June 2009

Building Low Latency Applications for Financial Markets

Rapid Addition leverages Microsoft .NET 3.5 Framework™ to build ultra-low latency FIX and FAST processing

Applies to: FIX and FAST message processing Low Latency Financial Application Architecture, Microsoft .NET 3.5 Framework™

Kevin Houstoun
Rapid Addition Limited

Ed Briggs
Microsoft Corporation

Summary

This whitepaper describes how Rapid Addition built their ultra low latency FIX and FAST message processing software using the Microsoft .NET 3.5 Framework. By following a disciplined design and development, Rapid Addition was able to meet stringent latency requirements while retaining the advantages that managed code brings.

Contents

Introduction

Latency matters. And Rapid Addition, the leading supplier of front office messaging components to the global financial services industry knows this better than most. Their founder and chairman, Kevin Houstoun, has been one of the leading innovators in this space since helping bring the FIX Protocol to Europe in the 1990's. "Being a few microseconds slower than their competitors can literally cost our clients millions of pounds, dollars or yen. It means the difference between a high speed arbitrage trade being profitable or a waste of time and money; for a hedge fund, it means the difference between posting an updated price and being hit on a stale price for a market maker; and for an exchange it means being the venue of choice for many legs of various trading strategies. For many of our clients low latency is not an option it is a necessity."

Houstoun has lead the FIX Protocol Limited's (FPL) Global Technical Committee (GTC) through the introduction of a data model behind the collection of protocols the group supports, the introduction of the FAST messaging compression standard, FAST, and the release of 3 versions of FIX targeting the exchange to sell side communication. FAST stands for FIX Adapted for Streaming Transport; it was a response to growing market data volumes and the exchange communities desire to avoid inventing further costly proprietary protocols. FAST is an open specification that was developed by FPL with financial support from Archipelago Exchange, the Chicago Mercantile Exchange, the International Securities Exchange, London Stock Exchange, Microsoft and the Singapore Stock Exchange.

In 2003 Houstoun left Salomon Brothers, then part of Citigroup, and teamed up with Clive Browning, to write components based on FPL standards utilizing the recently introduced repository. They had a vision of using the FPL data model to write better performing and easier to use components.

Motivation for Using .NET

Conventional wisdom has been developing low latency messaging technology required the use of unmanaged C++ or assembly language. But RA saw advantages to building this sort of technology in managed code. Asked about this choice Clive Browning, Rapid Addition's Chief Technology Officer said, “When you look at the unmanaged C++ solutions, you see that the approach the best of breed solutions use is to develop their own specialized engine and then generate dedicated handlers for each pattern that uses its specialized engine. Our approach is actually very similar to this except that our specialist engine is a sub set of Microsoft’s .Net CLR.” Browning continues, “This gives our clients certain advantages, we don’t have to update our engine for every hardware change, Microsoft does that for us; we have the full set of .NET features available for other modes of operation such as start up, and of course there is no overhead for communicating between the managed and unmanaged code if our end customers are using .NET for other parts of their project.”

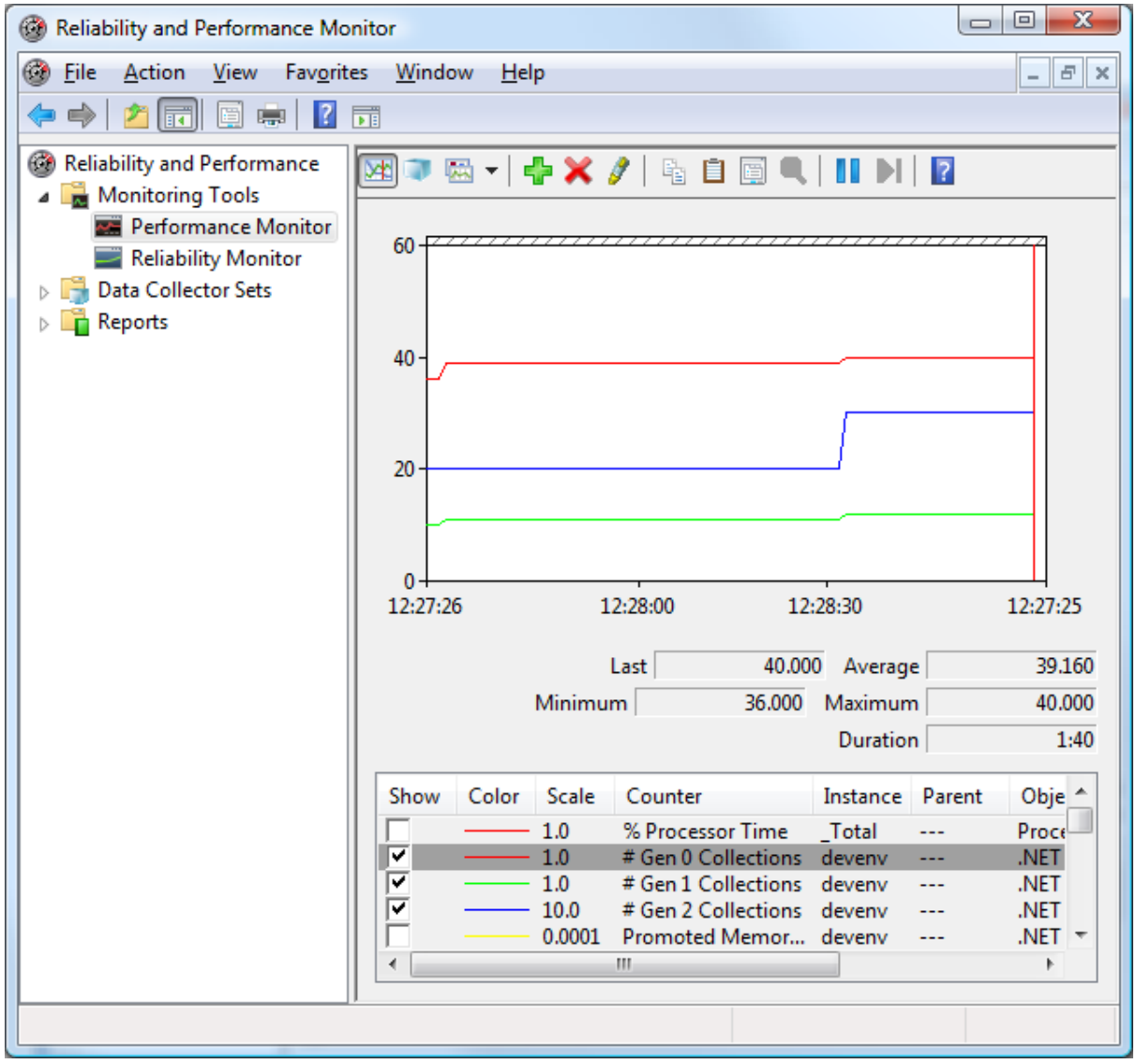
To meet the demanding latency requirements inherent in FIX and FAST message processing, there are two overriding rules Rapid Addition employs in their designs:

1) Actively manage any resources that are used in the steady state operation of the program. This is achieved through the use of resource pools. A number of resources are assigned in the startup phase and then these are recycled throughout the continuous operations phase.
2) Do not cause garbage collection in the continuous operation phase of the system.

From an overall design perspective when designing low latency systems in .NET, Rapid Addition adopts a number of disciplines:

1) Structure code into three distinct phases; startup, continuous operations and shut down sections. Code in the start-up is allow to make memory allocation both for use in resource pools and temporarily to initiate the process but at the end of the start-up mode the garbage collector is invoked to ensure that any unreferenced memory is released at this point. In the continuous operations phase no memory allocation is tolerated and in the shutdown phase the garbage collector is allowed to become active again.

Figure 1 below illustrates this, the program is started around 12:27:26 and we see some GC activity related to start-up, it runs in continuous mode sending 40,000 messages per second until 12:28:30 when it is shut down, as part of the shut down the resource pools are released and we see the Garbage Collector activity associated with cleaning up these resources. The important thing to note is that *there is no Garbage Collector activity* in the continuous operation phase. In live client systems this Garbage Collector free phase can be 10's or 100's of hours.



2) Tight coding standards and guidelines ensure that once in the continuous operating mode garbage is not created. Apart from the obvious approaches such as avoiding manipulating immutable objects, principle the .Net string data type, it also involves avoiding parts of the .Net runtime that Rapid Addition know create garbage.

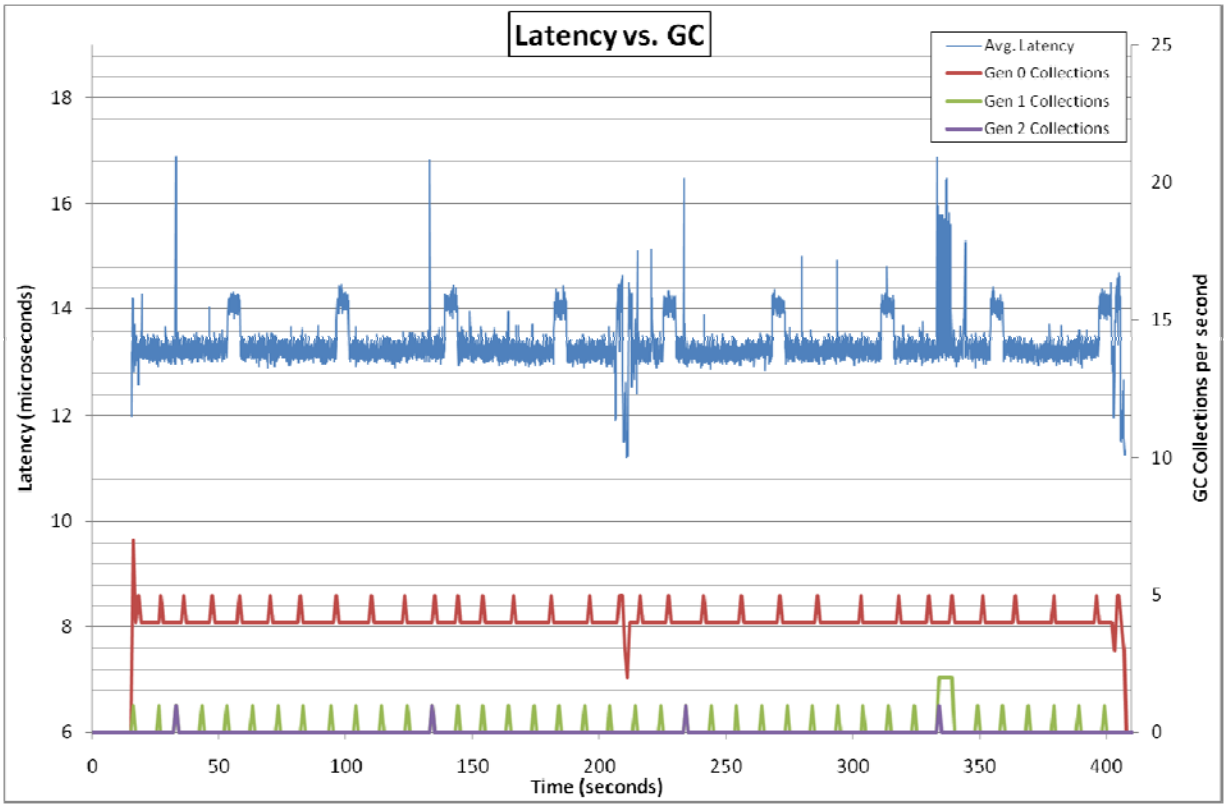
Garbage Collection

Garbage Collection is a charming term referring to the mechanism providing automatic memory reclamation employed by the Microsoft .NET Common Language Runtime™ (CLR). The CLR uses a generational garbage collector which divides the memory heap into partitions and segregates memory objects by age, or generation. This optimization is based on the following observations:

- Most memory allocations are short lived, and can be reclaimed quickly.
- Some allocations will last a long time. An heuristic to determine if an allocation will be long lived is the allocation's current age - if it is old, it will probably become older, and if necessary can be 'promoted' to a different generation.
- Reclaiming part of the heap is faster than collection the entire heap.

By partitioning the heap into generations, the time and computational effort required to perform garbage collection can be substantially reduced. The CLR also provides several different GC algorithms referred to as workstation and server which can be selected to best suit performance requirements. In the following sections we discuss GC overhead using workstations GC. Because a full discussion of GC is beyond the scope of this document, interested readers may consult Microsoft URL below for further information.

Although GC is performed quite rapidly, it does take time to perform, and thus garbage collection in your continuous operating mode can introduce both undesirable latency and variation in latency in those applications which are highly sensitive to delay. As an illustration, if you are processing 100,000 messages per second and each message uses a small temporary 2 character string, around 8 bytes (this a function of string encoding and the implementation of the string object) is allocated for each message. Thus you are creating almost 1MB of garbage per second. For a system which may need to deliver constant performance over a 16 hour period this means that you will have to clean up 16 hours x 60 minutes x 60 seconds x 1MB of memory approximately 56 GB of memory. The best you can expect from the garbage collector is that it will clean this up entirely in either Generation 0 or 1 collections and cause jitter, the worst is that it will cause a Generation 2 garbage collection with the associated larger latency spike.



The above graph illustrates the effect of Garbage collection on messaging latency. You can see that the GC2 collections (the 4 blue peeks at the bottom) correlate with the 4 highest peaks. We can also see a fair amount of jitter (probably due to lots of GC0's). By performing our own storage management using resource pools, RA avoid the latency arising from these GCs.

Some additional techniques RA use to reduce the quantity of garbage that we collect include:

1) Avoiding immutable reference data types, these effectively are read only within .NET and once created copies are produced for each modification and the original is discarded.
2) Avoid .NET operations which create temporary objects, for example boxing objects; this creates a copy of the value type as an object on the heap; this then needs to be garbage collected once it is de-referenced.
3) Some .NET' Framework functions create garbage, and these function calls need to be avoided during steady-state operation. For instance when you call

```
System.Globalization.DaylightTime DayLightTime =
System.TimeZone.CurrentTimeZone.GetDaylightChanges(2010);
```

the function creates more garbage than you would expect. The CIL for this operation shows why - the first few lines contain the following

```
IL_004d:  ldarg.1
IL_004e:  box       System.Int32
IL_0053:  stloc.0
```

It is the boxing operation at IL_004e that creates the unexpected garbage. Consequently, we do not use this call. RA has a list of similar calls that form a part of their coding standards.

Product Development and Test Methodology

RA has a slightly unusual approach to developing software, the development time for a component has two phases, typically equal in duration, first functional completeness second performance tuning, unless this is planned into the life cycle it does not happen effectively.

A lot of this design expertise is hard won knowledge teased out of years of constant refinement of these systems and in that long hard battle a number of loyal allies deserve special mention.

Working With Microsoft

Working closely with Microsoft through our partner manager and their specialist financial services team has meant that we have always had the right resources focused on any problem in a quick and responsive fashion. Microsoft also offers assistance to developers in a variety of other ways, and a visit to Microsoft websites reveal a wide variety of technical information and programs intended to support developers and architects.

The Visual Studio 2008™ Performance Tools

Over the years Microsoft has consistently improved the code profiling and instrumentation tools in Microsoft Visual Studio 2008™. The current performance tools are invaluable to quickly identifying any functions allocating objects and tracking the life of those objects. This tool has become an essential part of our development life cycle and all of our latency sensitive products spend a good percentage of their development time under its watchful eye.

Event Tracing for Windows (ETW)

Microsoft Event Tracing for Windows (ETW) is an essential useful tool for measuring latency as well as a very wide range of performance characteristics. First introduced in Windows Server® 2000, ETW consists of a extremely low overhead tracing mechanism that instruments the Microsoft Windows® kernel, network stack, .NET® Framework, and applications such as Microsoft SQL Server.® It can also be used to instrument other applications by using standard Win32 API calls or corresponding .NET 3.5 methods. ETW Traces can be synchronized between different servers providing an extremely high resolution end-to-end latency and performance profile.

When used in this way, ETW can be used to trace a message or transaction through the system, providing not just averages, but full information on isolated latency anomalies. It is possible to see, for example when a UDP datagram arrives, which it is read by the application thread, when the thread encounters a lock contention, page fault, or context switch, .NET CLR GC, JIT, or other application specific events of interest.

In addition, Microsoft provides ETW analysis tools to permit rapid evaluation of ETW Performance information as part of the Windows Performance Analysis Tools, and these events can also be used by Microsoft Visual Studio 2010.

Results

Sending billions of messages over an extended period of days provides an important perspective on the performance of our systems. It's very important to examine the performance of these systems over extended periods both to avoid sampling errors, and to insure that long running systems will exhibit the desired performance characteristics. For instance, if your memory allocation pattern is such that you trigger a garbage collection every 120 seconds it is possible to create a great benchmark for a 30 second window of operation but your numbers of a period of several hours tell a very different story.

The table below shows the result of a 5345 minute test run (almost 4 days) in which almost 4 billion messages were processed. The results show a mean processing time of 9.65 microseconds with a standard deviation 3.85 (send) and 9.8 microseconds with a standard deviation of 1.56. This represents very little fluctuation about the mean especially given the extended test duration in addition to the very low processing latency.

Item	Send Total	Receive Total
Message count	3,855,299,151	3,467,953,771
Run time (minutes)	5345	5345
Min (us)	6	8
Mean (us)	9.53	9.83
Std (us)	3.85	1.56
Median (us)	9.49	9.66
95th Percentile (us)	11.49	11.38
99th Percentile (us)	16.04	13.34
99.9th Percentile (us)	32.43	30.26
99.99th Percentile (us)	48.20	45.11
99.999th Percentile (us)	67.06	61.73

Conclusion

These results show it is indeed possible to write low latency code in managed languages but it requires care and an understanding of the underlying runtime and its operation. The results achieved in managed code can be just as good as those achieved with unmanaged but without the additional risks and associated costs of unmanaged code.

Rapid Addition brings best-of-breed performance and latency FIX processing to a broad range of financial services industry applications, and by leveraging Microsoft .NET Framework™ and the Microsoft Windows family of operating systems , also offers unbeatable cost-of-ownership, manageability, and flexibility.

References

Rapid Addition website: http://www.rapidaddition.com

Microsoft .NET website: http://www.microsoft.com/net