deep se
dependable evolvable pervasive software engineering group

StreamReasoning
Reasoning Upon Rapidly Changing Information

Università della Svizzera italiana

Stream and Complex Event Processing

Discovering Existing Systems: esper

G. Cugola E. Della Valle

A. Margara

Politecnico di Milano
cugola@elet.polimi.it
dellavalle@elet.polimi.it

Università della Svizzera italiana
alessandro.margara@usi.ch

Agenda

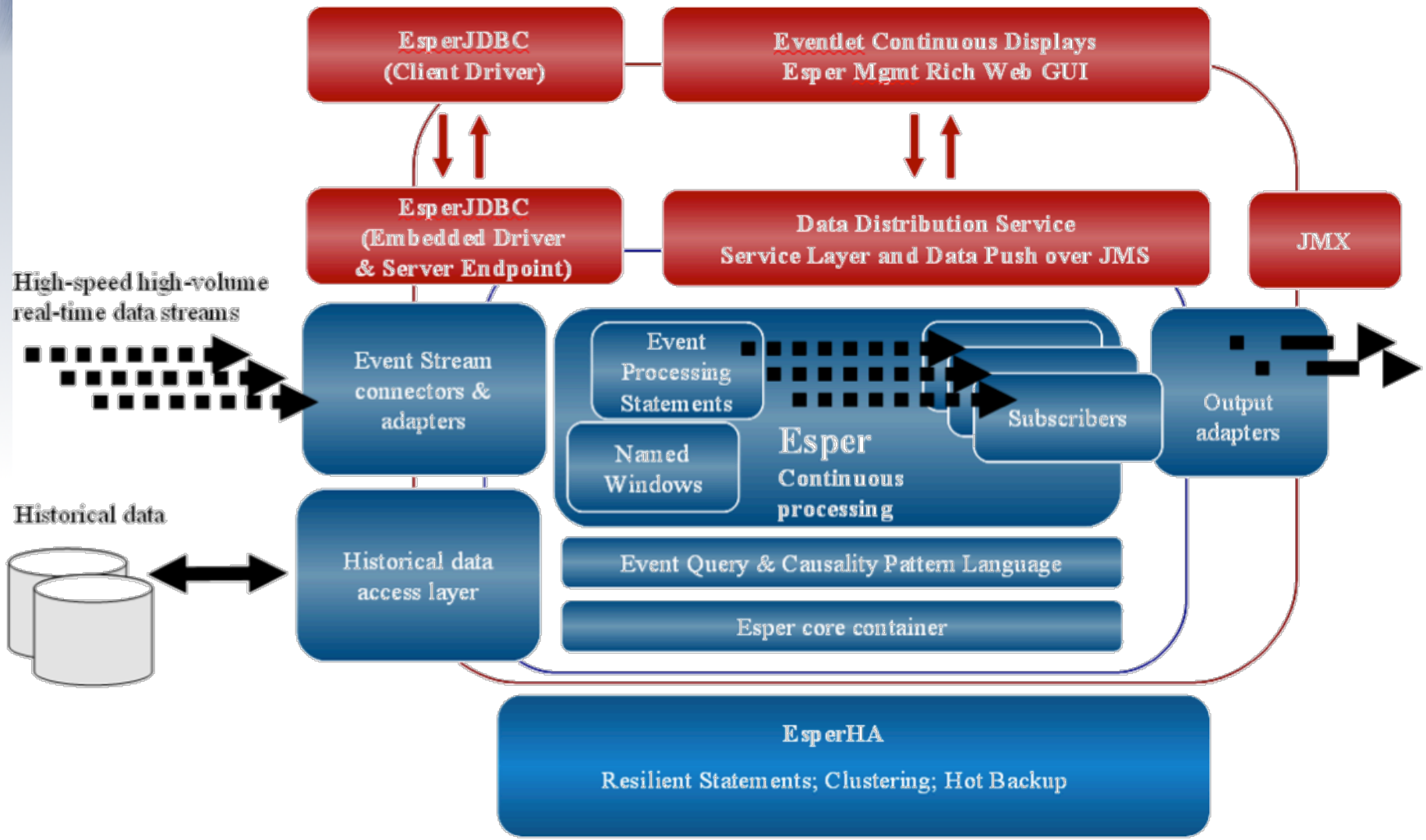
- Introduction
- Describing Events
- Event Stream Analysis
- Event Pattern Matching
- Combinations
- Resources

Esper

- Motto
  - 

'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

Esper Architecture



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

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

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

Event Stream and Complex Event Processing

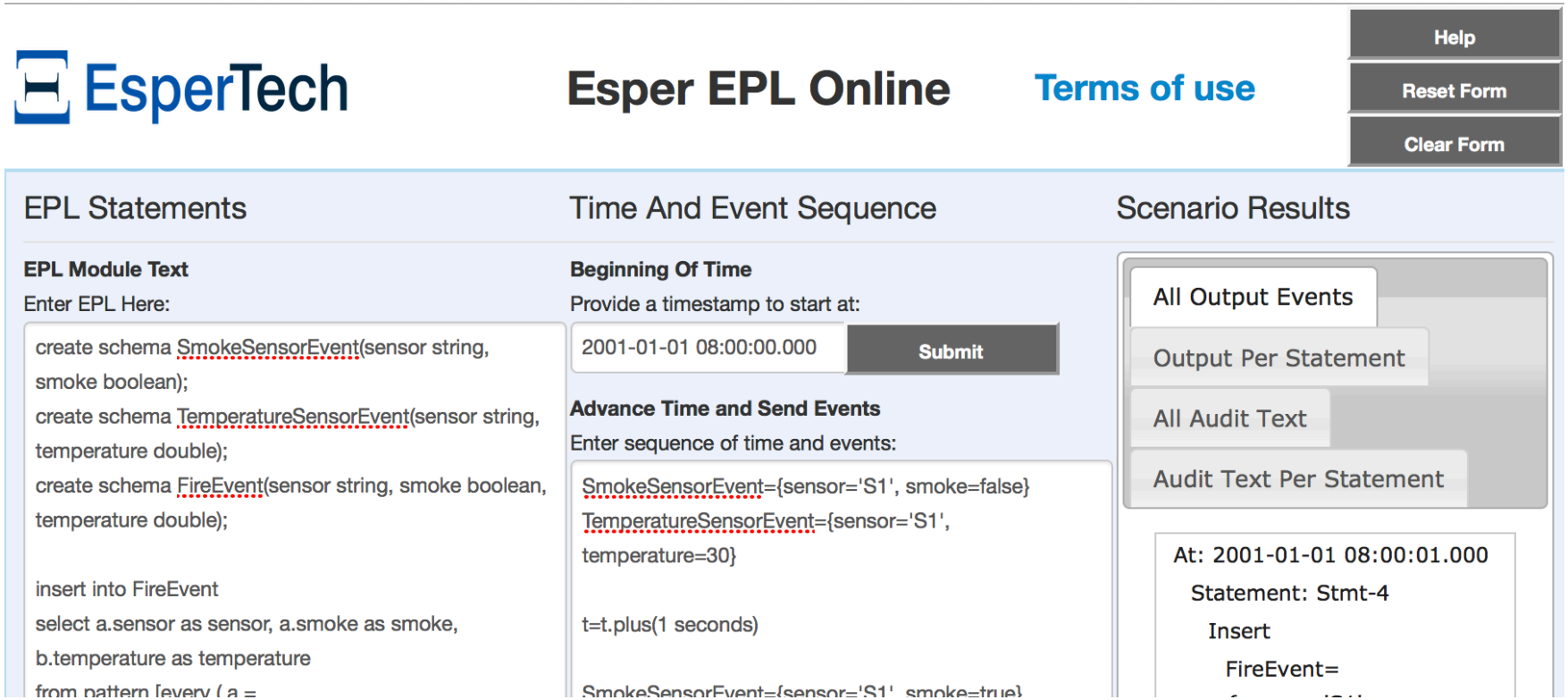
- Design continuous queries and complex causality relationships between disparate event streams with an expressive Event Processing Language (EPL).
- EPL statements are registered into (N)Esper and continuously executed as live data streams are pushed through.

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).

Trying out esper online

http://esper-epl-tryout.appspot.com/epltryout/mainform.html



Running Example

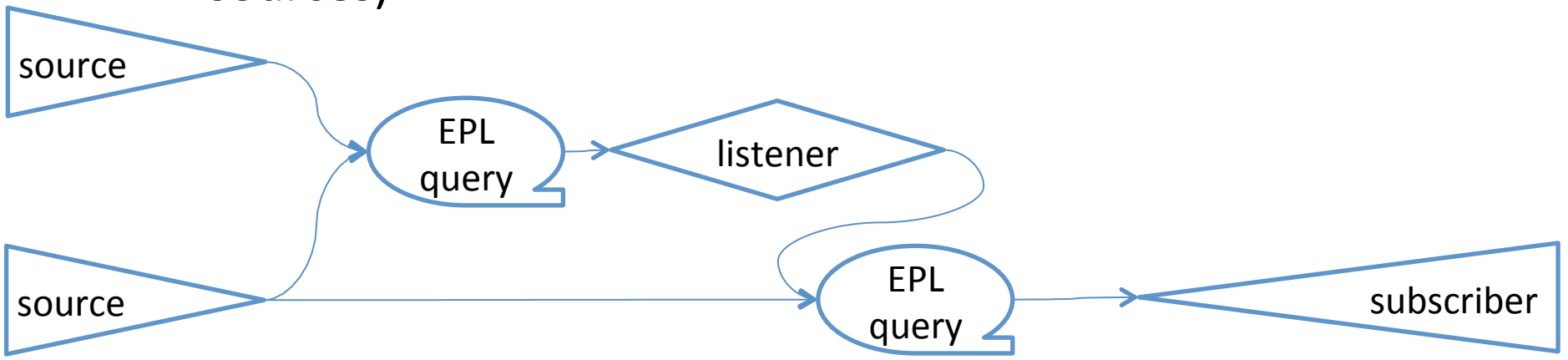
- Count the number of fires detected using a set of smoke and temperature sensors in the last 10 minutes
- Events
  - Smoke Event: String sensor, boolean state
  - Temperature Event: String sensor, double temperature
  - Fire Event: String sensor, boolean smoke, double temperature
- Condition:
  - Fire: at the same sensor smoke followed by temperature>50

Query Processing Model in Esper 1/2

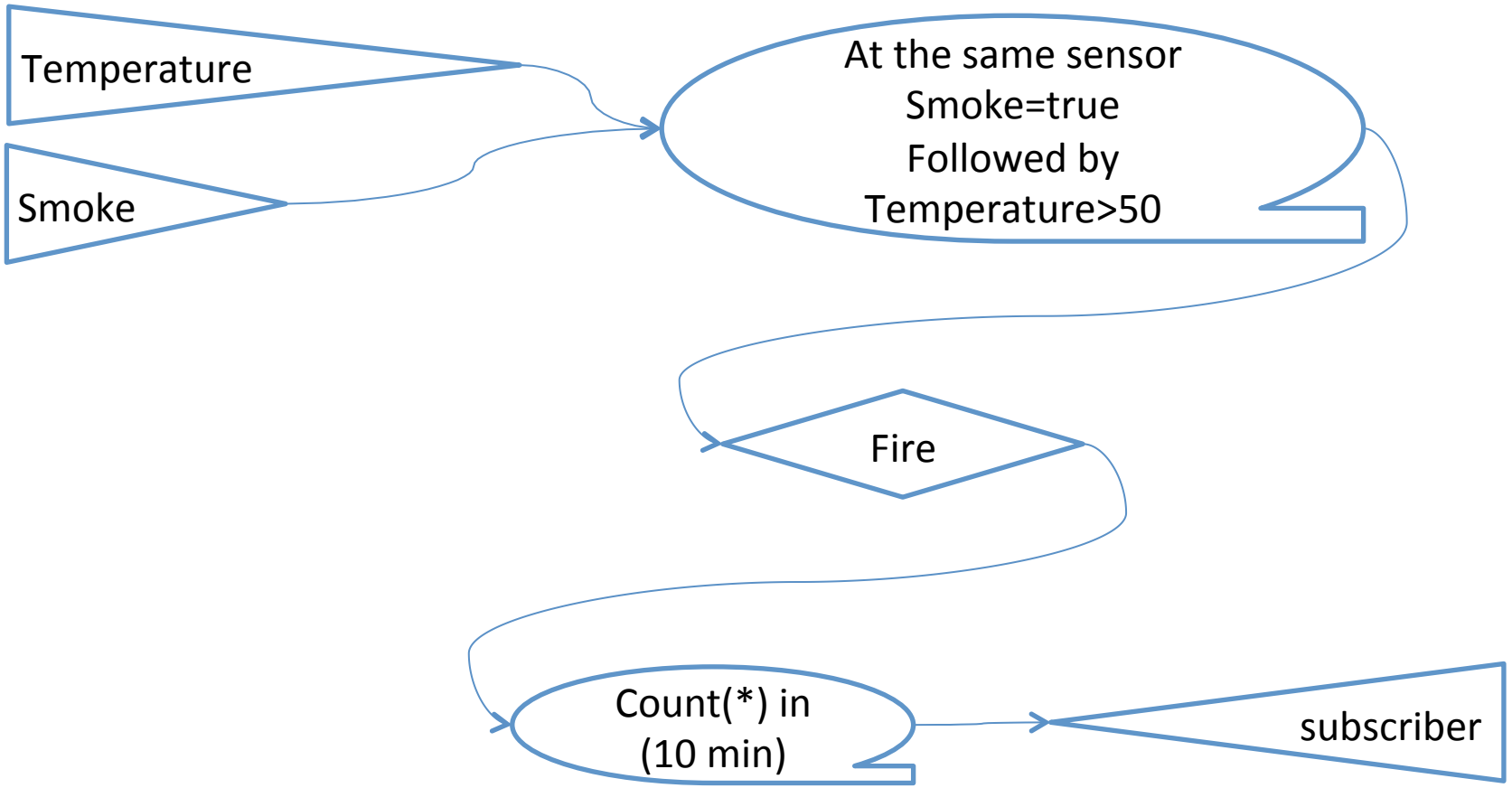
- The Esper processing model is continuous
- Four abstractions
  - Sources
    - Push based
    - Data tuples from sensors, trace files, etc.
  - Registered EPL Queries
    - Push Based
    - Continuously executed against the events produced by the sources
  - Listeners
    - Receive data tuples from queries
    - Push data tuples to other queries
  - Subscribers
    - Receive processed data tuples

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)



Graph for the running example



Describing events

- Possible methods:
  - Java classes
  - Maps
  - XML
  - EPL

Declaring an event type via EPL create schema

- 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*] [,...]]

Describing events

Temperature event for the running example

```
create schema
  SmokeSensorEvent(
    sensor string,
    smoke boolean
  );
```

Describing events

Smoke event for the running example

```
create schema
  TemperatureSensorEvent(
    sensor string,
    temperature double
  );
```

Fire event for the running example

```
create schema
  FireEvent(
    sensor string,
    smoke boolean,
    temperature double
  );
```

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 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 *output_specification*]
[order by *order_by_expression_list*]
[limit *num_rows*]

Simple examples

- Look for specific events
  - select *
    from TemperatureSensorEvent
    where temperature>50
- Aggregate several events
  - select avg(temperature)
    from TemperatureSensorEvent

Try them out with this sequence of events

SmokeSensorEvent={sensor='S1', smoke=false}

TemperatureSensorEvent={sensor='S1', temperature=30}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S1', smoke=true}

TemperatureSensorEvent={sensor='S1', temperature=40}

t=t.plus(1 seconds)

SmokeSensorEvent={sensor='S2', smoke=false}

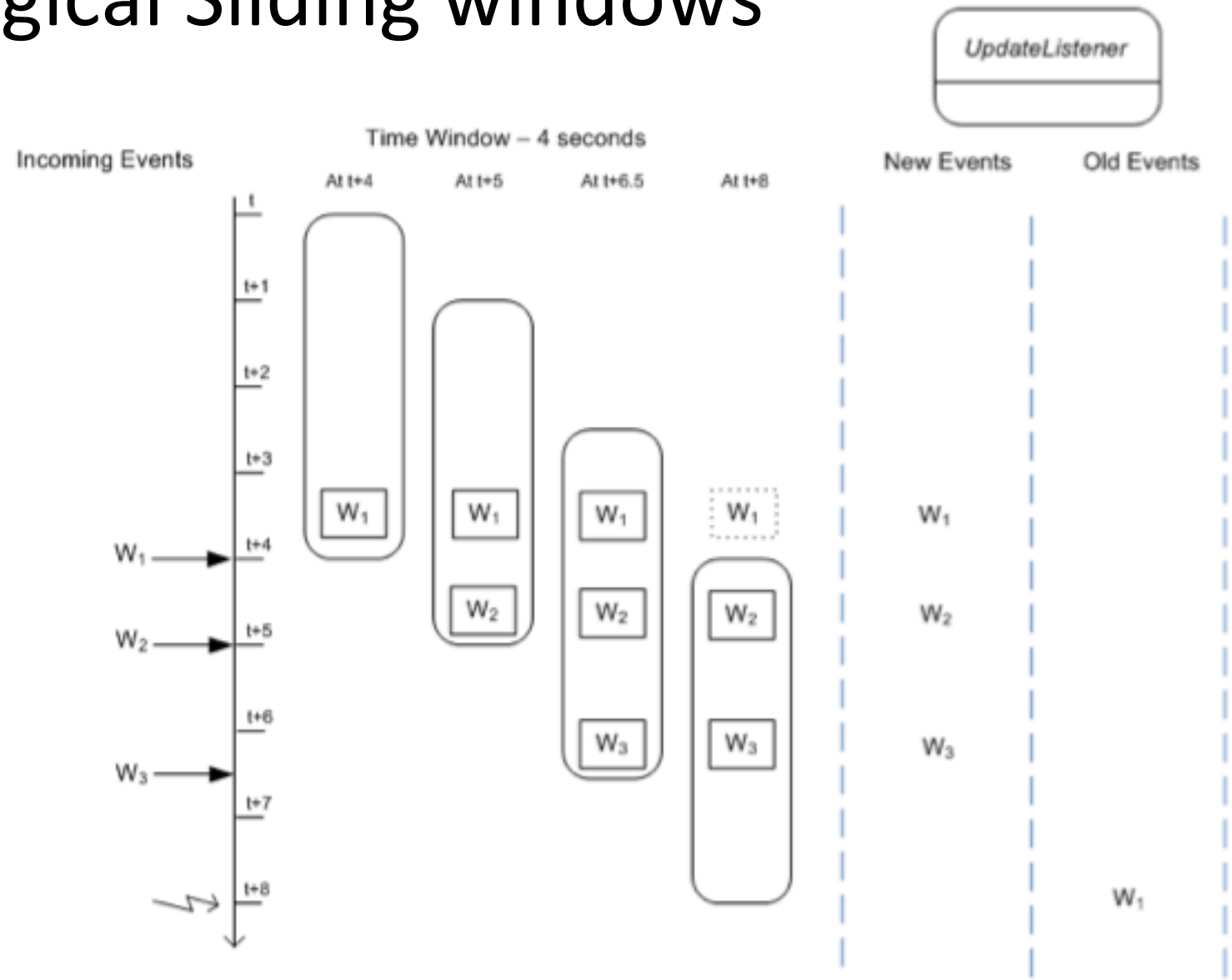
TemperatureSensorEvent={sensor='S1', temperature=55}

Advancing time manually is only required by the online interface, normally time pass by on its own :-)

Windows

Type	Syntax	Description
Logical Sliding	win:time(*time period*)	Sliding time window extending the specified time interval into the past.
Logical Tumbling	win:time_batch(*time period*[,*optional reference point*] [, *flow control*])	Tumbling window that batches events and releases them every specified time interval, with flow control options.
Physical Sliding	win:length(*size*)	Sliding length window extending the specified number of elements into the past.
Physical Tumbling	win:length_batch(*size*)	Tumbling window that batches events and releases them when a given min- imum number of events has been col- lected.

Logical Sliding windows



Logical Sliding windows: example

- Query

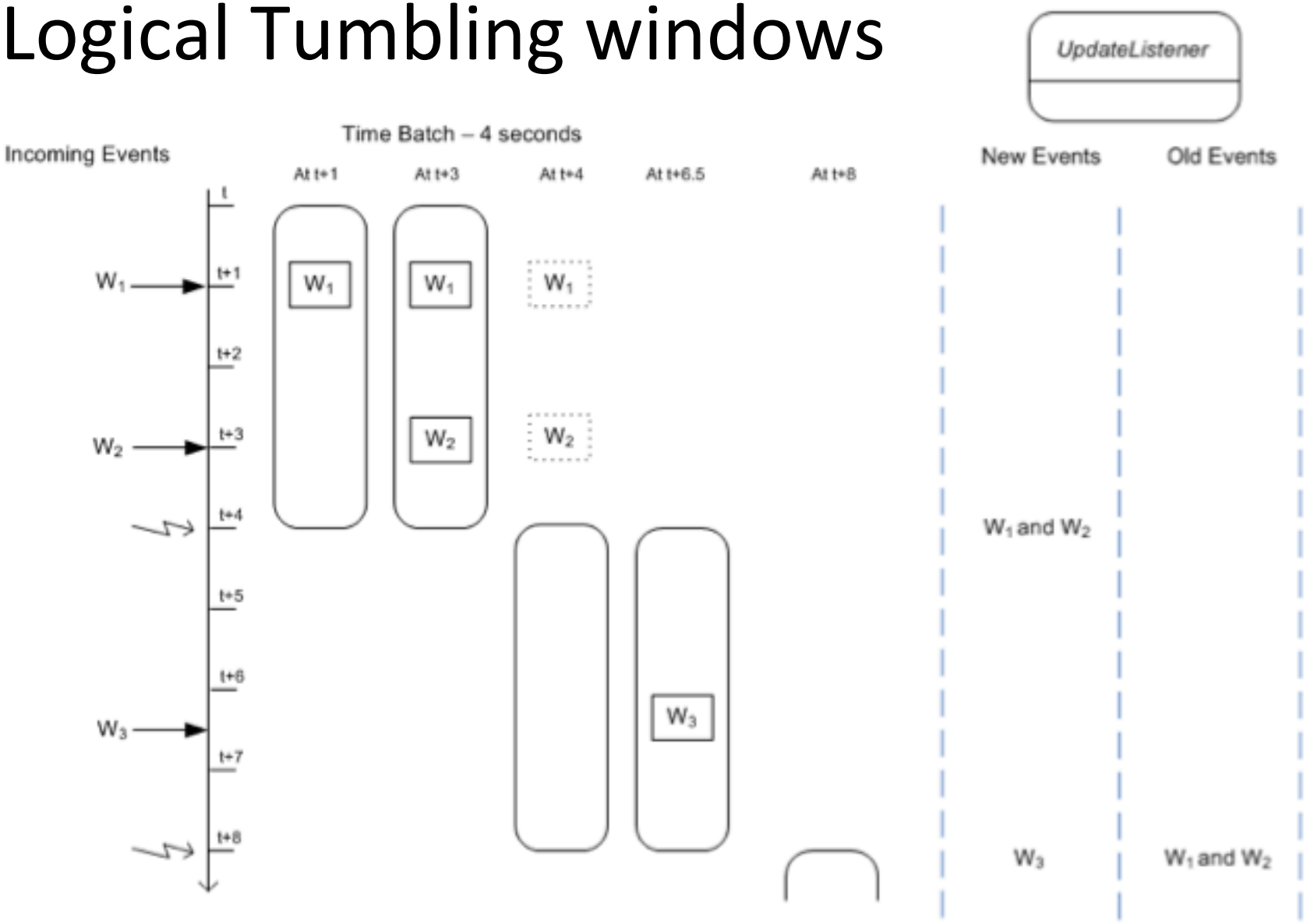
```
select avg(temperature)
from TemperatureSensorEvent.win:time(4 sec)
```

- Execution trace

```
At: 2001-01-01 08:00:00.000 Statement: Out
        Insert
                Out-output={avg(temperature)=30.0}
At: 2001-01-01 08:00:01.000 Statement: Out
        Insert
                Out-output={avg(temperature)=35.0}
At: 2001-01-01 08:00:02.000 Statement: Out
        Insert
                Out-output={avg(temperature)=41.66}
At: 2001-01-01 08:00:03.000 Statement: Out
        Insert
                Out-output={avg(temperature)=45.0}
At: 2001-01-01 08:00:04.000 Statement: Out
        Insert
                Out-output={avg(temperature)=50.0}
At: 2001-01-01 08:00:04.000 Statement: Out
        Insert
                Out-output={avg(temperature)=51.25}
```

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

Logical Tumbling windows



Logical Tumbling windows: example

- Query

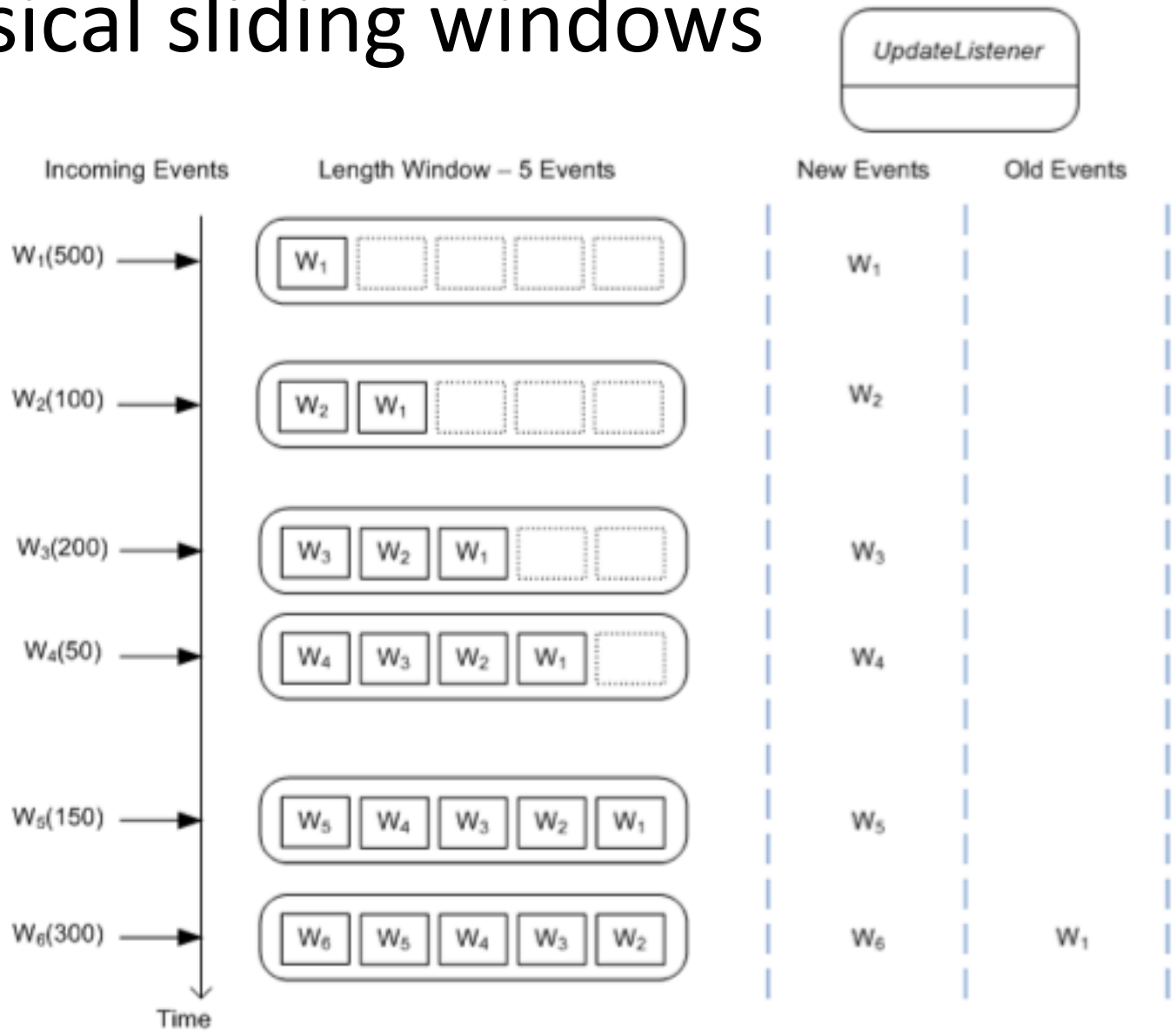
```
select avg(temperature)
from TemperatureSensorEvent.win:time_batch(4 sec)
```

- Execution trace

```
At: 2001-01-01 08:00:04.000 Statement: Out
   Insert
         Out-output={avg(temperature)=45.0}
```

Esper, when using logical tumbling windows, reports only when the window closes

Physical sliding windows



Physical sliding windows: example

- Query

```
select avg(temperature)
from TemperatureSensorEvent.win:length(5)
```

- Execution trace

```
At: 2001-01-01 08:00:00.000
  Statement: Out
    Insert
      Out-output={avg(temperature)=30.0}
At: 2001-01-01 08:00:01.000
  Statement: Out
    Insert
      Out-output={avg(temperature)=35.0}
At: 2001-01-01 08:00:02.000
  Statement: Out
    Insert
      Out-output={avg(temperature)=41.66}
At: 2001-01-01 08:00:03.000
  Statement: Out
    Insert
      Out-output={avg(temperature)=45.0}
At: 2001-01-01 08:00:04.000
  Statement: Out
    Insert
      Out-output={avg(temperature)=47.0}
```

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

Physical Tumbling windows: example

- Query

```
select avg(temperature)
from TemperatureSensorEvent.win:length_batch(5)
```

- Execution trace

```
At: 2001-01-01 08:00:04.000
   Statement: Out
      Insert
         Out-output={avg(temperature)=47.0}
```

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 TemperatureSensorEvent.win:time(4 sec)
    output snapshot every 2 sec
  - select avg(temperature)
    from TemperatureSensorEvent.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)

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.

Pattern example

- Query

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

- Execution trace

```
At: 2001-01-01 08:00:02.000
  Statement: Out
    Insert
      Out-output={a.sensor='S1'}
At: 2001-01-01 08:00:03.000
  Statement: Out
    Insert
      Out-output={a.sensor='S1'}
```

```
At: 2001-01-01 08:00:04.000
  Statement: Out
    Insert
      Out-output={a.sensor='S2'}
```

Resources

- Download Esper (for Java)
  - http://www.espertech.com/esper/download.php
- Download Nesper (for .net)
  - http://www.espertech.com/esper/nesper_download.php
- Quick start
  - http://www.espertech.com/esper/quickstart.php
- Tutorial
  - http://www.espertech.com/esper/tutorial.php
- Questions on EPL
  - http://www.espertech.com/esper/solution_patterns.php
- Documentation
  - http://www.espertech.com/esper/documentation.php

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"