but were behaving as shown in Figure 7. Whether or not requests were inbound, these children all continued with the same pattern of activity, failing in a specific system call (`fcntl`), returning an error status of `EAGAIN`, and trying again. This indicated that all of these other child

processes were attempting to set a lock (i.e., F_SETLK) on a file or segment of a file that was already locked by another process.

```
$ truss -p 11380
poll(0xEF4EFB80, 3, 120000)                     = 1
getcontext(0x00067A08)
setcontext(0x000727F8)
            :
read(22, " G E T   /   H T T P / 1".., 8192)    = 16
fstat(24, 0xEF401BA0)                           = 0
time()                                          = 867697774
getcontext(0x000727F8)
setcontext(0x00067A08)
            :
poll(0xEF4EFB80, 3, 300000)                     = 1
getcontext(0x00067A08)
setcontext(0x000727F8)
            :
writev(22, 0x00074E34, 3)                       = 3061
time()                                          = 867697774
write(15, " 1 2 8 . 2 3 7 . 3 . 7 6".., 72)     = 72
close(22)                                       = 0
```

Figure 6 WWW Server Child State (1)

When the healthy child terminated, the parent woke up from its suspended state (see Figure 5) to spawn a new child. At this point a race condition existed between the other three existing children (trying to set locks) and the new child. Only one could set the lock, after which the other three fell into the same failing status shown in Figure 7.

```
$ truss -p 11381
poll(0xEF4EFB80, 0, 1000)                       = 0
getcontext(0x00067A08)
setcontext(0x00065BA0)
fcntl(20, F_SETLK, 0xEFFFF7D4)                  Err#11 EAGAIN
getcontext(0x00065BA0)
setcontext(0x00067A08)
poll(0xEF4EFB80, 0, 1000)                       = 0
poll(0xEF4EFB80, 0, 0)                          = 0
getcontext(0x00067A08)
setcontext(0x00065BA0)
fcntl(20, F_SETLK, 0xEFFFF7D4)                  Err#11 EAGAIN
getcontext(0x00065BA0)
setcontext(0x00067A08)
```

Figure 7 WWW Server Child State (2)

It is useful to summarize these observations:

- When an actual query was submitted to the WWW server, the healthy child would make calls to obtain a connection to the RDBMS server.
- The healthy child would then make a `read` system call and block until the query completed execution.
- While the healthy child waited for the results of the query, the other browser skeletons were blocked.

These observations support parts, but not all, of the hypothesis. The expectation that not all of the children were handling inbound requests *was* supported. But rather than being idle, the child

processes were actually in an infinite loop and were trying to obtain a lock on a resource; this could not occur until the process holding the lock had terminated.

However, we also made another critical observation. We noted the WWW server was making use of the system calls `getcontext` and `setcontext` (as indicated in Figure 7), which are used in user-level threaded applications (we determined this by reading the Unix manual pages for these calls and using the documentation and configuration settings in the directory identified in diagnostic iteration two). This implied that the WWW server used user-level context switching between multiple threads of control within the child processes. This led to the next—successful—hypothesis.

3.4 Diagnostic Iteration 4

As noted in the initial description of the system (see Figure 2), the server agent we chose conformed to both the proprietary API defined for the WWW server and the publicly defined interface (the CGI protocol) between the WWW servers and associated helper applications. There are two fundamental differences between these coordination approaches (see Figure 8).

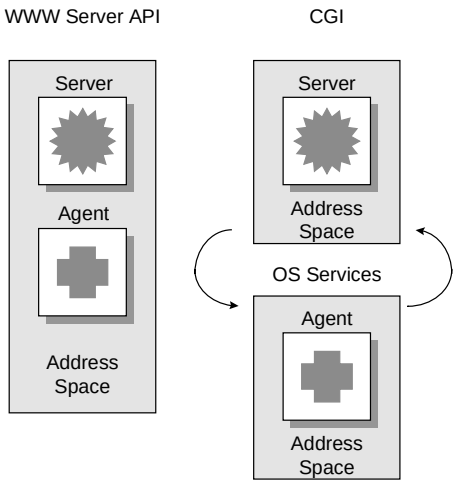

Figure 8 Server-Side API vs. CGI

The first difference involves the execution environment in which the WWW server and the server agent are run. In the WWW server-side API, the server agent is loaded, through a shared library, into the same address space as that of the WWW server. In contrast, the CGI protocol assumes a typical parent/child relationship between the WWW server (parent) and the server agent (child), thereby separating the address space and maintaining separate processes.

The second difference involves the mechanics by which the WWW server and the server agent communicate. In the WWW server-side API, the WWW server vendor defines structures and library callback routines that it expects to use to communicate with the server agent. Since the WWW server and the server agent share the same address space, once loaded, the communication between the two components is much like that of any tightly coupled application/library interface. However, in the CGI environment, the communication is loosely coupled, relying heavily on operating system services for interprocess communication. For CGI applications, these primarily consist of command line arguments and environment variables.

Given that the WWW server was actually a multi-threaded application using user-level threads, we had noted in the previous iteration that the working child process was not threading as expected on queries to the database. The hypothesis for this iteration was that the COTS server agent was not multi-threaded (or was not multi-thread safe) and was causing the WWW server to block. If this was the case, then replacing the server agent using the WWW server-side API with its CGI equivalent should solve the blocking problem.

To test this prediction, we first had to modify a number of configuration files for the WWW server to disable the loading and use of the server agent using the WWW server-side API. Next the suspect server agent had to be removed from the WWW system and reinstalled using the CGI version of the server agent. Finally, we recreated the initial test environment using the test harness created during the first diagnostic iteration. After a few minutes of operation, the browser skeletons in the test harness exhibited an average response time of 0.175 seconds. This was consistent with the response time seen earlier.

The actual test browser was launched and the test query was submitted to the WWW server for this iteration. At this point the query was submitted to the WWW server. This time, the browser skeletons in the test harness operated normally with little variance in the average response time, and after approximately 20 seconds, the test query returned the expected result to the actual test browser.

The test query was resubmitted a second time so that the process status (`ps`) of the WWW server and the child process could be observed during query execution (see Figure 9). The parent WWW server and four child processes appeared in the process table as expected. Further, during the execution of the submitted query, a child of one of the child processes (PID 14310) could be seen running. This confirmed that the server agent was using the CGI protocol, evidenced by the separate process execution of the server agent outside of the process space of the original child process.

```
$ ps -ef
UID    PID   PPID   C   STIME  TTY      TIME COMD
              :
nobody 14203     1  3   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 14204 14203 47   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 14205 14203 20   Jun 24 ?        0:05 ./httpd -d /export/home2/home/httpd-jc/config
nobody 14206 14203 20   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 14207 14203 32   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 14310 14205 20   Jun 24 ?        0:16 ./cgi-bin/webagent /cgi-bin/query.pl
              :
```

Figure 9 WWW Server Using CGI Agent

We confirmed during this iteration that by replacing the WWW server-side API version of the server agent with the CGI version of the same agent, the WWW server was no longer blocked when a query was submitted to the system. The evidence in this case supports the hypothesis that the non-CGI server agent was causing the blocking of the WWW server and the slow system behavior that had been reported.

4 Summary: Results of the Study

4.1 Final Post-Mortem of the Experience

The system discussed in this case study actually fell prey to more than one source of failure. The first failure (observed in diagnostic iteration 3) was that the child processes, intended as multi-threaded applications, were not properly threading as expected by the component developer. This drastically reduced the intended performance behavior down to that of a *single*, multi-threaded application. In this case, the parent was the overseer of all the child processes; its responsibility was to keep the number of (first generation) children at a constant. The parent was not responsible for responding to external service requests. Further, all but one of the child processes ignored external requests, since they were failing to get a resource needed to perform their duties. This left a single child process to handle all external requests to the WWW system.

The second failure (observed in diagnostic iteration 4) was that the server agent was causing the child process to block when the agent's services were invoked through the WWW server-side API. This caused a mismatch between the intended behavior of the WWW server's child process and the mechanisms used by the server agent (which were not multi-threaded). The combination of the two failures in this system caused a complete collapse in performance. On one hand, only one process was actually handling inbound service requests. On the other, the functioning child process was blocked for a period of time while submitted database queries were executed, thereby serializing customer access to the WWW system.

The data and information gathered in this case study were reported to the WWW server vendor and the RDBMS (and server agent) vendor. The WWW server vendor confirmed the problem of the failing multi-threaded child process discussed in diagnostic iteration 3. Further, we learned from the vendor that this problem only affected their WWW server running on a particular version of the operating system (which was used for this WWW system). A patch was later released by the vendor changing the child processes over to kernel-level threads (rather than user-level threads), thereby solving this problem.

4.2 Resources Required for Debugging the System

At the onset of system failure, we had discussions with both the RDBMS vendor and the WWW server vendor about the nature of the problem and the probable cause. Much of the discovery phase resulted in a number of messages about the system's configuration and environment. Based on information from the vendors we could only make futile attempts to "*tweak*" the system. We made additional attempts to reproduce the problem in the vendor's setting. One person spent approximately two staff-weeks on the effort during this discovery phase.

A succession of failing patterns of "*give a little, try a little*" with the vendors led us to the diagnostic iterations discussed in Section 3. After making little progress, it was clear that we needed to do something more. The actual process discussed in the case study took two staff-days to complete—that included the diagnosis, re-tooling the server agent for the CGI protocol, and documenting the findings.

In addition, the normal release process for testing, quality assurance, configuration management, and release packaging added an additional 30 calendar days (thus 45 days in total) to the schedule

(resulting in an overrun) and an additional two staff-months of effort before the repaired WWW system could be put into full operational status.

5 Afterword

Ultimately, it is the integrator who is responsible for delivering a system, and it is the integrator who makes a system function successfully. The corollary is that no one is (or should be) more motivated to repair a failing system than the system integrator. In general, the onus is on the system integrator to prove that a COTS component is at fault and guilty of system failure.

The need for expertise in system engineering when constructing a COTS-based system is at least on a par with, and perhaps greater than, that typically sought in traditional 'ground-up' development. The integrator will need technical prowess, diagnostic and deductive insight, and a willingness to try many approaches, if he/she is to build complex systems successfully using a mixture of commercial and custom components. In this case study, it is clear that if the integrator had not had the ability and knowledge to drill into the inner-depths of the COTS components that were incorporated into the WWW server, longer delays would have occurred.

Feedback

Comments or suggestions about these monographs are welcome. We want this series to be responsive to the real needs of government personnel. To that end, comments concerning inclusion of other topics, the focus of the papers, or any other issues are of great value in continuing this series of monographs. Comments should be sent to:

Editor
SEI Monographs on COTS
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
cots@sei.cmu.edu