User Experience: CICS as Web Services Requester
SHARE
August 2011
Session 09611

Craig Schneiderwent
cschneidpublic@yahoo.com
#include <std/disclaimer.h>

Administrative: Opinions expressed are my own and are not representative of my employer. I do not speak for the WI Department of Transportation. Please do not interpret mention of a product to constitute an endorsement thereof.

The Agenda

- My world, and welcome to it
- The problem, how we solved it, and why
- Tools we built
- Some comments on development and testing
- Our Production experiences

For the most part, I'm going to try not to read slides to you.

In order to understand why we solved our problem the way we solved it, it is necessary for me to tell you a bit about how our organizations are structured and how we go about developing systems. I'll try to keep it brief.

Me, Us, and Them

I work for the Information Technology Strategy and Architecture Section of the Bureau of Information Technology Services within the Division of Business Management which is part of the Department of Transportation.

We're a Bureau of about 200 people. We do the workstation setup, database, network, and firewall administration, and application development.

The Data Center housing the mainframe is operated by a separate agency, which also employs all the system programmers for z/OS, CICS, DB2, IMS, etc.

I'm the author of files 498, 758, 775, and 788 (formerly known as the MA1K MQ SupportPac) on the CBTTAPE web site, and was one of the winners of the 2000 International Obfuscated C Code Contest. Over the last 25+ years, I've been paid to write code in the following languages: - Assembler (z/OS) - BASIC (DEC, GW, Quick, PDS) - C (z/OS, OS/2, Win32, HP-UX) - CList - COBOL (OS/VS, VS II, OS/390 and VM, Enterprise) - Java (1.2 on Win32, 1.3 on z/OS) - MARK IV (z/OS) - Perl (Win32, z/OS) - PL/I (z/OS) - PL/SQL - Rexx (OS/2, Win32, z/OS) - RPG II (Burroughs) - SAS (Win32, OS/2, z/OS) - Smalltalk (OS/2). I'm also a contributor to the CICS Wiki. I am *not* a CICS Systems Programmer.

That part about development almost sounds flip: "Oh, by the way, we do application development." It's actually a pretty big part of what we do, much of it implementing legislative mandates.

This last part makes life interesting. The data center does chargeback, which of course affects system design. Tom Peltin, the man who hired me to work at the WI DOT, once said "All systems are built around the current billing algorithm."

The Technology

- CA-Gen
- Enterprise COBOL 3.4.1
- Change Man 5.3
- CICS TS 3.1
- DB2 8
- z/OS 1.9

CA-Gen is an I-CASE (Integrated Computer Assisted Software Engineering) tool, providing full lifecycle software development support. WI DOT has been using this tool for over 20 years and much of the business logic for the Division of Motor Vehicles is written with it. We use CA-Gen to generate client-server applications, C on Windows clients, COBOL for CICS servers. We follow the "skinny" client model.

We wanted to be able to manage the web services artifacts with Change Man. Our SOP with Change Man is to compile once, then copy to each execution environment as we progress towards Production. More about those execution environments on the next slide.

We have now been upgraded to Change Man 7.1; upgrades to CICS TS 3.2 and z/OS 1.11 occurred during the project. The upgrade to DB2 9 should be complete this fall just before we begin the z/OS 1.12 upgrade.

There are 53 total CICS regions, 12 of which are Production, 5 of those are for the Department of Transportation exclusively. There is exactly 1 CICS Systems Programmer, she has a backup in the DB2 Systems Programmer.

The Progression

- Development: Developer's playground, we expect abends in this environment
- System Test: Does your application play well with others, integration testing
- Acceptance: End User playground, QA, we expect errors but not abends here
- Education/ProdFix: Training, emergency fixes, as close to Production as we can make it
- Production

This is a short version of something I wrote 12+ years ago when we went to this model.

These execution environments are intended to be a progression series, as one moves through them one can test implementation plans, etc. The goal is for Production implementations to be boring, free from the dreaded "oh, we forgot to bring that along" sort of errors.

Each of these execution environments has a DB2 subsystem and a set of CICS regions (among other things) associated with it.

Moving from one execution environment to the next does *not* involve a recompile, even for Production. This is an important foundation concept for our IT ecosystem.

The Business Problem

The Division of Motor Vehicles within the Department of Transportation was presented with a Legislative Mandate: Improve the security and integrity of the Driver's License Issuance process.

The process involves interacting with two vendor systems, one for queueing customers for service, the other for imaging. These systems each provide a web services interface.

Ultimately, we used CICS Web Services with CICS as Requester to solve the business problem, which is why we're all here.

A digital photo of the applicant must "follow" the applicant through the process, and scans of the applicant's proof of identity documents must be kept. In order to do this, our code must query the queueing system for who is next in line, then the imaging system for the applicant's photo.

The Debate

Client or Server, from which should our service requests initiate? A lively discussion ensued. There were advantages to initiating requests from the client, but technical difficulties eventually pushed us toward initiating requests from the server side.

The clients are C programs. The initial thought was to write the service requesters in Java and communicate between C and Java. This required skill sets that are rare on the ground in our shop. The next iteration of debate involved writing service requesters in C, which would require the same rare skill set.

As previously indicated, we use the “skinny client” model. Accessing service requesters from the client would have on the slippery slope away from skinny clients.

The Training

I attended a couple of SHARE presentations.

The Design

Business Logic is CA-Gen, just like our developers are used to writing. The Movers are COBOL programs that take the data passed in by the Business Logic which is intended for a service and move it into the language structures created by DFHWS2LS, and take the data provided by the service and move it into data areas provided by the Business Logic to receive it. The CWS (CICS Web Services) Interface encapsulates the INVOKE WEBSERVICE API call and its attendant error handling.

Parameters for the CWS Interface are channel, web service definition name, URIMAP name, and web service operation. The arrows you see here represent dynamic CALLs (COBOL CALL identifier).

Why Encapsulate INVOKE WEBSERVICE?

The INVOKE WEBSERVICE API is encapsulated into its own program primarily for error handling. As this is a critical application, providing as much information about a web service failure in order to speed up error resolution was important.

As it turned out, we were also glad we had done this when it came to capturing information about how long the web services were taking. This became an important metric when dealing with our vendors.

Additionally, this design gave us the ability to deal with SOAPFaults in our own way. More on that a bit later.

Just allowing the application to abend when a web service call failed wasn't an option. Digging EIBRESP and EIBRESP2 out of the dump, looking them up in the CICS documentation, then going back to the dump to dig out the rest of the relevant information (channel name, pipeline name, etc.) is simply too time consuming. Additionally, the way CA-Gen is implemented, if we want to display any sort of meaningful message on the client we mustn't abend in the server.

So we specify RESP() and RESP2() on the API and construct a WTO message for specific combinations, translating e.g. INVREQ with a RESP2 of 3 into its English equivalent "The Web service binding file associated with the WEBSERVICE is invalid."

In the event of a SOAPFault, we dump the contents of all containers on the channel using a program that does "dump-like" format of hex and character side-by-side so that it fits on a standard 80 column display. We also dump the contents of the original DFHREQUEST and DFHRESPONSE containers

Why Specify URI on INVOKE WEBSERVICE?

CICS Web Services will, by default, connect to the URI specified in the soap:address section of the WSDL.

We were making requests from multiple execution environments, and the vendors we were requesting services from also had multiple execution environments. We didn't want to alter the WSDL and rerun DFHWS2LS, we wanted to treat the WSDL, WSBIND, and pipeline configuration files as normal (to us) objects within our Change Man environment.

"Normal (to us)" means we "compile" (in this case with DFHWS2LS) and *copy* to each execution environment as we progress towards Production. Altering the WSDL and rerunning DFHWS2LS just to change the embedded URI was not something we were interested in doing.

Mind you, when I say the vendors had multiple execution environments, that doesn't mean they had *corresponding* execution environments. One vendor had 3 execution environments which our development staff had to decide how they wanted mapped to our 5.

The Tools We Built

- Transport Handler
- HTTP “ping”
- Debugging Framework
- Statistics Writer
- Pipeline “newcopy”

These are things we found we needed. Over the the next few slides I’ll be explaining what the problems were and why we solved them the way we did.

The Transport Handler

- Originally the program found in the SG24-7206-02 Redbook
- Now restructured
- “Aware” of debugging framework
- Makes a copy of outgoing and incoming SOAP in TS queues

We needed to see the SOAP message when a SOAPFault was generated.

The transport handler was originally the program from the SG24-7206-02 Redbook. I’ve since rewritten it, structuring the code, and adding a bit more in the way of error handling. But the big deal was making a copy of the outgoing and incoming SOAP messages (DFHREQUEST and DFHRESPONSE) in separate temporary storage queues. In the event of a SOAPFault, we still have the original SOAP message; the original DFHREQUEST is overwritten by CICS as indicated in the technote at http://www-01.ibm.com/support/docview.wss?uid=swg21264885. This may be less of a problem with CICS TS 4.1, but we’re not there yet, and less of a problem doesn’t equate to “isn’t a problem.”

We found that, in the event of a SOAPFault, it was invaluable to be able to hand the offending SOAP message to either the developer or to the vendor, and sometimes our firewall admins. Sometimes a firewall will send a (non-SOAP) “sorry, no, you can’t get there from where you are” message and smudge the sending location to make it appear as if it is coming from the location to which you are denied access.

The HTTP “ping”

- Simple 3270 application to do a WEB OPEN, WEB CONVERSE, and WEB CLOSE on an arbitrary URL or URIMAP
- Allows us to reach out and touch a web address without invoking any business logic
- Easily repeatable test, “can you hear me now?”

In the past I’ve worked in software support roles, and I’ve always found it invaluable to have a method to verify the software works that doesn’t involve executing a business application. In this case, being able to show the results of attempting to get from CICS to some web address to a firewall admin or web service provider goes a long way in problem resolution.

The Debugging Framework

- Relatively low overhead method of checking to see if debugging (or logging, if you prefer that term) is requested for the current transaction
- Request debugging/logging by any combination of userID, tranID, and web service
- Hooks in place in the transport handler and the CWS Interface

The debugging framework came from the realization that we were very likely to need to be able to "trace" a SOAP conversation even in Production. A CICS system-level trace simply wasn't granular enough for our purposes.

We did have to use this, in order to find the *one and only* bug in one of the "mover" programs that made it into Production.

The Statistics Writer

- Relatively low overhead method of capturing how long (clock time) a web service call takes
- Multi-row insert into a DB2 table
- Used to benchmark service providers to see if they could keep up with projected production volume

The statistics writer is a separate transaction STARTed from the CWS Interface. The start data includes tranID, userID, web service ID, URIMAP, start time, end time, and optional data. The program accumulates this data and does a multi-row insert into a DB2 table for each 100 rows. The accumulated data was very helpful in determining whether or not the service providers were going to be able to keep up with our projected production volume.

The Pipeline “newcopy”

- Makes it possible to use our normal promotion procedures with WSDL, WSBIND, and pipeline configuration files
- Promotion into an environment triggers a pipeline scan automatically

The Pipeline “newcopy” program is part of the Change Man promotion process. When we made changes to the pipeline configuration file, or got new WSDL and had to run DFHWS2LS, the promotion into a given environment would trigger a pipeline scan. Originally this did an INQUIRE PIPELINE, DISCARD PIPELINE, and CREATE PIPELINE. With the advent of CICS 3.2, not all pipeline attributes can be set via CREATE PIPELINE, so we do a PERFORM PIPELINE SCAN, which is probably preferable anyway.

The Development

- Staff were initially uncomfortable with the “black box” approach represented by the layers in the design
- The language structures generated by DFHWS2LS required multiple nested CICS containers, and containers were new to the staff
- Stress testing, monitoring, and threadsafe considerations took some explaining

Once the black box was opened up and the staff looked around inside at all the work being done in a single module rather than duplicated in each of their dozen or so modules, they were quite happy with how it worked.

After working with the DFHWS2LS-generated structures for a while, the primary developer of the “mover” modules approached me and said, “Wait a minute, it works like this, doesn’t it?” and proceeded to explain (correctly) how the containers interlocked.

The Testing

- Test harnesses were built for the CWS layer and the Mover layer
- The latter found a use in Production when it turned out one of the distributed systems required “waking up”
- Differing numbers of environments were a challenge
- It was while we were testing we decided we would probably need to turn on some sort of logging even in a Production environment

By “test harnesses” I mean an application with a relatively primitive user interface that’s intended to execute the layer to be tested with known input values. This worked well, and allowed the developers of those layers to test independent of the layer above (or, looking back at the design graphic, to the left of) them.

Production Experiences

Overall, things work pretty well. On a bad day we might see an error rate of 0.1% on our web service calls (timeouts + SOAPFaults). We're working with the vendors on improving that. We don't have many bad days.

- Timeouts
- SOAPFaults
- SSL

That 0.1% translates into dozens of errors. Most of the time we see < 10 errors in a workday, most of those are timeouts. Most of the web service calls have sub-second response times.

We've received few SOAPFaults that weren't the result of a server being rebooted to recover from some other error.

There was some concern over SSL overhead, this turned out to be a non-issue.

Questions