

# Introduction to Complex Event Processing & Data Streams

OMG's Data Distribution Service data streams when you need high performance from complex event processing systems

WRITTEN BY SUPREET OBEROI

➤ With the evolution of distributed IT systems and the advent of Web Services, applications can now make more informed decisions by using real-time information from third-party sources. For example, today's automated trading applications can make over 30 trades a second by analyzing stock trends, market movements, news, and events that may only be relevant for a fraction of a second. A trading algorithm may infer a negative stock trend line for the next tenth of a second and may short the stock for that bit of time.

Such applications require Complex Event Processing (CEP) engines. These engines can detect patterns of activity from multiple data streams and infer events continuously. Many critical CEP use cases require peak performance from both the event engine and the data streams. In the example above, the trading algorithm can't profit from making buys and sells in a tenth of the second if the event engine and the data stream latency exceed the operable time window.

This article will survey the suitability of OMG's Data Distribution

Service data streams in use cases that demand high performance from complex event processing systems.

## What is Complex Event Processing?

Consider the following example: The Securities and Exchange Commission has different margin and reserve requirements for traders classified as "pattern day traders." The term "pattern day trader" means any customer who executes four or more day trades in the same stock in five business days. However, if the number of day trades is 6% or less of his total trades for the five-business-day period, the customer won't be considered a pattern day trader and the special margin and reserve requirements won't apply.

With Complex Event Processing, we can detect a "pattern day trader" in real-time as follows. Each time a trader makes a transaction, the system posts an `executedTradeEvent` `<traderId, stockId>` event. The CEP engine maintains a window of five days and searches for cases where the count (`count_n`) of `executedTradeEvents` for a given trader and a stock exceeds four. Detecting such a pattern it counts the total number of trades done in the last five days. If `count_n` is more than 6% of all trades in the last five days then it posts a `detectedDayTrader` event.

Based on this example, we can describe many key characteristics of Complex Event Processing:

- **Events are inferred:** In the example, the trading application doesn't and can't send a `detectedDayTrader` event. This event has to be inferred by processing other events like `execut-`

edTradeEvent, maintaining a count of events satisfying a query condition over a given period of time and integrating with non-streaming content like the number of total trades stored in a relational database.

- The system correlated the data from multiple sources to infer the event. In the example, it included explicit sources like the database that stored the total number of trades for a customer and implicit sources like time.
- The system wouldn't have sent a detectedDayTrader if there wasn't a set of executedTradeEvents preceding it. The executedTradeEvent plus some other criteria caused the detectedDayTrader event to occur.

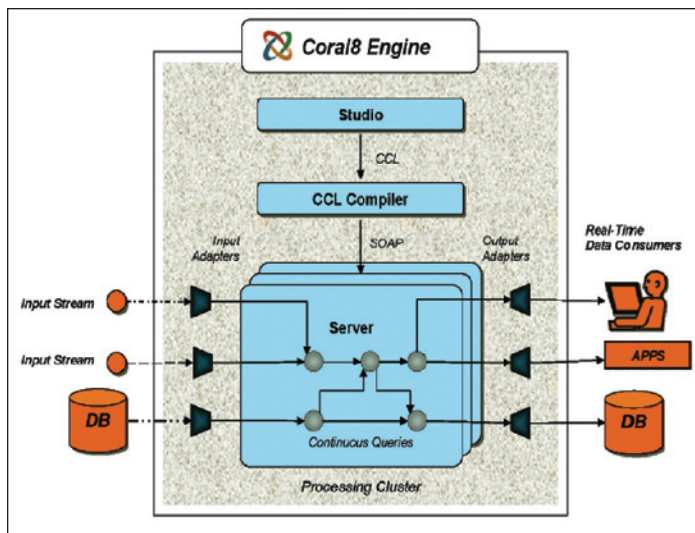


Figure 1 Architectural block diagram for a CEP engine

CEP engines manage event-driven information systems by employing techniques such as detecting complex patterns, building correlations, and relationships such as causality and timing between many events.

From a black box view, a CEP engine takes as input a set of input streams like RTI, database files, or JMS. Most CEP engines use an SQL-like programming language such as the Continuous Computation Language (CCL) with extensions for event processing.

While discussing the entire semantics of CCL or CEP is beyond the scope of this article, here's a simple example of how application developers can infer a weather event using the CCL programming language.

In this example, the query searches for events from the input data stream WindIn in which the wind speed changes by more than five miles an hour in two seconds. The events matching the pattern are then inserted into an output data stream WindPatternOut:

```
INSERT INTO WindPatternOut (Location, Speed1, Speed2)
SELECT W1.Location, W1.WindSpeed, W2.WindSpeed
FROM WindIn W1, WindIn W2
MATCHING [2 SECONDS: W1 && W2]
ON W1.Location = W2.Location
WHERE (W1.WindSpeed - W2.WindSpeed) >= 5;
```

As you can see from this example, CCL is loosely based on SQL semantics with extensions (such as matching patterns, viewing samples in a given time window) for complex events. The input and the output data streams are logically modeled as database tables, regardless of the underlying messaging protocol.

## Why Use Complex Event Processing?

From an oversimplified view, applications have implemented functionality for inferring events from existing data for a very long time. Credit and fraud-risk applications, for example, have existed for decades without an explicit design and implementation for CEP. However, the data avalanche produced by edge devices like Radio Frequency Identification (RFID) readers and sensors is rapidly changing design-detection algorithms and the need for a configurable and flexible way to detect patterns is becoming more vital. Here are some situations in which software architects might consider incorporating CEP into their application technology stack:

- Only the “processed” data is useful: In applications where edge devices such as sensors or RFID readers connect to the enterprise, all the raw data samples aren't of equal interest to the enterprise business process. The data might need to be cleansed, validated, and enriched before it's useful. In the wind example, the application isn't interested in each sensor read of the wind speed. The application is only interested when the wind speed changes by more than five miles an hour in two seconds, possibly inferring a hurricane or tornado.
- Software development cycles can't keep up with the changes in algorithms for detecting patterns: In trading applications, patterns for detecting buy and sell events may only be competitive for a few months, or even a few weeks. In some cases, new patterns are discovered, implemented, and deployed in a day. In such cases, it's necessary to parameterize and abstract out the pattern detection layer of the application. Having the trading algorithm embedded in the application code is not a good design practice.
- Event processing has to be done in real-time: Not all event processing requires a CEP engine. Data warehouse applications analyze trends by correlating multiple dimensions of the data in an “offline” manner (for example, send Home Depot discount promotions to all residents who have moved to zip code 95138 in the last two months). However, in many applications, events have to be inferred in real-time and the applications simply can't wait for the raw data to be persisted and analyzed. For example, radar tracking applications must process events in real-time. A trading desk application seeks a competitive advantage by analyzing an event microseconds ahead of its competition. Traditional event-detection applications require that the data is persisted first and then correlated. This methodology is too slow for applications such as trading desks where the data has to be analyzed for event patterns in real-time.
- Event processing has to scale: The media is now heralding the arrival of truly ubiquitous computing, where tiny microprocessors in our ambient surroundings communicate to deliver a more intelligent service. For example, in healthcare monitoring, pressure sensors in the shoes of patients at risk of a heart attack can monitor any change in their pattern of walking and alert the healthcare provider, who can immediately schedule a check-up. Enterprise

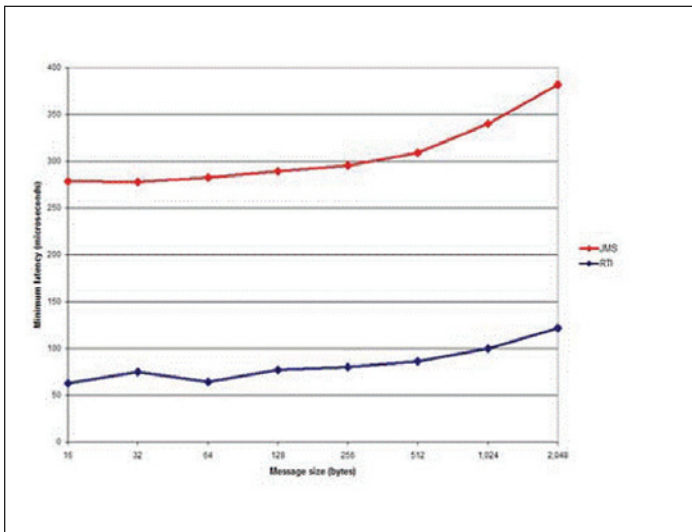


Figure 2 Comparison of latency for JMS and RTI's implementation of the Data Distribution Service

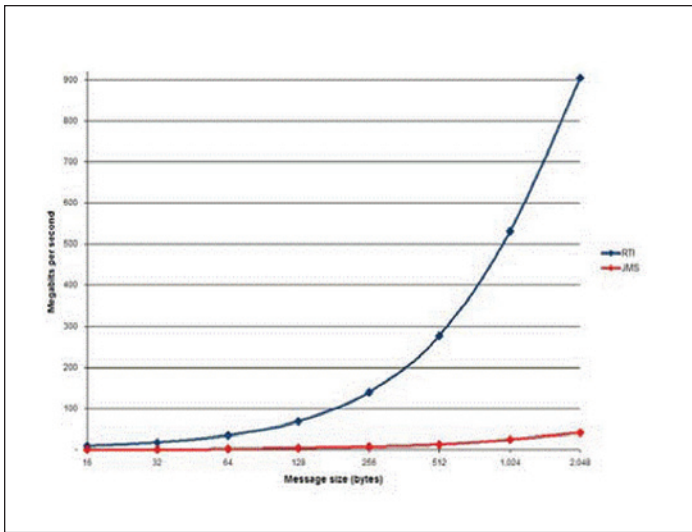


Figure 3 Network throughput comparison between JMS and RTI's implementation of the Data Distribution Service

applications that integrate with the edge have to cope with a large number of sensors simultaneously sending high rates of data. Traditional applications won't be able to handle this impedance mismatch of high data rate and volume. Using a CEP engine designed for performance, one can take steps to manage the load. Selecting the right messaging protocol for the data stream is the other step.

### How To Select the Right Messaging Protocol for Your CEP Engine

Selecting the right CEP engine is only one part of building your event processing solution. The other significant part of the architecture is selecting the right messaging bus for your data stream. With the correct selection, you can fully leverage the high-performance CEP engine, scale to a large number of nodes, and do more. Consider the use case of trading desk applications where microseconds matter in identifying a trend for buying or selling a stock. Regardless of how fast the CEP engine may be a typical JMS implementation that usually delivers messages in tens of milliseconds will be a

non-starter.

Here are some of the criteria to consider in adopting a messaging bus for your complex event processing needs:

- **Latency:** CEP isn't about speed. It's about correlating data from different data streams. However, chances are that if you're considering using CEP then latency is a significant concern for developing a successful application. In algorithmic trading, command and control, and fraud detection applications for electronic trades low latency of the entire application solution is critical. In trading desk applications, even the physical proximity of the trading floor to the exchange has become a critical issue. The extra nanosecond that a remote trade takes to register could cost the bank a deal. Many applications that rely on high-performance data streams simply can't afford the periodic (JVM) garbage collection that can degrade overall performance. Messaging products like implementations of OMG's Data Distribution Service, which have been designed and deployed for real-time mission-critical applications, can deliver event samples with a latency an order better than traditional JMS applications. In addition, while the overhead of Data Distribution Service depends on the message size and transport used. For reasonably sized messages and standard transports (100 Mbit-1 Gbit Ethernet) the overhead is typically less than 15% above the raw transport.
- **Managing network bandwidth:** Many developers who are considering CEP engines are also concerned with managing network bandwidth efficiently. In trading desk applications, the total volume of stocks traded daily has been following its own Moore's Law, doubling every 18 months since the start of electronic trading. This means the amount of data that must be processed follows the same curve (because the houses have to track all trades, not just their own). In addition, they have internal consumers of the data whose demands are expanding. New trading strategies are being developed that run in parallel with existing ones. Each model requires input and generates output.

Some messaging vendors can manage the network bandwidth more efficiently by packing more event samples into the network pipe and by sending only the relevant data over the network.

For example, with the Data Distribution Service, middleware can apply content-based filtering using SQL-like patterns so that only relevant data is sent over the network.

With pub-sub implementations of the Data Distribution Service, developers can configure the rate at which event samples have to be transmitted on a per data stream basis.

Consider the example of a market-data application that sends all stock ticks to the trading application. However, using CEP, sending a snapshot of the five-second stock high and low may not be required. By configuring a different transmission rate for the stock tick feed, and the "five-second high-low" feed, we can preserve network resources.

In addition, with intelligent network middleware for data streams, we can conserve network bandwidth by determining if the data has to be transmitted at all. For example, with the Data Distribution Service applications can subscribe to topics of interest such as stock news relating to a particular symbol or specific content within the topics of interest. If there are no subscribers for a given topic, RTI won't put the event samples on the network.

- **Quality of Service (QoS) for data streams:** QoS refers to a service contract made between two entities. Each data stream has

unique attributes or characteristics. For example, a trading desk application may require resending all lost stock ticker change events (reliable transmission). However, a security surveillance application may only require that the data stream for video transmission use best-effort (rather than reliable) techniques. The Data Distribution Service provides a rich set of QoS contracts that can be enforced out-of-the-box. Some examples of such pre-built contracts include ownership strength (selecting the right data source when multiple sources are generating the same data for failover), history (how many event samples are needed for late-joining consumers of information), reliability, time- and content-based filtering of event samples at the source, and persistence (in case the publisher dies and a late-joiner consumer of the information arrives).

- Number of messages a second: Use cases requiring Complex Event Processing typically demand that messages be transmitted at a rate of 1,000/second to 500,000/sec. This is understandable considering that the CEP application typically integrates with edge devices or, as in trading applications, process a large amount of market data from multiple exchanges to make trend inferences. In such cases high-performance middleware should be capable of intelligently compressing the messages to fit the given system's MTU. Vendors such as 29West and RTI provide performance numbers in this range. For example, with RTI the user can transmit up to 10 million four-byte messages a second.
- Supporting heterogeneous platforms: From one point-of-view, CEP engines are about integrating data streams from different sources. While different data streams can use their own messaging protocols, this poses a needless headache for application architects by having multiple technology stacks for their data stream implementations. Messaging protocols like JMS are supported on major enterprise platforms, but aren't prevalent in embedded ecosystems where edge devices reside. Traditionally, the Data Distribution Service implements a vast set of architectures including, but not limited to, enterprise platforms like Linux, Solaris, and Windows and embedded platforms like VxWorks, LynxOS, and Integrity operating systems.

## Summary

Remember the mid-90s? Some people still assembled their own PCs by purchasing the right combination of motherboards, processors, and disk. But a non-optimal combination of motherboard and processor ensured that you didn't get the right performance from your system. Similarly, while CEP engines herald the promise of delivering high-performance event detection and generation, you need to ensure that they run with data streams with compatible aims of low latency, high availability, throughput, and flexibility. ■

---

### About the Authors

Supreet Oberoi is vice-president of engineering at RTI, with over a decade of experience in building and deploying Web-based enterprise applications. He was a founding member and director of engineering for Trading Dynamics, which was acquired by Ariba in 1999. Later, he led the engineering organization for the Mohr-Davidow (MDV)-funded start-up, oneREV, Inc., that was acquired by Agile Software in 2002. Most recently, Supreet served as director of engineering at Agile Software. He received his BS in computer sciences with highest honors from the University of Texas at Austin and an MS in computer sciences from Stanford University.

[supreet.oberoi@authors.sys-con.com](mailto:supreet.oberoi@authors.sys-con.com)