WHITE PAPER

Distributed application performance troubleshooting with proof

Troubleshooting can be described as the process of finding out why something isn't working as expected and determining what to do about it. For our purposes, it begins with the premise that something isn't performing as desired (the measurement metric), then progresses to gathering data and identifying possible causes. Analysis may rule out some causes and focus on others according to the depth of our understanding of the system. Choosing a likely cause, the theory is tested; success means the performance problem has been solved, while failure loops us back through the process.

In the perception of many, troubleshooting expertise carries with it a certain mystique. This is why certain auto mechanics, physicians or network gurus have such elite reputations. But with the proper tools, techniques and practice, much of the mystery becomes clear. In this paper, we will focus on the techniques while relating these to examples of recent practice.

What is performance troubleshooting?

There exist a number of different well-defined troubleshooting methodologies (root cause analysis and bottleneck analysis are two); their goals and resulting value are shared. We could define the common goal as either:

a) return performance to normal (because something has changed)

b) investigate how to improve performance (because we continually improve quality).

The common value is based on our ability to achieve the goal as quickly and efficiently as possible. This value can be measured in terms of improving system efficiency and applying minimal resources.

It is interesting to note that one common approach to problem resolution, trial and error, is actually not a troubleshooting methodology at all. In the extreme case, we begin by attempting solutions without appropriate analysis. When (and if) we find one, we may then determine the cause (by reverse-engineering). Not only does this fail to provide the value of troubleshooting—it is certainly not quick and efficient—it frequently will introduce additional problems.

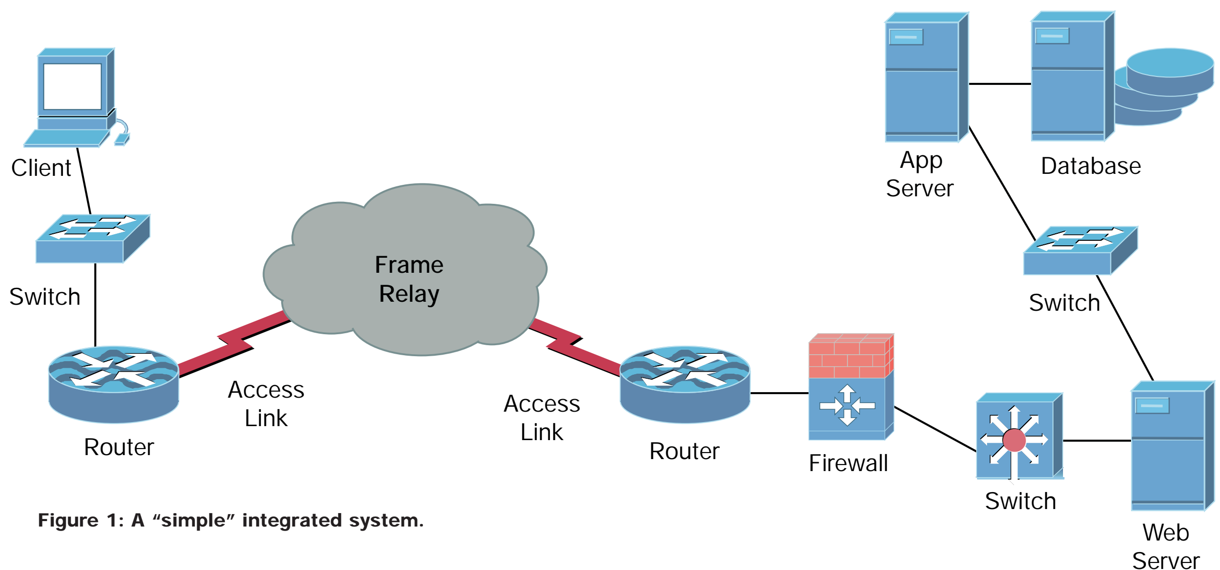


Figure 1: A "simple" integrated system.

For troubleshooting distributed application performance, no enterprise relies solely on trial and error. As network and systems architectures and infrastructures have become quite complex, monitoring and measurement instrumentation tracks the status, statistics and health of the components making up the integrated system. The first line of defense might be the up/down status of critical elements—WAN circuits, routers and servers, for example. If one of these fails, we can quickly initiate an appropriate corrective

response. Second, we may also monitor performance statistics of these elements—traffic volume across WAN circuits, CPU/memory utilization in a server, dropped packets and throughput for a router. Management of these statistics may help us define a "normal" range of behavior and generate alarms when thresholds exceed these ranges. Third, based on experience or testing, we may apply intelligence or rules to relate element-level statistics to overall system performance. If a particular link experiences load of greater than 75 percent, for example, we learn ERP application performance becomes unacceptable. With this knowledge, we could generate a proactive alarm when link utilization reaches 70 percent. These last two form the foundation of growth management, whereby "future alarms" can be forecasted based on growth rates.

The ROI of effective troubleshooting

Even in a well-instrumented enterprise, performance problems occur outside the boundaries of anticipation and measurement via instrumentation. The dynamic nature of the environment—carriers rerouting links, unforeseen congestion, device configuration changes, etc.—means there will always be cases where the instrumentation fails to give us advance warning. In fact, if our infrastructure instrumentation provides us the answer 80 percent of the time, we're in pretty reasonable shape. The first we become aware of these problems is when the help desk receives a call from a frustrated user or business manager. If the application is critical to the business, we scramble resources. Someone from the network group looks at routers and switches and makes a call to the carrier to check on link performance. Someone from security checks the firewalls. The webmaster checks the web server, the systems group checks the server logs and performance statistics, and the DBA traces database activity. Everything's fine, or at least operating within normal boundaries. At this point the network specialist takes some trace files and, using the expert assistance features of many protocol analyzers, discovers a number of errors. Unfortunately the errors are based on generic, often inappropriate, thresholds, and the actions taken to correct the problem have no impact. Days elapse, fingers point, results are demanded, late night hours are worked, vendors are blamed, mistakes are made—and money is lost.

What does this cost? Your best resources waste time and are removed from strategic projects. Relationships between operations groups suffer. Knee-jerk reactions, from inappropriate device configuration changes to ineffective addition of network bandwidth, create additional costs. The cost to the affected business depends on how many users are affected and the criticality of the application. Finally, your reputation as a responsive and capable IT shop takes a big hit. Months of perfection can be spoiled by a couple days of high-visibility wheel spinning. It should be easy to see why troubleshooting's goal (correct the performance problem) and value (do this efficiently) are important.

Let's take a step back and revisit the definition of troubleshooting. It is not trial and error, but rather like a combination of divide-and-conquer and bottleneck analysis. This does not mean divide into the smallest possible components, then work on each in isolation—that would be distributed trial and error. We quantify the significance of each component in relationship to its contribution to the whole, then analyze contributing bottlenecks. We can't do this by using internal component metrics—cache hits, link utilization or packet retransmissions are almost meaningless unless viewed in the context of the whole system. (Sure, there are cases where individual component analysis might correctly identify the problem, but this is simply trial and error with a dose of luck.) And that is where we started—we measured the system in terms of end-user response time, so we need to perform our analysis in the same context.

Application performance triage

So what do we do when all the lights on our monitoring consoles are green, yet we still have a performance problem? To achieve our troubleshooting goal and corresponding value, we need to quickly and accurately identify the performance bottleneck and assign the appropriate technical resource. We refer to this as "application performance triage," in that we are primarily concerned with classifying the problem for appropriate attention. Triage's goal is to assess quickly with enough supporting evidence that the recipient of the problem will accept it, in other words, to avoid the resource scramble.

There are three steps to performing application triage:

- *Measure.* Start with the failing metric of end-user response time, since that is typically what triggers the triage process.
- *Analyze.* Determine how the application interacts with the client, server and network's quality of service (QoS).
- *Prioritize.* From the analysis, define relevant constraints and assign to appropriate technical resource.

Let's examine each of these in more detail.

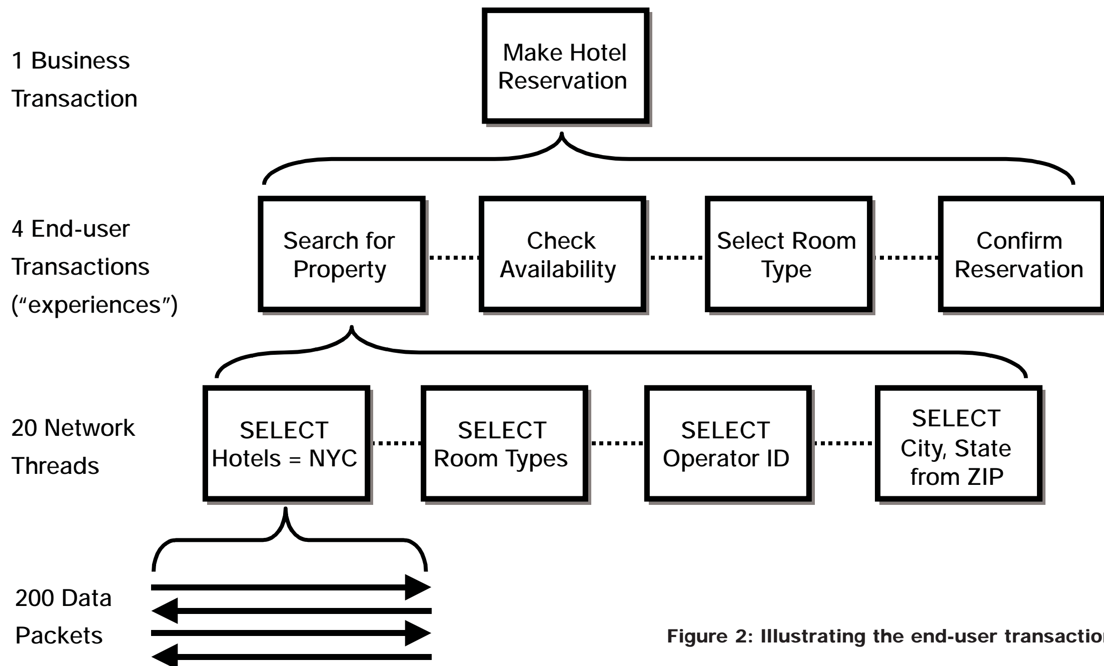


Figure 2: Illustrating the end-user transaction.

Measure

We can describe the problem as end-user (or transaction) response time; however, there are more than a few different definitions of the term transaction. For our purposes, we can describe this as an end-user perceived unit of work—enter key to screen update. It’s important because this represents the period of time that the system is doing work on the behalf of a user. Figure 2 should help clarify this definition.

Effective measurement has always been the most problematic part of troubleshooting—what should be captured from where. First, there is no *single point* of measurement that can possibly provide a complete and accurate end-to-end performance analysis of a distributed transaction. This is, in fact, the primary limitation of solutions to date—their single-segment or single-node view of a distributed system. If we measure from the client, the network and server become lumped together; likewise, if we measure from the server, the network and client time are indiscernible. While it occasionally may be possible to infer or interpolate these values from a single point, in reality, this only works well for simple transactions where the network characteristics (particularly congestion) are consistent. This might be OK for a lab environment, but not for a production network—particularly in a troubleshooting context. For true client/network/server breakdown and consistently accurate performance triage, we need to measure the transaction at both the client and server locations simultaneously, and then correlate these measurements into a single view.

To determine the contributions of network delays as well as client and server processing delays, we need to capture packet traces of the transaction; these will enable both the lower-layer network analysis as well as the upper-layer application analysis. So the first step is to implement a capture solution at both the client and server locations, preferably one that does not require two protocol analyzers, two experienced users and a long-distance phone conversation. If you already have implemented this type of capture instrumentation, great—you have addressed the primary logistical resistance to effective triage. If not, you should consider a software-based solution that does not require expensive hardware platforms, one that can be installed on clients and servers alike (discussed further in the Technology Implementation section of this paper). Keep in mind that the triage approach does not rely on detailed packet-level protocol analysis; it uses the packet traces to evaluate and quantify constraints on transaction performance.

It is important to set capture filters so that each trace file contains only the network traffic pertaining to the transaction under test. For instance, you may set a filter to capture traffic between the client and server IP addresses and ignore all other network traffic. With filtering set, simply start the captures, have the client run the offending transaction, then stop the captures and save the files.

Analyze

This is the core of the solution. We want to classify different types of application behavior so that we can quickly identify the sources of measured client, network and server delays. To accomplish this, we will need to relate the transaction measurements (trace files) to a basic understanding of TCP flow control and to key components or indicators of application behavior. To illustrate how this is done, we will use a few simple timing diagrams of measured transaction traffic between a client and server. Once we define a few key terms, we can interpret the diagrams, identify possible failure modes and suggest appropriate corrective action. We can actually define seven categories into which all application performance problems fall; each of these is illustrated by an example.

Prioritize

Since the analysis is thorough and informative, effective decisions can be made as to appropriate actions and expected results. We can replace expensive trial and error and educated guesswork with focused tuning and a well-understood ROI. The analysis quantifies various performance bottlenecks, so you can anticipate performance improvement resulting from tuning or removing these constraints and evaluating the benefits in relation to their cost. Remember, though, that the triage is primarily a classification exercise; while it will often pinpoint the source of a problem, there will be occasions where additional analysis tools and specific expertise will be required. The examples will reference these tools where appropriate.

Getting started

Merging packet traces

Before we begin, we must first correlate the client and server data—not a simple task. To avoid problems inherent in machine clock synchronization, we will assume that the trace files are merged using a packet-matching algorithm, as simplified and abstracted in Figure 3.

As we analyze the resulting merged data, we will identify various components of delay—essentially how our response time budget is spent within the system. For each node (client and server), we will identify node processing time and node sending time. For the network path (between the client and server), we will quantify the impact of bandwidth, latency, congestion and errors.

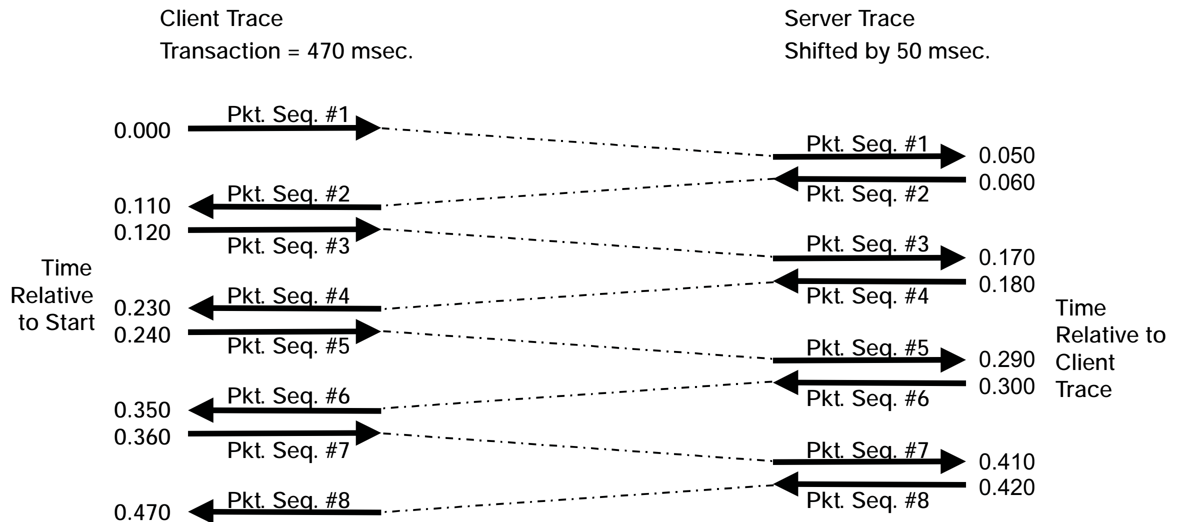


Figure 3: Merging two trace files of the same transaction.

Diagramming transaction traffic

To facilitate discussion of the analysis, we will use simple packet timing diagrams to illustrate different classes of transaction performance problems. These diagrams represent each network packet flowing between client and server by an arrow: light arrows from client to server are request packets, dark arrows from server to client are reply packets, and dashed arrows from client to server are TCP acknowledgements that do not include application payload. The transaction begins at the top of the graphs; time elapses from top to bottom.

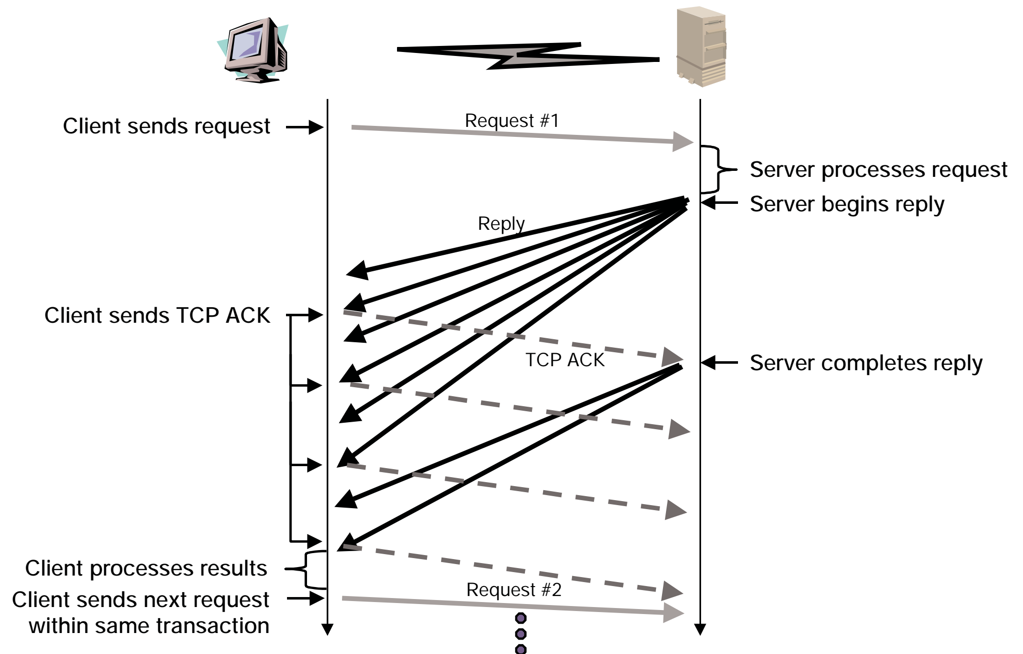


Figure 4: A visualization of packet flow for a simple transaction.

Figure 4 represents part of a simple transaction; note that there are usually multiple request/response sequences within a single end-user transaction.

Key definitions

We need to define a few terms to provide the semantics allowing us to relate application interaction with network traffic:

- *Application packet.* A packet that carries application-layer payload (as opposed to a TCP acknowledgement packet, which carries no application payload).
- *Application turn.* A series of application packets making up one complete request-response sequence. An application turn also can be described as a change in the direction of traffic flow at the application layer. This is quite important—since connection-oriented applications usually operate in a request-response paradigm, the more application turns, the more sensitive the application is to network delays. Some application development software vendors refer to application turns as “network round-trips.”
- *Application flow.* A sequence of application frames in one direction (from one node to another). An application flow is the sequence of packets that make up a single application request or reply, and is one half of an application turn.
- *Node sending duration.* The elapsed time between the first and last frames in an application flow. Measuring the efficiency of an application flow (as a sending duration) provides valuable insight into the transaction’s performance.

- *Node processing duration.* The elapsed time between the receipt of the last packet of one flow and the transmission of the first packet of the next flow (in the other direction). Also referred to as request-reply timing, this is usually only interesting in the context of the entire transaction, since these timings often vary wildly within a transaction.

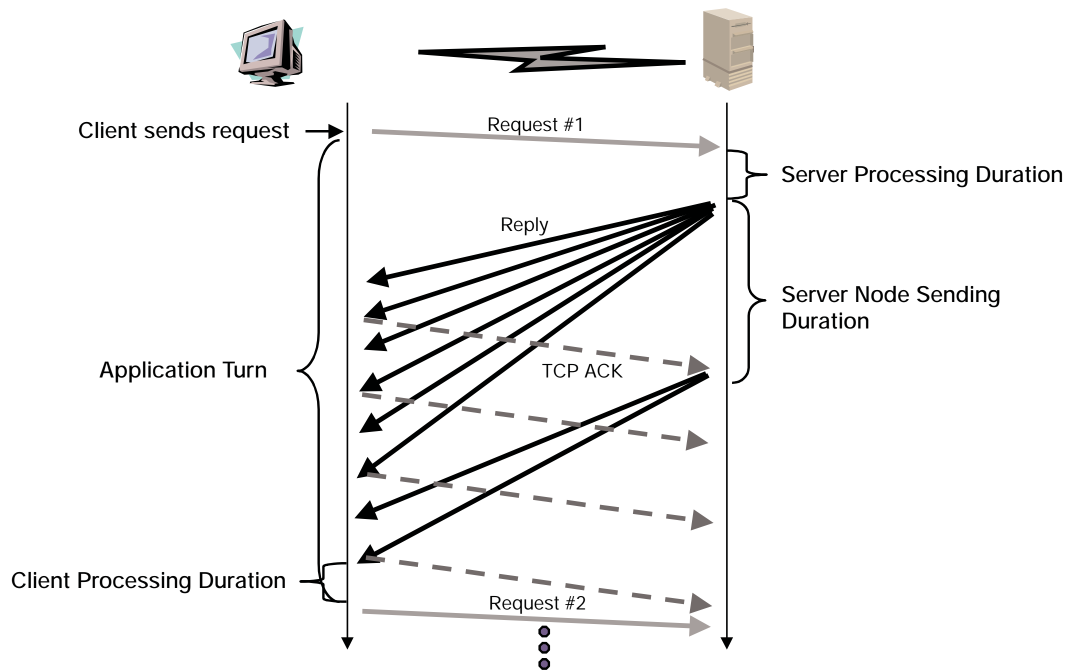


Figure 5: Visualization of key terms.

Classifying performance problems

We'll first classify the transaction's performance into one of three basic categories; every performance problem will fall into one (or more) of the following:

- *"Pure" node processing*—a delay or series of delays caused by client or server processing.
- *"Chatty" transaction*—one that is severely affected by latency or congestion based on the number of application turns required to complete the transaction.
- *Node sending delays*—where some other constraint is preventing the application from utilizing all of the available bandwidth.

We will need to define sub-categories for the node sending class, but let's look at the other two cases first.

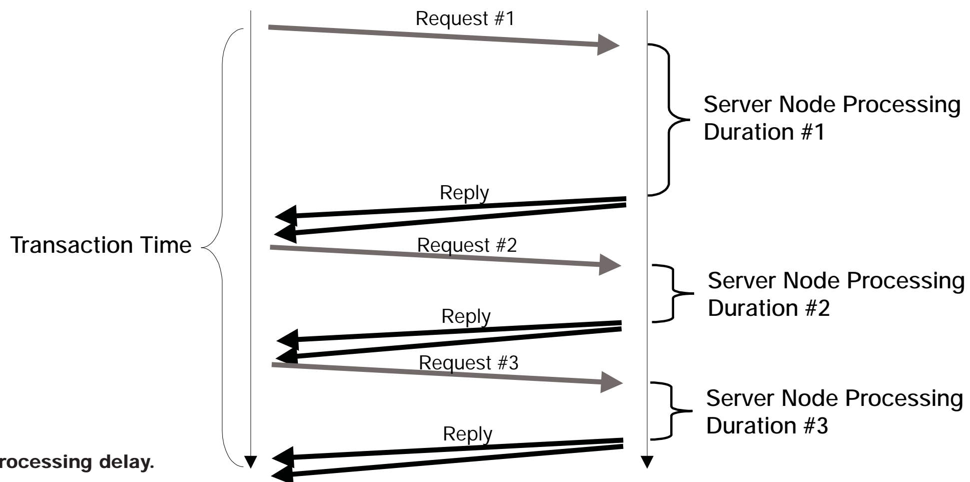


Figure 6: Server node processing delay.

Node processing delay analysis

This is one of the simplest cases to visualize. Referring to Figure 6, there will be one or more node processing delays, seen as gaps between request and reply flows. The total node processing time for the transaction is the sum of each individual processing duration.

Node processing delay actions

The first step is to identify the application thread or threads within which the most significant delays occur. This might be a particular SQL statement, a GET or POST request, an RPC, etc. For a transaction with multiple requests, you may find that all processing durations are slightly longer than expected (longer than the baseline or "normal" measurement). This means you are probably faced with a server resource bottleneck. If you find a single thread consuming most of the processing delay, you may have the opportunity to investigate this further. For instance, SQL performance might be easily tuned by a DBA, while CGI script or Active Server Page performance might be improved by configuration tuning or adding specific resources. In any case, identifying the thread lets you provide meaningful information to the developer or DBA—much more valuable than pointing a finger and saying "your server is slow."

If you decide that improving server performance is appropriate, you should take steps to instrument the server resources so that you understand where the bottleneck occurs—CPU, memory, disk I/O, etc.

If the application is multi-tiered, remember that the triage process treats the back-end servers as part of the first tier; processing delays related to any of the servers will appear as first-tier server delays. There may be clues to help you narrow down the resources you monitor on these back-end servers. For a web-based application, for example, are all client requests slower than normal, including locally stored static content? Or is just the active content that requires data from outside the web server environment (CGI scripts, ASP pages, etc.) slow?

"Chatty" transaction analysis

This is most clearly represented by the classic case where an application that performs well in a campus environment is deployed to remote users, where performance is unacceptable. The reason? A high number of application turns. For an application to read 100 80-byte records, for instance, the client might request each record one at a time. This would result in 100 request-response sequences, and for a link whose round-trip delay is 120 milliseconds, this behavior will contribute 12 seconds to response time. Of course, network congestion will increase this delay and the resulting response time.

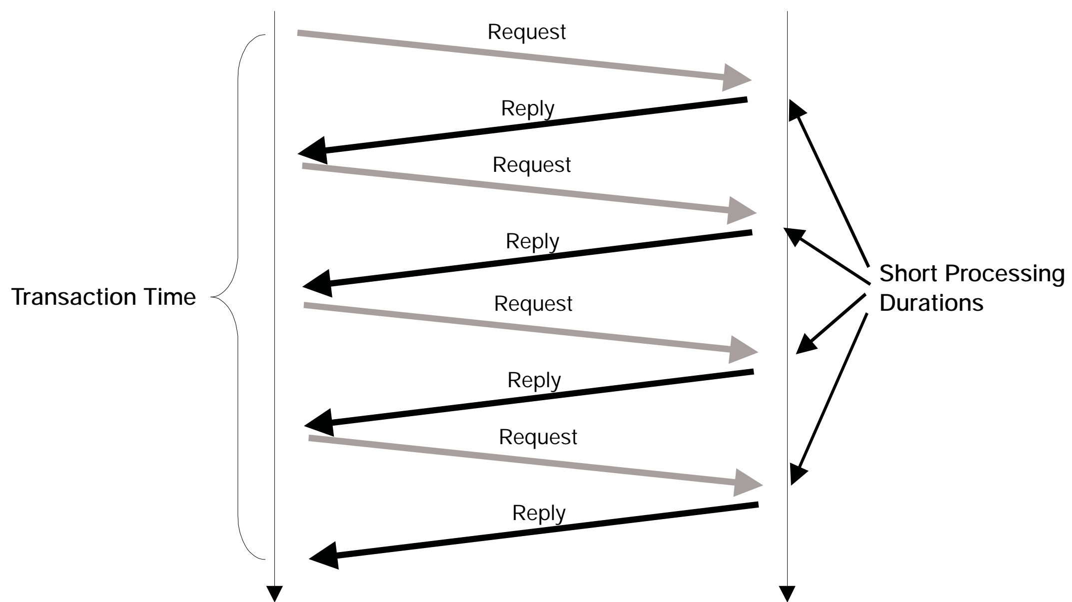


Figure 7: A chatty transaction.

Chatty application actions

Again, our first step is to narrow down the cause of the transaction's problem by identifying those threads which incur an inordinately high number of application turns. The worst possible case can be defined as two packets per turn—one request packet results in one reply packet, as would be the case reading our 100 80-byte records. The result is very inefficient use of network bandwidth, since only one small packet is on the network at any given time. (It is important to note that this has nothing to do with network packet size configuration; if the application only passes 80 bytes to the TCP/IP protocol stack, the resulting network packet will be small.)

If you are able to tune the application, you will want to reduce the number of application turns. For instance, in a database environment, a developer may use stored procedures instead of networked SQL calls. In a Java™ environment, using JAR files reduces the number of individual requests (and replies) for Java classes.

If you are unable to modify the application, there may be several steps you can take to improve performance across the network. Since the application is sensitive to latency, reducing latency will be the goal. If the link is frame relay, you may find that the carrier can reroute your circuit along a more direct path, reducing the distance your data travels and therefore the latency. Implementing a QoS policy for this application to provide a priority queue may help. This will ensure that this application's traffic is not delayed by other traffic, which might be less sensitive to delay. Reducing congestion will help in the same manner, allowing the router to forward the application's traffic more quickly, without waiting for other traffic. Determine which applications are using the link and decide whether some may be rerouted, given lower priority, served locally or restricted to certain off-peak time periods. Of course, if congestion is the source of the problem, you could add bandwidth to reduce congestion and improve latency. But be careful; adding bandwidth will not reduce base or inherent latency caused by distance and switching nodes (routers and switches). Finally, you may wish to evaluate a Windows Terminal Server solution. This popular approach to providing remote users access to database applications removes the effect of latency by co-locating the client software on the terminal server, sending only screen, keyboard and mouse updates across the network. You will need to determine whether this solution is appropriate for your environment.

Node sending delays

This is the "catch-all" category, where all other performance problems lie. Therefore, it is important to understand exactly what a node sending delay is. We have defined a node sending period as the time between the first and last frames of a traffic flow. In a perfectly tuned single-user system, the throughput of all the elements would match; the nodes would use all the bandwidth available to transmit data. A file transfer server, for example, would deliver data to the network as fast as the network can clock the data, while the client would receive it at the same rate. There would be no network flow control, no errors and no congestion. Transferring a 100MB file on a 100Mbps LAN connection under these conditions would take eight seconds. Some constraints, of course, are beneficial; TCP flow control ensures that multiple traffic flows can share the same network media, while servers and their operating systems are designed to service multiple concurrent users.

We need to analyze node sending durations carefully to determine how much time the client, network and server contribute. If the traffic doesn't utilize all of the available bandwidth, then there is some other constraint limiting performance. Identifying and quantifying this constraint will lead to effective performance improvement options.

There are five classes of node sending constraints; along with the two categories detailed earlier, these represent every type of application performance problem:

- Sending TCP stack "starved for data" (ready to transmit, but no application data)
- Network congestion (contention for network resources)
- TCP protocol effects (particularly TCP windowing)
- Network errors (particularly TCP retransmissions)
- Receiving TCP stack out of buffer space (unable to receive).

Sending TCP stack "starved for data" analysis

This represents the case where the network is "ready, willing and able" to transmit data, but the sending node has been delayed in placing additional data onto the network protocol stack. Note that Figure 8 is slightly different from the node processing delay diagram (Figure 6), where a processing delay occurs between a request and the beginning of a reply flow. In this case, the node has begun transmitting but temporarily runs out of data to send. The result is that the measured sending duration is longer than expected and data throughput suffers.

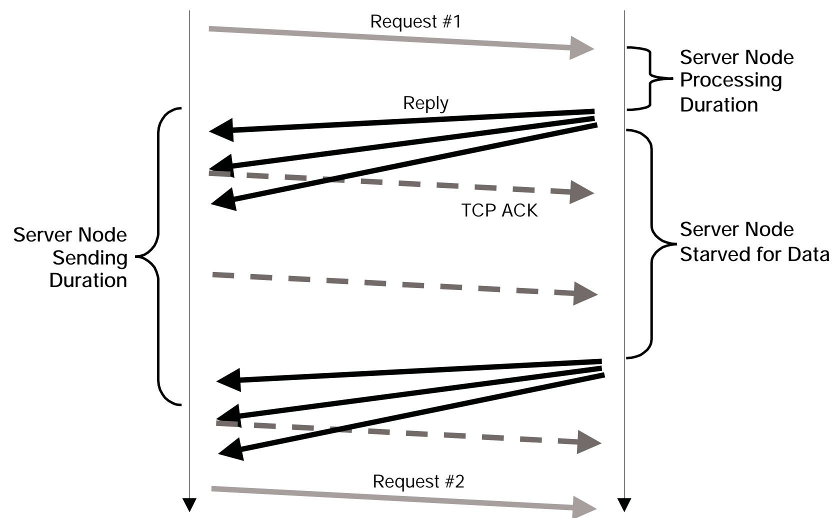


Figure 8: Sending node "starved for data."

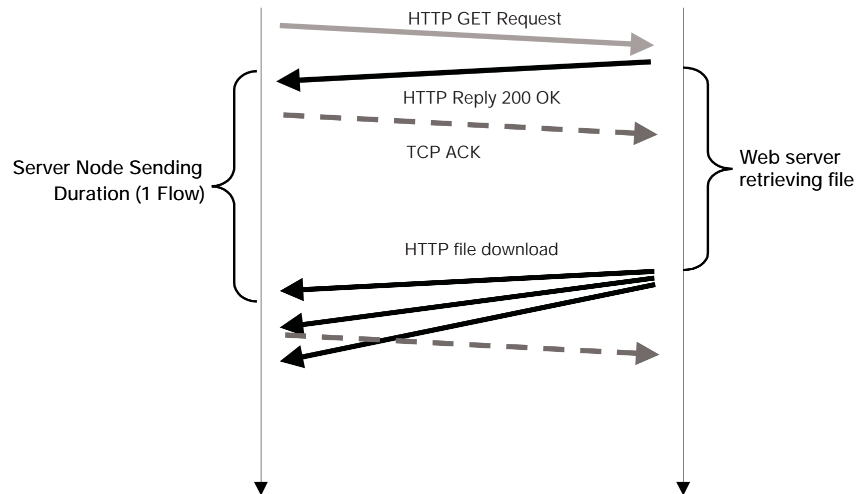


Figure 9: Node sending delay caused by server processing.

Sending TCP stack "starved for data" actions

Since the causes of this node sending delay are the same as those of a node processing delay, the actions of identifying long-running threads and measuring associated server resource constraints are also the same. Note that this particular case occurs rather frequently in web server environments, where the web server responds to a GET request (with a 100 "continue" or 200 "OK" message) and then begins to process the request. Once the reply is ready (which might include data from a back-end server), it is transmitted to the client. This is illustrated in Figure 9. What do you think would happen if you were simply measuring packet-level request/reply timing?

Network congestion and TCP protocol effect analysis

We'll look at these two conditions together, as they appear quite similar in behavior, although the corrective actions may be quite different. In both of these cases, the TCP window-based flow control inhibits data transmission. In Figure 10, we'll assume some basic TCP stack behavior: The client's receive window is 8760 bytes and the client will send a TCP acknowledgement every two frames or within 200 milliseconds after receiving a single frame. Network delays caused by distance (latency) or congestion will delay the transmitted packets on their way to the client. The server quickly sends as much data as possible (8760 bytes in six frames), and then must wait for the client acknowledgement. Since the data packets and the corresponding acknowledgements are delayed across the network, the server waits a significant amount of time between packet transmissions.

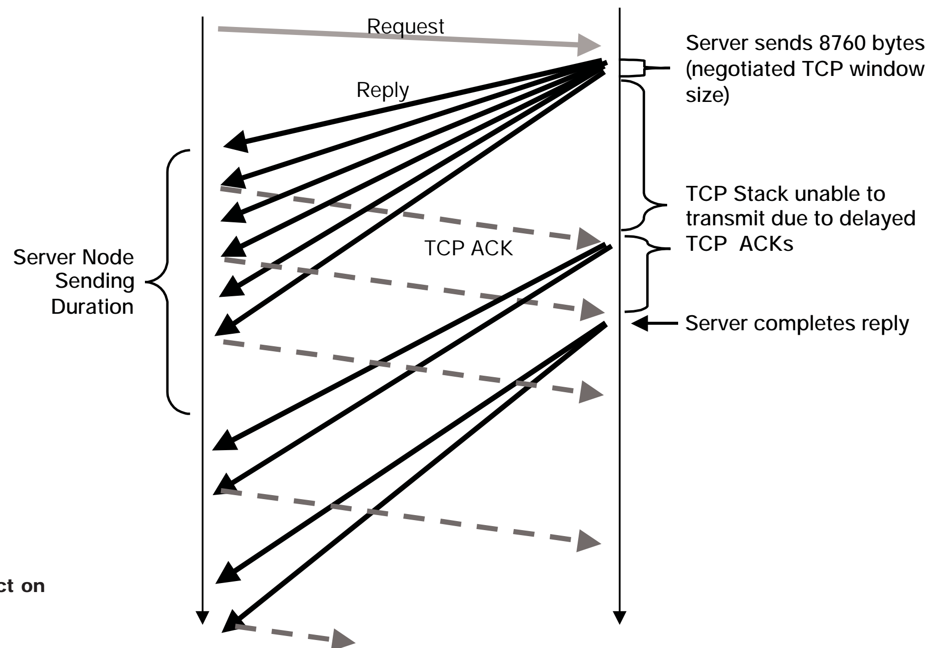


Figure 10: Latency's effect on TCP flow control.

Network congestion actions

If the delays are caused by network congestion, taking steps to reduce the sources of congestion will help. Determine which applications are using the link and decide whether some may be rerouted, given lower priority, served locally or restricted to certain off-peak time periods. Of course, adding bandwidth also will reduce congestion.

You might think that increasing the TCP window would help; in fact, a protocol analyzer might lead you to this conclusion, since it can only see one side of the conversation and therefore doesn't understand network delay. But this would only serve to exacerbate the problem, adding to the existing congestion caused by lack of bandwidth.

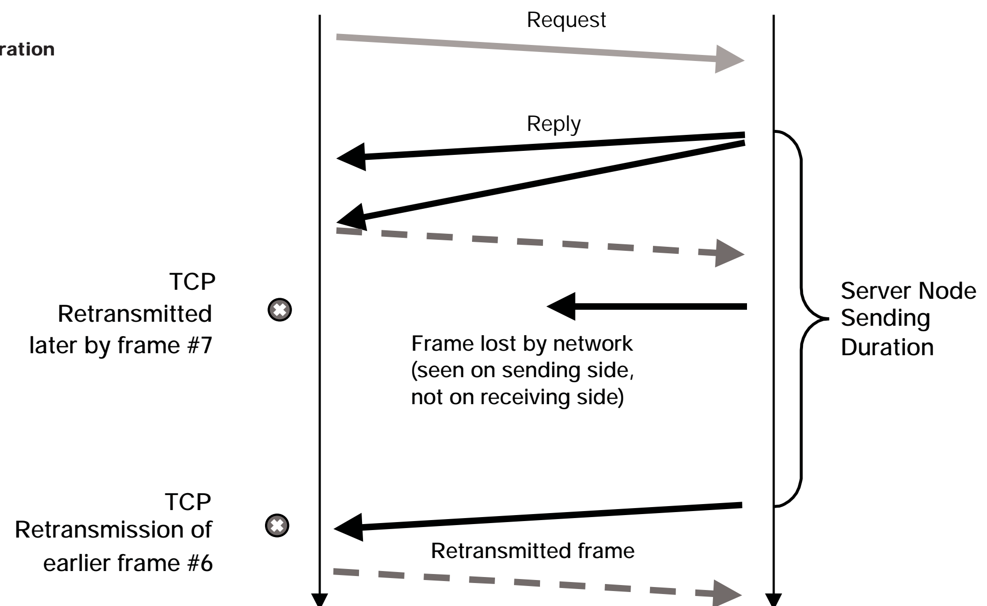
Network latency actions

If the delays are caused by distance—for instance, if the client location is in an overseas office—then you may very well achieve significant performance improvements by increasing the TCP window configuration. First, however, pay attention to the "burstiness" of the application by reviewing the size of the flows; if the largest flow is 4000 bytes, for example, increasing the TCP window from 8K to 32K will have no effect. Second, make sure there is enough bandwidth on the network path, since increasing the TCP window will permit the server to send data faster. Check with the application vendor to see if he or she has performed benchmarks or have specific tuning suggestions. You may wish to calculate the ideal TCP window size by referencing one of many documents available on the web, or you may prefer to experiment to get real-world results.

Network error analysis

Packet sequence numbers are used by TCP to ensure reliable end-to-end data delivery. When packets are lost by the network, this must be detected and corrected for the transaction to complete successfully. The impact that a dropped packet has on performance will vary depending in part on when (within a flow) it was dropped and how it was detected; as such, there are different ways this will manifest itself in a packet timing diagram. Figure 11 illustrates the case where one packet (the last in a flow) has been lost by the network. The server is expecting a TCP acknowledgement that the data was received; once its TCP retransmission timer expires, it will retransmit the packet. The configuration of the server's retransmission timer will determine the length of this delay. Obviously, multiple transmission errors can have a significant impact on throughput and therefore response time.

Figure 11: Long sending duration caused by dropped packet.



Network error actions

When network errors are prevalent, you will want to look for their source; often, they are related to an overloaded segment or internetwork device. You may want to look at router interface statistics or link monitors to help narrow down possible culprits. But be careful not to chase red herrings; errors may occur frequently in "normal" networks, and may in fact have very little effect on end-user response time. Use the node sending analysis to approximate how much delay they actually contribute.

You may also want to review your server configuration if you see consistent retransmissions at the receiving node. A TCP retransmission timer of 200 milliseconds may be quite effective for campus environments, but not long enough for international or Internet-based connections; the server will probably retransmit frames that were not actually lost. Conversely, too large a value will reduce performance for fast, but error-prone, connections. Use your system documentation as a starting point.

Receiving TCP stack buffer limit analysis

This final case, illustrated in Figure 12, represents the limitation of receiving node processing capability imposed on the sending node's permission to transmit data. As the receiving node accepts packets from the network, they are placed in a buffer controlled by the TCP stack where they sit until the application reads them. If the application itself is slow, the buffer becomes full and the receiving TCP stack must prevent the sending node from transmitting any more data until space becomes available. This is commonly done by sending a TCP acknowledgement with a TCP window size update of 0 bytes. This causes the sending node to wait until a new non-zero window size update is sent. In practice, upon receiving a 0-byte window update, the sending node will often send a packet with 1 byte of payload, actually retransmitting the last byte successfully received. (For this reason, these packets can be marked as "partial retransmissions" by protocol analyzers.) The corresponding acknowledgement will identify whether buffer space has become available or not. Note that some servers may "back off" exponentially, so that if the client doesn't automatically send an acknowledgement when buffer space becomes available there could be a rather lengthy delay.

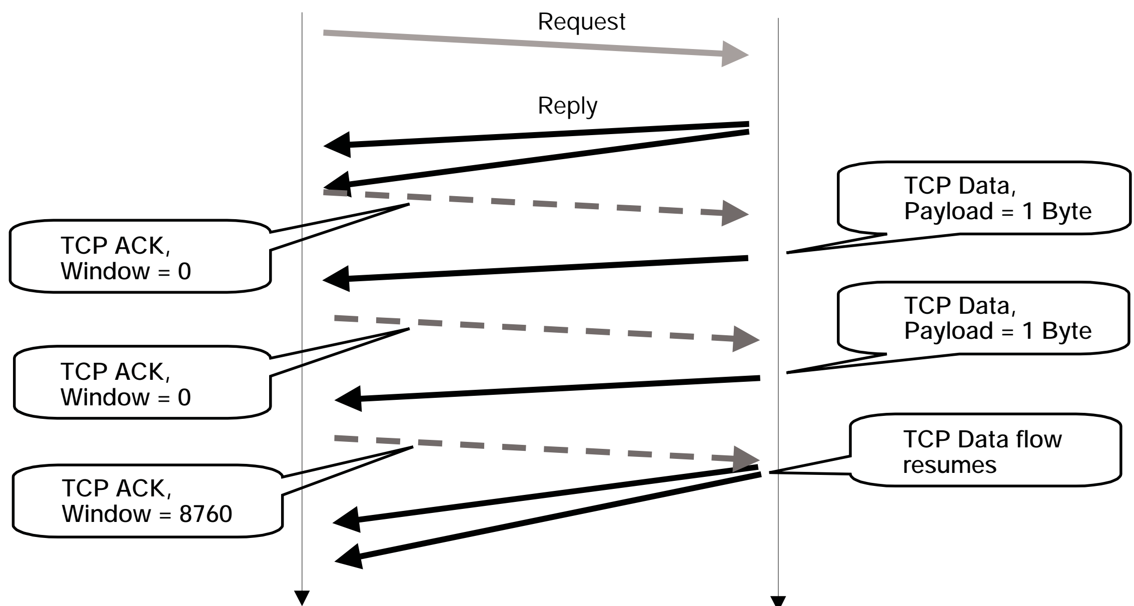


Figure 12: Client unable to receive data.

Receiving TCP stack buffer limit actions

The actions to take here relate to client machine performance. If the problem is isolated to a particular desktop, you will want to examine the machine configuration, not just in terms of CPU speed and memory but also in terms of network configuration, system tuning, background processes, etc. If the problem is pervasive, it may be appropriate to identify the thread or application section where the problem occurs (to see if the application might be modified), and to instrument the machine to determine internal resource constraints.

The technology implementation

While we may have successfully described the analysis process and some of the underlying techniques, the process is actually rather difficult to put in practice; the approach begs for an automated solution. Compuware's Application Vantage does this in a number of ways. The key enabling technology is the ability to merge two trace files of a transaction (typically from the client and server locations) into a single view. Performance analysis algorithms interpret the resulting data, allocating end-user experienced delays to network QoS metrics and processing nodes. Unique Thread Analysis capabilities provide the application-aware insight into transaction component performance, describing the relationship between logical program execution in the physical world of networks and processing nodes. Since one of the most significant hurdles is often capturing the transaction traffic itself, remotely managed software capture agents can be installed easily on clients and servers as required. Application Vantage presents the analysis in a series of transaction performance visualizations not only to provide the answer but also to support that answer with network-, node- and application-centric views.

The importance of thread analysis

One of the primary goals of application performance triage is to provide appropriate supporting evidence so that the assigned resource (a) accepts ownership of the problem, and (b) begins addressing the problem armed with relevant diagnostic information. This is a particularly significant challenge when communicating across network and application disciplines, since traditionally performance has different connotations within each of these "silos." Telling the web server or database administrator that "the server is slow" (even if you're right) is not the most effective way to solve the problem at hand, nor will it serve to build an open relationship for future communications. However, identifying the Active Server Page, CGI script or SELECT statement within the slow transaction will prove the value of the triage analysis and speed an appropriate resolution.

Thread Analysis capabilities are also important for measuring and analyzing multi-threaded application environments, those where multiple concurrent requests can be sent to a server on a single TCP connection. The calculation of the core performance metrics such as application turns and node sending durations depends on accurate correlation of application requests and responses; this is not possible without an understanding of the threading protocol used by the application. Application Vantage offers these unique Thread Analysis capabilities for many application protocols.

Baselines, multiple bottlenecks, intermittents and robots

We have illustrated how performance constraints affect application performance and how to quantify these constraints. Of course, there will be some constraints that are “normal” and some that will not be worth the effort to overcome. If your troubleshooting goals are to improve everyday performance, then you will analyze transaction performance and simply evaluate the bottlenecks, starting with the most significant. However, if you are troubleshooting a problem with a transaction in which performance has degraded, simple bottleneck analysis may not lead to a conclusive answer without a baseline for comparison. For example, lets say you measure a 60-second transaction that normally completes in 50 seconds, and your analysis identifies a server processing delay of 15 seconds and a network congestion delay of 11 seconds. Which would you decide is the cause of the problem? Without an understanding of what is normal (a baseline), you may make the “wrong” choice. This is not to say that you wouldn’t achieve the 50-second response time goal by removing either bottleneck, but you might miss identifying the change that caused the problem. (Of course, if you approach this as a bottleneck analysis and address both constraints, you might achieve a 40-second response time.) At a minimum, a baseline comparison provides a reference point that can quickly direct you toward a more efficient analysis based on what has changed.

Capturing a baseline can be as simple as tracing the transaction’s performance when everything is running smoothly. Of course, how many of us have the time to be this proactive? If the performance problem you are troubleshooting has become constant, you probably have no opportunity for creating a baseline; simply proceed with the methodology outlined above and you will succeed in identifying possible causes. An intermittent problem, on the other hand, may be a mixed blessing. On the plus side, it should be easy to capture a baseline for comparison. The downside is that it may be quite difficult to capture traces of the intermittent failure. It can be quite frustrating to attempt to capture sparse intermittent problems; they seem to occur only when you’re not watching. Using a “robot” or synthetic user to periodically run the transaction is one approach to reproducing the problem; the challenge then becomes capturing the associated trace files. Setting up continuous captures with appropriate address filters could allow you to capture multiple iterations of a transaction in a file of manageable size.

Complementary monitoring and analysis solutions

For some of the recommended actions, references to additional performance metrics require additional tools. Compuware's ServerVantage monitors the availability and performance of applications, databases and servers. When you decide to improve server performance based on either Thread Analysis or overall transaction performance, ServerVantage provides clear information on internal resource constraints. Compuware's NetworkVantage monitors distributed application traffic as it flows through the network, providing unique insight into the sources of traffic and network delays. If your performance analysis leads you to address the network's QoS, NetworkVantage supplies the information you need to make the correct decisions. Compuware's ClientVantage passive monitoring provides machine resource analysis and fault detection. If your analysis points to client problems, ClientVantage offers insight into machine and program performance. ClientVantage also includes active monitoring that lets you schedule the execution of scripted transactions, automatically providing performance measurement and trace file captures for service level threshold violations. If you are faced with intermittent problems, active monitoring can capture the data for you.

Compuware products and professional services—delivering quality applications

Compuware is a leading global provider of software products and professional services which IT organizations use to develop, integrate, test and manage the performance of the applications that drive their businesses. Our software products help optimize every step in the application life cycle—from defining requirements to supporting production service levels—for web, distributed and mainframe platforms. Our services professionals work at customer sites around the world, sharing their real-world perspective and experience to deliver an integrated, reliable solution.

Please contact us to learn more about how our comprehensive products and services can help your organization improve productivity, create higher quality applications and ensure performance in production.

All Compuware products and services listed within are trademarks or registered trademarks of Compuware Corporation. Java and all Java-based marks are the trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other company or product names are trademarks of their respective owners.
© 2002 Compuware Corporation

www.compuware.com

3/02