Eventlets: Components for the Integration of Event Streams with SOA

Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, Alejandro Buchmann
TU Darmstadt
Email: *lastname*@dvs.tu-darmstadt.de

***Abstract*—Cyber physical systems (CPS) react to changes in the environment and have become widely adopted in many domains. One key functionality to achieve this reactivity is the processing of event streams. To profit from this reactive behavior in service-oriented architectures (SOA), event stream processing needs to be encapsulated in a service-like manner. We thus introduce the concept of event applets, in short *Eventlets*, to provide developers and architects alike with a generic and reusable component model for encapsulating event stream processing logic. Eventlets have a managed lifecycle and are activated automatically upon arrival of appropriate events. We introduce our distributed Eventlet middleware architecture and implementation based on industry-strength message-oriented middleware. Our evaluation shows that Eventlets simplify the development of reactive components and that they can compete with traditional event processing approaches in terms of performance. Eventlets enable easy distribution of event stream processing components and are a suitable foundation for scalable applications that combine SOA with CPS.**

I. Introduction

In cyber physical systems (CPS) the pervasive use of mobile devices and sensors provides real-time information of small granularity in form of events. These events reflect changes in the real world and foster reactive behavior in software systems. Patient monitoring, smart home environments as well as traffic management systems are typical examples [1], [2], [3].

To integrate this reactive behavior in service-oriented architectures (SOA) appropriate mechanisms for the processing of events are required. Although there are several approaches to integrate event processing with SOA, e.g., event-driven SOA, these are usually restricted to single low-level events for inter-component communication [4]. In CPS, however, events are pushed into software components in form of *event streams* [5]. This push-based approach differs from the invocation-based (pull) approach SOAs were originally designed for. Thus, the abstraction paradigm of services in SOA still lacks an equivalent for the processing of event streams. In this paper we present a component model to encapsulate this processing of event streams in an intuitive, scalable, and distributed way. We refer to these new components as *Eventlets*. They are containers for generic reactive application logic referred to as *tasks*. Figure 1(a) shows that a generic task is defined by actions that are applied to different real-world entities and that are triggered by events. For each individual entity, a task instance has to be maintained; actions are generic and automatically applied per instance. Eventlets are designed to exploit the push-based nature of event streams. They are invoked implicitly and distributed automatically upon arrival of appropriate events and follow a lifecycle. Eventlets are managed by a middleware infrastructure that provides these automatisms for registration, instantiation, distribution, and lifecycle management. Like services, Eventlets encapsulate application logic and run inside a SOA-like infrastructure. This allows a similar development process for Eventlets and services; application logic is broken down in components that can be integrated and interact with each other.

As contributions of this paper, we

- introduce *Eventlets* as application logic containers for tasks on event streams;
- present a distributed architecture and implementation of an Eventlet middleware; and
- evaluate our implementation and show scalability and software engineering advantages with respect to traditional event stream processing approaches. We use the Esper complex event processing engine to show the advantages of Eventlets in terms of scalability, distribution, and efficient development.

The remainder of this paper is structured as follows: we present a motivating scenario that illustrates the application of Eventlets in Section II and discuss related work in Section III. In Section IV we introduce Eventlets from a software developer's perspective and present the distributed Eventlet middleware architecture. Our implementation is described in Section V and evaluated in Section VI. In Sections VII and VIII we summarize our contributions and present future work.

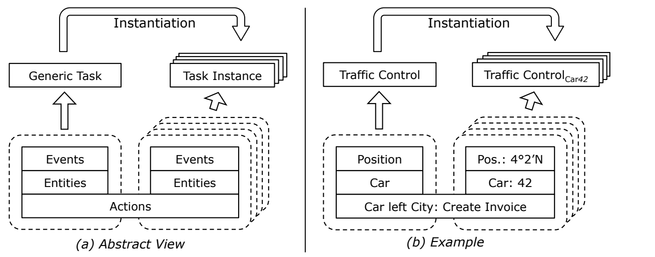


Fig. 1: Generic Task Model: Abstract view (a) and traffic control example (b). In (b), an instance exists per car.

II. Motivating Scenario: Traffic Management

Managing the increasing volume of traffic in large cities is a challenging task. To avoid gridlock and to share infrastructure-costs for highways or tunnels, cities rely on toll systems to charge vehicles entering certain areas. Vehicles are registered when they enter or exit certain zones and cities often have different rates for different areas, times or, kinds of traffic. An IT system for traffic management has to keep track of each car, its status, and location. Norway's AutoPASS system, for example, handles more than 2.1 million entities across Norway[1].

The process of toll collection consists of a dynamic sensing part and a standardized processing part: Cars have to be detected and tracked (sensing), toll rates and billing information have to be correlated with the detected movements (processing). These toll systems are often realized as cyber physical systems for the sensing and a SOA for the processing part. Vehicles are continuously detected at checkpoints throughout the city. The detections are represented by event streams, which are then used for toll processing.

While traditional event stream processing techniques can handle high volumes of events, they lack an intuitive abstraction mechanism designed for scalability on the architectural level that enables easy integration with SOAs. With Eventlets however, this integration is intuitive and the resulting application inherently scalable. The application logic is the same for each vehicle: detection events have to be recorded and toll has to be calculated. This generic task can be modeled with an Eventlet as shown in Figure 1(b). For each vehicle crossing the city limits an Eventlet instance is created automatically. During the instantiation further information, such as the toll category for the corresponding license plate, is retrieved by a SOA service invocation. Each Eventlet instance receives all detection events for its corresponding vehicle only. When a vehicle leaves the city the associated Eventlet instance performs the billing by invoking the invoice service and shuts itself down.

III. Related Work

We introduce Eventlets as containers for event stream processing on the same level of abstraction as services in SOA. They integrate SOA with event stream processing which is necessary to build reactive software systems [6]. We compare Eventlets with other existing concepts and systems related to event stream processing functionality. Due to limited space, we only mention related work exemplary for the different approaches and relevant for this work.

Architectures and models for push-based reactive software systems have been addressed in previous work; in [7] the authors present a survey where distributed event-based architectures are described from different points of view. In [8] Blanco et al. introduce a metamodel for distributed EBS based on a peer-to-peer system. Their system shares the idea of reactive components with Eventlets. However, they do not introduce a high level abstraction with a generic view on tasks. In [9] a taxonomy of distributed event-based systems (EBS) is given. This paper states that an EBS requires an event model and an event service. Many event services are distributed publish/subscribe (pub/sub) systems implementing different event models and broker network structures. Examples of content-based distributed pub/sub middleware are Hermes [10], PADRES [11], Rebecca [12], and JEDI [13]. With our approach of Eventlets we do not introduce another event model and event service. We rather build upon a, potentially distributed, pub/sub system and the respective event model. Eventlets and Eventlet middleware build a layer on top of the event service and thus the above-mentioned systems can be integrated with our idea. However, the underlying event model is not transparent to the developer since event handling inside Eventlets needs to comply with the underlying event model.

Another area of related research on push-based approaches is complex event processing (CEP). The distinction between distributed pub/sub systems and CEP, however, is ambiguous. Filter expressions used to subscribe to events require CEP to some extent, e.g., when evaluating events with respect to subscription expressions. Examples for CEP engines are Esper [14], Cayuga [15], and SASE [16]. CEP engines are typically integrated into an event-based infrastructure as components that subscribe to events. With respect to Eventlets, CEP occurs at different points. Eventlets subscribe to events using filter expressions; the complexity of those filter expressions and whether they involve CEP, depends on the underlying event dissemination infrastructure. Further, Eventlets can implement CEP functions on their own as application logic. But Eventlets can also integrate existing CEP solutions as we will show in our evaluation; with Eventlets the distribution of complex event queries can be realized easily.

Event-condition-action rules (ECA) are another mechanism to express reactive behavior [17], [18]. With dynamic ECA rule replication and generic rules it is possible to generalize the action part of ECA rules. However, this requires management components for rule replication and interpretation of generic expressions. In addition to pure reactive behavior expressed with ECA rules, Eventlets provide mechanisms for lifecycle management. Upon instantiation and removal of Eventlet instances application logic is executed. Further, the validity of Eventlet instances is checked during runtime.

Eventlets complement SOA services as building blocks for push-based architectures. SOA services encapsulate generic actions with respect to entities and allow for dynamic integration of system components. However, the approach in SOA is pull-based; an application requiring functionality invokes the respective service and waits for the data. Although in next-generation SOA the idea of events is integrated to realize reactive services (event-driven SOA) [19], [4], it still builds upon services originally designed with a different mind-set than event-based components. Thus we believe that encapsulating application logic with respect to entities' event streams is the more natural approach.

Eventlets are also related to mobile software agents [20].

[1]http://www.q-free.com/solutions/we-did-oslo/

In agent-based systems software components (agents) fulfill tasks autonomously. For example, Bromuri et al. present an approach with distributed agents reacting on events [21]. In their system agents are autonomous proactive components using events to coordinate the overall workflow across all agents. When comparing agent-based systems with Eventlets, a single software agent instance is similar to an Eventlet instance. The Eventlet concept, however, is different from agent-based systems; Eventlet instantiation is event-driven and dynamic depending on the actual events. Eventlets are not designed as autonomous units. They rely on Eventlet middleware components for creation and management.

Eventlets are language-agnostic. Language-specific approaches integrate event processing capabilities into programming languages. In EventJava [22], distributed event correlation is seamlessly integrated with methods. In EScala [23], events can be used in an aspect-oriented way in the source code. These extensions make events first class citizens in programming languages and provide event-based functionality without integrating, e.g., CEP engines. Further, dedicated programming languages are available to describe behavior of software components in a generic way, e.g., the Act3 actor language [24]. Compared to Eventlets however, the conceptual goal differs. Eventlets, as a generic paradigm, are designed to be independent from a certain programming language and help to express event-based reactive functionality in an abstract way. Desired Eventlet behavior can be expressed using standard programming languages without modifications.

In terms of general software engineering research, Eventlets are software components. Comparing Eventlets with software component models presented in [25] shows that Eventlets share technical properties with Enterprise Java Beans (EJB), especially Message-Driven Beans (MDB). However, EJB are not the best fit for scalable event stream processing in terms of performance and ease of development as we will show in our evaluation. In EJB, messages were introduced as asynchronous inter-component communication mechanism and MDB statically define their subscriptions at compile time. Eventlets in contrast issue subscriptions dynamically at runtime depending on the event stream.

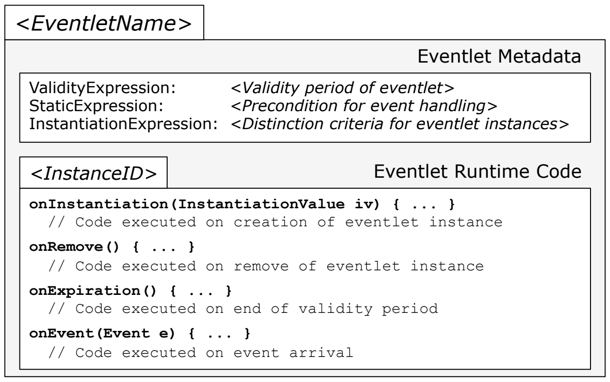
IV. Eventlet Structure

In the following we define the structure of Eventlets. We distinguish between the developers' view in Section IV-A and the architectural view in Sections IV-B. The developers' view describes the structure of Eventlets relevant to developers. The architectural view presents the Eventlet middleware necessary to execute, distribute, and manage Eventlets. The mechanisms inside the Eventlet middleware are transparent to developers.

For event dissemination we rely on a messaging middleware following the pub/sub paradigm [26]. Event consumers issue subscriptions and define filters to receive events of interest. We refer to this event delivery infrastructure as *event bus*. We assume that developers have knowledge about potentially available events, e.g., by means of an event type registry or advertisements published by event producers.

A. Eventlet Prototypes

Developers write *Eventlet prototypes* that contain the application logic for event stream processing. Eventlet prototypes contain meta data that determine when to create Eventlet instances at runtime. Eventlet prototypes follow a certain structure as shown in Figure 2; we distinguish between Eventlet *metadata* and Eventlet *runtime code*. Eventlet prototypes are identified by a unique name. Further, to identify Eventlet instances at runtime, an ID per instance is assigned by the middleware.



ValidityExpression:	*<Validity period of eventlet>*
StaticExpression:	*<Precondition for event handling>*
InstantiationExpression:	*<Distinction criteria for eventlet instances>*

```
onInstantiation(InstantiationValue iv) { ... }
  // Code executed on creation of eventlet instance
onRemove() { ... }
  // Code executed on remove of eventlet instance
onExpiration() { ... }
  // Code executed on end of validity period
onEvent(Event e) { ... }
  // Code executed on event arrival
```

Fig. 2: Structure of an Eventlet Prototype

1) Eventlet Metadata: Eventlet metadata is required by the middleware to instantiate and manage Eventlets appropriately. The metadata is shared amongst all instances of an Eventlet prototype.

a) Validity Expression: Eventlet instances can neither rely on a constant stream of events nor that event producers leaving the system inform subscribers. This requires to specify a validity condition to avoid that Eventlet instances remain active indefinitely. Examples for validity are a timeout, an insufficient event rate, or an expiration time/date.

b) Static Expression: An Eventlet might be interested only in certain types of events. The static expression is a precondition to identify the event stream relevant for an Eventlet prototype and all its instances. The static expression can be seen as a filter applied at subscription level; if an event does not match, no further processing is required. We chose the term *static* to indicate that the resulting filter is equal for all Eventlet instances of the same Eventlet prototype. An example for a static expression in the style of XPath is: `/event/type == PositionEvent`.

c) Instantiation Expression: The instantiation expression is the distinction criterion between Eventlet instances. It partitions the event stream associated with an Eventlet prototype into sub-streams. Each sub-stream contains events with respect to a certain grouping attribute value, e.g., a room number. At runtime, Eventlet instances exist for all grouping attribute values present in the Eventlet prototype event stream. An example for an instantiation expression in the style of XPath is: `/event/carID`; Eventlet instances are then created automatically for specific cars. The specific instantiation value, e.g., car number 42, is passed to the Eventlet instance.

2) Eventlet Runtime Code: Eventlets follow a lifecycle. Eventlet prototypes are registered with the middleware and Eventlet instances are created dynamically when required. The instances can expire, can be stopped, or removed. An Eventlet prototype has to implement four methods that contain the application logic for the Eventlet lifecycle management and for the handling of incoming events.

The `onInstantiation` method contains the code, which is executed when an Eventlet instantiation is triggered by the Eventlet middleware. Instantiation code is optional, depending on the use case. An example for instantiation code is opening a database connection to retrieve additional data or issuing a service call. The Eventlet middleware can trigger the removal of an Eventlet for various reasons, e.g., upon user request. The `onRemove` method is then responsible for a clean shutdown with respect to the Eventlet application logic. An example is persisting data for later reuse. If an Eventlet instance expires according to the validity expression, the `onExpiration` method is executed. The method can be used by developers to react appropriately on expirations. The core functionality of an Eventlet is the application logic executed upon event arrival. The corresponding code is located in the `onEvent` method.

B. Eventlet Middleware

The Eventlet middleware provides an interface to the developer for the registration of Eventlet prototypes. It provides automatisms to perform the instantiation, distribution, and management of Eventlet instances. The basic Eventlet middleware architecture along with relations amongst their components is depicted in Figure 3.

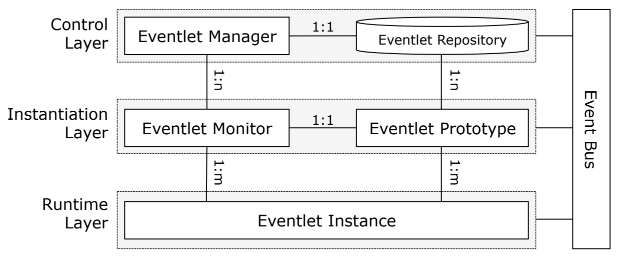


Fig. 3: Eventlet Middleware Components

Eventlet manager and *repository* are the core components of an Eventlet middleware and account for the control layer. *Eventlet monitors* and *prototypes* constitute the instantiation layer. *Eventlet instances*, created by Eventlet monitors, reside in the runtime layer. During runtime, Eventlet middleware components are connected to the event bus. The components are described in the following sections.

1) Event Bus: The event bus is the core mechanism to transport events from producers to consumers. Typically, the event bus follows a pub/sub paradigm to decouple event producers and consumers [27], [26]. The static and instantiation expressions of Eventlet prototypes are defined on event content. Thus, the pub/sub mechanism used by the Eventlet middleware components must be sensitive to the event content. This can be achieved with dedicated attributes in the event header or with a full content-based subscription model. The Eventlet middleware can be adapted to support both mechanisms.

2) Eventlet Manager: The Eventlet manager is the main component of an Eventlet middleware. It provides an interface to the developer for the registration and management of Eventlet prototypes. Upon registration of Eventlet prototypes the Eventlet Manager creates the Eventlet monitor.

3) Eventlet Monitor: Eventlet monitors are responsible for monitoring events and instantiating new Eventlets. Developers construct Eventlet prototypes corresponding to the structure shown in Figure 2. One Eventlet monitor is associated with each Eventlet prototype. Eventlet monitors support different strategies for the creation of new Eventlet instances: stream-controlled instantiation and manually-controlled instantiation. Upon the registration of an Eventlet prototype a stream-controlled Eventlet monitor is created and subscribes to all events of potential interest using the static expression of their associated Eventlet prototypes. A stream-controlled monitor receives all events for which Eventlet instances have to exist. At incoming events it evaluates the instantiation expression of the Eventlet prototype and performs a lookup to check whether an Eventlet instance for the results of the expression evaluation is available. If no Eventlet instance is available, the Eventlet monitor triggers the instantiation and passes the instantiation value to the instance. A manually-controlled monitor creates new Eventlet instances upon user requests only and monitors already running Eventlet instances. Manual instantiation requires users to know the instantiation values of Eventlet instances that should be created.

4) Eventlet Repository: The Eventlet repository holds Eventlet prototypes and their metadata. The repository allows the Eventlet manager to create Eventlet monitors and Eventlet instances on different nodes. The repository can be replicated for scalability.

5) Eventlet Prototypes and Instances: Eventlet monitors trigger the instantiation of Eventlets. During the instantiation process the newly created instance issues a subscription using a filter constructed corresponding to the static expression and the instantiation value. The `onInstantiation` method is invoked; afterwards the Eventlet instance reacts autonomously. Upon event arrival the `onEvent` method is called and gives developers access to the arrived event. Each Eventlet instance evaluates its validity expression during runtime to decide whether the validity condition is violated. Depending on the actual validity expression, the evaluation is triggered by a timer or by an incoming event. In case of a validity condition violation the `onExpiration` method is called to handle the expiration.

V. Implementation

In this section we present our distributed Eventlet middleware. We use state of the art software components and support events represented with attribute/value (att/val) pairs and XML.

A. *Distributed Architecture*

Our Eventlet middleware is implemented in Java and uses a Java Message Service (JMS) broker as event bus [28]. JMS is the de-facto industry standard for asynchronous messaging; messages are exchanged via a JMS broker network, e.g., IBM WebSphere MQ or Apache ActiveMQ. JMS is also the technology used in many enterprise service buses (ESB), e.g., Apache ServiceMix uses ActiveMQ as ESB. This allows the Eventlet middleware to leverage already deployed technology.

The pub/sub functionality required for Eventlets is provided by *JMS topics*. JMS provides basic content-based subscription capabilities via message properties. A message property is an att/val pair in the message header. When issuing a subscription a consumer can use a SQL-like statement (message selector) as a filter on message properties. The JMS broker ensures that only messages with matching properties reach the consumer. We use message properties and selectors to route events to Eventlet instances. We trade more elaborated content-based subscription capabilities of alternative pub/sub systems, e.g., filter subsumption, for the use of JMS. JMS has the advantage of being a standardized industry-strength API that allows the use of well-tested JMS brokers. We use a single topic as event bus; all event producers publish events to this topic.

The Eventlet middleware is designed for scalability as Eventlet monitors and Eventlet instances can run distributed across multiple machines. The architecture is shown in Figure 4.

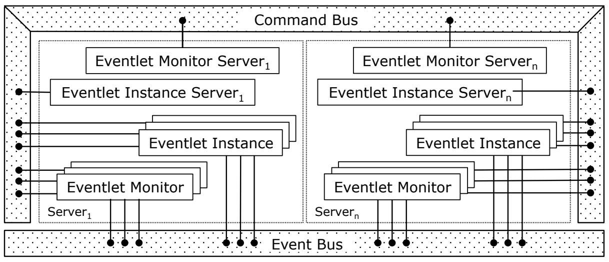

Command Bus
Eventlet Monitor Server$_1$
Eventlet Instance Server$_1$
Eventlet Instance
Eventlet Monitor
Server$_1$
Eventlet Monitor Server$_n$
Eventlet Instance Server$_n$
Eventlet Instance
Eventlet Monitor
Server$_n$
Event Bus

Fig. 4: Distributed Eventlet Middleware

To achieve distribution, queue- and topic-based inter-component communication is integrated in the middleware. Just like the event bus transports events from producers to consumers, the Eventlet middleware uses a command bus for asynchronous and decoupled communication between the components. It is possible to send commands to:

- a specific Eventlet instance;
- all Eventlet instances, monitors, and instance servers;
- all Eventlet monitors;
- all active Eventlet instances, not only to instances of a certain Eventlet prototype;
- a specific Eventlet monitor; and
- all Eventlet instances of a prototype.

To distribute Eventlet monitors and Eventlet prototype instances, Eventlet monitor servers and Eventlet instance servers are started on each machine participating in the middleware system. These servers form the Eventlet manager and connect to queues on which they listen for commands. In the current implementation queues deliver commands round-robin to all connected servers.

Our middleware supports different commands for inter-component interaction and lifecycle management of Eventlets. With the registration of an Eventlet prototype an EVENTLET MONITOR DISPATCH command is sent. Eventlet monitors send EVENTLET DISPATCH commands to trigger the creation of Eventlet instances. The middleware also supports Eventlet lifecycle management commands for Eventlet instances and Eventlet monitors: STOP, REMOVE, and RESUME. Stop commands pause Eventlet instances; instances remain active and keep state but do not receive events anymore and the validity check is disabled. When the stream-controlled instantiation policy is active, stopped Eventlet instances are not automatically reactivated upon matching events. Stopping an Eventlet monitor causes all associated Eventlet instances to be stopped. Resume commands reactivate stopped Eventlet instances and monitors. Remove commands delete Eventlet instances and trigger the `onRemove` method call. Removing an Eventlet monitor causes all associated Eventlet instances to be removed as well. Removed Eventlet instances are recreated when the Eventlet monitor is still active and matching events arrive again.

B. *Event and Expression Representation*

Our implementation supports att/val pairs as well as XML as representation for events [29]. For XML events static and instantiation expressions of Eventlets are expressed in XPath. For att/val events, the static expression is a JMS message selector and the instantiation expression is an attribute name.

In our solution Eventlet prototypes and monitors are represented by Java classes. An Eventlet prototype inherits functionality from its superclass for dynamic subscriptions and evaluation of the validity expression. The superclass further implements the JMS Message Listener interface to react on incoming messages. All of this is transparent to the Eventlet developer who only needs to implement the four core methods and to provide the appropriate static and instantiation expressions as described in detail in Section IV-A.

C. *Eventlet Instantiation*

Eventlet prototypes, identified by a name, are registered with the middleware. Upon the registration a command is sent to the Eventlet monitor dispatch queue. The Eventlet monitor server receiving this command starts the Eventlet monitor. We use Java Reflection for this instantiation process to dynamically identify Eventlet prototype classes at runtime. When an Eventlet monitor detects the need to create an Eventlet instance, it sends a command to the Eventlet instance dispatch queue. The Eventlet instance server receiving this command then retrieves the required classes from the repository and instantiates the corresponding Eventlet. Currently, Eventlet instance servers receive dispatch commands round-robin. For basic load balancing a physical machine can start multiple Eventlet instance servers. More elaborate Eventlet instance placement and load balancing strategies are part of our future work. A newly created Eventlet instance issues a subscription for relevant events based upon the static expression and the instantiation value. In addition a task is started to periodically check the validity expression. Our implementation currently supports timeouts.

VI. Evaluation

We evaluated our system and compared it with the Esper 4.3.0 complex event processing (CEP) engine. CEP is one of the most common event stream processing applications. We show: (1) that the distribution provided by Eventlets is necessary for scalable event stream processing; (2) that the overhead introduced by the Eventlet middleware compared to a traditional CEP solution is small; and (3) that the programming model of Eventlets simplifies the development of distributed event stream processing components. We used Apache ActiveMQ 5.5.1 as JMS broker; the test environment details are shown in Figure 5.

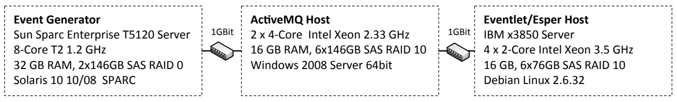

Event Generator
Sun Sparc Enterprise T5120 Server
8-Core T2 1.2 GHz
32 GB RAM, 2x146GB SAS RAID 0
Solaris 10 10/08 SPARC
1GBit
ActiveMQ Host
2 x 4-Core Intel Xeon 2.33 GHz
16 GB RAM, 6x146GB SAS RAID 10
Windows 2008 Server 64bit
1GBit
Eventlet/Esper Host
IBM x3850 Server
4 x 2-Core Intel Xeon 3.5 GHz
16 GB, 6x76GB SAS RAID 10
Debian Linux 2.6.32

Fig. 5: Test Environment

As test case we used a typical CEP setup where event processing components receive events via a broker. Events of multiple types and sources are sent to the event bus (JMS Topic). Depending on the event type and a threshold on an event attribute, a counter is increased. Events are represented in XML and event producers generate two different types of events representing 20 different sources; each event contains a type identifier, a source identifier, and a random value. The test case involves counting the number of events per source where the value is below/above a type-depending threshold.

We implemented the test case application in four ways: (1) purely with Eventlets, (2) via CEP queries with Esper (Esper A to E), (3) with Eventlets that use Esper (Eventlet-Esper), and (4) with Java EE message-driven beans (MDB) that use Esper (Esper-Beans). The Esper A to E scenarios follow the common implementation approach for stand-alone CEP applications. The Esper-Beans implementation is suitable for integration in enterprise environments where easy component deployment and lifecycle management is important. We combine both approaches with Eventlets and show the advantages.

The pure Eventlets implementation uses the source identifier as instantiation expression; each Eventlet instance receives only events originating from a certain source and counts events corresponding to given thresholds. In Esper the use case is realized with two CEP queries (one for each event type) that count events with respect to the given threshold and group output by source identifier. To evaluate scalability, we used Esper configurations with different levels of distribution. Distribution in our scenario leads to an increased number of JMS connection primitive objects and Esper instances. JMS connections are provided by connection factory (CF) objects. An application that requires a certain amount of JMS connections can request multiple connections from a single CF or create multiple CF and request less connections per CF. The same holds for JMS message listeners. We varied the number of CF, the number of Esper instances, and the number of JMS message listeners per CF to cover different ranges of distribution; the numbers are shown in Table I. In Esper B to E the increased number of connection primitives and Esper instances leads to more complex code. In terms of distribution the Eventlet-Esper and Esper E scenario are equal, however the Eventlet-Esper implementation benefits from the distribution and subscription automatisms provided by the Eventlet middleware. In the Esper-Bean scenario the connection primitives are managed by the ActiveMQ to GlassFish resource connector and are not disclosed to developers.

In the distributed Esper scenarios C and E subscription filters are used to receive only events relevant for particular query instances, i.e., a query receives only events containing a certain source identifier. In the hybrid solution (Eventlet-Esper) we used Eventlets to realize distributed event processing with Esper. With the creation of a new Eventlet instance an Esper instance is created and the Eventlet instance passes events on to the Esper instance. In the Esper-Beans scenario one MDB is created for each event source and deployed to a GlassFish 3.1.2 application server. Each MDB creates an Esper instance and forwards received events to it. The demand for state requires that MDB are configured with a pool size of one to have each sub stream processed by only a single bean instance (singleton pattern).

Scenario	CF	Esper Instances	Listeners per CF
Esper A	1	1	1
Esper B	1	1	20
Esper C	20	1	1
Esper D	1	20	20
Esper E	20	20	1
Eventlet	20	-	1
Eventlet-Esper	20	20	1
Esper-Beans	AS	20	AS

TABLE I: XML Event Processing: Esper and Eventlet scenarios with different scalability (CF: JMS Connection Factories; Listener: JMS Message Listener; AS: Determined by Application Server)

A. Measurement Results

We use CPU utilization of the Eventlet/Esper host as the indicator for our comparison. On the one hand CPU utilization allows to quantify the overhead of Eventlets. On the other hand scalability across multiple cores is an indicator for good distribution capabilities. The latency in our scenario is dominated by the network and message broker; both are not changed in the different scenarios so that we concentrate on CPU measurements here. To determine the limits of the different implementations we increased the event rate up to the point where ActiveMQ flow control throttled down the event producers, indicating that the consumers are saturated. The results of our tests are shown in Figure 6.

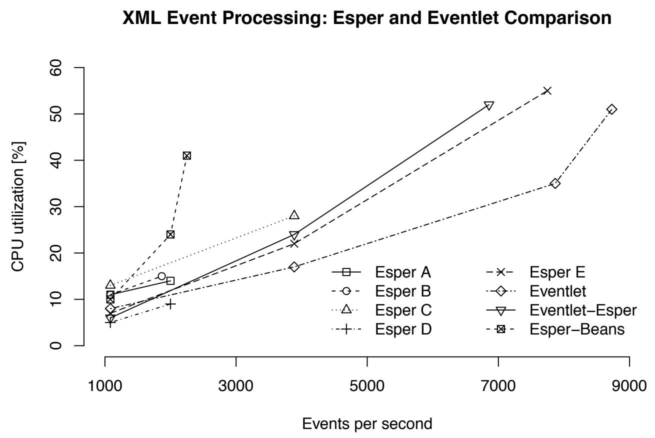


Fig. 6: XML Event Processing Performance Results

The Esper scenarios A,B, and D show that a single CF is a bottleneck. In these three scenarios the event rate was throttled by the consumer to 2000 events per second. The CPU is the limiting factor here: since we have eight CPU cores, a utilization of around 10 percent indicates that a single CPU core is saturated, e.g., due to not parallelized methods. This limits event arrival and triggers throttling mechanisms in the broker. In scenario Esper C multiple CF are used: a higher CPU utilization and event rate are reached but throttling keeps the event rate at about 4000 events per second making the single CEP instance the bottleneck. In the fully distributed scenario Esper E, an Esper instance for each event source identifier is created and registers for the relevant events using JMS message selectors. This is the implementation nearest to Eventlets since each Eventlet instance has its own CF and subscription. Our evaluation shows that only the distributed setups can process a high volume of events. Even on a single multi-core machine multiple connection primitives are required for scalability across CPU cores.

Our test case can be realized without a CEP engine. The resulting pure Eventlet scenario reaches the highest event rate in our evaluation. Since no external CEP library is included, we gain performance due to reduced complexity. However, often it is not reasonable to abstain from the use of a CEP engine. From a software engineering perspective it is desirable to apply a component model in these cases and encapsulate application logic to foster manageability and scalability. This led to our Eventlet-Esper and Esper-Beans scenarios.

The evaluation shows that the Esper-Beans scenario does not perform well compared to the other approaches. It suffers from the complex interplay between application server and message-oriented middleware. The CEP use case does not allow for using large bean pools so that scalability mechanisms of Java EE cannot be applied.

The Eventlet-Esper results show that this scenario is only slightly slower than the dedicated distributed implementation Esper E. This performance loss is introduced by the Eventlet middleware which adds additional layer of abstraction to provide the introduced functionality. We think this performance loss is acceptable given the ease of development and the integrated mechanisms for distribution with Eventlets; we will quantify this in Section VI-B.

In addition to the XML implementation we realized the evaluation use case using att/val representations of events. The reached event rate for the distributed Esper E scenario and Eventlets is about 20,000 events per second. At that point the broker machine was saturated while CPU of the Eventlet host was 22% for Esper E and 13% for Eventlets.

B. *Simplified Software Development with Eventlets*

Eventlets reduce the amount of code programmers have to write to implement distributed event stream processing applications. We analyzed the code of our fully distributed Esper E and Esper-Beans scenario. We compared it with the Eventlet-Esper implementation and counted core application logic code; this excludes code that is automatically generated by modern IDEs, i.e., class headers, exception detection, bean configurations, and constructors, as well as comments, logging, and debugging output. We do not include the pure Eventlets scenario since it implements only part of the Esper functionality. The results are shown in Table II; the Esper E and Esper-Beans implementations have significantly more lines of code than the Eventlet-Esper approach. Further, only two classes are needed for the implementation of Eventlet-Esper. This reduces the complexity of software maintenance tasks compared to Esper E and Esper-Beans.

Scenario: EVENTLET-ESPER

Component	*Lines of Code*
EL Prototype App. Logic	20
EL Prototype Meta Data	2
EL Registration	1
CEP Queries, Result Handling	18
Total: 2 Classes	**41**

Scenario: ESPER E

Component	*Lines of Code*
Main Class App. Logic	39
Event Listener	10
Distributed Instantiation (JMS-based)	61
CEP Queries, Result Handling	18
Total: 3 Classes	**128**

Scenario: ESPER-BEANS

Component	*Lines of Code*
Bean (generic/specific)	17/5
Bean Config (generic/specific)	2/1
CEP Queries, Result Handling	18
Total: 21 Classes	**157 (37+20*6)**

TABLE II: Lines of Code Comparison for different Components and Scenarios (EL: Eventlet)

In contrast to Eventlet-Esper and Esper-Beans the distribution across multiple machines has to be implemented manually in the Esper E. The Esper-Bean scenario uses the Java EE ecosystem which provides for lifecycle management and distribution. However, since EJB are not the natural fit to encapsulate event stream processing logic, code has to be adapted in each of the 20 MDBs. This code is referred to as *specific* while *generic* code remains unchanged and is reused. The savings with Esper distributed by means of Eventlets are mainly due to the automated dynamic subscription handling and integrated event handling provided by the Eventlet middleware.

VII. Conclusion

We introduced Eventlets as containers for application logic that processes event streams. The instantiation, execution, and distribution is handled by the Eventlet middleware. Conceptually, we see Eventlets as an abstraction layer that encapsulate event stream processing logic. Eventlets and services are defined on the same layer which allows for seamless integration. Services can register Eventlet prototypes, Eventlet instances can call services, or services can emit event streams that are processed by Eventlet instances.

An Eventlet prototype has a simple structure and does not introduce a new language. It provides a service-like abstraction layer on top of pub/sub middleware. The developer only has to provide three expressions to identify to which events an Eventlet prototype applies and what the validity is. This is sufficient for our Eventlet middleware to create Eventlet instances. Further, developers implement four methods to achieve reactive functionality and Eventlet lifecycle management. This makes Eventlet instances suitable containers for event stream application logic and provides a clear separation between subscription logic and applications logic.

The modular design of the Eventlet infrastructure enables easy distribution across multiple nodes. Our evaluation shows that this distribution is necessary for scalability. We further show that the overhead of distributing a traditional CEP application by means of Eventlets is low while the development is significantly simplified. Further, our Eventlet middleware can be integrated with existing IT infrastructures easily since it is built on top of a standard message-oriented middleware using a common programming language. In general, Eventlets allow developers to concentrate on specifying application logic and leave administrative tasks to the middleware.

VIII. Future Work

While developing the Eventlet middleware and applying it to different application domains we identified opportunities for future research on the architecture and application level. Future research on the architecture level addresses Eventlets as technical software components. Driven by our involvement in different research projects (see Section IX) one focus lies in the area of enterprise software systems; we plan to integrate the event-based functionality of Eventlets with existing enterprise software solutions. As described in our introduction, this integration is the foundation for modern reactive applications like Emergent Enterprise Software Systems [30].

Eventlet instances are self-contained; they receive events independently and thus have an intrinsic ability for distribution. Currently, Eventlet instances are bound to the network node of their creation. However, Eventlet instances could be moved from one node to another, e.g., for load balancing. Eventlet instances could be moved to their corresponding Eventlet monitor, or to the producer of the events they react to. We are working on metrics and strategies to make Eventlet instances mobile during runtime. Finally, a user interface will be part of our future work to allow an easy management, monitoring, and deployment of Eventlets.

On the application level, future research is related to the data Eventlets receive and process. Eventlets can be used to process sensitive data, e.g., patient data. It is therefore necessary to extend Eventlets with mechanisms to meet privacy and security requirements. The event bus and the Eventlet middleware have to ensure that events are not delivered to unauthorized participants and that malicious event producers can not publish potentially harmful events. An advantage of the Eventlet approach is that Eventlet instances are independent. Thus security mechanisms, e.g., certificates, can be integrated per instance to ensure isolation between components.

Another direction of research is related to the instantiation of Eventlets. Derived events or CEP expressions can be used as static and instantiation expressions and the creation of new Eventlet instances is triggered based upon the CEP result. We have not yet specified this instantiation behavior in detail and will address it in the future.

Event representation semantics is also important in future work. Currently developers have to know event schemas to define static and instantiation expression, and to access event data. We avoid this requirement of global system knowledge by adding a transformation layer to the middleware [31]. Transformations allow developers to combine different event representations in a single Eventlet application.

IX. Acknowledgements

Funding by German Federal Ministry of Education and Research (BMBF) under research grants ADiWa (01IS09020D) and Software-Cluster EMERGENT (01IC10S01) as well as by Hessen Ministry of Higher Education, Research and the Arts under research grant LOEWE Dynamo PLV. The authors assume responsibility for the content. We also thank Patrick Eugster and the anonymous reviewers for the valuable feedback.

References

[1] D. J. Cook and S. K. Das, *Smart Environments: Technology, Protocols and Applications*. John Wiley and Sons, Inc., 2005.
[2] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, “Middleware to support sensor network applications,” *IEEE Network*, vol. 18, no. 1, pp. 6–14, 2004.
[3] A. Hinze, K. Sachs, and A. Buchmann, “Event-based applications and enabling technologies,” in *DEBS*, USA, 2009.
[4] O. Levina and V. Stantchev, “Realizing event-driven SOA,” in *ICIW*. Italy, 2009.
[5] K. M. Chandy, “Sense and respond systems,” in *CMG*, USA, 2005.
[6] A. Buchmann, H.-C. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber, “Event-Driven services: Integrating production, logistics and transportation,” in *SOC-LOG*, USA, 2010.

[7] V. Cristea, F. Pop, C. Dobre, and A. Costan, *Distributed Architectures for Event-based Systems*, ser. Studies in Computational Intelligence. Springer, 2011.

[8] R. Blanco, J. Wang, and P. Alencar, "A metamodel for distributed event based systems," in *DEBS*, Italy, 2008.

[9] R. Meier and V. Cahill, "Taxonomy of distributed event-based programming systems," in *ICDCSW*, Austria, 2002.

[10] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *ICDCSW*, Austria, 2002.

[11] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski, "The PADRES distributed publish/subscribe system," in *ICFI*, UK, 2005.

[12] L. Fiege, M. Cilia, G. Mühl, and A. Buchmann, "Publish/Subscribe Grows Up: Support for Management, Visibility Control & Heterogeneity," *IEEE Internet Computing (Special Issue on Asynchronous Middleware and Services)*, vol. 10, no. 1, 2006.

[13] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE TSE*, vol. 27, 2001.

[14] EsperTech Inc., "Esper Complex Event Processing Engine 4.3.0," 2012.

[15] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: a high-performance event processing engine," in *SIGMOD*, China, 2007.

[16] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD*, USA, 2006.

[17] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite events for active databases: Semantics, contexts and detection," in *VLDB*, Chile, 1994.

[18] U. Dayal, A. Buchmann, and D. R. McCarthy, "Rules are objects too: A knowledge model for an active, object-oriented database system," in *OODBS*, Germany, 1988.

[19] G. Feuerlicht, *Next Generation SOA: Can SOA Survive Cloud Computing?*, ser. Advances in Intelligent and Soft Computing. Springer, 2010.

[20] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Communications of the ACM*, vol. 42, pp. 88–89, March 1999.

[21] S. Bromuri and K. Stathis, "Distributed agent environments in the ambient event calculus," in *DEBS*, USA, 2009.

[22] P. Eugster and K. R. Jayaram, "EventJava: An extension of java for event correlation," in *ECOOP*, Italy, 2009.

[23] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé, "EScala: modular event-driven object interactions in scala," in *AOSD*, Brazil, 2011.

[24] G. Agha and C. Hewitt, "Concurrent programming using actors: Exploiting large-scale parallelism," in *FSTTCS*, India, 1985.

[25] K.-K. Lau and Z. Wang, "Software component models," *IEEE Transactions on Software Engineering*, vol. 33, pp. 709–724, 2007.

[26] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus: an architecture for extensible distributed systems," *ACM SIGOPS Operating Systems Review*, 1993.

[27] S. P. Mahambre, M. Kumar, and U. Bellur, "A taxonomy of qos-aware, adaptive event-dissemination middleware," *IEEE Internet Computing*, vol. 11, pp. 35–44, July 2007.

[28] Sun Microsystems, Inc., "Java Message Service (JMS) Specification - Ver. 1.1," 2002.

[29] S. Rozsnyai, J. Schiefer, and A. Schatten, "Concepts and models for typing events for event-based systems," in *DEBS*, Canada, 2007.

[30] S. Frischbier, M. Gesmann, D. Mayer, A. Roth, and C. Webel, "Emergence as Competitive Advantage - Engineering Tomorrow's Enterprise Software Systems," in *ICEIS*, France, 2012.

[31] T. Freudenreich, S. Appel, S. Frischbier, and A. Buchmann, "ACTrESS - automatic context transformation in event-based software systems," in *DEBS*, Germany, 2012.