MICROSOFT: CAIRO ON HOLD? • IBM: NEW SUPPORT SERVICES

INFORMATIONWEEK

MARCH 27, 1995

FOR BUSINESS AND TECHNOLOGY MANAGERS

CAN $3.95 • US $2.95

Novell's Combination Shot

Building on NetWare, the company aims to expand into five new markets P. 34

13 >
0 71896 47283 7

IBM'S OS/2 STRATEGY P. 46

OPEN LABS

TIPS ON TWEAKING TUXEDO P. 62

A CMP Publication®

Technology Tutorial

How To Tweak Tuxedo

Tips to help you reap its client-server benefits

By Paul Cottey

Here are some techniques you won't find in the manual.

IF YOU'VE DECIDED to use Novell Inc.'s Tuxedo to boost the performance of your client-server online transaction processing (OLTP) application, you won't be disappointed. Like other transaction monitors, such as AT&T Global Information Solutions' Top End and Transarc Corp.'s Encina, Tuxedo can increase the number of transactions-per-second your application can process.

Middleware that sits between your client applications and database, Tuxedo accepts requests from clients and routes those to the appropriate server. Tuxedo's load-balancing facility optimizes performance as bottlenecks arise by dynamically routing client requests to available servers.

Tuxedo's forte, however, is managing transactions that are distributed across networks. By conforming to X/Open's Distributed Transaction Processing (DTP) model, Tuxedo can help ensure transaction integrity by performing two-phase commits across heterogeneous databases.

Some additional benefits of using Tuxedo are a decrease in the number of connections to your database server, which could reduce your memory requirements; and location transparency of application services, which

A Tuxedo Glossary

HERE IS AN EXPLANATION of some the terms used in this discussion of Novell Inc.'s Tuxedo:

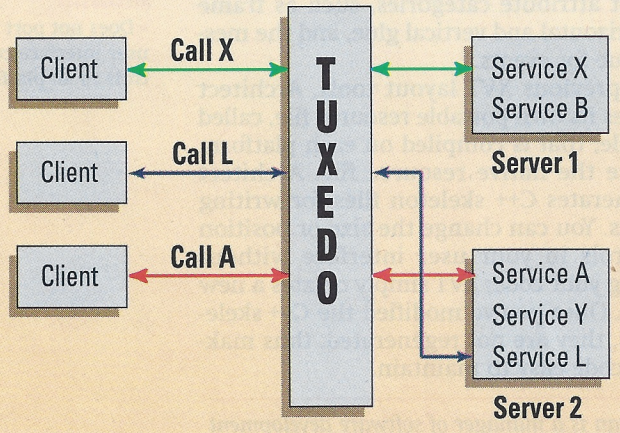
- **Service.** A business function, typically invoked by a call to Tuxedo—say, to update the database or look up a customer.
- **Server.** A collection of services. (When I'm referring instead to the physical machine, I'll explicitly say so.)
- **Client.** A program that initiates a request to a service. The client may be a workstation program or another service running on a server (in this case, I mean the actual machine).
- **Service Calls.** There are three types of service calls—synchronous (sync), asynchronous (async), and asynchronous without a reply (async no reply). In a sync call, the invoker sends a request and waits for a response. A sync call times out if a response is not received within a certain period of time. You might use a sync call to look up customer information—a credit limit, for instance—before processing an order.

In an async call, the invoker ultimately wants a response, but multiple requests may be sent at the same time, and the answers to those requests may come back in any order. This is sometimes referred to as "pipelining" the transactions. You might use an async call to send credit-card information for approval, then collect the shipping address for an order while you wait for the authorization to be returned.

In an async no reply call, the client program sends a request and continues processing, without checking for a reply. This type of call could be used to write performance information to a log file. —*Paul Cottey*

Tuxedo Service Processing

Novell's middleware passes client requests to services collected into server modules



gives you flexibility in distributing those services throughout a network.

But because Tuxedo inserts an extra layer (or tier) into your client-server application, you will need to carefully design your application's interaction with Tuxedo in order to reap those promised benefits.

Not In The Manual

Here are a few tips and techniques for using Tuxedo that you won't find in the manual. These tips can help you write more maintainable code, handle errors more gracefully, implement security, and, of course, optimize end-user response time—which is probably why you chose Tuxedo in the first place.

◆ Create service shells. It would be very easy to build applications that mix calls to Tuxedo services into the application code. The logic actually flows very nicely: Get whatever information you need from the user, format the buffer to send to a service, make the call, receive an answer, and then display the results to the user. The downside is that the user interface, transaction processing, and business logic become inseparable, making it difficult to maintain, or even understand, the code.

A better idea is to build a series of shells to handle standard types of Tuxedo services. For example, you might build one shell for synchronous calls and another for asynchronous calls. As shown at right, these shells should contain a call to the business function or service being invoked; they can then be modified independently of the business functionality.

The shells also should have hooks to error-handling and performance-monitoring routines, even if you don't have immediate plans to use them (you'll see why later on).

For code that is easy to understand and maintain, build a series of shells to handle standard types of Tuxedo services, complete with hooks for error handling.

By isolating the service shells from the application logic, updates to your application architecture and even to Tuxedo itself won't require modifying business services, only regenerating the new shells and recompiling.

To implement this, create a series of templates for these shells. Then create a simple program called a generator that replaces certain template data in the shells with the specifics of the business function. In addition to the name of the business function being invoked and the type of shell to use, there will be other inputs to your shell generator. If you start with a simple generator, you can enhance the generator to remove more complexities from the developer.

◆ Include performance hooks. Tuxedo has a built-in mechanism to capture performance information from your services, which is sufficient for the initial benchmarking of a service's responsiveness.

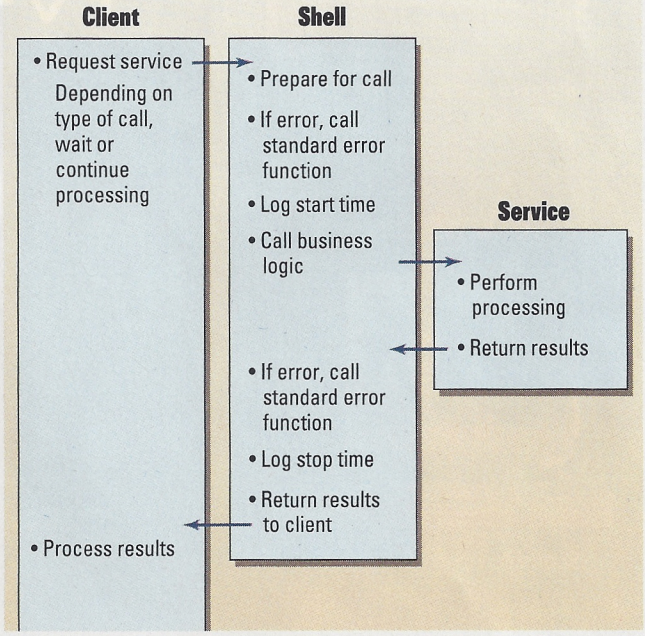
However, the servers must be rebooted to start or stop writing timing information; Tuxedo timings are intrusive because they run on the same physical machines as the services; and the timing information must be post processed before it can be used.

More important, since Tuxedo's timings begin and end with the service call itself, they do not capture what you probably want to measure: end-to-end user response time.

Since Tuxedo is typically used in a multitiered architecture, where delays can be introduced by network traffic or queuing of requests, capturing end-to-end time is especially critical.

To get a better measure of user response time, your custom performance-monitoring code should capture transaction start time from the user's perspective, and send results with an async-no-reply call to a log server (see chart, p. 66). Start your application's performance clock ticking when the user hits the Enter key and stop it when a response has been received. The shell code you wrote for your services also should capture start and stop times, and send those times to the same log server or to a different one. The log server itself should store the performance data in memory, periodically flushing the data to disk. This custom-monitoring code addresses all the

Calling A Tuxedo Business Function Through A Service Shell



limitations of the built-in Tuxedo timing and gives you true end-to-end response time.

To turn performance monitoring on or off, boot up or shut down the log server. When the log server is down, the async-no-reply calls take a minimum of overhead and effectively become no-ops. The log server can be run on a different machine from the application servers, so there is no contention for system resources. Finally, you can write a service for the log server that, when invoked, reports on up-to-the-second performance information. If you wish, you can post process the log server's output to create more traditional reports.

◆ Guarantee delivery. In a complex system, making a service call is like throwing a forward pass: 10 things can happen, and nine of them are bad.

The extreme ways to deal with the chance that a transaction may fail are either to have the user make only sync calls and wait until each transaction succeeds before continuing, or to just hope every service call works. Neither of these approaches is probably what you want.

Conveniently enough, Tuxedo provides a queuing mechanism called /Q (pronounced "slash queue") to let you write your transactions to stable storage so they will not be lost if a physical server goes down. You will still need to incorporate the use of /Q into your own Guaranteed shell type, but it can help eliminate the risk of a missed transaction.

Be forewarned, however: Writing a transaction to disk and subsequently reading it back introduces additional overhead. The Guaranteed shell should, therefore, be used only when no user is available to resolve an error immediately and the transaction really must be guaranteed to complete.

Making a service call is like throwing a forward pass: 10 things can happen, and nine of them are bad.

◆ Build in error handling. The more physical servers your requests must pass through, the greater the chance that something will go wrong. Sometimes all it takes to fix a failed transaction is to resubmit it. Perhaps the server was temporarily down, a Tuxedo queue filled up, or a database row was locked. Your architecture should provide a way to resubmit transactions without having to recreate the transaction.

Other transactions will never succeed, no matter how many times they are submitted—for example, a transaction containing incorrect data or one that tries to read a row that some other process has deleted. To resolve errors, these transactions will need to be examined. If the transaction was invoked as a sync call, it may be sufficient to report the error to the user. If the transaction was guaranteed to complete, it should end up on an error queue for manual resolution.

No matter what mechanisms you put in place for error handling, if they are part of the standard shell for all your client and service calls, you can update them as needed.

◆ Benchmark your services. The importance of benchmarking your services cannot be underestimated. You wouldn't buy a car without taking it for a test-drive, and you should not commit to an application architecture without taking it around the block a few times. You should build a prototype database with your major tables and test either early or representative versions of your key services against this database.

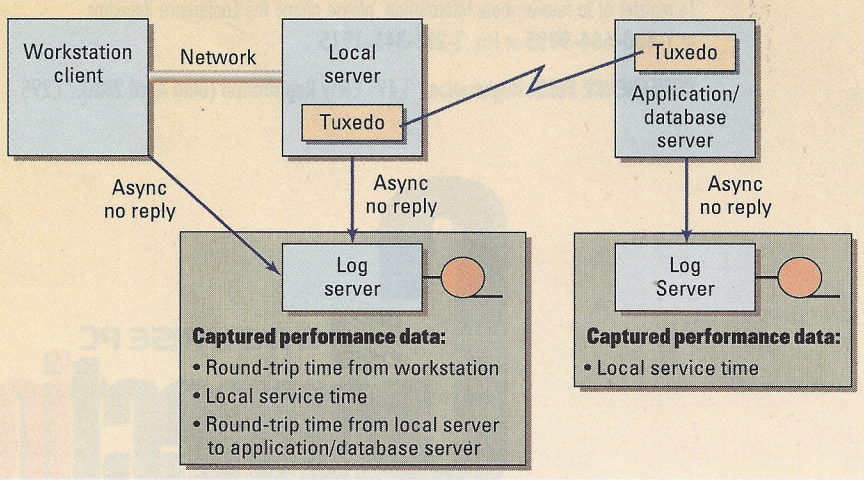
For synchronous calls, you should boot up one server for each simultaneous client request and then add one more to keep requests from queuing up. Then you should use the performance measurements you built into the service shells to give you an idea of user response time.

By benchmarking early, you can avoid preoptimizing the wrong portions of the architecture. You don't want to upgrade the network to asynchronous transfer mode when the performance problems really stem from an inefficient database layout. Similarly, buying more backend hardware is the wrong way to spend your money when you really should be optimizing network traffic.

Running a formalized benchmark with a prototype of your architecture also will give you an estimate of how many transactions will end up in your error queues. Feeding this information back to the designers of the application architecture may help change the way

Monitoring End-To-End Response Time

Through custom code, you can monitor performance of any element in your client-server architecture



you ultimately handle errors.

◆ Know your message sizes. Your client communicates to a service by passing a message. You should do a paper model of the messages when you are designing the service to help you find messages that might require multiple frames of data when they are transmitted over a wide area network (WAN). These service calls might be improved by sending only key data over the WAN and accessing the back-end database for detailed information.

The conventional wisdom for communicating over a slow WAN link dictates that smaller messages are better. That applies only if you are filling each message, however. If the last frame is not full, you may be able to save a back-end database access by sending more data in the message.

Because Tuxedo applies compression between machines in the same Tuxedo region, you will probably need a local area network (LAN) monitor to see the effects of compression on each of your messages. Actual compression ratios will be dictated both by how you store the data in the message and by the type of data being sent. Factor into your planning the knowledge that messages sent between regions will not be compressed.

As a rule of thumb, for shortest response time on synchronous calls, create one more server than there are simultaneous user requests.

◆ Group services. Each developer probably will code and test services separately. Although it is possible to create a server for every service, to realize the benefits of using Tuxedo, the services need to be grouped, or packaged, into servers prior to their being launched into a system-level test. When there are more requests than there are servers, requests queue up until a server is available. Thus, end-to-end user response time will be lengthened by the amount of time a request spends in a queue waiting to be processed.

It makes sense to group services by functionality and by the physical servers they access. Thus, you might group all sync services dealing with a specific application area and all the services that need to travel over the WAN.

As a rule of thumb, for shortest response time on sync calls, create one more server than there are simultaneous user requests. Then a request from one user will not need to wait until a request from another user finishes. If you are able to relax response-time requirements to allow a request to wait in a queue, you may boot fewer servers.

For shortest response time on asynchronous services, you will need enough servers to handle all simultaneous requests. This is more difficult to estimate, since you need to understand how your users will work in the new system. For this type of server, watch your performance logs and boot additional servers until end-to-end time minus service time is equal to your estimated network time.

◆ Remember security. Early in development, think security. Take advantage of the shells you built to help enforce security. Encryption of passwords is one of the first methods that spring to mind to secure an application. However, encryption can give a false sense of security over an unsecured network, since the potential security breaker could just resend the packet with the encrypted password to your application and never need to decrypt it.

Instead, associate a password with a Tuxedo region. This will bar accidental connections (such as a developer accidentally connecting to the production region) and will help prevent rogue applications introduced from an outside source from using Tuxedo resources. Remember, though, that this password is publicly known to your developers, so it is more a defense against accidental access than it is a bulletproof mechanism against intruders.

You could prompt the user for a password and send that prompt with each service call, but this is ineffective over an unsecured network. Plus, it adds a lookup for every request and gets users very annoyed.

Ultimately, you need to secure your application through a combination of procedures and programs. Tuxedo does provide hooks for user-supplied or third-party security mechanisms. Kerberos security, for example, could be implemented as an authentication service by including it in the Tuxedo server configuration file.

◆ Reconfirm benchmarks. Even though you built in performance monitoring and modeled your messages before you started coding, it is still important before you deploy the application to reconfirm that the back-end system can handle the actual load. A more formal benchmark that uses simplified versions of the services as they currently exist can help you spot network bottlenecks, Tuxedo queue-size deficiencies, and even database problems.

During this benchmarking, if your performance monitoring is not yet mature, you might turn on Tuxedo monitoring for one of each type of server. Although it will give you only an estimate of overall system response time, by limiting it to a small number of servers, you diminish the negatives of Tuxedo monitoring detailed above.

◆ Release your application. You've designed your application using standard shells and have built in performance monitoring and error handling. You've separated your Tuxedo code from your business logic and can regenerate a service if your shell code changes. You've benchmarked along the way, and everything seems fine. Once you have these pieces in place, you should be well prepared to handle difficulties as they arise—and, ultimately, to release your application into production.

Paul Cottey is a technical architect with Andersen Consulting in Chicago.