



BEA WebLogic Event Server Performance

A Benchmark Study
October 2007

Table of Contents

Summary.....	3
Event Server Architecture	3
Real Time Event Server Kernel.....	5
WebLogic Real Time JVM.....	6
Benchmark Application	7
Benchmark Configuration and Methodology	8
Benchmark Results	10
Conclusion	14

Summary

BEA WebLogic Event Server is a lightweight, Java-based (non-J2EE) application server designed specifically to support event driven applications. Event driven applications, including those in financial services as well as other markets, are frequently characterized by the need to provide low and deterministic latencies while handling extremely high rates of streaming input data. This presents performance challenges that are quite different from those faced by more traditional application servers that tend to focus on obtaining the highest possible throughput for transactional workloads.

WebLogic Event Server has taken a unique approach among the products that are targeting event processing, by providing not only a Complex Event Processing (CEP) engine and Event Processing Language (EPL), but an overall platform that allows queries written in EPL to be tightly integrated with custom Java code written to a POJO/Spring programming model. WebLogic Event Server is truly an application server that provides container services and allows applications including a mix of EPL and Java code to be deployed and managed. An important design goal for the product was that performance not be sacrificed as a consequence of providing this rich development platform.

The benchmark study described in this report demonstrates WebLogic Event Server's ability to provide low latency at very high data rates with a use case that is very typical of financial front office applications in Capital Markets. The benchmark application implements a Signal Generation scenario in which the application is monitoring multiple incoming streams of market data watching for the occurrence of certain conditions that will then trigger some action.

For this application, WebLogic Event Server was able to sustain an event injection rate of up to 1 million events per second while maintaining low average and peak latencies. At this injection rate the average event latency for the full processing path within the server was 67.3 microseconds, with 99.4 percent of the events processed in less than 200 microseconds and 99.99 percent processed in less than 5 milliseconds.

The remainder of this paper includes a discussion of the product features that enable this level of performance, and a detailed description of the benchmark and its results.

Event Server Architecture

Figure 1 shows the high level architecture of the BEA Time and Event Driven platform. At the lowest level is the Java Runtime consisting of the JVM and core Java classes. Above this is the WebLogic Event Server. The Event Server is implemented using standard Java SE 5 APIs and is certified on BEA's JRockit and WebLogic Real Time JVMs. The lowest latencies and highest level of determinism is obtained when running on the WebLogic Real Time runtime environment which is based on JRockit with extensions for deterministic garbage collection.

WebLogic Event Server is a Java container implemented with a lightweight, modular architecture based on the OSGi framework. The container services include common core services such as logging, security, and management as well as services more specific to event driven applications including stream management and a complex event

processing engine. The WebLogic Event Server core also includes a real time kernel providing thread scheduling and synchronization support tuned for low latency and determinism. Applications consisting of a mix of Plain Old Java Objects (POJOs) and EPL queries can be dynamically deployed to the server and access the various services via Spring-based dependency injection. Rounding out the overall architecture is an integrated monitoring framework for precise monitoring of event latencies and a development environment based on the Eclipse IDE.

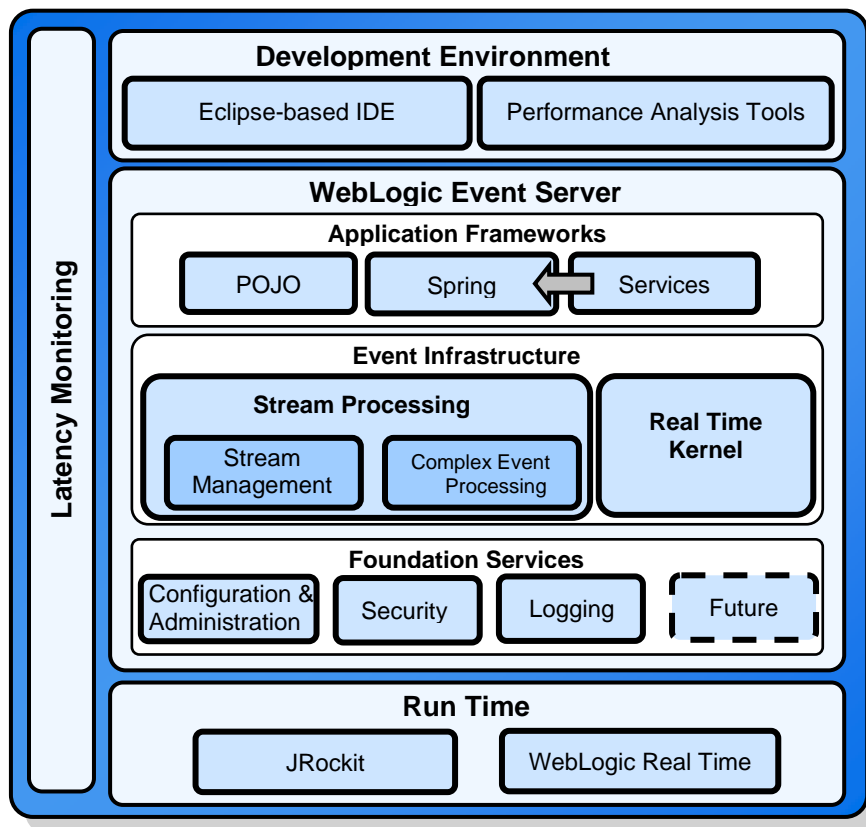


Figure 1 – BEA Time and Event Driven Platform

Figure 2 illustrates the typical data flow through a WebLogic Event Server application. On the inbound (left) side are event data streams from one or more event sources. The incoming data is received, unmarshalled, and converted into an internal event representation within an adapter module. The internal event representation can be an application defined Java Object or a Java Map. As the adapter creates event objects it sends them downstream to any components that are registered to listen on the adapter. In Figure 2, the listening components are “Stream” components. A Stream component is essentially a queue with an associated thread pool that allows the upstream and downstream components to operate asynchronously from each other. There is no requirement to include a Stream component in the event processing path but it can be very useful in increasing concurrency for applications that might otherwise have limited concurrency (e.g. a data feed coming in over a single connection). The next component in the Figure 2 data flow is the Processor component. A Processor represents an

instance of the Complex Event Processing engine and hosts a set of queries written in Event Processing Language (EPL). EPL queries support filtering, aggregation, pattern matching, and joining of event streams. The output of the configured EPL queries is sent to any downstream listeners. In this example a POJO is configured to listen to the Processor output. The POJO may perform additional processing on the events output from the queries and may trigger actions or send the output data to external systems via standard or proprietary messaging protocols.

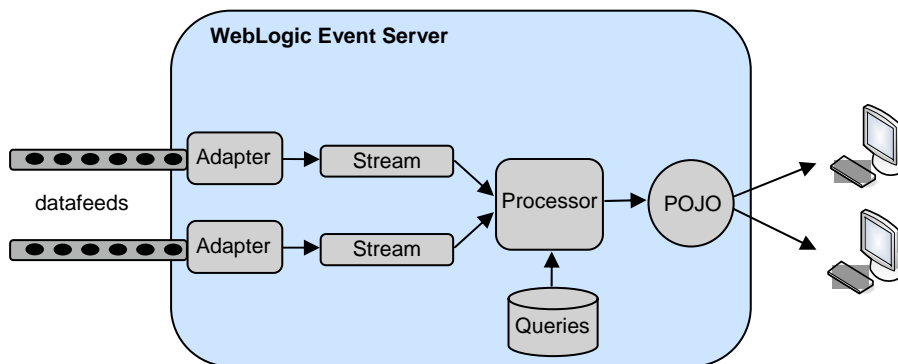


Figure 2 – Typical WebLogic Event Server Data Flow

The collection of interconnected Adapter, Stream, Processor, and POJO components is collectively referred to as the Event Processing Network (EPN). Although the example in Figure 2 shows a common topology, arbitrary EPN graphs may be wired together consisting of any number of components of each type in any order.

Real Time Event Server Kernel

Meeting the stringent latency requirements of typical event driven applications requires specialized support in the areas of thread scheduling, synchronization, and I/O. Techniques used by the WebLogic Event Server kernel to support low latency processing include:

- Thread scheduling that attempts to minimize blocking and context switching in the latency critical path. Whenever possible, a given event will be carried through its full execution path on the same thread with no context switch. This approach is optimal for latency and also ensures in-order processing of events for applications that require this. However in some cases handoff of an event between threads may be desirable. For example an application may wish to handle data from a single incoming network connection concurrently in multiple threads. The kernel provides flexible thread pooling and handoff mechanisms that allow concurrency to be introduced wherever it is desired in the processing path with minimal impact on overall latency.
- Synchronization strategies that minimize lock contention which could otherwise be a major contributor to latency.

- Careful management of memory including object reuse, use of memory efficient data structures, and optimized management of retain windows within the complex event processing engine. The memory optimizations benefit latency by reducing both the allocation rate and the degree of heap fragmentation, both of which help the garbage collector achieve minimal pause times.
- A pluggable adapter framework that allows high performance adapters to be created for a variety of network protocols, with support for multiple threading and I/O handler dispatch policies.
- Use of the WebLogic Real Time JVM described in the following section.

WebLogic Real Time JVM

While the WebLogic Event Server is certified on the standard version of BEA's JRockit JVM, the lowest latencies and highest level of determinism are obtained when running on the WebLogic Real Time JVM. WebLogic Real Time consists of the JRockit JVM with enhancements for low latency and deterministic garbage collection. Typical garbage collection algorithms stop all of the threads of an application when they perform a collection. The resulting "GC pause time" can be very long (several seconds or longer) in some environments and is a major contributor to latency spikes and jitter. WebLogic Real Time's deterministic garbage collector uses a different approach designed to make GC pause times both shorter and more predictable. The deterministic collector handles much of the collection while the application is running and pauses only briefly during critical phases of the GC. In addition the WebLogic Real Time collector will monitor the duration of individual pauses to ensure that the amount of time spent in a given GC pause doesn't exceed a user specified pause target. For example with a user specified pause target of 10 milliseconds, the deterministic collector would limit the duration of individual GC pauses to no more than 10 milliseconds providing a high degree of predictability compared to traditional GC algorithms.

In addition to the deterministic GC feature the WebLogic Real Time product also includes a Latency Analyzer Tool, integrated with the JRockit Runtime Analyzer (JRA) tool. The Latency Analyzer Tool (LAT) is a unique performance analysis tool that identifies and analyzes sources of latency within a Java application. While typical profiling tools focus only on where CPU time is spent while the application is running, the Latency Analyzer provides detailed information about where, when, and for how long the various threads of the application block or wait. The Latency Analyzer Tool can identify the cause and duration of a thread wait whether the cause is due to garbage collection, I/O, synchronization, or an explicit sleep or wait requested by the application. The ability to locate and analyze these sources of latency is indispensable in tuning a latency sensitive application and the Latency Analyzer Tool was used extensively in tuning the benchmark described in this paper.

Benchmark Application

The application used for this benchmark study implements a Signal Generation scenario in which the application is monitoring multiple incoming streams of market data watching for the occurrence of certain conditions that will then trigger some action. This is a very common scenario in front office trading environments. Figure 3 shows the overall structure of the benchmark.

The incoming data is generated by a load generator which creates simulated stock market data and sends it to the server over one or more TCP connections at a configured, metered rate. The format of the data on the wire is specific to the implementation of the load generator and adapter and is designed for compactness. Within the event server the adapter reads the incoming data from the socket, unmarshalls it, creates an event instance (a Java object conforming to certain conventions) for each incoming stock tick, and forwards the events to the event processor. Within the event server the adapter reads the incoming data from the socket, unmarshalls it, creates an event instance (a Java object conforming to certain conventions) for each incoming stock tick, and forwards the events to the event processor.

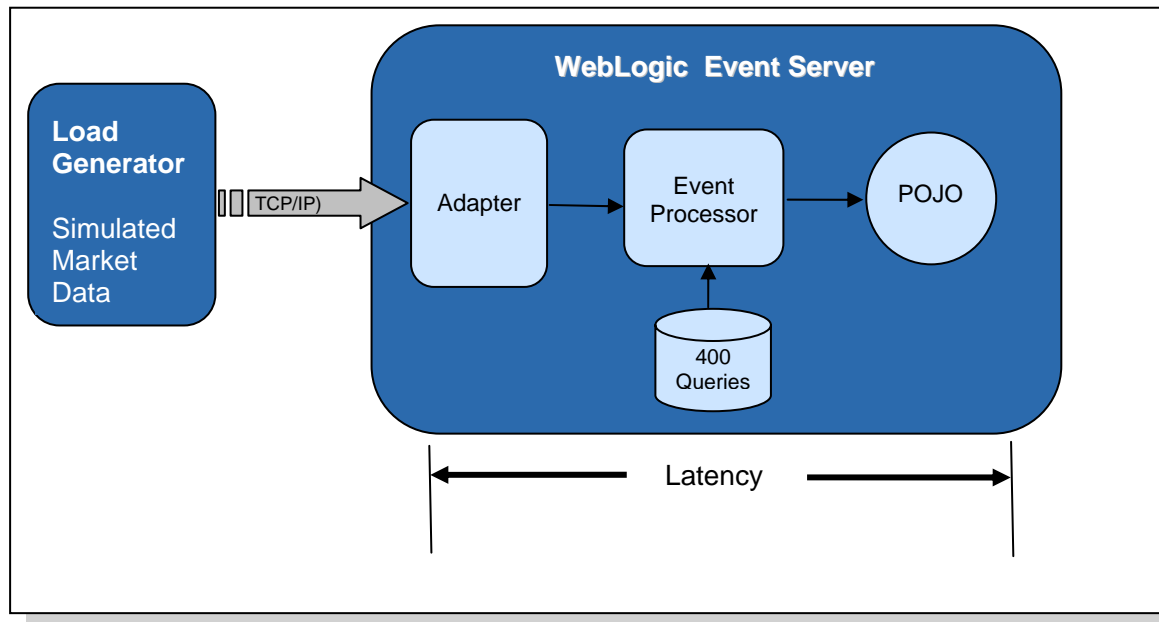


Figure 3 – Benchmark Application Structure

The event processor is configured to monitor the incoming data for any one of 200 different stock symbols. Each of these stock symbols is monitored for the following two conditions:

- The stock price increases or decreases by more than 2% from the immediately previous price
- The stock price has 3 or more consecutive upticks without an intervening downtick

The Event Processing Language syntax that is used to implement these rules for the stock symbol “WSC” is shown below:

```
SELECT symbol, lastPrice, perc(lastPrice), clientTimestamp, timestamp
FROM (select * from StockTick where symbol='WSC') RETAIN 2 EVENTS
HAVING PERC(lastPrice) > 2.0 OR PERC(lastPrice) < -2.0
```

```
SELECT symbol, lastPrice, trend(lastPrice), clientTimestamp, timestamp
FROM (select * from StockTick where symbol='WSC') RETAIN 3 EVENTS
HAVING TREND(lastPrice) > 2
```

These two queries are replicated for each of the 200 symbols being monitored, resulting in a total of 400 queries that the event processor must execute against each incoming event. When an incoming event matches one of the rules, an output event is generated with the fields specified in the select clause and sent to any downstream listeners. In this case the downstream listener is a Java POJO which computes aggregate statistics and latency data for the benchmark based on the output events it receives.

Latency data for the benchmark is computed based on timestamps taken in the adapter and POJO. The adapter takes the initial timestamp after reading the data from the socket and prior to unmarshalling. This initial timestamp is inserted into each event created by the adapter, is passed through the event processor and inserted into any output events generated by a matching rule. When the POJO receives an output event it takes an end timestamp and subtracts the timestamp generated by the adapter to compute the processing latency for that event. These latencies are then aggregated to produce overall latency data for the duration of the benchmark run.

Benchmark Configuration and Methodology

Load Injection

The load generator can be configured to specify the number of connections it should open to the event server and the rate at which it should send data over each connection. We will refer to the aggregate send rate across all connections as the aggregate *injection rate*. For this benchmark the data sent by the load generator for each event consists of a stock symbol, simulated price, and timestamp data. The average size of the data on the wire is 20 bytes per event not including TCP/IP header overhead. The stock symbols are generated by repeatedly cycling through a list of 1470 distinct stock symbols. If the load generator is configured to open multiple connections to the server, the symbol list is partitioned evenly across the set of connections. The price data is generated dynamically based on a geometric brownian motion algorithm and the price for a given symbol is updated each time the symbol is sent.

Event Server Configuration

The event processing network (EPN) configuration within the event server consists of a single adapter instance, single processor instance, and a single POJO as described in the previous section.

The adapter is configured to use a blocking thread-per-connection model for reading the incoming data and dispatching the events within the server. The adapter feeds all of the injected input events to the processor, which is configured with a total of 400 queries (200 distinct symbols with 2 rules per symbol) as described in the previous section. Each of the configured queries is run against each input event, and for each match an output event is sent downstream to the POJO.

Hardware and Software Stack

The hardware consists of one machine for the event server and one machine for the load generator, connected by a gigabit ethernet network. The server and load generator machines each have an identical hardware configuration and identical software stack as described below:

Hardware Platform:	Intel “Caneland” Platform 4 quad-core Intel X7350 processors at 2.93 GHz (16 cores total) 8 MB L2 cache per processor, shared across the 4 cores 32 GB RAM
Operating System:	Red Hat Enterprise Linux 5.0, 32 bit. Kernel 2.6.18-8.
JVM:	BEA WebLogic Real Time 2.0 (JRockit R27.3.1-1 1.5.0_11) 32 bit. 1 GB Heap Size, Deterministic GC enabled
Event Server:	BEA WebLogic Event Server 2.0 (with support patch ID XQWK)

Methodology

The benchmark data was collected as follows:

- An initial 15 minute warmup run was done with the load generator opening 10 connections to the server and sending data at a rate of 100,000 events per second per connection.
- The warmup was followed by a series of 10 runs scaling the number of connections from 1 to 10 with the load generator sending 100,000 events per second per connection in all cases (maximum injection rate of 1,000,000 events/second). The duration of each run was 10 minutes.
- An additional series of 10 runs was done holding the number of connections fixed at 10 and scaling the injection rate per connection from 10,000 to 100,000 events (maximum injection rate of 1,000,000 events/second). The duration of each run was 10 minutes.
- The injection rate, output event rate, average latency, absolute maximum latency, and latency distributions were collected for all runs.

Benchmark Results

Table 1 and Figures 4 and 5 show the results scaling from 1 to 10 connections at 100,000 events per second per connection. As discussed earlier, the latency values are collected only for those events that are forwarded to the POJO as a result of a match, and represent the latency from an initial timestamp in the adapter (prior to unmarshalling and creation of the internal event object) and a timestamp when the event is received by the POJO.

Connections	Injection Rate Per Connection (events/sec)	Total Injection Rate (events/sec)	Output Event (Match) Rate	Average Latency (microsecs)	99.99% Latency (milliseconds)	Absolute Max Latency (milliseconds)
1	100,000	100,000	3911	42.6	0.2	8.77
2	100,000	200,000	7811	44.0	2.1	12.77
3	100,000	300,000	11686	47.2	2.2	12.93
4	100,000	400,000	15595	49.3	2.4	13.45
5	100,000	500,000	19466	51.5	2.5	15.73
6	100,000	600,000	23351	53.6	2.6	16.64
7	100,000	700,000	27234	55.5	2.6	18.60
8	100,000	800,000	31235	58.3	3.1	19.50
9	100,000	900,000	35080	62.0	3.7	19.21
10	100,000	1,000,000	38890	67.3	4.3	21.52

Table 1 – Scaling from 1 through 10 connections at 100,000 events per second per connection

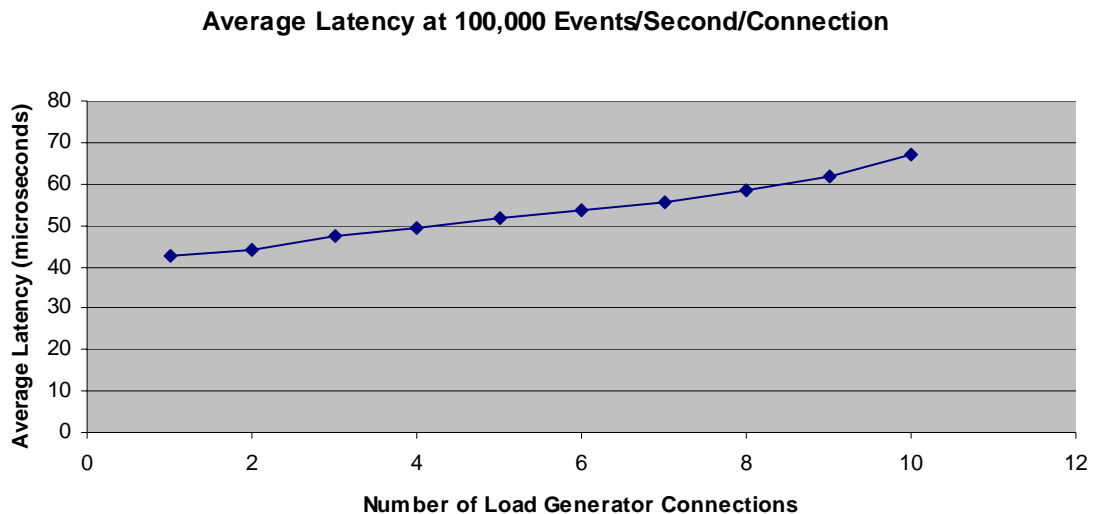


Figure 4 – Average latency scaling from 1 through 10 connections (100,000 – 1,000,000 events/second)

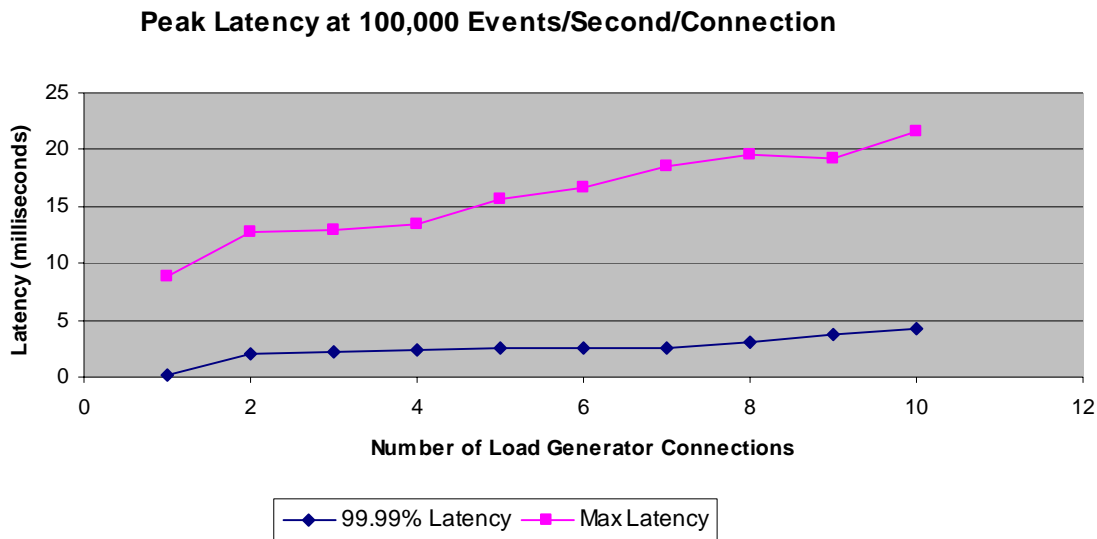


Figure 5 – Peak latency scaling from 1 through 10 connections (100,000 – 1,000,000 events/second)

As Table 1 shows, the output event rate was a fixed percentage (3.9%) of the injection rate as the load increased. There was a gradual increase in average and maximum latencies as the number of connections and overall injection rate increased. The 99.99 percentile latencies remained fairly flat (between 2.1 and 2.6 ms) with increasing load from 200,000 through 700,000 events per second and then increased slightly as the injection rate approached 1,000,000 events/second. At the maximum benchmarked load of 1,000,000 events/second the average and 99.99% latencies are still quite low and the even the absolute maximum has degraded only slightly with the increased load.

Table 2 and Figures 6 and 7 show the results when holding the number of connections fixed at 10 and scaling the injection rate.

Connections	Injection Rate Per Connection (events/sec)	Total Injection Rate (events/sec)	Output Event (Match) Rate	Average Latency (microsecs)	99.99% Latency (milliseconds)	Absolute Max Latency (milliseconds)
10	10,000	100,000	3893	45.2	0.4	13.97
10	20,000	200,000	7793	45.7	0.8	13.26
10	30,000	300,000	11676	47.2	1.1	13.81
10	40,000	400,000	15564	49.3	1.3	16.12
10	50,000	500,000	19460	51.6	1.7	19.21
10	60,000	600,000	23348	54.2	2.1	20.96
10	70,000	700,000	27239	56.6	2.6	20.91
10	80,000	800,000	31123	59.4	3.4	20.31
10	90,000	900,000	35001	62.9	3.9	21.83
10	100,000	1,000,000	38890	67.3	4.3	21.52

Table 2 – Scaling from 100,000 to 1,000,000 events per second with 10 connections

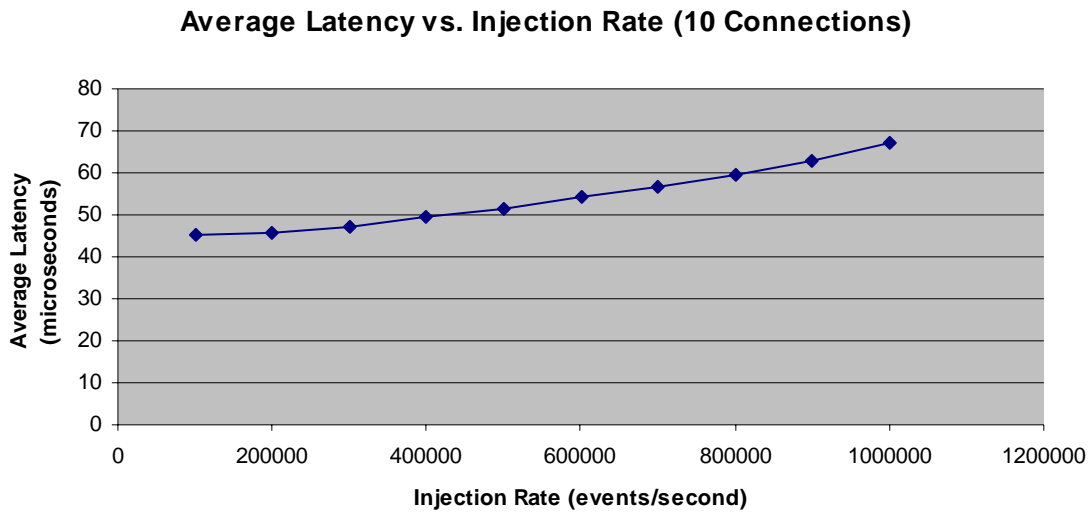


Figure 6 – Average latency scaling from 100,000 to 1,000,000 events per second with 10 connections

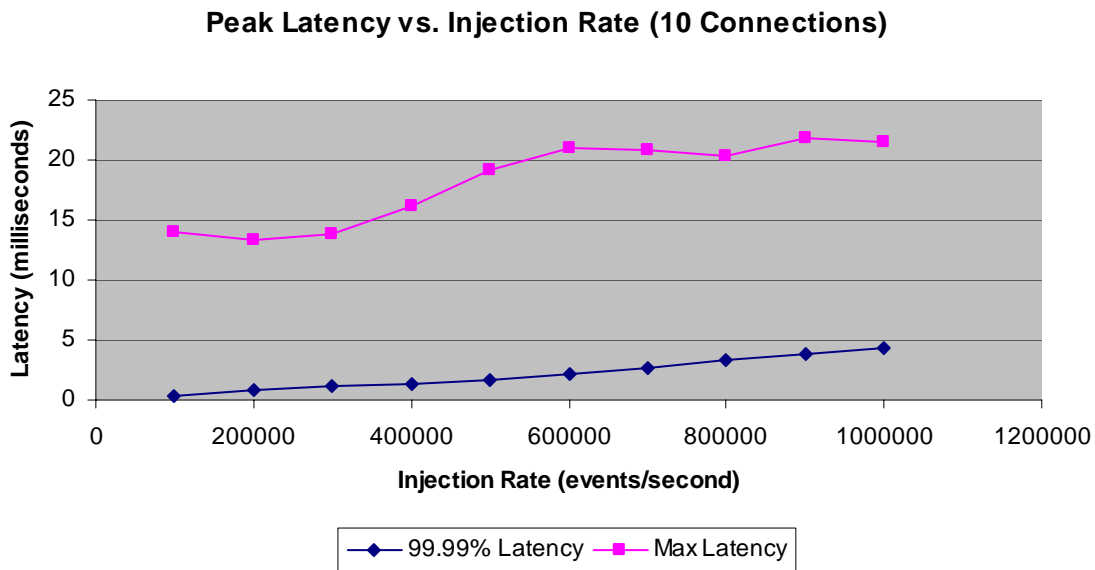


Figure 7 – Peak latency scaling from 100,000 to 1,000,000 events per second with 10 connections

The effect of increasing load on latency when scaling with a fixed number of connections is very similar to the results shown above when the load was scaled up by increasing the number of connections. This similarity in results suggests that the performance of the system at a given input load is mostly independent of the number of connections used to inject the data. A difference is visible in the max latency curves in the range of 400,000 to 700,000 events per second suggesting that in this range max latencies may be reduced at a given load by using a smaller number of connections.

The histograms in Figure 8 show the latency distribution at an injection rate of 1,000,000 events/second. The first histogram uses a linear scale on the Y axis and consolidates the event count for the range > 200 microseconds into a single (barely visible) bar. The second histogram displays the same data using a log scale on the Y axis to provide additional detail in the > 200 microsecond latency range. The histograms illustrate how strongly skewed the distribution is toward the lower end of the latency range with 86.3% of the latency values below 100 microseconds and 99.4% of the latency values below 200 microseconds at an injection rate of 1,000,000 events/second.

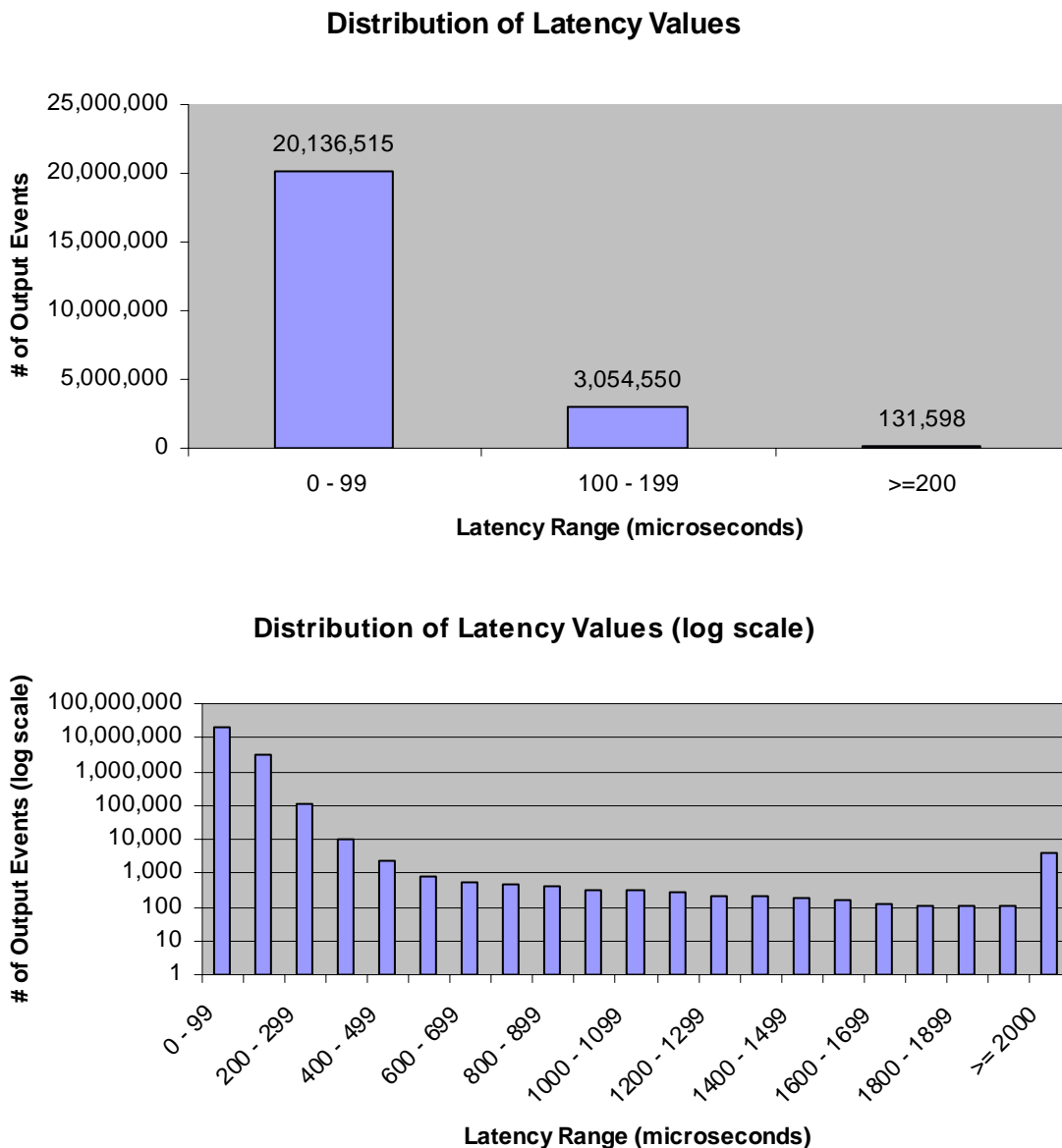


Figure 8 – Distribution of output latency values over 10 minute run at 1,000,000 event/second injection rate.

Finally, Figure 9 shows the garbage collection pause times for the duration of the benchmark run at 1,000,000 events/second. There were a total of 477 garbage collections over the course of the 10 minute run. The maximum GC pause during the run was 17 milliseconds and 97 percent of the pauses were at or below 15 milliseconds. The ability of the WebLogic Real Time JVM to maintain these short and predictable GC pauses under load was a major factor in limiting the peak application latencies.

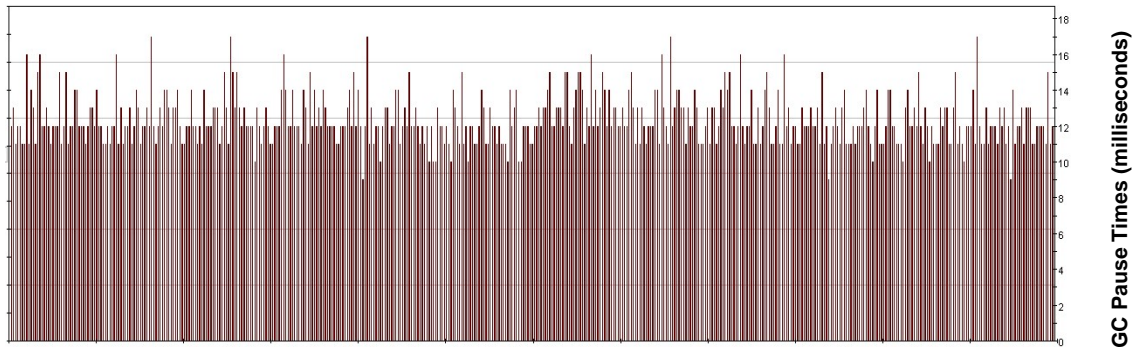


Figure 9 – Garbage collection pause times over 10 minute run at 1,000,000 event/second injection rate.

Conclusion

This paper has reviewed the overall architecture of BEA WebLogic Event Server and some of the specific features and design characteristics that allow it to provide high performance for event driven applications. The performance characteristics of WebLogic Event Server were studied using a very common event processing use case. The results demonstrate very clearly WebLogic Event Server's ability to achieve low and predictable latency under very high loads.