Expert Systems with Applications 39 (2012) 10408–10419

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

A lightweight and extensible Complex Event Processing system for sense and respond applications

Ivan Zappia *, Federica Paganelli, David Parlanti

National Interuniversity Consortium for Telecommunications (CNIT), c/o Dipt. di Elettronica e Telecomunicazioni, Università di Firenze, Firenze, Italy

ARTICLE INFO

Keywords:
Complex Event Processing
Staged Event-Driven Architecture
Event processing declarative language
Goods monitoring
Sense and respond applications

ABSTRACT

Complex Event Processing (CEP) is considered as a promising technology for enabling the evolution of service-oriented enterprise systems towards operational aware systems. CEP effectively supports the implementation of "sense and respond" behaviours, as it enables to extract meaningful events from raw data streams originated by sensing infrastructures, for enterprise processes and applications consumption. This paper proposes a novel CEP engine conceived with ease of use, extensibility, portability, and scalability requirements in mind. More specifically, we propose a Lightweight Stage-based Event Processor (LiSEP) based on a layered architectural design. Thanks to the adoption of Staged Event-Driven Architecture principles, core event processing logic is decoupled from low-level thread management issues. This results in an easy-to-understand and extensible implementation while testing results show performance scalability. We report on the carrying out of a case study on dangerous goods monitoring in maritime transport. The objective of the case study is to develop a Proof of Concept application leveraging on LiSEP capabilities in sensor and RFID events processing for monitoring and alerting purposes.

1. Introduction

Service-oriented enterprise platforms are increasingly called to support sense and respond capabilities in a wide range of application scenarios (finance, logistics, goods transport, homeland security, just to mention a few). Complex Event Processing (CEP) is considered as a promising asset for enabling the evolution of SOA enterprise systems towards such operational aware systems.

While CEP is a relatively not new technological paradigm, as Luckham first introduced it in 1998 (Luckham & Frasca, 1998), the expectations on the exploitation of CEP in real applications are growing exponentially (Leavitt, 2009).

One of the factors motivating this trend is the increasing amount of heterogeneous and distributed data sources that have become available and exploitable. In this context, the adoption of Service Oriented Architecture (SOA) models and Enterprise Service Bus (ESB) technology is acting as a driver for and a beneficiary of the increasing adoption of CEP technologies: driver, since it facilitates the access to heterogeneous data sources and the integration of CEP capabilities to existing systems; beneficiary, as CEP technologies may enhance capabilities of business processes in responding to business events with minimal latency (Hermosillo, Seinturier, & Duchien, 2010; Schiefer, Rozsnyai, Rauscher, & Saurer, 2007).

In last years, an increasing number of efforts have been made in the adoption of CEP technologies in sense and respond applications in order to bridge SOA enterprise environments with pervasive technologies and applications such as Internet of Things and context-aware systems (Haller, Karnouskos, & Schroth, 2009; Ku, Zhu, & Hu, 2008; Zang, Fan, & Liu, 2008).

As discussed by Wei, Ari, Li, and Dekhil (2007), there is a gap between business processes and service platforms on one side and existing CEP engines on the other side. As a matter of fact, most existing engines, which have not been conceived with SOA-inspired principles and requirements in mind, weakly focus on extensibility, interoperability and modularity requirements. They rather focus on language semantics and performance and often try to embed CEP capabilities into a middleware, thus resulting in a big proprietary package.

In order to address these issues, in this work we propose a lightweight general-purpose CEP engine, called LiSEP (Lightweight Stage-based Event Processor). LiSEP design has been driven by ease-of-use, extensibility, scalability and portability requirements.

In this paper, we present and discuss main architectural design and implementation choices aiming at fulfilling such requirements. To this purpose, we provide insights on the LiSEP layered architecture and the adoption of Staged Event-Driven Architecture (SEDA) principles, helping to clearly separate core event processing logic from lower level resource management issues. This stage-based and event-driven approach enables a highly modular implementation, thus easing extension and reuse of the implemented

* Corresponding author.
E-mail addresses: ivanzappia@gmail.com (I. Zappia), federica.paganelli@unifi.it (F. Paganelli), david.parlanti@gmail.com (D. Parlanti).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.01.197

features, the overall system maintainability and performance scalability with respect to hardware resources.

In an ongoing research project, we are experimenting the use of LiSEP capabilities in a sense and respond application scenario focusing on dangerous goods monitoring during maritime transport. The main objective of the case study is to experiment LiSEP capabilities in filtering and processing sensors and RFID events for monitoring and alerting purposes.

The paper is structured as follows: in Section 2 we present background information on sense and respond systems and related work on Complex Event Processing; in Section 3 we briefly provide an overall view of LiSEP capabilities, while Section 4 describes the defined SQL-like Event Processing Language. Section 5 describes main LiSEP design and implementation choices. Section 6 shows testing results for performance evaluation and Section 7 introduces a case study for dangerous goods monitoring in maritime transport. Section 8 outlines our conclusions and future work.

2. Background

Sense and respond systems have been defined as "systems that sense what is going on and respond appropriately" (Chandy, 2005). Application of sense and respond systems is gaining attention in several application domains, such as finance, homeland security, critical infrastructure management, logistics and goods transport monitoring.

The increasing attention on sense and respond systems has been boosted also by the increasing availability of low-cost RFID tags and sensors. Response activities may range from an alert notification to the invocation of a complex business process, depending on the specific application domain.

The perspective is that the potential impact will be on all aspects of daily living (Estrin et al., 2010), from a global to a personal scope across a wide range of application domains, such as border surveillance and global environmental applications, maritime surveillance (Parlanti, Paganelli, & Giuli, 2010), smart grid and monitoring of critical infrastructures (Estrin et al., 2010), smart cities (Hernández-Muñoz et al., 2011) and participatory sensing applications (Estrin et al., 2010).

Sense and respond systems are inherently based on event-based processing. As a matter of fact, phenomena under observation in the physical environment may be represented as discrete changes of variables within a certain time interval, as measured by physical sensors.

Although the development of sense and respond systems is not a novel research topic, current research challenges for industrial and academic communities include:

- abstraction layers: layers of abstraction are needed in order to abstract the essential structural and functional properties of sense and respond capabilities, hiding implementation and low-level details e.g., interfaces of specific devices, messaging, timing and exception handling of communications among devices and computing platforms (Baker, Zafar, Moltchanov, & Knappmeyer, 2009; Chen, Chou, Cohen, & Duri, 2007);
- design and implementation reuse: different sense and respond applications may share similar conceptual and functional models or some implementation details. Availability of reusable design and processing components may facilitate the rapid development of sense and respond applications (Chen et al., 2007) by enabling reconfiguration and composition of available building blocks (Liu & Zhao, 2010).

With respect to these challenges, Complex Event Processing (CEP) techniques are considered as promising technology assets (Leavitt, 2009). CEP is a technology for processing and analyzing multiple simple events from distributed sources, with the objective of extracting semantically richer events from them in a timely, online fashion. It provides capabilities for specifying patterns of events at different layers of abstraction in a high-level language. More specifically, event abstraction layers may be defined by specifying (Luckham & Frasca, 1998): (a) a set of event types for a specific level of abstraction and system activities triggered by the occurrence of such event types; (b) a set of event abstraction conditions that rule the creation of event instances according to pattern matching conditions on lower-level events.

2.1. Existing approaches for CEP

To the state of our knowledge, only a few analyses have been performed to provide a classification of existing CEP solutions (Eckert, 2008; Kavelar, Obweger, Schiefer, & Suntinger, 2010).

Eckert (2008) argues that CEP approaches can be distinguished according to the adopted event querying language style: languages based on composition operators (also called event algebras), data stream query languages, and production rules.

The first language class expresses complex events by composing single events using different composition operators (such as sequence, conjunction, disjunction, negation, temporal relations, event occurrences counting) and nesting of expressions. Examples are: SnoopIB (Adaikkalavan & Chakravarthy, 2006), where the authors formalize the detection of event operators using interval-based semantics; the work by Zang et al. (2008), proposing an event meta model, event pattern rules and operators for complex event detection; CEDR (Complex Event Detection and Response), which is an event processing system, based on a declarative language for specifying patterns that include high level event operators and support for detection lifetime (Barga et al., 2006).

Data stream query languages enable to express queries in an SQL-like language. They put their focus more strongly on data than on composition-operator-based languages, thus paying less attention to issues such as timing and temporal relationships. A relevant example is the Continuous Query Language (CQL) (Arasu, Babu, & Widom, 2006), which is an SQL-based declarative language for registering continuous queries against streams and stored relations. CQL is an instantiation of an abstract semantics for continuous queries, based on two data types (streams and relations) and operators over these data types.

The third group, "production rules", refers to approaches based on the use of rule-based systems to implement event-based queries. Event queries may be implemented by evaluating a condition expression over a set of facts. Drools (Drools, The Business Logic integration Platform, 2011) is a rule-based engine that includes a module providing the engine with native support for events evaluation and temporal logic analysis.

Several systems may be considered as hybrid approaches, i.e. solutions combining aspects of the above-mentioned categories, such as Esper (EsperTech, Event Stream Intelligence, 2011).

2.2. CEP in sense and respond systems

Complex Event Processing techniques are gaining increasing attention in both academic and industrial communities for the design and development of sense and respond systems. An especially active research field is RFID and sensor data processing. Hereafter we mention just a few examples in some relevant application domains: health, intelligent transport systems, and logistics.

- Health: Yao, Chu, and Li (2011) experiment complex event processing techniques in an RFID-enabled framework for managing hospital data for surgical procedures. They use Drools 5.0 as CEP

engine to model basic events and event patterns in hospitals for detecting medically significant events. In Mouttham, Peyton, Eze, and Saddik (2009), the authors use the aforementioned open source CEP engine Esper in a data-integration system for personal health monitoring.

- Intelligent transport systems: Dunkel, Fernandez, Ortiz, and Ossowski (2008) propose an event-driven road traffic management systems, where CEP capabilities are used to extract meaningful events from raw measurements provided by traffic sensors and to deliver resulting events as input to traffic control systems for decision support.
- Logistics: In Kim, Yoo, and Park (2006) Complex Event Processing capabilities are used in a business process automation system for harbor operations in container depot. More specifically, CEP is part of an RFID Event Management System serving to reduce redundant data and aggregate relevant data for minimization of data traffic.

In such a wide range of possible application domains, CEP capabilities have to be properly configured and customized according to the requirements of the case at hand. As discussed in details by Chandy (2005) and Voisard and Ziekow (2010), different application domains may require specific characterizations of the needed event processing approach (e.g., in terms of performance requirements, distributed vs. centralized processing, hardware configurations).

As discussed by Wei et al. (2007), most existing engines weakly focus on extensibility, portability and modularity requirements. They rather focus on language semantics and performance aspects and often try to embed CEP capabilities into a middleware, thus resulting in a big proprietary package.

In the following section we discuss how LiSEP has been designed and developed in order to address these issues.

3. Lightweight Stage-based Event Processor

The Lightweight Stage-based Event Processor (LiSEP) is a general-purpose java-based CEP engine. It receives input event instances and verifies whether they validate pre-registered query statements.

Events are defined as "anything that happens, or is contemplated as happening" (Luckham et al., 2008); the corresponding event instance is a concrete semantic object containing data describing an event occurrence. Event processing systems can be arranged to detect particular events or, more generally, specific event patterns and react in user-defined ways. Events may be distinguished into simple and/or composite events. Composite events are defined as composition of simple and/or other composite events and are typically placed at a higher abstraction level (Hinze, Sachs, & Buchmann, 2009).

In LiSEP the events model includes a unique name (identifier) and a collection of attributes. Attributes values may be simple types (literal values) as well as other event object types. Events are represented internally as Plain Old Java Objects (POJOs) according to the JavaBeans conventions.

In a typical deployment configuration, input events are generated by external applications and routed to the CEP engine by a proper messaging infrastructure (e.g., Enterprise Service Bus). Input data can be delivered by sources in different message formats (e.g., XML, JSON) and then transformed into the Java-based internal representation by proper adapter components.

Query statements are used to express target event patterns. Some software components may be registered as listeners for a target query statement. When a specific event pattern is detected, registered listeners are notified and execute specific actions as a reaction to successful event detection (as shown in Fig. 1). Actions performed by listeners may include, for instance, arithmetic and statistic functions on some events data, storage operations or notifications to external applications.

The LiSEP processor can be properly configured in order to represent hierarchies of event abstraction layers in specific application domains. This can be achieved by properly customizing the listeners behavior for producing higher-level events. These events can then be routed to feed the input event stream and analyzed for detecting possible further events.

LiSEP is built around a set of features aiming to characterize it beyond generally expected event processing behavior.

- Portability: the adopted Java language assures code portability and adaptability to software-wise and hardware-wise heterogeneous environments.
- Modularity and extensibility: LiSEP is designed with the "separation of concerns" principle in mind so to mold a highly modular tool with its main functional units well isolated in order to ease maintenance and extension operations. This is achieved implementing clear but effective programming patterns resulting in easy to understand and manipulate code.
- Scalability: LiSEP can be easily employed in applications with different load requirements; this is possible thanks to LiSEP ability to exploit available computational resources.
- Minimal configuration and deployment requisites: deployment procedure is really simple since it consists only of the Java Virtual Machine installation; in addition, independence from third party libraries, contributes to build a lightweight event processor and therefore, a tool easily integrable in preexistent applications.

Statements are written in a specially defined high-level language. The following section describes this Event Processing Language and related parsing components that have been defined for LiSEP implementation.

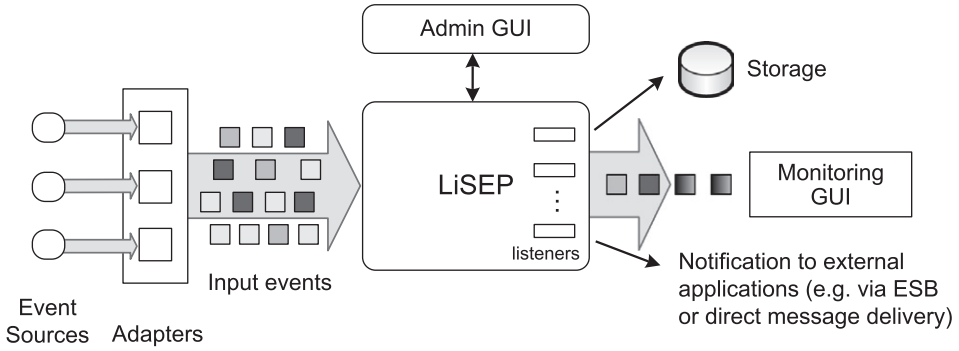

Fig. 1. LiSEP Event Processing flow.

4. Event Processing Language

A relevant role in any event processing system is played by the high-level query language used to describe constraints and event patterns; instead of adopting the same language used to develop the event processing application itself, a dedicated declarative and specially designed language grants a good wealth of benefits. Working on a higher level of abstraction set the programmer free to focus on event patterns definition rather than on implementation details specific to usually complicated detection algorithms; this general ease of use consequently produces more flexible and easy to maintain code also thanks to the deriving decoupling of event logic from the rest of the application logic; this is especially important in highly dynamic contexts where event processing capability has to be adapted frequently to meet changing requirements or it has to be integrated with other independently developed software. Finally an Event Processing Language is required to build query compilers and optimization-based evaluation engines so to lessen programmer's workload (Eckert, 2008).

The Event Processing Language (EPL) developed for this project is based on the SQL syntax; extensions and customization were made to adapt the language to the specificities of the event processing context like the inclusion of time related operators.

In this chapter the aforementioned language is introduced through a comprehensive tutorial in Section 4.1, while Section 4.2 is dedicated to clause parsers description.

4.1. Language tutorial

In this section the Event Processing Language (EPL) is introduced through a tutorial. Starting with a minimal complexity statement, more and more clauses and features are described and shown, also through some examples, exposing an increasing level of detail.

4.1.1. Minimal query "Select-From" type

The simplest and minimal complexity query declarable with the defined EPL follows the "SELECT *a* FROM *b*" structure where, as in standard SQL syntax, *a* represents the list of attributes that need to be selected from the events that match the object type declared in *b*. For example:

```
SELECT o.id, o.customerCode, o.amount
  FROM <eventNamespace>.Order o BATCH 5
```

The FROM clause requires explicit queuing mode declaration; the two available queuing mechanisms are referred to by the following keywords: BATCH (like in the example), followed by an integer, defines the length of a queue buffering incoming events; WHEN is used to specify temporal constraints and will be described later in this section.

The <*eventNamespace*> placeholder is used to specify the qualified event name so the system can correctly identify the proper event class to use.

Results can be shown without repetitions via the DISTINCT keyword. Optionally, as a presentation feature, elements in the selection list can be accompanied by an alias definition that renames events fields:

```
SELECT o.id AS 'Order ID',
       o.customerCode AS 'Customer Code',
       o.amount AS 'Amount (Euro)'
  FROM <ns>.Order o BATCH 5
```

4.1.2. Before filtering, the FILTER keyword

Slightly incrementing query complexity, the FROM clause can be extended to filter incoming events using the keyword FILTER as in:

```
SELECT o.id, o.customerCode
  FROM <ns>.Order o BATCH 5
FILTER o.amount > 30000
```

This kind of filtering happens before events get enqueued in batches.

4.1.3. After filtering, the WHERE clause

Another filtering technique is achieved using the WHERE clause. As in any SQL-like language, this clause filters instances of incoming events according to a Boolean-arithmetic expression.

```
SELECT o.id, o.customerCode
  FROM <ns>.Order o BATCH 5
 WHERE (o.amount >= 30000) &
       (o.dept='hardware')
```

This kind of filtering happens after events gets enqueued in batches.

4.1.4. Projections with the JOIN clause

To correlate different event types, a Join clause is needed. The current implementation offers two different joining modes using the INNER_JOIN and CROSS_JOIN keywords.

```
      SELECT o.id, o.customerCode
        FROM <ns>.Order o BATCH 5,
             <ns>.Payment p BATCH 5
INNER_JOIN o, p
        ON o.id = p.orderId
     WHERE o.amount > p.amount
```

Analogously for cross joining:

```
      SELECT o.id, o.customerCode
        FROM <ns>.Order o BATCH 5,
             <ns>.Payment p BATCH 5
CROSS_JOIN o, p
```

Sometimes could be useful realizing a self join between events of the same class; for this purpose, so to remove ambiguity, renaming is available also for this clause like in:

```
      SELECT o.id, o.customerCode
        FROM <ns>.Order o BATCH 5
INNER_JOIN o AS a, o AS b
        ON a.date-b.date < 5
```

4.1.5. Time intervals

A specific effort has been devoted in implementing a set of features enabling users to declare time relations and constrains between events. The first way to do that is by using the aforementioned WHEN keyword in the FROM clause. It can be used

to set a time interval, either open or closed, using the three keywords WITHIN, STARTING_FROM and ENDING_ON as in the following example:

```
SELECT o.id, o.customerCode, o.amount
  FROM <ns>.Order o WHEN 5
WITHIN '2011/01/01 08:30:00:000 CEST' AND
       '2011/01/31 18:30:00:000 CEST'
```

The FROM clause parser is implicitly configured to use the system time the event is detected as reference time attribute. If needed, any date field in the original event type can be used; the corresponding keyword is USING:

```
     SELECT o.id, o.customerCode, o.amount
       FROM <ns>.Order o WHEN 5
  ENDING_ON '2011/01/01 08:00:00:000 CEST'
      USING o.date
```

Dates may be represented according to the standard format: "yyyy/MM/dd HH:mm:ss:SSS z". In order to enable processing of events detected by geographically distributed and/or moving devices, while assuring coherence of data processing, the reference time zone has to be explicitly defined.

4.1.6. The ORDER BY clause

Finally the Order By clause is used, as intuitively apparent, to force a specific sorting order to the output event frames. Said order can be set ascending or descending.

```
  SELECT o.id AS 'Order ID',
         o.customerCode AS 'Customer Code',
         o.amount AS 'Amount (euro)'
ORDER BY 'Amount (euro)' DESCENDING
    FROM <ns>.Order o BATCH 5
```

4.1.7. Statement expiration

To accommodate special needs, the FROM clause offers one more keyword: EXPIRES_ON; this can be used to define an expiration time for the statement as a whole; at that time the statement is being removed from the system.

```
     SELECT o.id, o.customerCode, o.amount
       FROM <ns>.Order o WHEN 5
  ENDING_ON '2011/01/01 08:00:00:000 CEST'
EXPIRES_ON '2011/01/01 09:00:00:000 CEST'
```

This feature, used in conjunction with the listeners mechanism, allows users to configure particular actions to be triggered after the statement lifecycle is set to end; for example to create a new statement parameterized on preceding results. We chose to introduce a specific keyword to model this behavior, instead of using the above mentioned time analysis keywords. This choice derives from the fact that the underlying semantics refers exclusively to the system time, while the above-mentioned keywords may be tied to custom time event fields.

4.2. Parsers

Given the specific nature of any SQL-like language, and especially in the introduced EPL, is easily understandable that, given a particular batch of events, evaluation flow runs through clauses following a well known order: in fact, after the initial type-check and accumulation phase specific to the From clause (with optional a priori filtering), Join and Where clauses are applied (when present) in this order; finally Select and Order By are used to extract relevant components and force a specific sorting order.

Consequently the evaluation process was split in five different units, one for each clause. When a new plain text statement is submitted to the system, each clause is extracted, parsed and compiled into an expression by a specific parser; each of these expressions consists primarily of a binary tree hierarchically representing the operations needed for clause validation. For the Where clause the expression tree simply follows the structure of the Boolean-arithmetic expression is based on. For other clauses, such as the From one, expressions are built depending on the associated keywords that define the overall clause semantics; for instance, different From clause expression trees are built whether the BATCH or WHEN keywords are used. Fig. 2 represents the parse tree for the From clause for the following EPL statement:

```
       SELECT o.amount AS 'OA'
         FROM ns.Order o WHEN 10
STARTING_FROM '2011/12/31 08:00:00:000 GMT'
       FILTER o.amount > 400
```

At runtime said expressions are used to perform evaluation checks in the aforementioned prefixed order when events flow through the system.

5. LiSEP architecture

To achieve modularity and extensibility an approach based on the separation of concerns principle was chosen. Keeping clearly apart each peculiarity of the tool grants freedom to perform even deep changes without affecting non-strictly related areas.

As shown in Fig. 3 the LiSEP tool is composed of two layers. The upper LiSEP Event Processing Layer deals with event evaluation and processing tasks and is implemented as a graph of event-driven stages connected with explicit event queues in accordance to the SEDA architectural pattern (Welsh, Culler, & Brewer, 2001); it encloses the core logic of the LiSEP engine and leverages on thread management and stage building capabilities provided by the lower SEDA Framework Layer thus allowing to separate specific core application logic from thread management and scheduling issues (Welsh et al., 2001).

The other layers in Fig. 3 represents an abstraction of the execution environment with the JVM playing as unique strict requirement for application deployment. There is actually no need for a J2EE platform as the Java SE is perfectly sufficient; therefore application lifecycle can be managed with just the two public init and cleanUp methods. Deployment is perfected setting up a connection between the LiSEP processor and any component enabled to event acquisition; said component will then forward events to the processor public interface through the sendEvent method.

5.1. Staged Event-Driven Architecture

To ensure the greater level of concurrency and scalability the event-driven architecture was also used to approach the internal component design process. Specifically we chose to adopt the Staged Event-Driven Architecture (SEDA) framework, developed from scratch within previous research activities (Parlanti et al., 2010). A SEDA-style application is built around a set of stages connected by event queues. This programming model aims at

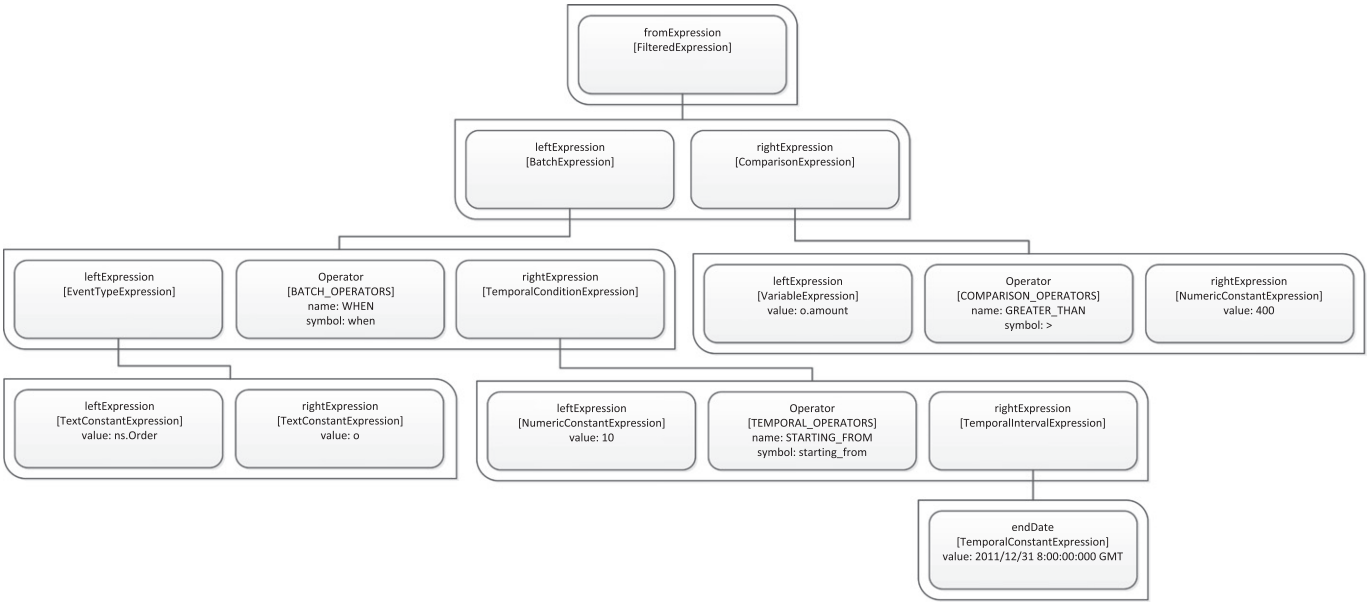

Fig. 2. Parsing tree for example From clause.

decoupling thread scheduling from application logic. Since thread management is completely entrusted to this specialized layer every change in policy or implementation is made without affecting any other part of the event processing logic in the upper layer; this grants freedom to pursue optimization in an area so affected by specific platform features: thread management is in fact tightly tied to the particular combination of hardware, operating system and java virtual machine; use of java technology comes with portability as a strong key feature but variability in deployment environment may produce the need for specific resource management tuning.

Capitalizing on the SEDA pattern, the application is split into self-contained functional units (i.e. stages). When the LiSEP processor is instantiated a total of eight stages are generated, each of them designed to resolve a specific aspect of computation; a per stage description is found later in this section. Communication among the stages happens entirely through messages exchange strictly following SEDA pattern specifications; this topic is covered in subsection 5.4.

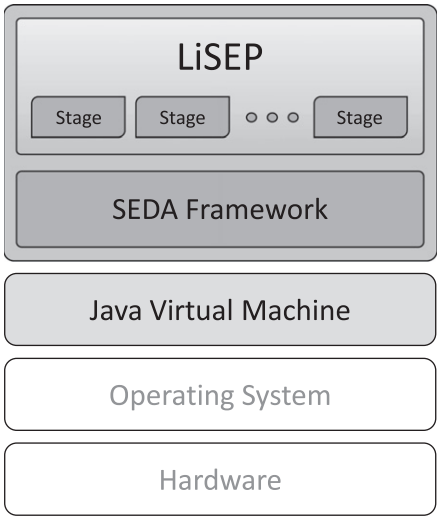

Fig. 3. LiSEP layered architecture.

5.2. Public API

A public API enables LiSEP users (i.e. end users via Administration GUIs or client applications) to configure and administer the event processing system:

- the createStatement method is used to add statements to the system; it accepts three parameters: a string containing the plain text input statement, a list of listeners to be invoked whenever the statement is triggered and an optional string used to force the assignment of a particular statement id (this can be used as an handle to manipulate statements at runtime); the removeStatement method is used to delete one or more statements from the system; it accepts the target statements identifiers as input parameters;
- the addStatementListener method can be used at runtime to add one or more listeners to an already registered statement and accepts as parameters the target statement id and the list of desired listeners; the removeStatementListener method is used to remove one or more listeners and accepts the same parameters.

5.3. LiSEP stages

In this section the eight stages are introduced and described in detail.

5.3.1. StatementBuilder

This is the stage responsible exclusively for statement creation; after receiving the plain text statement as input string, it creates the statement object and forwards it to the StatementManager stage. Actual creation process consists of three steps:

- Regular expression based input string validation. To ensure consistency and general syntax correctness, regular expression based validation is executed on the input string so to extract individual clauses.

- Clause expressions parsing. Single clause strings are parsed and compiled into expressions by specific clause parsers.
- Evaluation routing table building. In accordance with the statement structure, an optimized evaluation routing table is built (as detailed in subsection 5.5). Using this precompiled table each stage knows which clause manager is the next hop in the evaluation chain.

In summary the registered statement object consists of a unique statement id, the precompiled clause expressions and the customized routing table.

5.3.2. StatementManager

This stage is invoked to add and remove statements. As mentioned above, each statement is composed of precompiled clause expressions that have to be stored in each corresponding clause manager. To do so this stage has to contact each clause manager stage individually. To achieve consistency through the clause stages but in a lightweight way only one rule is followed: since, during evaluation, events navigate clause stages starting with the FromManager stage without a centralized registry of on-line statements, there could be a chance that, at a given time, not all stages already received and processed every clause; consistency is thus achieved forcing the From stage to update as first when removing and as last when adding a statement to the system.

5.3.3. Clause managers

The Clause Manager stages are specialized units that store compiled expressions clauses for each registered statement and evaluate events against them. These stages offer identical interface regarding clause storing and removing logic but implement different evaluation methods.

Following directly the EPL language structure each stage is named after the corresponding statement clause: From Clause Manager; Join Clause Manager; Where Clause Manager; Select Clause Manager; Order Clause Manager.

5.3.4. ListenersManager

The eighth and last stage is the ListenersManager stage that is in charge of handling listeners registration and removal, as well as invoking the corresponding listeners according to the triggered statements and the events reaching this final processing phase.

5.4. Messages-based stage interaction

Stages interact via message exchange. We defined three message types, one for each class of recipients in the system. Each message can be customized to represent different commands using the field actionCode; the actual message payload changes accordingly.

5.4.1. StatementBuilderMessage

This message is directed to the StatementBuilder stage and is used to trigger a statement building process through the BUILD_STATEMENT action code.

5.4.2. StatementManagerMessage

This message can be used to interact with the StatementManager stage. Two action codes are available: ADD_STATEMENT and REMOVE_STATEMENT.

5.4.3. ClauseManagerMessage

This message type is used to reach all the clause managers both for clause management and for event evaluation. In the first case the STORE and the REVOKE action codes are used; in the second case only the EVALUATE action code is needed. Even if the ListenersManager has a different role than the one covered by the clause managers, it shares with them much of its behaviour and, therefore, of its implementation logic; for this reason the same ClauseManagerMessage is used but, besides the three action codes already introduced, messages delivered to this stage may include further specific action codes: ADD_LISTENER, REMOVE_LISTENER and ON_EXPIRE_EVENT. The first two action codes are used to add and remove listeners to already registered statements while the last one supports the statement expiration mechanism described in Section 4.1.7.

5.5. Event processing

As previously described in Section 4.2, the event evaluation process consists in letting events flow through the clause managers according to a specific order; in each stage events are validated against precompiled clause expressions.

Given the high volume of messages exchanged within the system, a worthwhile approach to optimization would be ensuring that only the strictly essential messages are produced and sent among LiSEP stages. Since statements seldom use all clauses the idea was to implement a customized routing table. This table is compiled during the first statement-building phase simply by taking note of missing clauses; at evaluation time the non-mentioned stages can be easily skipped; this particular case is illustrated in Fig. 4. This is viable because of proper stage design: events frames are built and sent without recipient assumptions so to be compatible among all the possible target stages.

5.6. Self-tuning

As a possible future extension, a self-tuning module could be added to the SEDA Framework capabilities. The adoption of the separation of concerns principle, by entirely devoting thread management to a specific layer, eases the design and implementation of self-tuning policies for optimizing performance and resource management by automatically adjusting parallelization level and JVM parameters. This could be used to overcome limits or capitalizing on strong features of the deployment environment. For

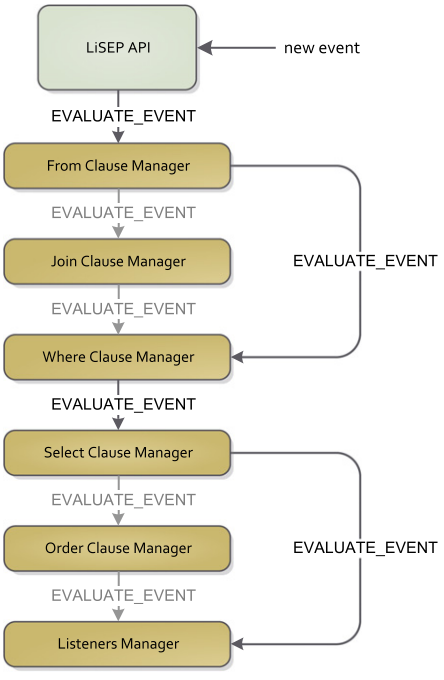

Fig. 4. Message routing among clause stages.

instance, queues length could be adjusted in relation to system memory size or the number of thread pools used could be arranged to take advantage of the CPU core count as discussed also in Welsh et al. (2001).

6. Performance

In this section we present testing activities and resulting measures used to estimate LiSEP performances and scalability. Since our aim was to build a lightweight, modular and extensible Complex Event Processing tool, raw performances never were a priority. These tests are therefore to be seen as a qualitative validation of LiSEP as an enabling technology.

6.1. Testing methodology

Tests were performed measuring the maximum throughput values, in terms of computed events per second, a correctly configured LiSEP system can achieve. Events are evaluated against two parameters: *S* is the size of event batches (10, 25, 50, 100 and 200) and *K* is a threshold used to force desired query selectivity (20%, 40%, 60% and 80%).

We designed two different test cases in order to cover different levels of LiSEP Event Processing Language expressivity and, therefore, produce different levels of computational complexity; with this approach we try to estimate real use performances through the use of two upper and lower bounding statements which will be presented in the following subsections.

Tests were executed on two different machines: an Intel Core 2 Duo P8600 (2.4 GHz) notebook with 4 GB of DDR3 PC3–8500 (1066 MHz) RAM and an Intel Core i5-750 (2,72 GHz) desktop with 4 GB of DDR3 PC3–12800 (1600 MHz) RAM.

6.2. First test case: filtering

The first case is based on a testing statement reproducing a common filtering scenario and consisting of the two strictly required clauses; only one type of events is analyzed.

```
SELECT o.amount AS 'Order Amount'
  FROM ns.Order o BATCH S
FILTER o.amount< K
```

Fig. 5 shows throughput results achieved in a simple test by sending an input stream of 120.000 events. The throughput is calculated by measuring the time needed for processing the whole input event stream. Fig. 5 shows how the throughput curve varies with the increase of the batch size (S) in both testing configurations.

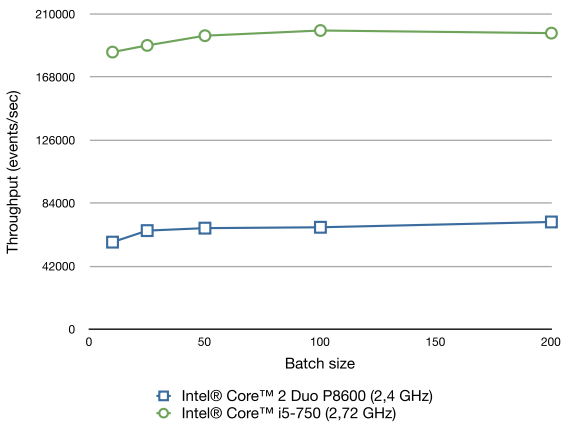

Fig. 5. Throuhput values for the two testing environments.

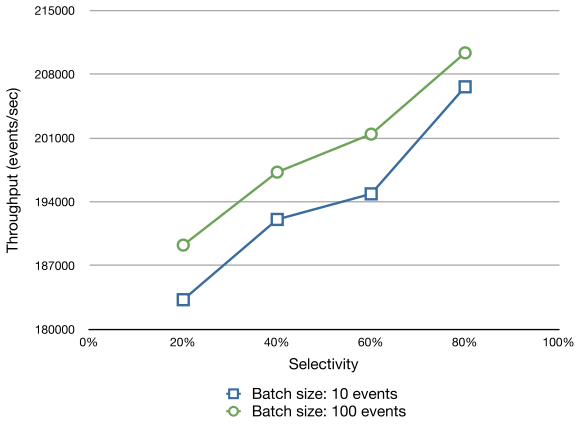

Fig. 6. Selectivity-based testing results.

By increasing the batch size, system performance slightly improves. This is due to the reduction in number of messages exchanged among the stages. Moreover, by comparing the two curves, it comes evident that LiSEP stage-based architecture takes advantage of the higher number of available cores as throughput increases accordingly. We then performed the remaining tests on the more performing desktop machine.

Fig. 6 shows how throughput varies in relation to the selectivity rate with different batch sizes (10 and 100). As expected, greater selectivity leads to better performance, as less computation is required.

6.3. Second test case: correlation

In the second test case, the statement correlates events of two different types, using a wide set of available EPL language features resulting in a much more computationally demanding query.

```
    SELECT o.amount AS 'OA', p.amount AS 'PA'
ORDER BY 'PA' ASCENDING
   FROM ns.Order o WHEN S
  WITHIN '2011/02/22 08:00:00:000 CEST'
     AND '2011/02/22 11:30:00:000 CEST',
         ns.Payment p WHEN S
  WITHIN '2011/02/22 08:00:00:000 CEST'
     AND '2011/02/23 11:30:00:000 CEST'
INNER_JOIN o, p ON o.id = p.orderId
    WHERE o.amount< K
```

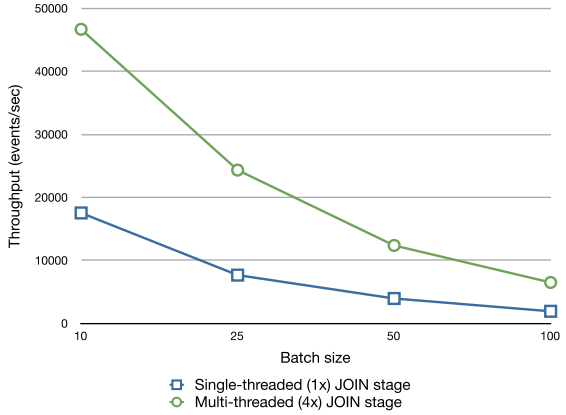

Fig. 7. Throughput values for Single- and Multi-threaded JOIN stage.

Table 1
Event Model.

Event levels	Event types	Event attributes
Raw Events	Sensor Reading	(tagId, eventId, timestamp, temperatureValue, pressureValue, humidityValue)
	Opening	(tagId, eventId, timestamp, eventType[OpeningDetected])
	Light Detection	(tagId, eventId, timestamp, eventType[LightDetected])
Domain Entity Events	Container Update	(containerId, timestamp, parameterType [Temperature | Pressure | Humidity | Opening | Light Detection], value)
Warning Events	Warning	(containerId, timestamp, warningLevel [High | Medium | Low], eventType [PossibleFire | Breaking | UnauthorizedOpening | OutOfThresholdMeasurement])

The query complexity mostly derives from the JOIN operator, since it is a computationally demanding operation.

As shown in Fig. 7, LiSEP performance drastically decreases with batch size; by leveraging on thread management capabilities provided by the SEDA framework, it is easy to allocate more resources where needed. In this case, we configured the JOIN stage to use up to 4 threads in order to match the number of available cores. This leads to more than doubled throughput values.

7. Case study

At present we are experimenting the use of the LiSEP engine in a case study on dangerous goods monitoring during maritime transport. 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.

Maritime transport of dangerous goods represents a relevant example of a sense and respond scenario. As a matter of fact, security and safety assurance across the maritime supply chain is considered a critical factor (Barnes & Oloruntoba, 2005), since maritime transport is susceptible to the effects of terrorism and other vulnerabilities because of its global and open nature and its inherent complexity (van de Voort, O'Brien, Rahman, & Valeri, 2003). In this context, the need of crisis management capabilities, ranging from anomaly detection to crisis response management, is widely recognized (Barnes & Oloruntoba, 2005). Radio Frequency IDentification (RFID) and sensing technologies are considered as promising technology assets for detection of possible threats in goods maritime transport (Tsilingiris, Psaraftis, & Lyridis, 2007). For this reason their adoption in container transport and handling monitoring applications is gaining interest in both industrial and academic communities.

In this context, SITMAR aims at experimenting the use of RFID, sensors, event processing and workflow technologies in a dangerous goods maritime transport scenario in order to timely detect and handle possible security and safety risks.

The SITMAR prototype design is composed of: a distributed sensing infrastructure for monitoring environmental conditions in containers and ship storage areas; a distributed service-oriented middleware which collects and processes monitoring data and executes business processes to provide monitoring, information and alerting services to interested users. The middleware is deployed on local nodes (e.g., on board ships) and on a central node collecting data from registered ship nodes. The LiSEP engine is deployed as a component of the distributed service-oriented middleware. RFID and sensor readings are delivered to the LiSEP engine, which may be properly configured to perform filtering and transformation actions on Raw Events and to detect event patterns modeling possible anomalous conditions.

7.1. Proof-of-Concept implementation

We implemented a Proof of Concept (PoC) of a monitoring application to perform a preliminary validation of LiSEP capabilities in the reference scenario.

The Demo includes two main components:

- an Event Producer that simulates the readings of RFID and sensors in a ship storage area. The Event Producer can be configured to simulate the periodical acquisition of temperature, humidity and pressure sensor readings (e.g., every thirty minutes) and to trigger single events for simulating container openings and light variations inside the container. These kind of events may be used to detect different types of threats, such as door opening, fire, chemical reaction, container break, as discussed by Tsilingiris et al. (2007);

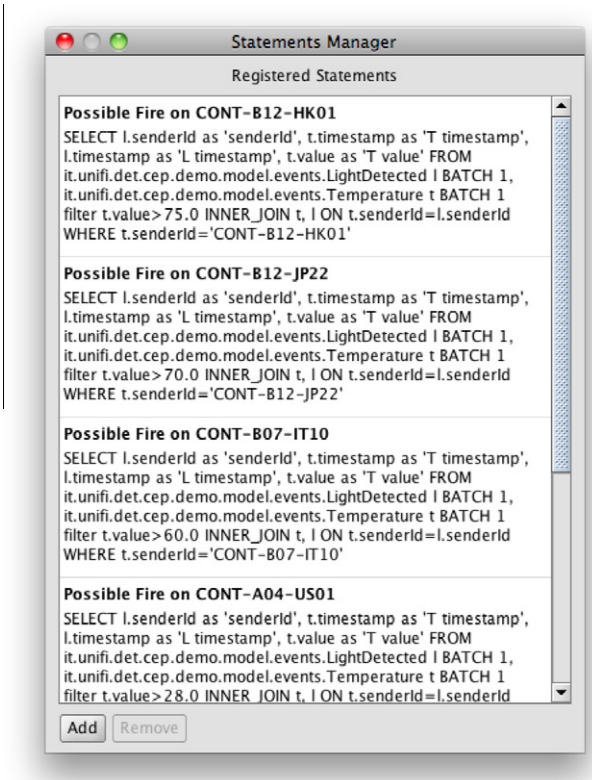

Fig. 8. Proof of Concept – administration view.

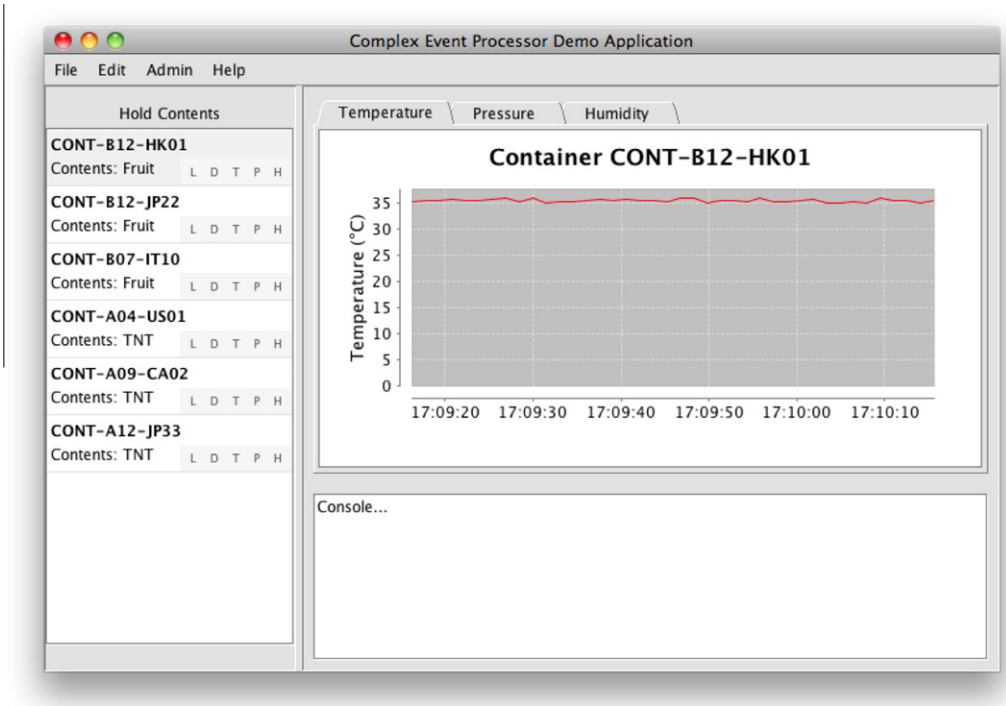

Fig. 9. Proof of Concept – monitoring view.

- a Graphical User Interface (GUI) that provides users with administration and monitoring views. The administration view enables end users to manage EPL statements, while the monitoring view shows a graphical representation of the simulated events and notifies possible anomalies detected by LiSEP.

7.1.1. Event Model and EPL statements

We defined an Event Model and we configured LiSEP with a set of registered EPL statements for enabling proper processing of events delivered by the Event Producer component.

As reported in Table 1, the Event Model is structured in a three-layered hierarchy: Raw Events produced by sensors, which contain low-level data that have no meaning for the application-level; Domain Entity Events which translate raw data in the status change of entities in the application domain (i.e. the container); Warning Events that represent possible critical situations caused by abnormal conditions in the status of domain entities.

Raw Events produced by RFID and sensors are listed hereafter.

- Sensor Reading: a Sensor Reading event contains sensor measurements. This event type contains the following attributes: the RFID identifier, measurement values (temperature, humidity, pressure), a timestamp, an event identifier.

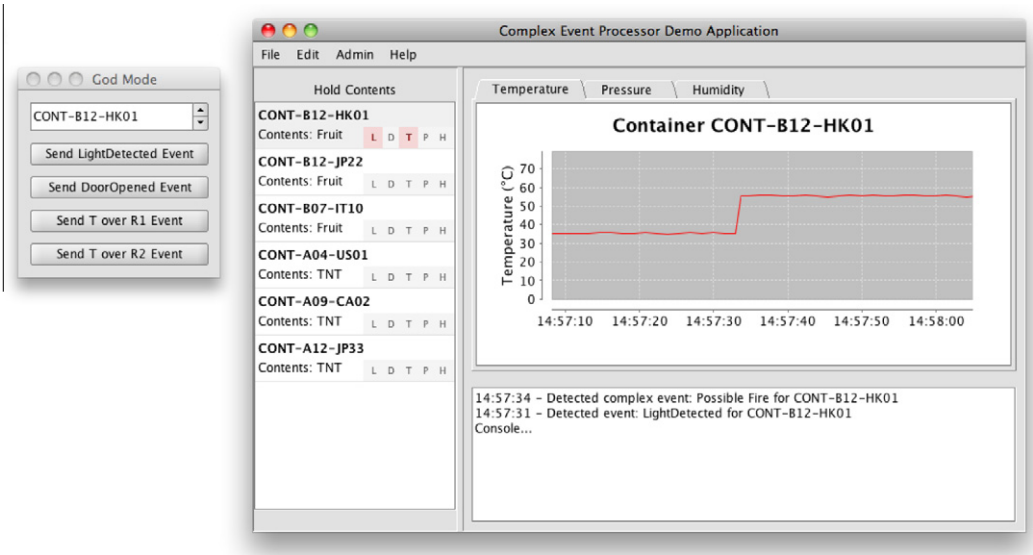

Fig. 10. Proof of Concept – monitoring view for a Warning Event.

- Occurrence Events: these events are triggered when specific boolean conditions are matched. In this case study occurrence events may detect a container opening (Opening event) or light inside the container (Light Detection event).

We defined a filtering statement for selecting "Sensor Reading Events" from the stream of input events.

```
SELECT o.tagId as 'tagId',
       o.eventId as 'eventId',
       o.temperature as 'temperature',
       o.pressure as 'pressure',
       o.humidity as 'humidity'
  FROM model.events.TagOutput o BATCH 1
```

LISEP routes matching events to a registered listener component which is in charge of deleting possible multiple copies of the same Sensor Reading event and translating Raw Events into higher-level events related to entities handled in the application. More specifically, sensor readings are mapped into proper updates of container parameters (Container Update event). As shown in Table 1, a Container Update event includes the following attributes: the container identifier, a timestamp, a parameter indicating the type of update [Temperature | Pressure | Humidity | Opening | Light Detection] and the corresponding value.

Specific statements may be registered in order to detect anomalies and possible threats by analyzing and correlating events produced by RFID and sensors. For every container or groups of container, users may specify a set of thresholds that relate measured values with different warning levels. When one or more Container Update events fall out of these thresholds, registered listeners fire corresponding Warning Events. The structure of Warning Events comprises the following minimal attributes: container identifier, timestamp, warning level, warning type. More complex event detection logic may be implemented by correlating different event types in order to detect possible threats. Hereafter an example of a statement expressing a pattern event for detecting the co-occurrence of a Light Detection event and a Temperature value over a specified threshold within a predefined time interval (e.g., 10,000 ms). This event pattern might be used to originate a Warning Event for a Possible Fire threat.

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

7.1.2. Graphical User Interface

The GUI-based application provides users with basic administration and monitoring features for validating the adoption of LiSEP capabilities in the goods maritime transport scenario. It is a Java-based application designed according to the Model-View-Controller (MVC) pattern. Fig. 8 shows a snapshot of the administration view which allows users to visualize and delete registered EPL statements, and insert new ones.

Raw Events simulated by the Event Producer and higher-level events produced by LiSEP processing are consumed by listener components to update the data model and the corresponding widgets of the graphical interface. Fig. 9 depicts the monitoring view. Users may select a specific container on the left menu to view the graphical representation of sensor parameters for the target container.

The application offers also a menu for simulating a set of warning scenarios by triggering some predefined Raw Events (panel on the left side of Fig. 10). The window on the right side of Fig. 10 shows how the application notifies the occurrence of a Warning Event caused by a temperature increase and a light detection occurrence in a container.

8. Conclusions

In this paper, we described the design and implementation of a lightweight and extensible Complex Event Processing engine, called LiSEP.

LiSEP is based on a layered architecture, whose design clearly separates the core logic devoted to event processing from low-level thread management handled by the SEDA framework. The design has been driven by the principle of minimizing dependency on external software components. Consequently, LiSEP depends solely on the Java Standard Edition libraries, thus minimizing deployment requirements. Moreover, the LiSEP logic is strictly focused on core event processing, thus resulting in a lightweight and minimal implementation. The overall JAR package is limited to 360 Kbytes. The adopted modular and layered design promotes code maintainability, while favoring the processor extensibility to accommodate possible new general-purpose or application-specific processing features (such as arithmetic and statistical computations) according to the specific requirements of the target scenario.

The LiSEP Event Processing Language provides an expressive and user-friendly querying modeling capability, based on an SQL-like syntax.

We also described an optimization technique for minimizing message exchange among LiSEP stages and we showed performance measurements. Testing results demonstrated that LiSEP performance scales well with hardware resources. The adopted design and implementation choices pave the way for possible future optimizations. By relying on the SEDA framework, we plan to extend the processor with self-tuning policies for automatically adjusting system configuration parameters for performance and resource management optimization.

We also reported on current research activities in the carrying out of a case study in a dangerous goods monitoring scenario. Through a Proof of Concept implementation we demonstrated how LiSEP can be easily configured in order to filter and process Raw Events produced by RFID and sensor measurements and route relevant and meaningful events to interested application components. Future extensions of the prototype will include the specification of further event correlation patterns (e.g., possible fire threat detected in two or more containers located in the same area) to improve and refine the detection of possible threats and the accuracy of the resulting situation awareness.

Acknowledgments

The authors thank Mr. Luca Capannesi for his technical support.

References

Adaikkalavan, R., & Chakravarthy, S. (2006). Snoopib: Interval-based event specification and detection for active databases. *Data & Knowledge Engineering, 59*(1), 139–165.

Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal, 15*, 121–142. URL http://dx.doi.org/10.1007/s00778-004-0147-z.

Baker, N., Zafar, M., Moltchanov, B., & Knappmeyer, M. (2009). EContext-Aware Systems and Implications for Future Internet. In *Future Internet conference and technical workshop*. URL http://doi.acm.org/10.1145/1266894.1266934

Barga, R., & Caituiro-Monge, H. (2006). Event correlation and pattern detection in CEDR. In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, & S. Müller, et al. (Eds.), *Current trends in database technology – EDBT 2006. Lecture Notes in Computer Science* (Vol. 4254, pp. 919–930,). Berlin/ Heidelberg: Springer. 10.1007/11896548_70.

Barnes, P. H., & Oloruntoba, R. (2005). Assurance of security in maritime supply chains: Conceptual issues of vulnerability and crisis management. *Journal of International Management, 11*(4), 519–540.

Chandy, K. M. (2005). Sense and respond systems. In *International CMG conference*. pp. 59–66.

Chen, H., Chou, P. B., Cohen, N. H., & Duri, S. (2007). Extending SOA/MDD to Sensors and Actuators for Sense-and-Respond Business Processes. In *E-Business Engineering, IEEE International Conference on 0*, pp. 54–61.

Drools, The Business Logic integration Platform. (2011). Official website. URL http://www.jboss.org/drools

Dunkel, J., Fernandez, A., Ortiz, R., & Ossowski, S. (2008). Event-Driven Architecture for Decision Support in Traffic Management Systems. In *11th International IEEE conference on intelligent transportation systems*, 2008. ITSC 2008. pp. 7–3.

Eckert, M. (2008). Complex Event Processing with XChangeEQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events. Phd thesis, University of Munich (LMU), December 2008.

EsperTech, Event Stream Intelligence, 2011. Official website. URL http://www.espertech.com/products/esper.php

Estrin, D., Chandy, K. M., Young, R. M., Smarr, L., Odlyzko, A., & Clark, D., et al. (2010). Internet Predictions. *IEEE Internet Computing* 14, pp. 12–42.

Haller, S., Karnouskos, S., & Schroth, C. (2009). The internet of things in an enterprise context. In J. Domingue, D. Fensel, & P. Traverso (Eds.), *Future Internet – FIS 2008. Lecture Notes in Computer Science* (Vol. 5468, pp. 14–28). Berlin/ Heidelberg: Springer. doi:10.1007/978-3-642-00985-3_2.

Hermosillo, G., & Seinturier, L., Duchien, L. (2010). Using Complex Event Processing for Dynamic Business Process Adaptation. In *Proceedings of the 2010 IEEE international conference on services computing SCC '10*. IEEE Computer Society, Washington, DC, USA, pp. 466–473.

Hernández-Muñoz, J., Vercher, J., Muñoz, L., Galache, J., Presser, M., Gómez, Hernández L., et al. (2011). Smart cities at the forefront of the future internet. In J. Domingue, A. Galis, A. Gavras, T. Zahariadis, D. Lambert, & F. Cleary, et al. (Eds.), *The future internet. Lecture Notes in Computer Science* (Vol. 6656, pp. 447–462). Berlin/ Heidelberg: Springer. doi:10.1007/978-3-642-20898-0_32.

Hinze, A., Sachs, K., & Buchmann, A. (2009). Event-based applications and enabling technologies. In *Proceedings of the third ACM international conference on distributed event-based systems DEBS '09*, ACM, New York, NY, USA, pp. 1:1–1:15.

Kavelar, A., Obweger, H., Schiefer, J., & Suntinger, M. (2010). Web-Based Decision Making for Complex Event Processing Systems. Services, IEEE Congress on 0, 453–458. URL http://doi.ieeecomputersociety.org/bib4.

Kim, Y., Yoo, J.-W., & Park, N. (2006). RFID Based Business Process Automation for Harbor Operations in Container Depot. unpublished. URL http://iseresearch.eng.wayne.edu/2006/Proceedings2006/JungWoon.pdf

Ku, T., Zhu, Y., & Hu, K. (2008). A Novel Complex Event Mining Network for RFID-Enable Supply Chain Information Security. In *International Conference on Computational Intelligence and Security CIS '08*, Dec 2008. Vol. 1., pp. 516 –521.

Leavitt, N. (2009). Complex-event processing poised for growth. *Computer, 42*, 17–20.

Liu, J., & Zhao, F. (2010). Composing Semantic Services in Open Sensor-Rich Environments.

Luckham, D., Schulte, R., & Luglio. (2008). Event Processing Glossary - Version 1.1. Event Processing Technical Society.

Luckham, D.C., & Frasca, B. (1998). Complex Event Processing in Distributed Systems. Techical Report.

Mouttham, A., Peyton, L., Eze, B., & Saddik, A. (2009). Event-driven data integration for personal health monitoring. *Journal of Emerging Technologies in Web Intelligence, 1*(2).

Parlanti, D., Paganelli, F., & Giuli, D. (2010). A service-oriented approach for network-centric data integration and its application to maritime surveillance. *IEEE Systems Journal* (99), 1.

Schiefer, J., Rozsnyai, S., Rauscher, C., & Saurer, G. (2007). Event-driven rules for sensing and responding to business situations. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems DEBS '07*, ACM, New York, NY, USA, pp. 198–205.

Tsilingiris, P. S., Psaraftis, H. N., & Lyridis, D. V. (2007). RFID-enabled Innovative Solutions Promote Container Security. In *International symposium on maritime safety, security and environmental protection (SSE07)*, Athens, Greece.

van de Voort, M., O'Brien, K. A., Rahman, A., & Valeri, L. (2003). ''Seacurity'': Improving the Security of the Global Sea-Container Shipping System.

Voisard, A., Ziekow, H., 2010. Designing Sensor-Based Event Processing Infrastructures. In *Hawaii International Conference on System Sciences 0*, pp. 1–10.

Wei, M., Ari, I., Li, J., & Dekhil, M. (2007). ReCEPtor: Sensing Complex Events in Data Streams for Service-Oriented Architectures. Technical report hpl-2007-176, HP Laboratories Palo Alto.

Welsh, M., Culler, D., & Brewer, E. (2001). SEDA: An architecture for well-conditioned, scalable internet services. *SIGOPS Operation Systems Review, 35*, 230–243.

Yao, W., Chu, C.-H., & Li, Z. (2011). Leveraging complex event processing for smart hospitals using RFID. *Journal of Network and Computer Applications, 34*(3), 799–810. rFID Technology, Systems, and Application.

Zang, C., Fan, Y., & Liu, R. (2008). Architecture, implementation and application of complex event processing in enterprise information systems based on rfid. *Information Systems Frontiers, 10*, 543–553. http://dx.doi.org/10.1007/s10796-008-9109-0.