LiSEP: a Lightweight and Extensible tool for Complex Event Processing

Ivan Zappia, David Parlanti, Federica Paganelli

National Interuniversity Consortium for Telecommunications
Firenze, Italy

Context

Need for integrating heterogeneous information systems has promoted the increasing adoption of architectural patterns based on loose-coupling and message exchange.

Event-Driven Architecture is based on a push asynchronous communication mechanism; messages (events) are used to propagate system state alterations.

Specific analysis tools are needed to manipulate great volumes of events, non necessarily organized, in order to obtain a more aggregate and manageable view (Event Processing).

Complex Event Processing to correlate (time and causality) heterogeneous event instances so to infer and manage higher abstraction levels of information.
Complex Event Processing

A family of technologies concurring at the elaboration of large amount of simple data. Goal is to identify the most valuable information samples and/or infer new entities placed at higher abstraction levels.

- Events are available as streams or clouds of data, each one marked with a timestamp.
- Event patterns are represented through query statements modeling temporal and causality-based relations.
- Received events are continuously evaluated against registered statements so to produce prompt feedback (“inversion” of the database concept).
This paper proposes a novel Complex Event Processing engine conceived with extensibility, portability, modularity and scalability requirements in mind. Ease of use and “lightness” are also well-accepted features.
Requirements analysis

Functional requirements:
- system should allow event patterns definition and memorization;
- system should be able to interpret a sufficiently expressive high level language designed to define said event patterns;
- system should be able to accept and evaluate incoming events;
- system should allow the definition of (complex) actions to be executed when a particular pattern is detected;
- system should expose a public administration interface.

Non-functional requirements:
- portability;
- modularity and extensibility;
- scalability;
- minimal configuration and deployment requirements.
Event Processing Language (1/2)

High level and SQL-like language; clauses have been “extended” to enable event patterns definition (event types, different kinds of constraints).

```
SELECT o.amount AS 'OA', p.amount AS 'PA'
ORDER BY 'PA' ASCENDING
FROM ns.Order o WHEN 10
  WITHIN '2011/02/22 08:00:00:000 CEST'
  AND '2011/02/22 11:30:00:000 CEST',
ns.Payment p WHEN 10
  STARTING_FROM '2011/01/01 08:00:00:000 CEST'
  EXPIRES_ON '2011/12/31 18:00:00:000 CEST'
INNER_JOIN o, p ON o.id = p.orderId
WHERE o.amount < 30000
```
To interpret the EPL language and translate statements in objects the CEP engine can actually use we need a parser.

A parser analyzes the syntactic structure of input statement strings and produces a semantically equivalent data structure, a tree in this case.

As a consequence of its SQL-like syntax, this EPL language is made up of independent clauses, which are elaborated in sequence according to a pre-defined and fixed order.

Adopting a divide et impera approach, five distinct parsers were designed, each of them tailored for a specific language clause.

These parsers are used to compile textual clauses into clause expressions; said expressions enable the following evaluation phase.
Design choices

Non-functional requirements reception:

• Java programming language was adopted to match the **portability** constraint;

• an internal structure based on the Staged Event-Driven Architecture (SED) pattern by Matt Welsh (2001) was adopted to achieve **modularity, extensibility and scalability**;

• the adoption of only strictly Java SE libraries (thus avoiding any *application container* usage) and as few as possible third party libraries (thus preventing *vendor lock-ins*) led to **deployment requisites minimization**.

For these reasons our CEP tool is named Lightweight Stage-based Event Processor (**LiSEP**).
Staged Event-Driven Architecture

- Idea: decomposition of a complex event-driven system in stages connected by event-queues.

- Main pattern goals:
  - efficient, event-driven concurrency (thread pools, non-blocking I/O patterns);
  - stages decoupling;
  - stages have a unique control point for incoming events;
  - easy load balancing (event-queues);
  - code modularity.
Layered architecture
Internal structure
Use case example:
statement registration
Evaluation

Submitted events are enqueued in batches according to registered statements (From Clause Manager).

When full, each batch is forwarded to the next stage on the evaluation chain, according to an optimized, and statement-specific, Evaluation Routing Table.

From there on, multiple computation paths are executed in parallel (messages asynchronicity).

The Listeners Manager stage is assigned to call listeners interested in those batches that reaches this final evaluation phase.
Performance analysis (1/2)

- Testing machine: Intel Core i5-750 (2.72GHz) desktop with 4GB of DDR3 PC3-12800 (1600MHz) RAM.
- Two test cases parameterized on variables *batch size* and *selectivity*.

1st test case, filtering statement:
- greater *batch size* leads to fewer exchanged messages and therefore higher throughput values;
- greater *selectivity* leads to fewer events in the latter stages and therefore higher throughput values.
Performance analysis (2/2)

• Testing machine: Intel Core i5-750 (2,72GHz) desktop with 4GB of DDR3 PC3-12800 (1600MHz) RAM.

• Two test cases parameterized on variables \textit{batch size} and \textit{selectivity}.

2nd test case, high-expressivity correlation statement:

• greater \textit{batch size} leads to a more computationally challenging projection phase;

• configuring four threads on the Join stage, performances triplicate on the reference multi-core machine.
Case study (1/3)

Case study on dangerous goods monitoring in maritime transport routes.

This activity is part of an ongoing research project, called SITMAR (acronym for the Italian equivalent of “Integrated system for goods maritime transport in multi-modal scenarios”), funded by the Italian Ministry for Economic Development.

The monitoring and control infrastructure is based on a set of RFIDs and sensors, deployed in the container (the reference loading unit) and/or in the ship storage area in order to monitor and control the environment conditions.

LiSEP can be configured in order to:

- deliver filtered and aggregated events to a monitoring application component deployed locally
- detect anomalies in the physical parameters of containers and the storage area in order to trigger alert services
Case study (2/3)

The following statement is built to detect the “possible fire” complex event from raw temperature sensor readings and simple event LightDetected occurrences.

```sql
SELECT l.senderId AS 'senderId',
       t.timestamp AS 'T timestamp',
       l.timestamp AS 'L timestamp',
       t.value AS 'T value'
FROM model.events.LightDetected l BATCH 1
    INNER_JOIN t, l ON t.senderId = l.senderId
WHERE t.senderId = <containerId> &
  (t.timestamp - l.timestamp < 10000 |
   l.timestamp - t.timestamp < 10000)
```

Case study (3/3)
Conclusions

Distinctive LiSEP traits:

- **stage-based** and **modular** architecture according to the SEDA pattern; this approach clearly separates the core logic devoted to event processing from the inner communication mechanism and low-level thread management;

- SEDA framework adoption eases system configuration, thus allowing available resources exploitation (**scalability**);

- each stage is extremely **simple** and **intelligible**;

- high **maintainability** and **extensibility** given the high separation of concerns through the autonomous and specialized units constituting the engine; adjustments can be performed with limited impact and only on strictly related areas;

- **deployment procedure** is **simple** given the independence from specific libraries other than standard Java SE platform;

- independence from third party libraries concurs in producing a **lightweight** engine, which can be **easily integrated** in pre-existing applications.
Further investigations

Areas that could benefit from specific further investigations:

• the Event Processing Language may be extended to increase system features (new negation operator, extension of present temporal operators);

• the SEDA framework may be extended with a self-tuning module so to optimize exploitation of available resources (e.g., changing number of threads allocated per stage).
LiSEP: a Lightweight and Extensible tool for Complex Event Processing

Thanks!
Appendix 1
Performance analysis