

Elvis and HostBridge: We Do Performance

A HostBridge® Technical White Paper

By Russ Teubner, Founder & CEO



Copyright Notice

Copyright © 2011 by HostBridge Technology. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any computer language, in any form or by any means, electronic, mechanical, optical, chemical, manual, or otherwise, without prior written permission. You have limited permission to make hardcopy or other reproductions of any machine-readable documentation for your own use, provided that each such reproduction shall carry this copyright notice. No other rights under copyright are granted without prior written permission. The document is not intended for production and is furnished “as is” without warranty of any kind. All warranties on this document are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

Revision date: 4/10/2011

Trademarks

HostBridge and the HostBridge logo are trademarked by HostBridge Technology. All other trademarks mentioned are property of their respective owners.

Elvis and HostBridge: We Do Performance

Tupelo, Mississippi, is the birthplace of Elvis, and residents are darned proud of it. Tupelo is also the home of a HostBridge customer, and we are darned proud of them. This customer, a financial services company, has embraced our latest and greatest (HostBridge v6) JavaScript-based Process Automation Engine as part of a CICS application integration project. Obviously, we love it when customers use our latest technology with their existing CICS apps to create new and innovative solutions for their customers.

Recently I traveled to Tupelo to review the performance of their latest HostBridge project. But it wasn't just a social call. The customer had been doing load testing and got word to us that there were some potential response-time issues. Knowing I love nothing better than to review CICS aux traces late into the night, the HostBridge guys asked if I would run point on the project.

This particular customer has a near-perfect setup for performance testing:

- They were using the latest version of our software (exploiting every feature)
- Their systems programmers and software developers are very competent
- They can control their own system

Perhaps this last bullet sounds odd, but you'd be surprised at the number of customers with competent systems professionals who can't control their own system (due to outsourcing arrangements). I also jumped on this chance because this customer has a smaller, i.e., slower, system than we use for our development (we use the big, bad z10 at the IBM Dallas Data Center). It is always interesting to compare performance on smaller systems (this system's speed was about half that of ours).

Load Testing Methodology

It turned out that the load testing was being performed by an outside service provider. Thus, the first step was to figure out exactly what they were doing. This is not always as easy as it sounds. Everybody has their own perspective about load testing, and the tooling is widely divergent – from homegrown to over-the-top sophisticated.

In this situation, the objective was to determine if CICS/HostBridge could process a given number of web services requests per second. However, it seemed that the presentation rate was virtually unconstrained. When a load test has the basic characteristics of a denial-of-service attack (unconstrained presentation rate), there's only so much information you can glean from it. It will definitely test/assess something, but it may not be what you intended. And it probably won't help you find the "knees" in the performance curve.

Let's create a hypothetical scenario to illustrate the point. Let's assume that the objective is to process 10 web service requests per second (RPS). Further, let's assume that we construct a load test that presents (sends) 10 RPS to CICS for processing. Does this make sense? Yes – the presentation rate equals the target processing rate. So we will be able to assess if we can handle that sustained RPS load. But what if we double the presentation rate to 20 RPS? That will also be instructive, but the test will implicitly test other things as well – such as CICS's ability to manage concurrent workload. Or what about 100 RPS? Will that be instructive? Perhaps, but whatever we learn may have little to do with the original objective (can we process 10 RPS?). The bottom line is this: *the presentation rate matters*.

In this situation, the load testing methodology involved an unconstrained presentation rate. And since the load testing service was remote to the customer's network, a fair bit of (variable) latency was creeping into the observations. Thus, while it was clear that bad things were happening, it was difficult to separate out the constituent issues. What we needed was a more controlled load testing approach – one that we could repeat at will on a local network.

Taking Matters into Our Own Hands

There are a couple of free yet robust HTTP/web load testing tools available. Microsoft has one. Another we frequently use is Apache JMeter. Per the Apache web site:

Apache JMeter is open source software, a 100% pure Java desktop application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions. Apache JMeter may be used to test performance both on static and dynamic resources. It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types. You can use it to make a graphical analysis of performance or to test your server/script/object behavior under heavy concurrent load.

The customer downloaded JMeter onto one of their desktops. Installation is a snap. Configuration, however, takes a bit of time because you have to get your brain cells twisted into JMeter's testing metaphor and vocabulary.

Once set up, it is flawless. Using JMeter, we prepared a battery of load tests that sent the same HTTP web service request to CICS/HostBridge. We varied three things:

1. The number of simultaneous threads (concurrency)
2. Wait time between each request sent on a thread (think time)
3. The total number of requests sent by each thread (iterations)

This allowed us to differentiate between and assess the primary and secondary performance issues. For our purposes, the primary issue was response time when the presentation rate was equal to or slightly higher than the target processing rate.

Secondary issues related to how everything behaved when we drove up the presentation rate.

Seeing Is Believing

Once we started running controlled load tests we were able to gain insight into the response time patterns. Sure enough, they were not to our liking. But now we began to see why. As we gained general insight from the load tests, we then drilled down on the various hot spots (mostly using CICS aux traces). When all the dust settled, the optimization issues fell into three buckets: HostBridge, CICS, and TCP/IP.

The HostBridge bucket had only one item, but it was important. The copybook object used by an HB v6 script to manipulate data described by a COBOL copybook was not performing to our expectations on the customer's machine. I'm going to gloss over this pretty quickly for two reasons. First, no software vendor, for obvious reasons, likes to dwell on the fact that the first version of a subroutine may need some polishing every now and then. Second, it's no longer an issue. While I was in Tupelo, and continuing over the weekend, Scott Glenn and James Alexander (HostBridge developers extraordinaire) completely reworked the copybook object to yield a huge performance improvement with a corresponding latency reduction.

So let's move on to the more instructive CICS and TCP/IP buckets.

Basic CICS Tuning

Our Tupelo customer has a System z9 BC with two General Purpose processors (GPs). Question: what happens if you spray HTTP requests at a CICS test region according to the following parameters:

- 32 concurrent threads/sessions
- 0 think time between each request
- All work runs on the QR TCB
- No transaction classes limiting concurrency
- The system is already devoting 50% to production work?

Answer: something akin to digital constipation.

For each concurrent request, CICS created one CWXN transaction and one HBJs transaction. CICS was so busy managing work in progress that the time it devoted to actually getting stuff done suffered. Thus, we made a few adjustments to the customer's CICS region configuration to bring a bit of discipline to the environment:

- We defined the HBJs transaction to be in a class that allowed 2 concurrent transactions to be executed (we later increased to 3 or 4).

- We defined HBR\$JSE program (and a handful of referenced programs) as THREADSAFE. This allows HostBridge to run on CICS Open TCBs. In turn, CICS/HostBridge can actually absorb both GPs' processing power.

CICS File Control Threadsafety

Once we had HostBridge set up to run on the Open TCBs, we observed about 12 context switches between the L8 and QR TCB's. Context switching should always be monitored and avoided if possible, especially on smaller systems.

The context switches in question stemmed from VSAM file I/O. Hmmm.... I thought VSMA file I/O was supposed to be threadsafe. One side effect of dealing with customers running different versions of CICS TS, especially while working on HostBridge product design/development work for the next release of CICS TS, is that you tend to forget which versions introduced which features (at least I do). Since I was under the gun, I asked my friend Russ Evans, of Evans Consulting Group, to circumvent my learning curve and shed some light on the situation. He informed me that APAR PK45354, a File Control Threadsafety Enabler for Local VSAM LSR Files, is applicable in CICS TS 3.2 (quite the relevant bit of knowledge).

What happened next was almost unbelievable. The CICS system programmer supporting our efforts was able to download the relevant PTFs, research them, and apply them in such a way that we could isolate and accurately test their effect – all in a mere *3 hours!* It was truly one of the more impressive displays of systems programming prowess I have seen.

When he had the test region ready we changed the SIT parameter FCQRONLY to specify NO and resumed testing. Sure enough, all the context switches disappeared and the CICS aux trace cleaned up very nicely.

TCP/IP Tuning

HostBridge customers tend to implement web services using informal/lightweight protocols (e.g., REST as opposed to SOAP). The most frequent choice we see is the use of HTTP GET/POST requests with XML payloads. Customers do this because their volumes are high and they usually control both ends of the connection. They thus avoid unjustified SOAP overhead. Since their volumes are high and response time must be low, we inevitably end up examining CICS aux traces (and TCP/IP packet traces) to insure that things are running optimally.

When we examined the customer's traces, we saw some problems. This type of analysis and tuning on a customer's system can be a pain, so we modeled their environment on our system and began performing WireShark traces from there. Here is a synopsis of the interactions between JMeter (as the distributed app test driver) and CICS/HostBridge:

Pkt No.	Time (Relative)			Function	Info
1	0	Web App	→	CICS	Establish Connection [SYN] Len=0 MSS=1460
2	0.011466	Web App	←	CICS	" [SYN, ACK] Len=0 MSS=536
3	0.011492	Web App	→	CICS	" [ACK] Len=0
4	0.012611	Web App	→	CICS/HB	Rqst Pkt 1/3 (headers) [PSH, ACK] Len=316
5	0.012648	Web App	→	CICS/HB	Rqst Pkt 2/3 (data) [ACK] Len=536
6	0.024611	Web App	←	CICS/HB	Ack [ACK] Len=0
7	0.024632	Web App	→	CICS/HB	Rqst Pkt 3/3 (data) [PSH, ACK] Len=408
8	0.024894	Web App	←	CICS/HB	Ack [ACK] Len=0
9	0.050962	Web App	←	CICS/HB	Resp Pkt 1/6 [ACK] Len=536
10	0.051909	Web App	←	CICS/HB	Resp Pkt 2/6 [ACK] Len=536
11	0.051927	Web App	→	CICS/HB	Ack [ACK] Len=0
12	0.052359	Web App	←	CICS/HB	Resp Pkt 3/6 [ACK] Len=536
13	0.053019	Web App	←	CICS/HB	Resp Pkt 4/6 [ACK] Len=536
14	0.053030	Web App	→	CICS/HB	Ack [ACK] Len=0
15	0.053585	Web App	←	CICS/HB	Resp Pkt 5/6 [PSH, ACK] Len=536
16	0.067779	Web App	←	CICS/HB	Resp Pkt 6/6 [PSH, ACK] Len=45
17	0.067862	Web App	→	CICS/HB	Ack [ACK] Len=0

What do you see in this trace? Here's what catches my eye:

- The request was sent as 3 packets
- There was an ACK between request packet 2 and 3
- The response was delivered as 6 packets, which necessitated 3 ACKs
- Waiting for ACKs from CICS seemed to interject considerable delay

Needless to say, some TCP/IP tuning was in order. It's hard to create a narrative out of what ensued. Let's just say that there are a multitude of factors that shape TCP performance between CICS and a distributed app. I'll hit the highlights.

MSS and MTU Sizes

Per Wikipedia:

The Maximum Segment Size (MSS) is the largest amount of data, specified in bytes, that TCP is willing to send in a single segment. For best performance, the MSS should be set small enough to avoid IP fragmentation, which can lead to excessive retransmissions if there is packet loss. To try to accomplish this, typically the MSS is negotiated using the MSS option when the TCP connection is established. In this case it is determined by the Maximum Transmission Unit

(MTU) size of the data link layer of the networks to which the sender and receiver are directly attached.

This description certainly focuses on the potential evils associated with an MSS that is too large. However, the description lacks an important counterbalancing argument. It's also important to understand that too small an MSS can negatively impact end-to-end application performance.

In the trace above, notice that the MSS indicated by the distributed app was 1460, but the MSS indicated by zOS was 536. Oh, my. For TCP/IP routes under System z, 536 is the default. And it should NEVER be used! 1460 is probably the minimum MSS you would want in effect for routes that ultimately boil down to an Ethernet connection. This small MSS size is why the inbound request was chopped into three packets and the outbound request was chopped into six packets. Thus, the first change was made to the applicable ROUTE statement in the TCP/IP configuration file.

Just to clarify, if you have a ROUTE statement that looks something like this (which doesn't specify MTU):

```
ROUTE 172.29.127.0 255.255.255.0 = OSDL
```

Or like this (which specifies it as DEFAULTSIZE):

```
ROUTE 172.29.127.0 255.255.255.0 = OSDL MTU DEFAULTSIZE
```

Then change it to specify *at least* an MTU of 1500! For example:

```
ROUTE 172.29.127.0 255.255.255.0 = OSDL MTU 1500
```

Bottom line: 576 is the default MTU size for IPv4 routes and should be avoided because it results in an MSS of 536 (576 less 40 bytes for the TCP header).

Delayed Acknowledgements (ACKs)

TCP/IP has all sorts of "knobs" that seek to optimize bandwidth. Many of them date back to days when bandwidth was slow and expensive while many others are very helpful. However, some can really cause problems for short lived TCP/IP sessions in which response time is critical. Take, for example, the DELAYACK parameter. Per the IBM doc, the DELAYACKS or NODELAYACKS parameter:

Controls transmission of acknowledgments when a packet is received with the PUSH bit on in the TCP header. NODELAYACKS specifies that an acknowledgment is returned immediately when a packet is received with the PUSH bit on in the TCP header. DELAYACKS delays transmission of acknowledgments when a packet is received with the PUSH bit on in the TCP header.

And guess what? DELAYACKS is the default!

We advise all HostBridge customers to use NODELAYACKS on HostBridge ports. This parameter can be specified in a number of different ways, but is often specified on the PORT reservation statement. For example:

```
PORT
23028 TCP CICSA NODELAYACKS ; HBJNSSL
23029 TCP CICSA NODELAYACKS ; HBRHTTP
23030 TCP CICSA NODELAYACKS ; HBXNSSL
```

Slow Starters Are Not So Swift

After analyzing the trace, it appeared that another MSS-related TCP congestion control algorithm might be affecting performance. Though results were inconclusive, I'll describe it here, for the sake of completeness. The algorithm is called Slow-Start and is described by Wikipedia as follows:

Slow-start is part of the congestion control strategy used by TCP, the data transmission protocol used by many Internet applications. Slow-start is used in conjunction with other algorithms to avoid sending more data than the network is capable of transmitting, that is, to avoid causing network congestion.

So we know its intentions are honorable. But let's keep reading to see how it does this:

... slow-start works by increasing the TCP congestion window each time the acknowledgment is received. It increases the window size by the number of segments acknowledged. This happens until either an acknowledgment is not received for some segment or a predetermined threshold value is reached. The algorithm begins...initially with a congestion window size (cwnd) of 1 or 2 segments and increases it by 1 Segment Size (SS) for each ACK received. This behavior effectively doubles the window size each round trip of the network. This behavior continues until the congestion window size (cwnd) reaches the size of the receiver's advertised window or until a loss occurs.

This explanation from ssfnet.org seems a bit more concrete:

[Slow Start] operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end.

Slow start adds another window to the sender's TCP: the congestion window, called "cwnd". When a new connection is established with a host on another network, the congestion window is initialized to one segment (i.e., the segment size announced by the other end, or the default, typically 536 or 512). Each time an ACK is received, the congestion window is increased by one segment. The sender can transmit up to the minimum of the congestion window and the advertised window.

For clarification, they include the following comment:

The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion; the latter is

related to the amount of available buffer space at the receiver for this connection.

This all makes sense. But could all this goodness have a downside? Apparently so! Among a few items mentioned in the Wikipedia article is:

The slow-start protocol performs badly for short-lived connections.

Yikes! Short-lived connections are exactly what a typical web service request directed to CICS/HostBridge should be. This may, after all, have something to do with us. But how? Let's dig a bit deeper into the concept of a congestion window. Wikipedia to the rescue again:

... the congestion window is one of the factors that determines the number of bytes that can be outstanding at any time. Maintained on the sender, this is a means of stopping the link between two places from getting overloaded with too much traffic. The size of this window is calculated by estimating how much congestion there is between the two places. The sender maintains the congestion window. When a connection is set up, the congestion window is set to the maximum segment size (MSS) allowed on that connection. Further variance in the [congestion]¹ window is dictated by an Additive Increase/Multiplicative Decrease approach.

Okay. The picture is clear. The MSS and the rate of acknowledgements are pretty important to the performance of these so-called short lived connections.

TCP/IP Tuning Results

After digging through all the minutia related to TCP/IP tuning and making the above changes, the big question is: *did it make a difference?* The answer is a resounding YES! After changing the MTU size on the ROUTE statement and specifying NODELAYACKS on the PORT statement, the request/response exchange looked like this:

Pkt. No.	Time (Relative)			Function	Info
1	0	Web App	→	CICS	Establish Connection [SYN] Len=0 MSS=1460
2	0.011989	Web App	←	CICS/	" [SYN, ACK] Len=0 MSS=1400
3	0.012026	Web App	→	CICS	" [ACK] Len=0
4	0.013035	Web App	→	CICS/HB	Rqst Pkt 1/2 (headers) [PSH, ACK] Len=316
5	0.013078	Web App	→	CICS/HB	Rqst Pkt 2/2 (data) [PSH, ACK] Len=944
6	0.026903	Web App	←	CICS/HB	Resp Pkt 1/3 [ACK] Len=0
7	0.040810	Web App	←	CICS/HB	Resp Pkt 2/3 [ACK] Len=1400
8	0.041964	Web App	←	CICS/HB	Resp Pkt 3/3 [PSH, ACK] Len=1325
9	0.042000	Web App	→	CICS/HB	Ack [ACK] Len=0

¹ Wikipedia has "collision window" here; we're confident "congestion window" was intended.

Clearly, this flow is more efficient.

- Only 9 packets were exchanged, instead of 17
- Only 4 changes of direction were involved, instead of 10
- Only 1 “solo” ACK packet, instead of 5
- No ACK delays

Even though we were not trying to formally benchmark response time by holding all other system or network activity constant, you can see that response time decreased by 38% – entirely attributable to TCP/IP tuning.

Nagle’s Algorithm

Since this is turning into my *opus magnum* about TCP/IP tuning as it relates to CICS and HostBridge web services, I might as well cover Nagle’s algorithm. While we saw no evidence of its impact on this customer’s performance level, it has certainly been noticed by others.

Per Wikipedia: “Nagle’s algorithm, named after John Nagle, is a means of improving the efficiency of TCP/IP networks by reducing the number of packets that need to be sent over the network.”

Come on now. Who can argue with that? Sounds like motherhood and apple pie (a truly American idiom). The excellent Wikipedia article continues:

Nagle’s document, *Congestion Control in IP/TCP Internetworks* (RFC 896) describes what can be called the ‘small packet problem’, where an application repeatedly emits data in small chunks, frequently only 1 byte in size. Since TCP packets have a 40 byte header (20 bytes for TCP, 20 bytes for IPv4), this results in a 41 byte packet for 1 byte of useful information, a huge overhead. This situation often occurs in Telnet sessions, where most keypresses generate a single byte of data that is transmitted immediately. Worse, over slow links, many such packets can be in transit at the same time, potentially leading to congestion collapse.

This makes a lot of sense, but its rationale harkens back to the grand and glorious days of asynchronous ASCII terminals interacting with applications on a keystroke-by-keystroke basis (imagine line-mode Unix or DEC VAX VMS applications). Continuing:

Nagle’s algorithm works by combining a number of small outgoing messages and sending them all at once. Specifically, as long as there is a sent packet for which the sender has received no acknowledgement, the sender should keep buffering its output until it has a full packet’s worth of output, so that output can be sent at once.

The logic seems solid, but what happens if you combine the use of the algorithm with other TCP transmission optimizations? Nothing good. The Wikipedia article touches on this too:

This algorithm interacts badly with TCP delayed acknowledgements, a feature introduced into TCP at roughly the same time in the early 1980s, but by a different group. With both algorithms enabled, applications that do two successive writes to a TCP connection, followed by a read that will not be fulfilled until after the data from the second write has reached the destination, experience a constant delay of up to 500 milliseconds, the “ACK delay”. For this reason, TCP implementations usually provide applications with an interface to disable the Nagle algorithm. This is typically called the TCP_NODELAY option.

The Wikipedia article continues with advice that is technically correct, but often irrelevant if you are a CICS developer living off of the EXEC CICS WEB interface:

It is, however, not recommended to disable this [the TCP_NODELAY option]. A solution, recommended by Nagle himself, is to keep from sending small single writes and buffer up application writes then send...: *“The user-level solution is to avoid write-write-read sequences on sockets. Write-read-write-read is fine. Write-write-write is fine. But write-write-read is a killer. So, if you can, buffer up your little writes to TCP and send them all at once.”*

Why is this irrelevant? Two reasons. First, you can’t always control how CICS chooses to write buffers on the TCP socket. Second, in most cases, CICS/HostBridge is the *receiver* of a request (the server) not the *originator* of a request (the client). Thus, Nagle’s algorithm usually rears its ugly head when the request is sent from the distributed application to CICS/HostBridge.

Summary

It seems that a visit to Tupelo, Mississippi, became a well-spring of observations and insights that resulted in significant performance improvements for a HostBridge customer. If I back off from the details of this particular situation, my high-level conclusions would be:

- Whether it’s HostBridge, CICS, or TCP/IP – *tuning matters*
- It’s terribly difficult to tune what you cannot see and control
- Don’t assume everything is fine under the covers

If you would like us to peer under the covers of your CICS-based web services, please let us know (even if you don’t use HostBridge). Who knows what sort of TCP/IP gremlins we might find?