WebLogic Event Server: A Lightweight, Modular Application Server for Event Processing

Seth White
BEA Systems, Inc.
475 Sansome St
San Francisco, CA 94111
+1.415.402.7767
swhite@bea.com

Alexandre Alves
BEA Systems, Inc.
475 Sansome St.
San Francisco, CA 94111
+1.415.402.7244
aalves@bea.com

David Rorke
BEA Systems, Inc.
150 Allen Road
Liberty Corner, NJ 07938
+1.908.580.3404
drorke@bea.com

ABSTRACT

This paper describes WebLogic Event Server (WL EvS), an application server designed for hosting event-driven applications that require low latency and deterministic behavior. WL EvS is based on a modular architecture in which both server components and applications are represented as modules. The application programming model supports applications that are a mixture of reusable Java components and EPL (Event Processing Language), a query language that extends SQL with stream processing capabilities. WL EvS applications are meta-data driven, in that application behavior can be changed without recompilation or redeploying an application. The paper also presents the results of a benchmark performance study. The results show that the approach used by WL EvS can handle extremely high volumes of events while providing deterministic latency.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Algorithms, Performance, Design

Keywords

application server, complex event processing, stream processing, Java, OSGi, Spring

1. INTRODUCTION

WebLogic Event Server (WL EvS) [2] is a Java [9] application server designed to support real-time, event-driven applications. Real-time applications that require extremely high throughput and deterministic latency represent a new frontier for Java-based middleware, in large part due to the challenges posed by garbage collection. To satisfy the demands of these applications, WL EvS employs a lightweight, modular software architecture that runs on top of a Java VM featuring deterministic garbage collection. The architecture enables a service-oriented approach to be used, in which modules implement interdependent services that together provide the functionality needed by WL EvS applications.

The programming model that WL EvS offers to application developers is a hybrid model featuring both Java and the Event Processing Language (EPL), an SQL-based query language for complex event processing. The model allows application developers to mix and match Java and EPL seamlessly within the same application. WL EvS applications thereby gain the benefits of both complex event processing technology, including incremental evaluation of EPL queries, and the ability to write business logic in a high level programming language with built-in memory management.

WL EvS applications are composed of reusable components that are configured using dependency injection techniques. WL EvS applications are also configuration-driven. Arbitrary configuration changes may be made to the server without interrupting running applications. For example, an EPL rule set can be changed on-the-fly resulting in a change in program behavior without interrupting the running application. This support for dynamic change results in a new level of flexibility for enterprise applications.

To demonstrate the capabilities of WL EvS, the paper presents the results of a benchmark performance study. The results show that WL EvS can process extremely high event volumes with deterministic latency. The benchmark application was derived from event-driven applications in the financial domain.

The remainder of the paper is organized as follows. Section 2 discusses the modular architecture of WL EvS and how WL EvS fits into a distributed architecture. Section 3 presents the application programming model. Section 4 covers some of the important aspects of the EPL language. Section 5 discusses the server configuration. Section 6 presents the benchmark performance results. Section 7 contains our conclusions.

2. ARCHITECTURE

2.1 Server Architecture

WL EvS is architected as a set of relatively fine-grained software modules. Currently, there are 140+ modules. Each module contains a set of Java classes that implement some aspect of the server's functionality. A module may act as a library for other modules by exporting classes for client modules to use. A module

DEBS '08, July 2–4, 2008, Rome, Italy.

may also register services in a service registry for use by other modules.

Figure 1 shows the high level software architecture of a WL EvS instance. At the lowest level there is a Java virtual machine [13] (JVM) with deterministic garbage collection (DGC) [3]. The JVM provides the foundation for support of applications that demand deterministic latency by limiting the length of garbage collection pauses. The next layer is composed of the modularity framework [15] which allows modules to control the import and export of Java classes. The modularity layer also handles classloading and provides versioning support for classes. The service framework [16] is responsible for instantiating the classes that implement a service and for resolving dependencies between services. The service framework uses dependency injection (DI) [7] to provide service instances with configuration data and references to other services that they need.

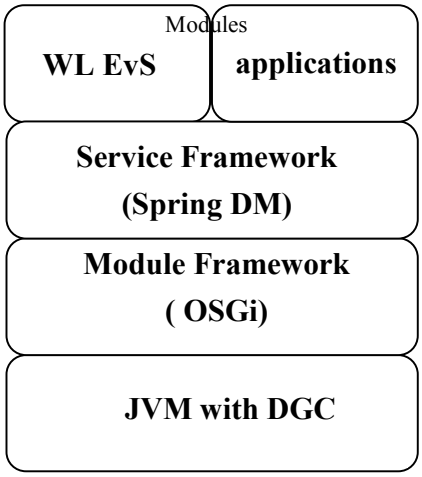

Figure 1. WL EvS software stack.

The final layer represents the modules themselves. There are two fundamental types of modules--modules that are part of the server implementation and modules that make up applications deployed to a server instance. Typically, one or more modules will implement a logical subsystem in the server, such as the configuration or deployment subsystems. Applications may also be composed of one or more modules. The architecture is uniform in that there is no physical distinction between server and application modules. Application modules use services provided by server modules. In addition, application modules can extend the functionality of the server by providing services that are invoked by server modules.

Figure 2 shows a sampling of the server modules that make up WL EvS. Most of the subsystems implemented by these modules, such as configuration, deployment, security, logging, etc. would also be found in more traditional, i.e. Java EE [10][11][14][4], application server implementations. An example of a library module is JAXB [12] which is used by the Configuration module for processing XML configuration. The Configuration module in turn exports a service that is used by the Management module to read and update server-wide and application-level configuration. These relationships are illustrated in Figure 3.

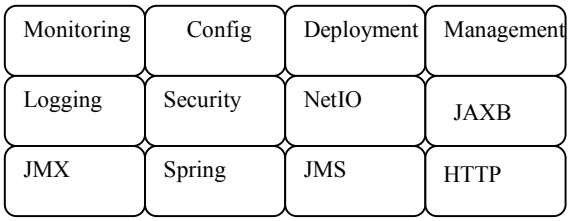

Figure 2. Examples of WL EvS server modules.

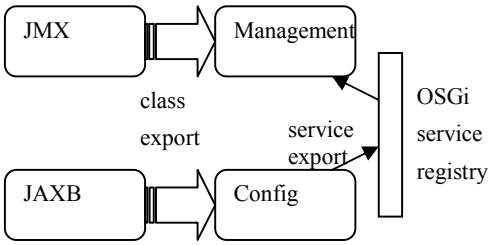

Figure 3. Module class and service dependencies.

Given the large number of modules in the server, the dependency relationships could become quite complex. WL EvS uses a layered approach and attempts to avoid cyclic dependencies as much as possible. Server services are implemented as components using a generic component framework [16] that provides services such as dependency injection, and service lifecycle management. All services are implemented using the Java programming language.

2.2 Distributed Architecture

Event-driven architecture is a distributed architectural style composed of decoupled applications that interact by exchanging events. These applications are called event-driven applications. Event-driven applications play the role of emitter of events, and of responder or processor of events. Event driven applications are sense-and-respond applications; that is, applications that react to and process events.

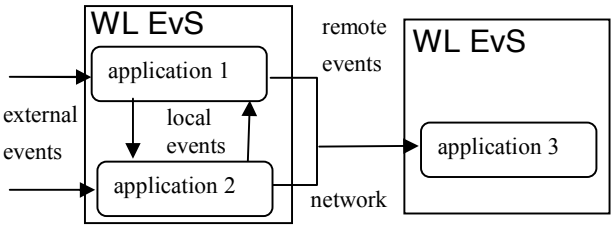

Figure 4. Event-driven applications deployed in WL EvS.

Figure 4 illustrates WL EvS as part of a distributed, event driven architecture. In Figure 4, application modules *application1* and *application2* are deployed in one instance of WL EvS and send events to a third application deployed in a separate instance of WL EvS. In addition, both application1 and application2 receive events from an external event source, such as a market data feed in a financial application.

Event-driven architecture is important, because the real-world is event-driven. One example is the financial world, in which trader applications react to events (or changes) made to the financial exchange market. Event-driven situations should be modeled by event-driven architecture. Due to its asynchronous and decoupled nature, event-driven architecture has an intrinsic quality for scaling and supporting real-time constraints.

3. PROGRAMMING MODEL

3.1 Event-driven Components

WL EvS provides a native programming model for authoring event-driven applications. In WL EvS, a user defines an event-driven application by assembling a set of event-driven components. The component types are:

- event source: a component that generates events
- event sink: a component that consumes events
- stream: a component through which events flow, provides queuing and concurrency
- processors: a component capable of processing events, the type of processing is specific to a processor
- event types: metadata defining the properties of events

Within an application, events flow starting at event sources. Events then flow through a set of processors and finally reach the event sink(s). Event sources and sinks are also termed adapters because they are primarily responsible for converting events from their external wire format to the Java format understood by the application and vice versa. Streams link components together forming an event flow graph, which is called an Event Processing Network (EPN). Formally, an EPN is a non-rooted directed graph, in which vertices (i.e. nodes) are instances of event sources, event sinks, or processors, and arcs are streams.

Figure 5 provides an example of an EPN for a simple financial market pricing application. This event-driven application contains two event sources, each receiving stock tick events from two different exchange markets. The application further defines a processor that is configured to calculate and output the price of a stock symbol as being the average price received from the exchange market event sources. Finally, there is a single event sink that publishes the calculated average stock price to a well-known JMS [20] destination.

An EPN is a formal model, based upon Petri Networks [17], which allows:

- the specification of the concurrency between the EPN nodes
- the prioritization of EPN paths
- the composition of applications

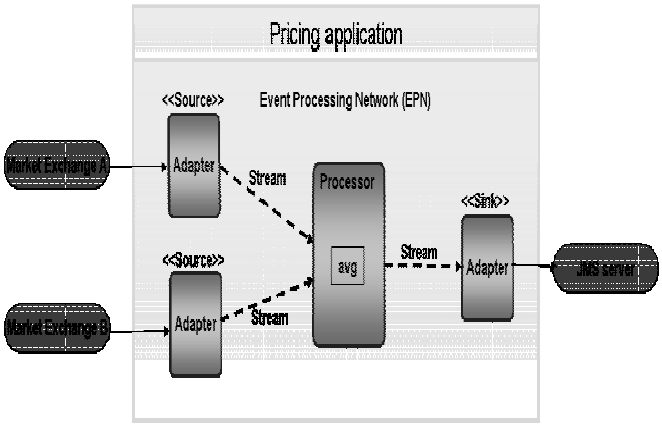

Figure 5. EPN for pricing application.

3.2 Concurrency

The EPN allows the specification of different concurrency models. The paths initiated at the roots of the EPN, that is the event sources, are executed concurrent to each other. Adjacent nodes in the EPN establish an ordering dependency, so in the context of an event, two adjacent nodes process the event sequentially. Conversely, nodes that are not adjacent, for example, two nodes that are the fan-out of a common parent, may execute the same event concurrently, as determined by the runtime infrastructure.

Thus the application developer has the means to model the application logic and restrictions in the EPN. For example, if an application has to first cleanse an event and then calculate a property value, it should model these two processing functions into adjacent processor nodes, as illustrated in Figure 6.

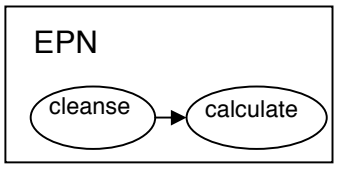

Figure 6. Sequential processing of events.

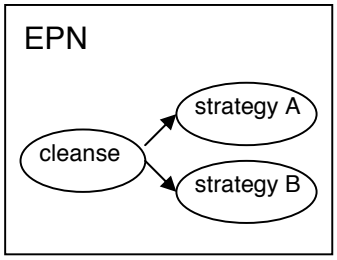

Figure 7. Concurrent processing of events.

However, if there are several different ways of calculating the property value and these calculations can happen concurrently,

then each calculation should be placed into parallel nodes in the EPN, as illustrated in Figure 7.

Another concurrency design pattern is to partition events into separate categories that can be processed concurrently and place these as separate roots at the EPN. For example, if events from different clients can be processed concurrently, one can create a separate event source for each client, or for a set of clients.

3.3 Prioritization

WL EvS is designed and implemented with the goal of providing a deterministic execution environment for event-driven applications. Consider the pricing application discussed previously. It is important that the application calculate the price of the stock symbol within the bounds of a guaranteed worst-case time, otherwise, considering the fast pace of the stock market, the price could be stale and not usable. In particular, the guaranteed worst-case time must hold even when the load on the system is high, which generally happens during peak stock trading hours.

To accomplish this, WL EvS is implemented using several real-time design patterns and practices, such as avoiding unbounded data structures like linked lists. For example, a user may specify the maximum number of events that a stream component in the EPN is able to hold at a given time. The user may also specify different discard policies to handle situations in which the maximum is reached. By assigning a maximum size to the streams in the EPN, the user can model the worst-case behavior of an application. The user specifies which paths in the EPN have more resources to process events and thus have higher priority, particularly when the load in the system is high.

3.4 Application Composition

A user authors a WL EvS application by wiring EPN components together. The components perform distinct functions--collaborating to fulfill the goal of the application. In this way, a user is able to decompose the application logic into separate stages, and manage application complexity and component re-use. A user can implement a processor component using either the Java or EPL languages. This allows the user to seamlessly integrate a domain-specific language, such as EPL, with a general purpose programming language, like Java.

Component implementations are hosted by an event-driven container that is implemented as an extension of a general purpose container [16] supporting dependency injection, Plain-Old-Java-Object (POJO) components, aspect-oriented programming (AOP), and declarative XML configuration. This gives WL EvS application developers access to WL EvS extensions as well as the full power of the underlying container. An EPN is specified declaratively using a WL EvS-specific extension of the generic container's XML configuration file format. The generic XML configuration has been extended to support the WL EvS event-based programming model, in which users create the EPN graph formed of event sources, event sinks, streams, and processors.

WL EvS provides a pre-packaged processor implementation that supports the Event Processing Language (EPL). The EPL processor implementation delegates to the configuration module to handle the persistence of EPL queries, and supports dynamic changes to its configuration, such as the addition and removal of EPL queries. A user may also write a custom POJO component to act in the role of a processor. In another words, the POJO is an intermediate node in the EPN graph and performs event processing functions. Generally, event processing functions performed by POJO components are geared towards functions such as event passing, event routing, and event mediation, or tasks that are simply better suited to being written in Java. EPL processors handle complex query processing, such as aggregation, and pattern matching.

WL EvS is a domain-specific application server; that is, an application server targeted for the development and execution of event-driven applications. Due to its modular and service-oriented architecture, it is highly customizable. The extensible architecture of the programming model allows for other processor implementations -- supporting additional query languages -- to be plugged into the server.

4. EVENT PROCESSING LANGUAGE

The Event Processing Language (EPL) is an SQL-based query language for complex event processing. EPL extends SQL with a number of features needed to query streams of events. EPL was inspired by many of the ideas that have come out of the research and industrial CEP communities [1][4][18][19][8].

Figure 8 shows an example EPL query that filters StockTick events -- only outputting events that have a price greater than 100.

```
INSERT INTO BigStockTick
SELECT symbol, price, timestamp
FROM StockTick
RETAIN 2 EVENTS
WHERE  price > 100.0
```

Figure 8. Simple EPL query.

The example in Figure 8 demonstrates a number of the basic features of EPL. The From clause references the StockTick event type. Event types are mapped to a Java class. In the example, StockTick events have symbol, price, and timestamp properties which must also be present on the associated Java class. The Where clause contains a simple expression that evaluates to true whenever the price of the StockTick event is greater than 100. The Select clause selects the symbol, price, and timestamp properties from StockTick events that satisfy the Where clause.

The 'Insert Into' and 'Retain' clauses in Figure 8 are examples of EPL extensions to the SQL language. The Insert Into clause references a second event type, BigStockTick, which defines the type of the events that are output by the example query. The example query essentially maps input events of type StockTick into output events of type BigStockTick. In the example, the mapping is very simple, but EPL does allow for much more complex mapping between event types. The Retain clause is used to define a window of events in the input stream over which the query is evaluated. In the example in Figure 8, the window consists of the two most recent StockTick events that have arrived. The input window is conceptually similar to a table in a relational database, however, EPL queries are not generally evaluated against the entire contents of the input window. Query evaluation is incremental and typically only involves evaluation against the most recent event to enter or leave the input window.

In addition to the SQL language clauses discussed above, EPL also supports Group By, Having, and Order By clauses. A Matching clause can be used to detect temporal patterns of events. Other EPL features include user-defined functions, inner and outer joins, and sub-queries.

5. CONFIGURATION

Like most commercial application servers, WL EvS supports dynamic configuration changes for server-level configuration data, such as connection pool and logging configuration. In addition, WL EvS goes beyond traditional configuration and component models [6] by allowing dynamic configuration changes to be made to individual application components. The ability to dynamically change the behavior of application components through configuration updates gives administrators and end-users of WL EvS applications increased flexibility and control over applications. It allows applications to be modified in predefined ways without the need to take the application off-line or requiring that a new version of the application be deployed.

Configuration changes can span multiple components and even applications. Configuration changes are atomic in that either all components involved in a configuration change will be updated, or none of them will. A standard two-phase algorithm is used notify components of a dynamic configuration change.

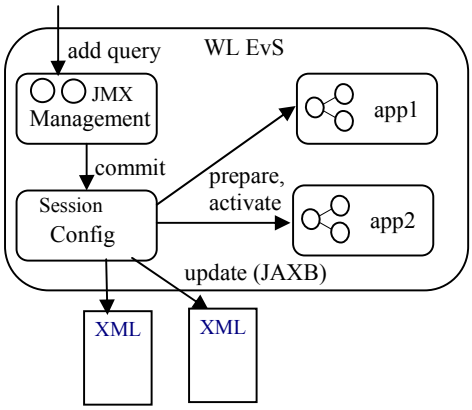

Figure 9. Dynamic configuration change.

Figure 9 shows how dynamic configuration changes are implemented within WL EvS. In the example, two separate applications have been deployed to the WL EvS instance. Each application is packaged as a separate OSGi module. We assume that the applications cooperate in some way and that their configuration needs to be kept consistent. The configuration change being made in Figure 9 is the addition of a new EPL query to a processor component in each application.

The configuration update begins in the management module. A remote JMX client invokes the "add query" operation on the management components for the two processors. Internally, the management module begins a Session with the configuration subsystem, uses the Session to retrieve and update Java objects that represent the processors' configuration data, and then commits the Session. The Configuration module provides services such as transparent persistence and version-based concurrency control to configuration clients.

When the configuration session is committed, the configuration module invokes prepare and activate operations on the affected application components, passing each component the Java representation of its updated configuration data. Components may reject a change during the prepare operation which will cause a rollback operation to be invoked on each component instead of these subsequent activate operation. Once all components have accepted the configuration change, the configuration module persists the updated configuration data to XML documents stored in the file system. JAXB is used to map between the Java and XML representations.

The configuration system of WL EvS is also extensible. Application developers can define custom configuration data for the components that they write. Configuration data for custom components can be updated dynamically, using the same techniques as those employed for built-in components such as processors and streams.

6. PERFORMANCE STUDY

6.1 Benchmark Overview

The ability to process event data at very high rates, while maintaining low and average and peak latencies, is a critical requirement for many event-driven applications. The benchmark study described in this section was designed to measure the average and worst case latency performance of an event-driven application running on WL EvS under heavy load.

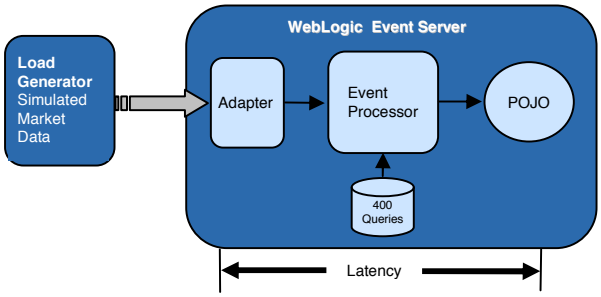

Figure 10. Benchmark Application Structure.

The application used for this benchmark study implements a signal generation scenario in which the application is monitoring multiple incoming streams of financial market data watching for the occurrence of certain conditions that will then trigger some action. This is a very common scenario in financial trading environments. Figure 10 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.

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 EPL queries that are used to implement these rules for the stock symbol "WSC" are shown below.

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

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

The 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 component 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.

6.2 Configuration and Methodology

6.2.1 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.

6.2.2 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.

6.2.3 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® Xeon® 7300 based Server

4 Quad Core Intel® 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 1.5.0_11) 32 bit.

1 GB Heap Size, Deterministic GC enabled

Event Server:

BEA WebLogic Event Server 2.0 (with patch ID XQWK)

6.2.4 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.

6.3 Benchmark Results

Table 1 shows the results scaling from 1 through 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. The average latencies reported in the table are in units of microseconds and the 99.99 percentile and max latencies are in milliseconds.

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 percentile latencies are still quite low and even the absolute maximum has degraded only slightly.

Conn - ections	Total Injection Rate (evnt/sec)	Output Event Rate	Avg. Latency (usecs)	99.99% Latency (msecs)	Max Latency (msecs)
1	100,000	3911	42.6	0.2	8.8
2	200,000	7811	44.0	2.1	12.8
3	300,000	11686	47.2	2.2	12.9
4	400,000	15595	49.3	2.4	13.5
5	500,000	19466	51.5	2.5	15.7
6	600,000	23351	53.6	2.6	16.6
7	700,000	27234	55.5	2.6	18.6
8	800,000	31235	58.3	3.1	19.5
9	900,000	35080	62.0	3.7	19.2
10	1,000,000	38890	67.3	4.3	21.5

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

Table 2 shows the results when holding the number of connections fixed at 10 and scaling the injection rate per connection. 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 injection rate is mostly independent of the number of connections used to inject the data. A difference is visible in the max latency data 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.

Rate per Conn- ection	Total Injection Rate (evnt/sec)	Out- roput Event Rate	Avg Latency (usecs)	99.99% Latency (msecs)	Max Latency (msecs)
10,000	100,000	3893	45.2	0.4	13.9
20,000	200,000	7793	45.7	0.8	13.3
30,000	300,000	11676	47.2	1.1	13.8
40,000	400,000	15564	49.3	1.3	16.1
50,000	500,000	19460	51.6	1.7	19.2
60,000	600,000	23348	54.2	2.1	20.9
70,000	700,000	27239	56.6	2.6	20.9
80,000	800,000	31123	59.4	3.4	20.3
90,000	900,000	35001	62.9	3.9	21.8
100,000	1,000,000	38890	67.3	4.3	21.5

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

The histogram in Figure 11 shows the latency distribution at an injection rate of 1,000,000 events/second. Note that a log scale is used on the Y axis to provide additional detail in the > 200 microsecond latency range. The histogram illustrates how strongly skewed the distribution is toward the lower end of the latency range with 86.3% of the latency values below 100

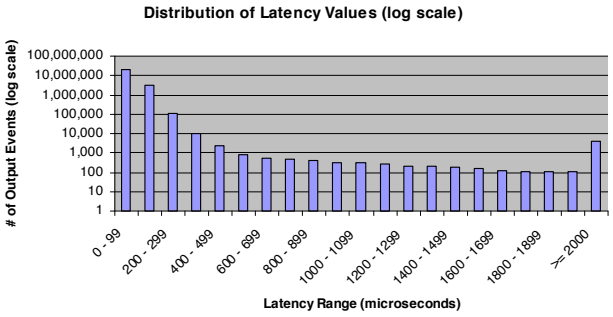

Figure 11. Distribution of output latency values over 10 minute run at one million events/second injection rate.

microseconds and 99.4% of the latency values below 200 microseconds at an injection rate of 1,000,000 events/second.

Finally, the garbage collection pause times were examined to understand their contribution to the overall latencies measured at the application level. 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 deterministic garbage collector to maintain these short and predictable GC pauses under load was a major factor in limiting the peak application latencies.

7. CONCLUSIONS

This paper has presented WebLogic Event Server, a next-generation (i.e. post-J2EE) application server designed specifically for event processing applications that require high throughput and deterministic latency. The paper highlighted the innovative aspects of the server design. WL EvS is based on a modular, service-oriented architecture. Server modules export classes and services for client modules to use. The modular design helps to keep the server lightweight since modules are pluggable. WL EvS provides a hybrid application programming model that supports both the Java programming language and the EPL query language. The programming model allows application developers to seamlessly mix-and-match languages. Application developers can take advantage of their existing Java skill set while also leveraging the power of a stream query language. WL EvS also supports dynamic configuration changes for application components which makes applications more flexible and manageable by business and end users.

The paper also presented the results of a benchmark performance study which validates the approach taken by WL EvS. The results show that a Java application server can deliver high throughput and deterministic performance for even the most demanding event processing applications. WL EvS delivers deterministic performance by addressing determinism throughout the software stack. This includes running on a Java virtual machine that provides deterministic garbage collection, as well as employing specialized algorithms and design principles within the application server itself.

8. ACKNOWLEDGMENTS

We would like to thank all of the members of the WL EvS team for their contributions, especially: Larry Dino, Andy Piper, James Taylor, Dana Bergen, and Mayur Shah. Thanks to Mike Carey for encouraging us to write this paper.

9. REFERENCES

[1] Babu, S. and Widom, J., "Continuous Queries Over Data Streams", SIGMOD Record, September 2001.

[2] BEA WebLogic Event Server 2.0 product documentation, BEA Systems Inc., 2007.

[3] BEA WebLogic RealTime 2.0 product documentation, BEA Systems Inc., 2007.

[4] BEA WebLogic Server 10.0 product documentation, BEA Systems Inc., 2007.

[5] Chen, J., DeWitt, D., Tian, F., and Wang, Y., NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[6] DeMichiel, L. and Keith, M., Enterprise JavaBeans,Version 3.0, Sun Microsystems, Inc., May 2006.

[7] Dependency Injection, Wikipedia, http://www.wikipedia.org, 2008.

[8] Esper 2.0 product documentation, EsperTech Inc., 2007.

[9] Gosling, J., Joy, B., Steele, G., Bracha, G., "The Java Language Specification, 3rd Edition", Prentice Hall, June 2005.

[10] IBM WebSphere Application Server 6.1 product documentation, IBM, 2007.

[11] JBoss Enterprise Application Platform 4.3 product documentation, RedHat Inc., 2007.

[12] Kawaguchi, Kohsuke, "Java Architecture for XML Binding (JAXB) 2.0 ", Sun Microsystems, Inc., 2006.

[13] Lindholm, T. and Yellin, F., "Java Virtual Machine Specification, 2nd Edition", Prentice Hall, April 1999.

[14] Oracle Application Server 10*g* product documentation, Oracle Corp., 2007.

[15] OSGi Service Platform Core Specification, Release 4, Version 4.1, OSGi Alliance, April 2007.

[16] Spring Dynamic Modules for OSGi Service Platforms product documentation, SpringSource, January 2008.

[17] Peterson, James Lyle, "Petri Net Theory and the Modeling of Systems", Prentice Hall, 1981.

[18] Shah, M., Madden, S., Franklin, M., and Hellerstein, J., Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System, ACM SIGMOD Record, 30(4), Dec, 2001.

[19] Streambase 5.1 product documentation, Streambase Systems, 2007.

[20] Tharakan, George, "Java Message Service (JMS) API", Sun Microsystems, Inc., 2003.