Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

**SEI Monographs on the Use of Commercial Software in Government Systems**

# Isolating Faults in Complex COTS-Based Systems

Scott Hissam
David Carney

*February 1998*

# 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 overview of a method for isolating and overcoming faults in COTS-based systems. It provides a method and mechanisms that are useful for engineers and integrators who are tasked with assembling complex systems from heterogeneous sources. While other readers may find value in this report, it is specifically written for a technical audience. The method described in this monograph has been used on various systems. One such use is described in the SEI monograph *Case Study: Correcting System Failure in a COTS Information System.*

# Isolating Faults in Complex COTS-Based Systems

## 1 Introduction: Some Realities of COTS-Based Systems

There are many advantages to using commercial off-the-shelf (COTS) components in modern systems. The first and most obvious advantage is lower cost, but others include immediacy of acquisition, greater ease of modernization, and reduced development costs.

This monograph describes one method for debugging COTS-based systems. The method is based on several real instances where systems were fielded, failed, and needed correction.[1] This paper describes the conceptual basis of the method and suggests various mechanisms through which it can be used to isolate faults in complex, COTS-based systems.

## 2 Why Things Go Wrong

There is an unavoidable truth that accompanies the decision to use COTS components: Individual component evaluation will typically not illuminate all of a given component's shortcomings, no matter how extensively an assessment is done. Further, as the complexity of a system *increases* (e.g., system capability, heterogeneity, number of components to be integrated), the likelihood that all components and their interactions can be accurately appraised *decreases*. This means that the use of COTS components magnifies the problem that plagues all complex systems, namely that system failure may require extensive and difficult debugging to isolate the source of the problem.

System failures can occur for many reasons. When commercially-based systems fail, however, some circumstances particular to the use of COTS components become apparent (e.g., dependence on vendor priorities, loss of support for old releases). Possibly the worst circumstance among these is the absence of source code. Debugging becomes a different, and much more difficult, activity when source code is unavailable. Even the basic meaning of "debugging" shifts. Debugging no longer centers on finding erroneous code (i.e., "bugs"), but focuses on observing the behavior of large, opaque components and making inferences from that behavior. It is also possible that the system may not have been integrated properly.

Regardless of the actual cause of failure, the person who does this "debugging" is not the engineer who created the component and knows its internal workings, but the integrator who combined it with other components into the complex system: This will be the focus of this monograph. The outcome of the integrator's "debugging" is not correcting source code, but rather determining whether, and how, the component's unexpected behavior can be reconciled with the desired behavior of the whole system.

Finally, even when the source of failure is isolated in a commercial component, getting the vendor to take notice and repair the fault may prove difficult or impossible. Many system integrators have found themselves at an impasse because a vendor is unable or unwilling to divert

1 One of these instances is documented in another monograph in this series. See *Correcting System Failure in a COTS Information System*, Scott A. Hissam, SEI Monographs, September 1997.

resources, change release schedules, or accommodate requests for corrections or improvements to the product.

These three points—insufficiency of evaluation techniques, system opaqueness, and the vendor's response to trouble reports—can affect the task of isolating and correcting faults in a COTS-based system. We will examine each point below.

## 2.1 Insufficiency of COTS Software Evaluation Techniques

Current techniques for software evaluation are immature and unreliable. This is especially true of techniques for evaluating COTS products. Testing is not easily done, for instance, and a large part of understanding of the component depends on the vendor's claims for the product. Another problem in evaluating COTS components is that it is rare for two components to be fully comparable. More often the evaluator will be comparing two components, each of which satisfies only some subset of the desirable features. The evaluator must balance conflicting imperatives, since the system's customers seldom differentiate between "hard" and "soft" requirements.

A final problem in evaluating COTS components appears, paradoxically, after the fact. After COTS selection, evaluation will likely cease. This means that any unidentified shortcomings in the chosen component will probably not be found until much later in the system's life cycle. This may be at a point where previous purchasing or licensing commitments or schedule factors mandate the use of the component, no matter how severe the shortcoming.

## 2.2 System Opaqueness

A major complication for any system is the extent of its diversity. As the number of components to be integrated and the number of possible combinations increase, so does the likelihood that failures will occur in one of three possible places:

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

If the system fails, fault isolation becomes progressively more difficult: Is the failure caused by one component? If so, which? Is failure caused by an interaction of components? If so, which combination?

While these questions are applicable to any kind of system, the presence of commercial components adds a new level of complexity. To the integrator of a COTS-based system, isolating the fault involves first ascertaining *how* the components work and then finding out *why* they fail. This complexity results from the integrator's lack of visibility into the components and lack of control over their workings.

The integrator may also lack insight into the design philosophies, assumptions, or rationale that of the component developers. The integrator's view of the requirement being addressed and the role the component has in implementing a solution may be different from that of the component's developer. For example, many POSIX-compliant operating systems support loadable device drivers. The initial intent of these device drivers was to allow new devices to be added without

having to rebuild the operating system kernel. Most vendors assume that these device drivers will be loaded once upon booting the system. However, it is possible for a third-party integrator to base the design of his system on the ability to load and unload these drivers many times during system execution. This leads to a potential mismatch of assumptions: The vendor is not concerned with releasing resources (since the drivers were not expected to be reloaded without rebooting the system), and the integrator expects that drivers can be loaded frequently. This mismatch creates a situation in which the device drivers may fail to reload, may fail to operate as expected, or may corrupt system integrity.

### 2.3 A Vendor's Response to Trouble Reports

When a COTS-based system fails, the initial tendency is to rely on the COTS vendors to provide a solution. In most cases, the system integrator has the task of proving to one vendor that *his/her* component is at fault. When dealing with multiple vendors, it is very likely that each vendor will suggest that the fault lies not with his/her component but with someone else's. This is perhaps an understandable reaction: Customer complaints are often unfounded, and a vendor's perceived inaction can be the result of a lack of sufficient evidence upon which to act.

However, when serious technical shortcomings in a particular COTS component exist, getting the component's vendor to correct the shortcomings is often the best way to achieve a solution. To bring this about, the integrator must collect and analyze as much hard evidence as possible. A typical first action is to call the vendor's support line. Presuming that the integrator gets to speak with a knowledgeable engineer (or more unlikely, the actual developer), he/she will need to describe the problem in as much detail as possible. At best, the integrator will convince the component's developer that a correction is necessary, which will typically be introduced in some future release of the product. At worst, the component developer will refuse to make any change to his product. In either case this means that the integrator must find a workaround, or the system cannot function.

To summarize, if the system's failure is due to the interaction of two or more components, then it is very improbable that the vendors will be able to offer any substantive assistance. If the failure is due to a single component and the cause of failure can be clearly identified to the vendor, then it is possible that the vendor will provide a solution. But in either of these cases, it is far more likely that the integrator will have to debug the system. And this requires a very different notion of "debugging" than exists for a system whose source code is available.

## 3 A Method for Debugging COTS-Based Systems

The method we propose defines a systematic approach for diagnosing and correcting failures in COTS-based systems. It is a method that derives from the classic scientific method of observation, hypothesis, and prediction. This method defines a systematic approach for observing cause and effect (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 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, refinement of the hypothesis and subsequent experimentation are needed.

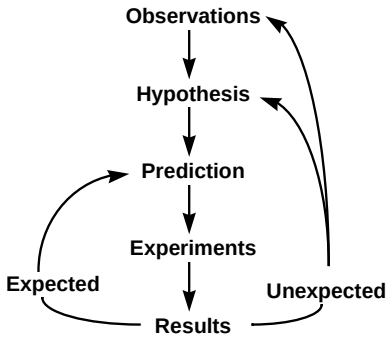



**Figure 1: The Scientific Method**

This method, applicable to many sciences, has equal validity in software engineering, for traditional software debugging uses the method implicitly. For example, in a hypothetical software system, a problem report (the observation) indicates that a sorting routine fails when the number of elements in the list gets too big. An initial assumption (the hypothesis) is that the list is bounded by a hard-coded fixed limit; the prediction is that a review of the source code will reveal a constant that provides the list's upper boundary. If the review does not locate the constant (thus contradicting the hypothesis), the hypothesis must be refined and other debugging activities will begin.

This iterative sequence is very different when the system makes use of COTS components because now the components are opaque: Their *behavior* can be observed, but their inner workings are not visible, so predictions about their code have no meaning. To illustrate this point, assume the previous hypothetical scenario of a defective sorting routine. The first recourse described above is no longer available. Instead, it becomes necessary to isolate the problem by observing the system's behavior in one or more contexts. Thus, a hypothesis that conforms to this context (i.e., the failure of the sort function) might be that scarce memory resources adversely affect the sort routine. A prediction could be made 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. We can then build an experiment to consume available memory resources and re-run the sort component with a controlled data set; if the component fails as expected, then the hypothesis (i.e., that the sort component is affected by low memory conditions) has been verified.

Notice the different approaches used by these examples. When source code was available, it was fairly straightforward to determine whether a constant upper bound was the cause of the problem. If this was not the cause of the failure, then successive iterations could have included debug statements, code profilers, static and dynamic analyzers, and other techniques readily available to code developers. In the second case, however, such means are not available to the integrator. He/she must choose a diagnostic path that manipulates the environment in which the component operates to observe cause and effect.

Given the assumption of this method (that there is something to observe) and given that with COTS components, such observations—and the insights they provide—cannot depend on examining the source code, it is obvious that other types of visibility must be found. We now describe some mechanisms by which integrators can gain visibility into COTS-based systems.

# 4 Mechanisms for Component and System Observation

There are many tools and techniques that provide visibility into COTS components. The choice of tool or technique is based on the kind of component and the type of suspected failure. If the component itself is suspect, the most useful mechanisms are those targeted towards peering through the component's outer boundary. If the failure is thought to be between two components, then the mechanisms will look at the interplay between the suspected components. Finally, if the entire system (or subsystem) has failed, then the techniques needed will be those that can assess a group of components working in unison.

All of these mechanisms represent "observation posts" that provide insight into off-the-shelf components. We summarize these vantage points as follows:

- *Intra-component visibility:* tools that observe the behavior of an individual component; they seek to understand and illuminate the component developer's assumptions and intended use of the component.
- *Inter-component visibility:* tools that observe the behavior of two or more components; they seek to understand potential mismatches between components.
- *Extra-component visibility:* techniques that observe a system of components in ensemble; they seek to understand macro-level issues dealing with performance, maintenance, misfits, etc.

## 4.1 Intra-Component Visibility

Tools in this category provide insight into the behavior of an individual component *in itself*. This is generally accomplished by *probing*. These tools obtain visibility into parent/child relationships, thread performance, user stack, resource utilization, signal and event disposition, system calls (including parameters and return codes), and open files. Table 1 provides a sampling of tools available to perform some of these types of probes.

| Probing Tool | System Availability | Function |
|---|---|---|
| `ps` | Unix and Unix derivatives | parent/child relationships, aggregate resource utilization |
| `trace/truss` | Unix and Unix derivatives | system calls, parameters, return codes, error codes |
| `crash` | Unix and Unix derivatives | detailed resource utilization, signal disposition, open files, user stack |
| `Process Viewer` | WinNT/Win95 | memory allocation, processor utilization, thread information, API calls |
| `Spy/Spy++` | WinNT/Win95 | view/manipulate message streams for a window, thread, or process |

**Table 1: Probing Tools**

These tools let the observer see how a component makes use of the native operating system's resources. Since such information tends to be highly specific to individual operating systems, tools such as these are typically provided with the operating system or development environment. Reference material about these tools is generally available to help the integrator understand the

tools' resources, values, limits, settings, and codes. This material is commonly in the form of manual pages, documentation, API references, library references, and help files.

These probing techniques help the integrator to get past the "black box" notion surrounding off-the-shelf components, i.e., that all internal workings and behaviors of a COTS product are hidden. The specific benefits that can come from "peeking inside the black box" include

- ability to create detailed problem reports for the component's developer, thus providing the evidence needed to convince the developer to address the problem quickly
- placement of the integrator on a more level playing field with the developer; thus shortening the communication loop
- increased understanding of the underlying assumptions and design characteristics used by the component developer
- enhanced ability of the integrator to quickly evaluate new releases or component repairs

Thus, probing techniques can improve the understanding of a component's internal behavior and reveal inherent ambiguities, thereby increasing the ease with which a component can be integrated.

## 4.2 Inter-Component Visibility

Tools in the inter-component visibility category provide insight into the interactions between components. Such insight is generally accomplished by *snooping*.[2] These tools obtain visibility into logical and physical protocol streams, state information, procedure calls, and data exchange. Table 2 provides a sampling of tools available to perform this type of technique.

| Probing Tool | System Availability | Function |
|---|---|---|
| `Etherfind (a.k.a. tcpdump, a.k.a snoop)` | Unix and Unix derivatives | capture, view, and filter TCP/IP-based network traffic; view TCP and IP headers and data |
| `ipcs` | Unix and Unix derivatives | status and information about shared memory, message queues, and semaphore use |
| `protocol analyzer` | (Not OS specific) | capture, view, and filter analyzer-specific logical and physical network layers |
| `Object Viewer` | WinNT/Win95 | monitor OLE 2.0-enabled applications' interaction with the operating system |
| `DDESpy` | WinNT/Win95 | monitor dynamic data exchange in the operating system |

**Table 2: Snooping Tools**

[2] These snooping techniques are very often the same as those used for illegal and intrusive attacks on computer systems.

These tools let the observer see the interplay *between* components. This interplay is not limited to software component interfaces, but includes hardware interfaces as well. In fact, snooping can occur at any point where data and control extend past a single component's boundary. This can include: inter-component communication across process and processor boundaries, parent/child communication, remote procedure calls, client/server communication, and dynamic data exchange. Similarly, such snooping is not limited to network traffic (e.g., Ethernet, FDDI), but can be used on other physical transport layers such as RS-232 or SCSI.

The specific benefits that can come from snooping include

- isolation of the point of failure to a particular component pairing
- identification of deviations from standards
- understanding of component external interfaces (such understanding also provides a potential entry point for adapting a component)

Snooping techniques can thus improve understanding of the external behavior and interfaces of an off-the-shelf component and its interplay with other components within the system.

## 4.3 Extra-Component Visibility

Techniques in this category provide insight into the overall workings of the integrated components from the system- (or subsystem-) level perspective. The general technique involves finding objective measurements about the functioning of the system. These include static or dynamic data about performance, system throughput, resource utilization, and response time. This data is found in audit logs, error files, and similar sources.

The principal approach is to capture and analyze system-level behavior. The data that is obtained can be used for early detection, causal analysis, and intervention. This information is more valuable when data on the system (and appropriate subsystems) is collected over time, thus establishing a historical portrait for the system. As components are upgraded in the system, fluctuations from historical norms (particularly negative fluctuations) can be an early indicator of failure. For example, suppose that a network server for a corporate local area network is being upgraded to a computer with parallel processors. On completion of the upgrade, data collected after installation shows a downward trend in server response time. The realized performance impact was less than expected, and some further inspection of the system's behavior is necessary. It is also true that the same kind of data can indicate that the upgrade has improved the system, and the degree to which the change was positive.

The specific benefits that can come from these types of measurements include

- isolation of system/subsystem (and individual component) failure points
- calculation of the impact of new releases on performance (expected vs. realized)
- performance tuning hints

Essentially, such techniques improve the integrator's ability to objectively quantify and assess changes in the system, measure the effects of those changes, and optimize time and resources needed for debugging.

# 5 Some Uses of the Scientific Method

In this section we describe how the scientific method has been used by integrators in debugging COTS-based systems.[3]

## 5.1 An Example of Intra-Component Visibility: Debugging a COTS-Based Information System

The system in question is an information system that provided dynamic access, through the World Wide Web (WWW), to data and information contained in a commercial database. The system was composed of a commercial WWW server and a commercial relational database management system (RDBMS) operating on a Unix platform, and a COTS product called a "server agent," that provided an interface between the browser and the database. Late in the development cycle it was noticed that system performance was extremely poor for no apparent reason.

Four iterations of the debugging method described here were needed to correct the system. In the first, the execution of the RDBMS was confirmed as the point of failure. In a multi-user system like this, it is expected that "child processes" will be spawned to accommodate new queries, so the second hypothesis was that this spawning was not occurring. However, this hypothesis was not true: numerous child processes were present as the system executed.

We therefore conjectured that these child processes might be idle. Again, this hypothesis turned out to be false. However, in testing this hypothesis, we also found that the child processes were all trying (vainly) to set a lock on certain system resources. This failing attempt was immediately attributable to the server agent, in particular to its assumptions about the use of system "threads." The server agent was reconfigured to use a different protocol, thereby circumventing the use of system threads, and the degraded system performance disappeared.

## 5.2 An Example of Inter-Component Visibility: Debugging a Java Applet and CORBA Server Application

The system under investigation was a simple client/server application where a Java applet[4] communicated with a CORBA[5]-based server. The applet was to be loaded by a Java-enabled WWW browser, display a graphical-user interface (GUI), establish an IIOP[6] connection to the server, and invoke operations provided by that server. This system consisted of a legacy server for which a new front-end client was being developed. It was discovered during early development of this Java application that communication with the server was failing while the legacy client was continuing to operate properly.

3 A full description of this process can be found in: *Case Study: Correcting System Failure in a COTS Information System* (http://www.sei.cmu.edu/ /cbs/monographs.html).

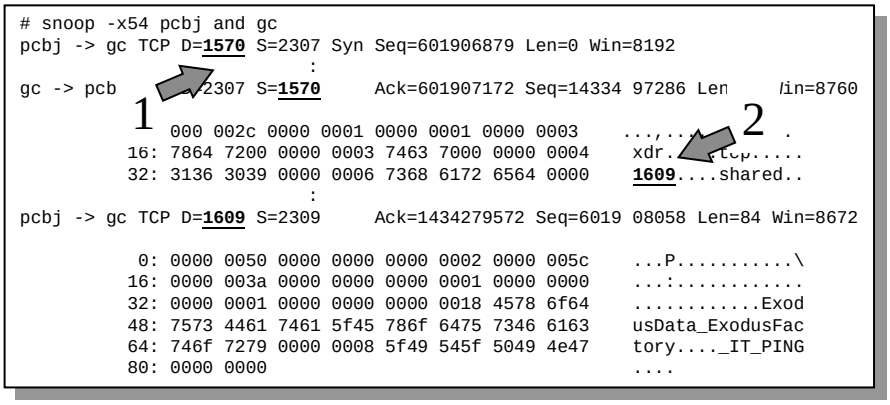
4 More information on Java can be found at http://www.javasoft.com. Applet is a term used to describe a thin client application that is loaded on demand by a Java-enabled browser such as Microsoft's Internet Explorer or Netscape's Navigator or Communicator browsers.

5 CORBA—Common Object Request Broker Architecture: more information can be found at http://www.omg.org.

6 IIOP—CORBA/Internet Inter-ORB Operability Protocol: more information can be found at http://www.omg.org/corba/corbiiop.htm.

An initial hypothesis was that the Java applet was not establishing a connection to the legacy server via IIOP or that the IIOP protocol between the client and server was failing. We predicted that by observing the network traffic between the legacy client and the server, we would see the client and server using the proprietary communications protocol while the new Java client's attempts at communication via IIOP would either go unanswered or appear to fail.

To test this prediction, we used a Unix network analysis tool called snoop. In the first stage of the experiment we isolated the network traffic that was present during a healthy session between the legacy client and server. Figure 2 shows some of what was observed during that experiment.




```
# snoop -x54 pcbj and gc
pcbj -> gc TCP D=1570 S=2307 Syn Seq=601906879 Len=0 Win=8192
                        :
gc -> pcb  D=2307 S=1570     Ack=601907172 Seq=14334 97286 Len     Win=8760
                        :
          000 002c 0000 0001 0000 0001 0000 0003    ...,...... .
         16: 7864 7200 0000 0003 7463 7000 0000 0004    xdr.....tcp.....
         32: 3136 3039 0000 0006 7368 6172 6564 0000    1609....shared..
                        :
pcbj -> gc TCP D=1609 S=2309     Ack=1434279572 Seq=6019 08058 Len=84 Win=8672

          0: 0000 0050 0000 0000 0000 0002 0000 005c    ...P...........\
         16: 0000 003a 0000 0000 0000 0001 0000 0000    ...:............
         32: 0000 0001 0000 0000 0000 0018 4578 6f64    ............Exod
         48: 7573 4461 7461 5f45 786f 6475 7346 6163    usData_ExodusFac
         64: 746f 7279 0000 0008 5f49 545f 5049 4e47    tory...._IT_PING
         80: 0000 0000                                  ....
```

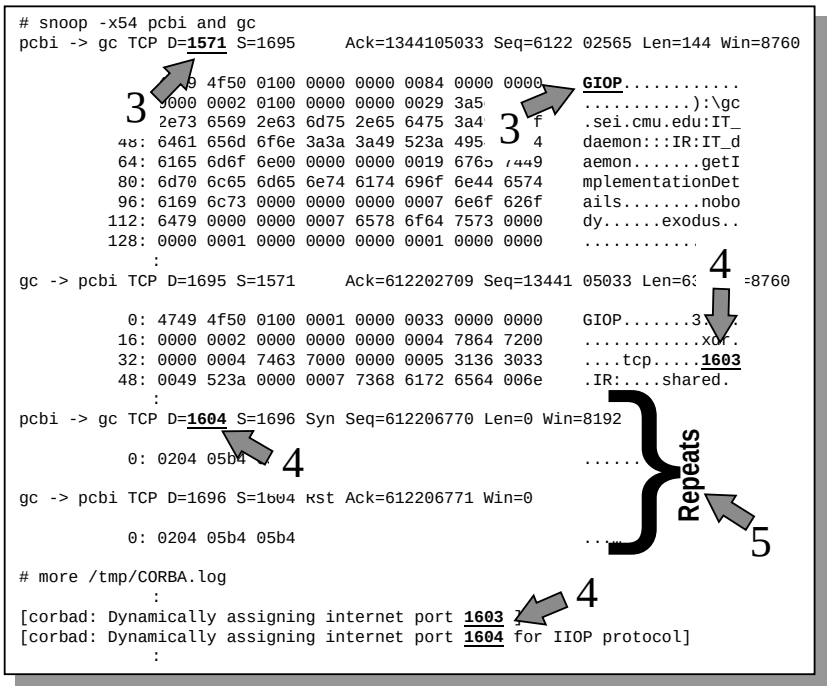
**Figure 2: Snooping Healthy Legacy Application**

There are two significant points in Figure 2:

1. The initial communication with the CORBA services occurs on TCP port **1570**, which for this system is the well-advertised port number for this vendor's proprietary communications protocol.

2. As can be observed through Item #2 in Figure 2, data returned from the server host shows that TCP port number **1609** is used in subsequent communication between the legacy client and server.

With knowledge of the healthy system, the second stage of this experiment was to observe the network traffic for the failing portion of the system. The same tools were used and some of the observations appear in Figure 3.




```
# snoop -x54 pcbi and gc
pcbi -> gc TCP D=1571 S=1695     Ack=1344105033 Seq=6122 02565 Len=144 Win=8760

                 9 4f50 0100 0000 0000 0084 0000 0000    GIOP............
                0000 0002 0100 0000 0000 0029 3a5       ...........):\gc
                2e73 6569 2e63 6d75 2e65 6475 3a4   f   .sei.cmu.edu:IT_
              48: 6461 656d 6f6e 3a3a 3a49 523a 495   4   daemon:::IR:IT_d
              64: 6165 6d6f 6e00 0000 0000 0019 676   7449    aemon.......getI
              80: 6d70 6c65 6d65 6e74 6174 696f 6e44 6574    mplementationDet
              96: 6169 6c73 0000 0000 0000 0007 6e6f 626f    ails........nobo
             112: 6479 0000 0000 0007 6578 6f64 7573 0000    dy......exodus..
             128: 0000 0001 0000 0000 0000 0001 0000 0000    ............
                :
gc -> pcbi TCP D=1695 S=1571     Ack=612202709 Seq=13441 05033 Len=6   =8760

               0: 4749 4f50 0100 0001 0000 0033 0000 0000    GIOP.......3....
              16: 0000 0002 0000 0000 0000 0004 7864 7200    ............xdr.
              32: 0000 0004 7463 7000 0000 0005 3136 3033    ....tcp.....1603
              48: 0049 523a 0000 0007 7368 6172 6564 006e    .IR:....shared.
                :
pcbi -> gc TCP D=1604 S=1696 Syn Seq=612206770 Len=0 Win=8192

               0: 0204 05b4 0   4                             ......

gc -> pcbi TCP D=1696 S=1604 Rst Ack=612206771 Win=0

               0: 0204 05b4 05b4                              ...

# more /tmp/CORBA.log
                :
[corbad: Dynamically assigning internet port 1603 f
[corbad: Dynamically assigning internet port 1604 for IIOP protocol]
                :
```

**Figure 3: Snooping Failing Application**

There are three significant points in Figure 3:

1. The initial communication with the CORBA services occurs on TCP port **1571**, which for this system is the well-advertised port number for the CORBA/IIOP communications protocol. Use of IIOP is confirmed through the "`GIOP`" marker found in the data stream (which is consistent with the CORBA/IIOP specification).

2. As seen in Item #4 in Figure 3, data returned from the server host shows that TCP port number **1603** is returned, but **1604** is actually used for subsequent communications. According to the log file generated by the CORBA daemon, **1604** is the port number specifically assigned to the legacy server for CORBA/IIOP connections.

3. Item #5 in Figure 3 shows that subsequent attempts to communicate on TCP port **1604** repeat and then fail once the timeout threshold is reached. The Java applet then terminates.

It was clear from the experiments and the resulting observations that the legacy server was refusing to operate on the assigned CORBA/IIOP port as expected by the CORBA specification. The problem was not with the Java applet or Java class libraries but with the legacy server. This information was provided to the maintainers of the legacy server and proper configuration changes were made to permit the legacy server to interoperate with the Java applet via the CORBA/IIOP specification.

## 5.3 An Example of Extra-Component Visibility: Debugging a Distributed Relational Database System

The system in question was a typical distributed three-tiered application that used user-level components; middle-layer servers containing application-specific logic, business rules, and COTS wrappers; and back-end relational database servers (RDBMS). The system served developers, testers, and users from the operational area as well. Based on user input, the middle-layer servers would make queries to the RDBMS, aggregate the resulting data sets, and summarize those results for the user. After deployment, an intermittent error began to occur that could not be consistently reproduced: if the user resubmitted a failing request, it might not occur again.

One hypothesis was that the code in the middle tier was responsible for the error. Since these modules had been developed in-house, they were suspected as a source of error. However, an alternative hypothesis (and the one that we acted upon) was that the fault actually lay with the relational databases.

Since previous attempts to reproduce the errors consistently had failed, we needed to take a system-wide view to locate the source of the error. We began a detailed analysis of the available system logs and error reports for our initial observation. We analyzed log files from five different middle-level servers; two of these were from the operational area (`OPSrv1`, `OPSrv2`), two were from the development area (`DEVSrv1`, `DEVSrv2`), and one was from the test area (`TSTSrv`). Data from each of these servers was plotted against the date on which the error occurred; the results are shown in Figure 4.

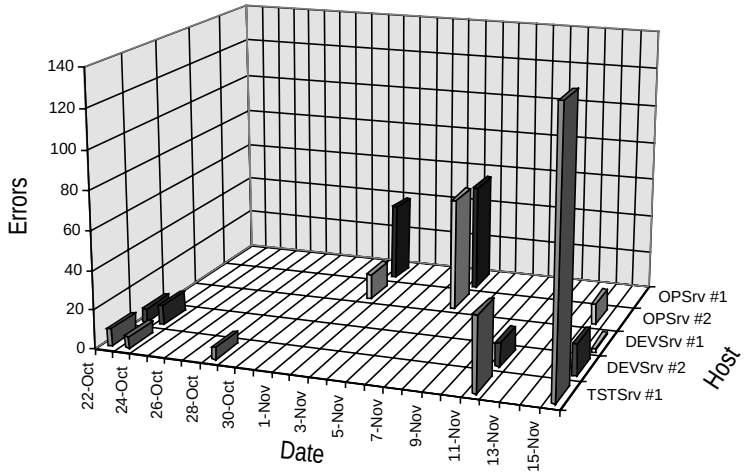



**Figure 4: Distribution of Errors by Data and by Server**

Some significant patterns of behavior emerged:

1. `TSTSrv`, `DEVSrv1`, and `DEVSrv2` (we will call these server group 1, or SG1) show a significant correlation of the error on four dates.
2. `OPSrv1`, and `OPSrv2` (server group 2, or SG2) showed a comparable correlation for the error on three different dates.
3. Errors were thus clustered to either the operational servers (SG1) or the development/test servers (SG2).

The working hypothesis was that the relational databases were responsible for the intermittent errors; we then needed to provide sufficient evidence to support this hypothesis. What the data from the analysis shows is that the servers of SG1 and SG2 were failing in the same manner, but at different times. Since each group was isolated, the stimulus that provoked the failure was chronologically unique to each server group (SG1 was affected by stimulus1, SG2 was affected by stimulus2). Figure 5 shows that each of the server groups interacts with users (at the top tier) and RDBMS (at the lower tier). Since the error originated after the query was submitted to the RDBMS, the evidence was strong enough to warrant an in-depth investigation into the lower tier (the RDBMS).

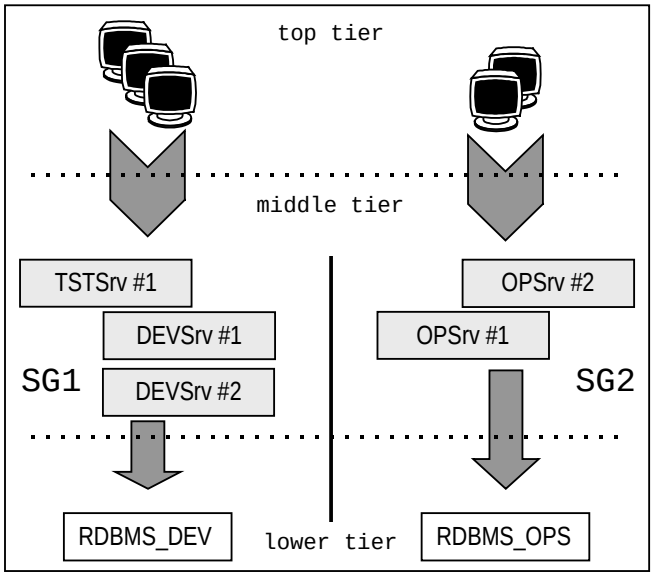



**Figure 5 High-Level System Configuration**

As shown in Figure 5, there is a distinct parallel between the pattern of behavior we noted and the high-level system configuration. Each portion of the system used an independent data source for the lower tier; in one case it was an operational RDBMS, in the other a Dev/Test RDBMS. These rules indicated that the resource constraint that was causing the error somehow lay with these databases.

We therefore examined the lower tier RDBMS searching for resource constraints and erroneous conditions. We eventually determined that temporary database table space was being exhausted

as the databases responded to queries after long periods of time. These failures were dependent on high load factors (thus explaining the inconsistency), and the resulting failures cascaded through the entire system.

# 6 Summary

The increasing prevalence of COTS in today's systems is bringing new challenges to systems development and engineering as we know it. By their very nature, COTS components are difficult to evaluate carefully, are opaque to the integrator, and are not open to traditional source-code based debugging. While newly evolving evaluation techniques and improving vendor support are hopeful signs, ultimately the onus is on the system integrator to diagnose and solve problems when things go wrong.

As we have shown in this monograph, the scientific method is applicable to software engineering and is especially applicable in those systems involving COTS components. As we have also demonstrated, observation is the key to this method, and obtaining useful observation into COTS components is often more challenging than traditional debugging.

Sophisticated tools exist to aid the integrator in building the observational evidence needed for system repair and in establishing a better understanding of the COTS component and its underlying assumptions. However, a structured method for using these tools is needed. We have described one such method in this monograph. In addition to this method and tools, the integrator will need technical prowess, diagnostic and deductive insight, and a willingness to try many approaches to build complex systems successfully using a mixture of commercial and custom components.

# 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