Stream and Complex Event Processing
Discovering Existing Systems: esper

G. Cugola   E. Della Valle   A. Margara

Politecnico di Milano  Vrije Universiteit Amsterdam

chema@elet.polimi.it   a.margara@vu.nl
dellavalle@elet.polimi.it
Agenda

• Introduction
• Describing Events
• Event Stream Analysis
• Event Pattern Matching
• Combinations
• Resources
Introduction

Esper

• Motto
  • EsperTech’s Event Stream and Complex Event Processing software turns large volume of disparate real-time event streams into actionable intelligence.

• Esper
  • Event Processing for Java

• Nesper
  • Event Processing for .Net
Introduction

Esper Architecture
Introduction

Esper Features at a glance 1/4

• Efficient Event Processing
  • Continuous queries, filtering, aggregations, joins, sub-queries
  • Comprehensive pattern detection
  • Pull and Push
  • High performance, low latency
Introduction

Esper Features at a glance  2/4

• Extensible Middleware
  • Java, .Net, Array, Map or XML events
  • Runtime statement management
  • API or configuration driven
  • Plug-in SDK for functions, aggregations, views and pattern detection extensions
  • Adapters: CSV, JMS in/out, API, DB, Socket, HTTP
  • Runtime management, operational visibility, interoperability
  • Data distribution service for data push management and service layer
Esper Features at a glance  3/4

• Rich Web-Based User Interface
  • Real-time event displays: Eventlet technology allows customizable and interactive continuous displays
  • CEP engine management
  • Design EPL Statements
  • Drill-down and browser script
  • integration
Introduction

Esper Features at a glance  4/4

• HA enabled (EsperHA)
  • Per statement configuration
  • Transient combinable with fully resilient behaviour
  • Hot standby API, hot backup
  • Highly optimized and fast data storage technology
  • Engine state RDBMS storage option
Introduction

Event Stream and Complex Event Processing

• Design continuous queries and complex causality relationships between disparate event streams with an expressive Event Processing Language (EPL).

• EPL statements are registered into (N)Esper and continuously executed as live data streams are pushed through.
Introduction

Rapid development and deployments

• EPL has a "SQL look alike”
• EPL statement matches trigger plain Java or .Net/C# objects for real-time customized actionable intelligence.
• (N)Esper is pure Java/.Net and can run standalone or embedded into existing middleware systems (application servers, services bus, in- house systems).
Running Example

• Count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes

• Events
  • Smoke Event: String sensor, boolean state
  • Temperature Event: String sensor, double temperature
  • Fire Event: String sensor, boolean smoke, double temperature

• Condition:
  • Fire: at the same sensor smoke followed by temperature>50
Introduction

Query Processing Model in Esper 1/2

• The Esper processing model is continuous

• Four abstractions
  • Sources
    – Push based
    – Data tuples from sensors, trace files, etc.
  • Registered EPL Queries
    – Push Based
    – Continuously executed against the events produced by the sources
  • Listeners
    – Receive data tuples from queries
    – Push data tuples to other queries
  • Subscribers
    – Receive processed data tuples
Introduction

Query Processing Model in Esper 2/2

- Sources, queries, listeners and subscribers are manually connected to form graphs
  - Sources act as input
  - Subscribers act as output
  - EPL Queries integrate sources
  - Listeners propagates query results (they act internal sources)
Introduction

Graph for the running example

At the same sensor
Smoke=true
Followed by
Temperature>50

Temperature
Smoke

Fire

Count(*) in
(10 min)

subscriber
Describing events

• Possible methods:
  • Java classes, Maps, XML, EPL
  • Java classes are a simple, rich and versatile way to represent events in Esper.
  • Follow JavaBeans-style getter methods and property names

<table>
<thead>
<tr>
<th>Method</th>
<th>Property Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>getQ()</td>
<td>q</td>
</tr>
<tr>
<td>getQN()</td>
<td>qn</td>
</tr>
</tbody>
</table>
Describing events

Temperature event for the running example

```java
class TemperatureSensorEvent
{
    String sensor;
    double temperature;
    long timeStamp;

    TemperatureSensorEvent(String sensor, double temperature, long timeStamp)
    {
        this.sensor = sensor;
        this.temperature = temperature;
        this.timeStamp = timeStamp;
    }

    String getSensor()
    {
        return sensor;
    }

    double getTemperature()
    {
        return temperature;
    }

    Date getTimeStamp()
    {
        return new Date(timeStamp);
    }
}
```

- Note: in this and in the following examples the `timeStamp` property is not necessary
Describing events

Smoke event for the running example

```java
SmokeSensorEvent
  △ sensor
  △ smoke
  △ timeStamps
  SmokeSensorEvent(String, boolean, long)
  getSensor() : String
  getSmoke() : boolean
  getTimeStamp() : Date
```
Describing events

Fire event for the running example

```java
public class FireComplexEvent {
    public FireComplexEvent(String sensor, boolean smoke, double temperature) {
        // constructor implementation
    }
    public FireComplexEvent(String sensor, boolean smoke, double temperature, long timeStamp) {
        // constructor implementation
    }
    public String getSensor() {
        return sensor;
    }
    public boolean getSmoke() {
        return smoke;
    }
    public double getTemperature() {
        return temperature;
    }
    public Date getTimeStamp() {
        return timeStamp;
    }
}
```
Describing events

Declaring an event type via the create schema clause

• EPL allows declaring an event type via the *create schema clause* and also by means of the static or runtime configuration API *addEventType* functions.

• Syntax
  
  • `create schema schema_name [as] (property_name property_type [,property_name property_type [,...]]) [inherits inherited_event_type [, inherited_event_type] [,,...]]`

• Example
  
  • `create schema FireComplexEvent (sensor string, smoke boolean, temperature double);`
Event Stream Analysis

Event Processing Language (EPL)

• EPL statements
  • derive and aggregate information from one or more streams of events,
  • to join or merge event streams, and
  • to feed results from one event stream to subsequent statements.
Event Stream Analysis

Event Processing Language (EPL)

• EPL is similar to SQL in its use of the select clause and the where clause.
• EPL statements instead of tables use event streams and a concept called views.
• Views are similar to tables in an SQL statement
  • They define the data available for querying and filtering.
  • They can represent windows over a stream of events.
  • They can also sort events, derive statistics from event properties, group events or handle unique event property values.
EPL Syntax

[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output outputSpecification]
[order by order_by_expression_list]
[limit num_rows]
Event Stream Analysis

Simple examples

• Look for specific events
  • select * from SensorEventStream
    where temperature>50

• Aggregate several events
  • select avg(temperature) from SensorEventStream

• Joining two streams
  • select Tstream.sensor, Tstream.temperature,
    Sstream.smoke
  from TemperatureEventStream as Tstream,
  SmokeEventStream as Sstream
  where Tstream.sensor = Sstream.sensor and
    Tstream.temperature>50 and
    Sstream.smoke=true
Event Stream Analysis

The EPL alone is not enough ...

Configuration cepConfig = new Configuration();
cepConfig.addEventType("TemperatureEventStream",
    TemperatureSensorEvent.class.getName());
cepConfig.addEventType("SmokeEventStream",
    SmokeSensorEvent.class.getName());
String query = "<<any of the three in the previous slide>>";
EPServiceProvider cep =
    EPServiceProviderManager.getProvider("myCEP", cepConfig);
EPRuntime cepRT = cep.getEPRuntime();
EPAdministrator cepAdm = cep.getEPAdministrator();
EPStatement cepStatement = cepAdm.createEPL(query);
cepStatement.addListener(new CEPListener());

See also HelloWorldEsper in the Esper ready to go pack on the course Website
Event Stream Analysis

Listening to EPL query results 1/3

The interface for listeners is com.espertech.esper.client.UpdateListener. Implementations must provide a single update method that the engine invokes when results become available.

```
update(EventBean[] newEvents, EventBean[] oldEvents)
```
Listening to EPL query results 2/3

• The engine provides statement results to update listeners by placing results in `com.espertech.esper.client.EventBean` instances. A typical listener implementation queries the `EventBean` instances via getter methods to obtain the statement-generated results.
Event Stream Analysis

Listening to EPL query results 3/3

For instance, the following code prints each new event received:

```java
public class CEPLListener implements UpdateListener {
    public void update(EventBean[] newData, EventBean[] oldData) {
        for (EventBean e : newData) {
            System.out.println("Event received: " + e.getUnderlying());
        }
    }
}
```

NOTE: similar code can be used to access the events that are exiting the window (oldData), see also next slides.
## Windows

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Sliding</td>
<td><code>win:time(time_period)</code></td>
<td>Sliding time window extending the specified time interval into the past.</td>
</tr>
<tr>
<td>Logical Tumbling</td>
<td><code>win:time_batch(time_period[,optional reference point] [, flow control])</code></td>
<td>Tumbling window that batches events and releases them every specified time interval, with flow control options.</td>
</tr>
<tr>
<td>Physical Sliding</td>
<td><code>win:length(size)</code></td>
<td>Sliding length window extending the specified number of elements into the past.</td>
</tr>
<tr>
<td>Physical Tumbling</td>
<td><code>win:length_batch(size)</code></td>
<td>Tumbling window that batches events and releases them when a given minimum number of events has been collected.</td>
</tr>
</tbody>
</table>
Event Stream Analysis

Logical Sliding windows

Stream & Complex Event Processing - Introduction
Event Stream Analysis

Logical Sliding windows: example

• Query

    select avg(temperature)
    from TemperatureEventStream.win:time(4 sec)

• Execution trace

Sending Event:[S1|24.0|Mon Apr 29 13:05:44 CEST 2013]
Event received: {avg(temperature)=24.0}
Sending Event:[S1|55.0|Mon Apr 29 13:05:45 CEST 2013]
Event received: {avg(temperature)=39.5}
Sending Event:[S0|1.0|Mon Apr 29 13:05:46 CEST 2013]
Event received: {avg(temperature)=26.666666666666668}
Sending Event:[S1|81.0|Mon Apr 29 13:05:47 CEST 2013]
Event received: {avg(temperature)=40.25}
Event received: {avg(temperature)=45.666666666666664}
Sending Event:[S0|20.0|Mon Apr 29 13:05:48 CEST 2013]
Event received: {avg(temperature)=39.25}
Event received: {avg(temperature)=34.0}
Sending Event:[S0|915.0|Mon Apr 29 13:05:49 CEST 2013]
Event received: {avg(temperature)=254.25}
Event received: {avg(temperature)=338.6666666666667}
Event received: {avg(temperature)=467.5}
Event received: {avg(temperature)=915.0}
Event received: {avg(temperature)=null}
Sending Event:[S0|30.0|Mon Apr 29 13:05:54 CEST 2013]
Event received: {avg(temperature)=30.0}

Esper, when using logical sliding windows, reports as soon as a new event arrives and an old one expires
Event Stream Analysis

Logical Tumbling windows
Event Stream Analysis

Logical Tumbling windows: example

• Query

```
select avg(temperature)
from TemperatureEventStream.win:time_batch(4 sec)
```

• Execution trace

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[S1</td>
<td>42.0</td>
<td>29]</td>
<td>[S1</td>
</tr>
<tr>
<td>[S1</td>
<td>25.0</td>
<td>29]</td>
<td>[S1</td>
</tr>
<tr>
<td>Event received: {avg(temperature)=33.0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[S0</td>
<td>23.0</td>
<td>29]</td>
<td>[S1</td>
</tr>
<tr>
<td>Event received: {avg(temperature)=149.5}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[S0</td>
<td>76.0</td>
<td>29]</td>
<td>[S0</td>
</tr>
<tr>
<td>Event received: {avg(temperature)=48.0}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Esper, when using logical tumbling windows, reports only when the window closes.
Event Stream Analysis

Physical sliding windows

Stream & Complex Event Processing - Introduction
Event Stream Analysis

Physical sliding windows: example

- Query
  
  select avg(temperature)
  from TemperatureEventStream.win:length(5)

- Execution trace

  Sending Event:[S1|6.0|Mon Apr 29 13:35:50 CEST 2013]
  Event received: {avg(temperature)=6.0}
  Sending Event:[S0|48.0|Mon Apr 29 13:35:51 CEST 2013]
  Event received: {avg(temperature)=27.0}
  Sending Event:[S0|23.0|Mon Apr 29 13:35:52 CEST 2013]
  Event received: {avg(temperature)=25.666666666666668}
  Sending Event:[S1|89.0|Mon Apr 29 13:35:53 CEST 2013]
  Event received: {avg(temperature)=41.5}
  Sending Event:[S0|54.0|Mon Apr 29 13:35:54 CEST 2013]
  Event received: {avg(temperature)=44.0}
  Sending Event:[S0|877.0|Mon Apr 29 13:35:55 CEST 2013]
  Event received: {avg(temperature)=218.2}

  Sending Event:[S1|42.0|Mon Apr 29 13:36:00 CEST 2013]
  Event received: {avg(temperature)=217.0}
  Sending Event:[S1|7.0|Mon Apr 29 13:36:01 CEST 2013]
  Event received: {avg(temperature)=213.8}
  Sending Event:[S0|23.0|Mon Apr 29 13:36:02 CEST 2013]
  Event received: {avg(temperature)=200.6}
  Sending Event:[S0|10.0|Mon Apr 29 13:36:03 CEST 2013]
  Event received: {avg(temperature)=191.8}

Esper, when using physical sliding windows, reports as soon as a new event arrives
Event Stream Analysis

Physical Tumbling windows: example

• Query

\[
\text{select } \text{avg(temperature)} \\
\text{from TemperatureEventStream.win:length_batch(5)}
\]

• Execution trace

Sending Event:[S1|66.0|Mon Apr 29 13:40:06 CEST 2013]
Sending Event:[S0|42.0|Mon Apr 29 13:40:07 CEST 2013]
Sending Event:[S1|51.0|Mon Apr 29 13:40:08 CEST 2013]
Sending Event:[S0|10.0|Mon Apr 29 13:40:09 CEST 2013]
Sending Event:[S1|61.0|Mon Apr 29 13:40:10 CEST 2013]
Event received: \{avg(temperature)=46.0\}
Sending Event:[S0|621.0|Mon Apr 29 13:40:11 CEST 2013]
Sending Event:[S0|40.0|Mon Apr 29 13:40:16 CEST 2013]
Sending Event:[S0|84.0|Mon Apr 29 13:40:17 CEST 2013]
Sending Event:[S1|21.0|Mon Apr 29 13:40:18 CEST 2013]
Sending Event:[S0|43.0|Mon Apr 29 13:40:19 CEST 2013]
Event received: \{avg(temperature)=161.8\}

Esper, when using physical tumbling windows, reports only when the window closes
Controlling Reporting

• The *output* clause is optional in Esper
• It is used
  • To control the rate at which events are output
  • to suppress output events.
• Syntax
  • Output [[all | first | last | snapshot] every *output_rate* [seconds | events]]
Controlling Reporting: examples

• Controlling the sliding in logical and physical windows
  • select avg(temperature)
    from TemperatureEventStream.win:time(4 sec)
    output snapshot every 2 sec
  • select avg(temperature)
    from TemperatureEventStream.win:length(4)
    output snapshot every 2 events
Event Pattern Matching

• Event patterns match when an event or multiple events occur that match the pattern's definition.
• Patterns can also be temporal (time-based).
• Pattern matching is implemented via state machines.
Pattern atoms

• Filter expressions specify an event to look for.
  • TemperatureEventStream(sensor="S0", temperature>50)

• Time-based event observers specify time intervals or time schedules.
  • timer:interval(10 seconds)
  • timer:at(5, *, *, *, *)

Every 5 minutes
Event Pattern Matching

Types of operators

• Operators that control pattern finder creation and termination: every, every-distinct, [num] and until
• Logical operators: and, or, not
• Temporal operators that operate on event order: -> (followed-by)
• Guards are where-conditions that filter out events and cause termination of the pattern finder, such as timer:within, timer:withinmax and while-expression
• Note: Pattern expressions can be nested arbitrarily deep by including the nested expression(s) in () round parenthesis.
Event Pattern Matching

Pattern example

• Query

    select a.sensor
    from pattern [every ( a = SmokeEventStream(smoke=true) -> TemperatureEventStream(temperature>50, sensor=a.sensor) where timer:within(2 sec) ) ]

• Execution trace

Sending Event:[S0|false|Mon Apr 29 14:33:54 CEST 2013]
Sending Event:[S1|28.0|Mon Apr 29 14:33:55 CEST 2013]
Sending Event:[S1|true|Mon Apr 29 14:33:56 CEST 2013]
Sending Event:[S1|43.0|Mon Apr 29 14:33:57 CEST 2013]
Sending Event:[S0|true|Mon Apr 29 14:33:58 CEST 2013]
Sending Event:[S0|74.0|Mon Apr 29 14:33:59 CEST 2013]
Event received: {a.sensor=S0}
Sending Event:[S0|true|Mon Apr 29 14:34:00 CEST 2013]
Sending Event:[S0|70.0|Mon Apr 29 14:34:01 CEST 2013]
Event received: {a.sensor=S0}
Event Pattern Matching

Pattern operators: every

• The *every* operator indicates that the pattern sub-expression should restart when the sub-expression qualified by the every keyword evaluates to true or false.

• Without the every operator the pattern sub-expression stops when the pattern sub-expression evaluates to true or false

• Every time a pattern sub-expression within an every operator turns true the engine starts a new active sub-expression looking for more event(s) or timing conditions that match the pattern sub-expression.
Event Pattern Matching

Pattern operators: every

- This pattern fires when encountering an A event and then stops looking:
  - A

- This pattern keeps firing when encountering A events, and doesn't stop looking:
  - every A
Event Pattern Matching

Pattern operators: every ( A -> B )

• Events
  - A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4

• Pattern
  - every ( A -> B )

• Results
  - Detect an A event followed by a B event. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for the next A event.
    1. Matches on B_1 for combination \{A_1 , B_1\}
    2. Matches on B_3 for combination \{A_2 , B_3\}
    3. Matches on B_4 for combination \{A_4 , B_4\}
Event Pattern Matching

Pattern operators: every A -> B

• Events
  • $A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4$

• Pattern
  • every $A \rightarrow B$

• Results
  • The pattern fires for every $A$ event followed by a $B$ event.
    1. Matches on $B_1$ for combination $\{A_1, B_1\}$
    2. Matches on $B_3$ for combination $\{A_2, B_3\}$ and $\{A_3, B_3\}$
    3. Matches on $B_4$ for combination $\{A_4, B_4\}$
Event Pattern Matching

Pattern operators: A -> every B

• Events
  • \(A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4\)

• Pattern
  • A -> every B

• Results
  • The pattern fires for an A event followed by every B event.
    1. Matches on \(B_1\) for combination \(\{A_1, B_1\}\)
    2. Matches on \(B_2\) for combination \(\{A_1, B_2\}\)
    3. Matches on \(B_3\) for combination \(\{A_1, B_3\}\)
    4. Matches on \(B_4\) for combination \(\{A_1, B_4\}\)
Event Pattern Matching

Pattern operators: every A -> every B

- Events
  - $A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4$
- Pattern
  - every A -> every B
- Results
  - The pattern fires for every A event followed by every B event.
    1. Matches on $B_1$ for combination $\{A_1, B_1\}$
    2. Matches on $B_2$ for combination $\{A_1, B_2\}$
    3. Matches on $B_3$ for combination $\{A_1, B_3\}$, $\{A_2, B_3\}$ and $\{A_3, B_3\}$
    4. Matches on $B_4$ for combination $\{A_1, B_4\}$, $\{A_2, B_4\}$, $\{A_3, B_4\}$ and $\{A_4, B_4\}$
Event Pattern Matching

Limiting sub-expression lifetime 1/3

• As the introduction of the every operator states, the operator starts new sub-expression instances and can cause multiple matches to occur for a single arriving event.

• New sub-expressions also take a very small amount of system resources and thereby your application should carefully consider when sub-expressions must end when designing patterns. Use the \texttt{timer:within} construct and the \texttt{and} \texttt{not} constructs to end active sub-expressions.

• Note: the data window onto a pattern stream does not serve to limit pattern sub-expression lifetime.
Event Pattern Matching

Limiting sub-expression lifetime 2/3

• Events
  • $A_1 A_2 B_1$
• Pattern
  • every $A \rightarrow B$
• Results
  • $\{A_1, B_1\}$ and $\{A_2, B_1\}$

• Events
  • $A_1 A_2 B_1$
• Pattern
  • every $A \rightarrow (B \text{ and not } A)$
• Results
  • $\{A_2, B_1\}$
  • The *and not* operators cause the sub-expression looking for $\{A_1, B?\}$ to end when $A_2$ arrives.
Event Pattern Matching

Limiting sub-expression lifetime 3/3

• Events
  • $A_1$ received at $t_o + 1$ sec
  • $A_2$ received at $t_o + 3$ sec
  • $B_1$ received at $t_o + 4$ sec

• Pattern
  • every $A \rightarrow B$

• Results
  • $\{A_1, B_1\}$ and $\{A_2, B_1\}$

• Events
  • $A_1$ received at $t_o + 1$ sec
  • $A_2$ received at $t_o + 2$ sec
  • $B_1$ received at $t_o + 3$ sec

• Pattern
  • every $A \rightarrow (B \text{ where timer:within(2 sec)})$

• Results
  • $\{A_2, B_1\}$
  • The $where \text{ timer:within}$ operators cause the sub-expression looking for $\{A_1, B\}$ to end after 2 seconds.
Combining Event Pattern Matching and Stream Analysis

Example

• Query

select count(a.sensor)
from pattern [every ( a = SmokeEventStream(smoke=true) -> TemperatureEventStream(temperature>50, sensor=a.sensor) where timer:within(4 sec) )].win:time(10 sec)

• Execution trace

Sending Event:[S0|true|Mon Apr 29 15:18:10 CEST 2013]
Sending Event:[S0|64.0|Mon Apr 29 15:18:11 CEST 2013]
Event received: {count(*)=1}
Sending Event:[S1|true|Mon Apr 29 15:18:12 CEST 2013]
Sending Event:[S1|63.0|Mon Apr 29 15:18:13 CEST 2013]
Event received: {count(*)=2}
Event Pattern Matching and Stream Analysis in a graph Example

• The *insert into* clause forwards events to other streams for further downstream processing.

• Query

  ```java
  cepConfig.addEventType("FireStream", FireComplexEvent.class.getName());
  insert into FireStream
  select a.sensor as sensor, a.smoke as smoke, b.temperature as temperature
  from pattern [every ( a = SmokeEventStream(smoke=true) -> b = TemperatureEventStream(temperature>5, sensor=a.sensor) where timer:within(2 sec) ) ]
  ```

• Downstream query

  ```sql
  select count(*) from FireStream.win:time(10 sec)
  ```
Event Pattern Matching and Stream Analysis in a graph

Example: execution trace

Sending Event:[S0|false|Mon Apr 29 15:30:23 CEST 2013]
Sending Event:[S0|52.0|Mon Apr 29 15:30:24 CEST 2013]
Sending Event:[S1|true|Mon Apr 29 15:30:25 CEST 2013]
Sending Event:[S0|65.0|Mon Apr 29 15:30:26 CEST 2013]
Sending Event:[S0|true|Mon Apr 29 15:30:27 CEST 2013]
Sending Event:[S0|65.0|Mon Apr 29 15:30:28 CEST 2013]
Event received: Fire:[S0|true|65.0|Mon Apr 29 15:30:28 CEST 2013]
Event received: {count(*)=1}
Sending Event:[S0|true|Mon Apr 29 15:30:29 CEST 2013]
Sending Event:[S0|71.0|Mon Apr 29 15:30:30 CEST 2013]
Event received: Fire:[S0|true|71.0|Mon Apr 29 15:30:30 CEST 2013]
Event received: {count(*)=2}
Sending Event:[S0|true|Mon Apr 29 15:30:31 CEST 2013]
Sending Event:[S1|93.0|Mon Apr 29 15:30:32 CEST 2013]
Sending Event:[S1|true|Mon Apr 29 15:30:33 CEST 2013]
Sending Event:[S1|761.0|Mon Apr 29 15:30:34 CEST 2013]
Event received: Fire:[S1|true|761.0|Mon Apr 29 15:30:34 CEST 2013]
Event received: {count(*)=3}
Event received: {count(*)=2}
Event received: {count(*)=1}
Resources

• Download Esper (for Java)
  • http://esper.codehaus.org/esper/download/download.html
• Download Nesper (for .net)
  • http://esper.codehaus.org/nesper/download/download.html
• Quick start
  • http://esper.codehaus.org/tutorials/tutorial/quickstart.htm
• Tutorial
  • http://esper.codehaus.org/tutorials/tutorial/tutorial.html
• Questions on EPL
  • http://esper.codehaus.org/tutorials/solution_patterns/solution_patterns.html
• Documentation
  • http://esper.codehaus.org/esper/documentation/documentation.html
• A not-trivial example: DEBS 2011 Challenge
  • http://esper.codehaus.org/tutorials/tutorial/debs2011_challenge.html
Acknowledges

• Large part of the content of are taken from
  • EsperTech: “Event Stream Intelligence Continuous Event Processing for the Right Time Enterprise Products Data Sheet”
  • EsperTech: "Reference Documentation Version: 4.2.0"