SocEDA

Cloud based platform for large scale social aware EDA

ANR-10-SEGI-013

Document name:	Overall Framework Model
Document version:	1.0
Task code:	T0
Deliverable code:	1.2.1
WP Leader (organisation):	PetalsLink
Deliverable Leader (organisation):	PetalsLink
Authors (organisations):	Nicolas Salatgé
Date of first version:	20/09/11

Change control

Changes	Author / Entity	Code of version
Creation of the document	Nicolas Salatgé	0.1

Table of Contents

1. Introduction

This document wants to outline a framework for dynamic and complex, event-driven interaction in large, highly distributed and heterogeneous service systems. Such an architecture will enable the exchange of contextual information between heterogeneous services, providing the possibilities of optimizing and personalizing the execution of the services themselves, resulting in context-driven adaptivity.
The Soceda platform is a global (web-scale) structure to combine events from many sources with the goal of connecting and orchestrating services, things and people.

2. Architecture Overview

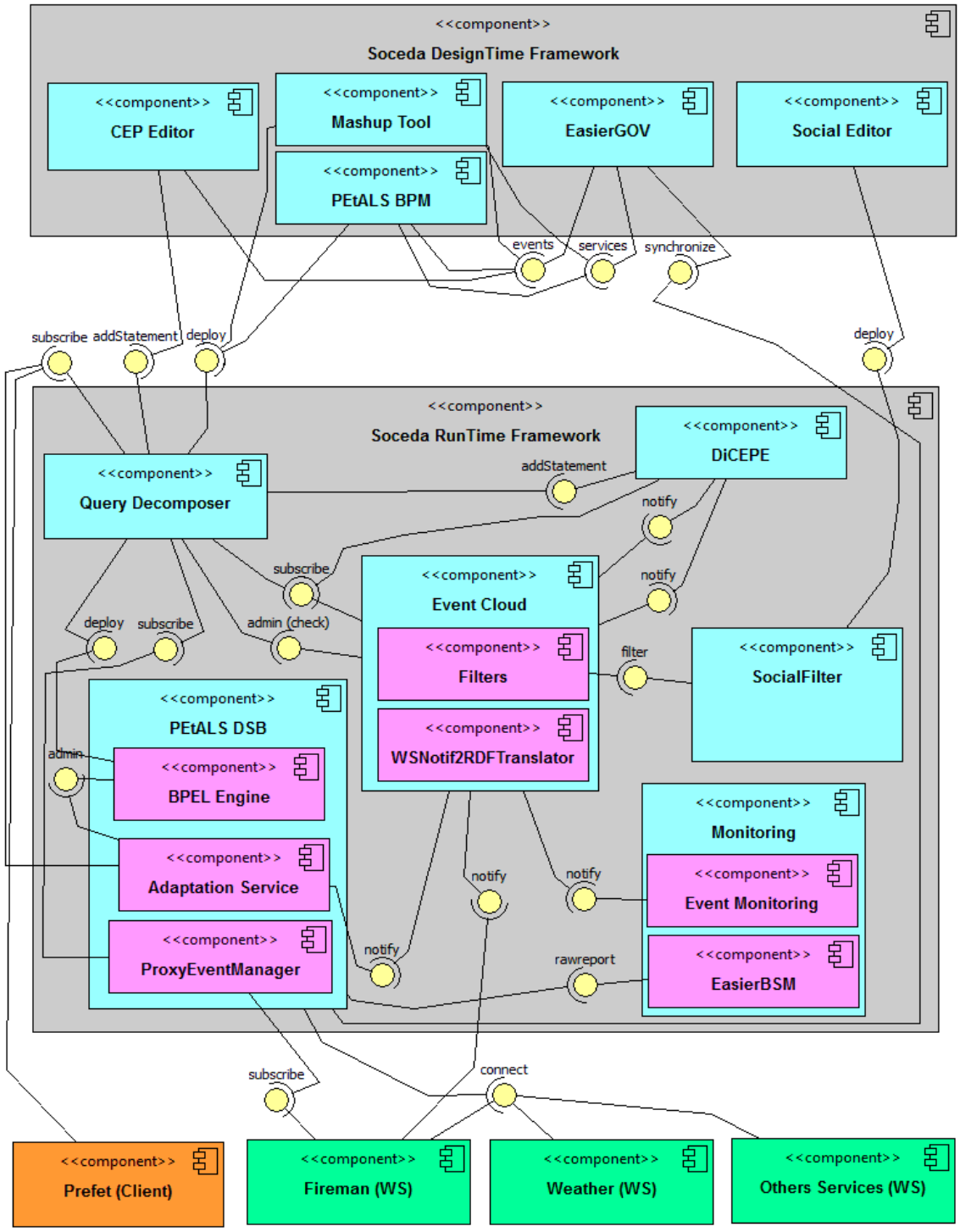


Figure 1: Framework Overview

In this chapter, we describe the Soceda platform from a bird's eye view. We explain how the platform breaks down into functional components and what their responsibilities are. This is followed by a brief look at high-level interfaces to interact with the platform.

The Figure 1 presents the overall architecture of Soceda. This Framework is composed of two parts:

- The Soceda Design Time Framework is composed of: CEP Mashup Editor, EasierGOV, Social Editor and Petals BPM.
- The Soceda Run Time Framework is composed of: Query Decomposer, DSB, Adaptation Service, Proxy Event Manager, Event Cloud, WSNotif2RDF Translator, Filters, Social Event Editor, Event Monitoring and EasierBSM.

All these components are described below.

3. Design Time

3.1. Mashup Tool

3.1.1. Requirements

In order to keep available the richness of a largely tooled framework, and to inherit improved mechanisms, the mashup designer will be based on a BPMN editor.

The mashup high level components would encapsulate predefined BPMN components, keeping complexity hidden at design time. Then at deploy time, mashup would be translated to BPMN internal representation, and then to BPEL runnable representation, in order to execute the flow in its complexity.

3.1.1.1. Functionality

The main and most visible mashup tool functionality is graphical design. The tool must allow components to be dragged and dropped from a palette to a flow panel, to connect components, and to parameterize setting on all graphical objects, depends on their underlying function. As part of design functions, the palette structure will be a main issue, in order to facilitate the components finding and the process design principles understanding

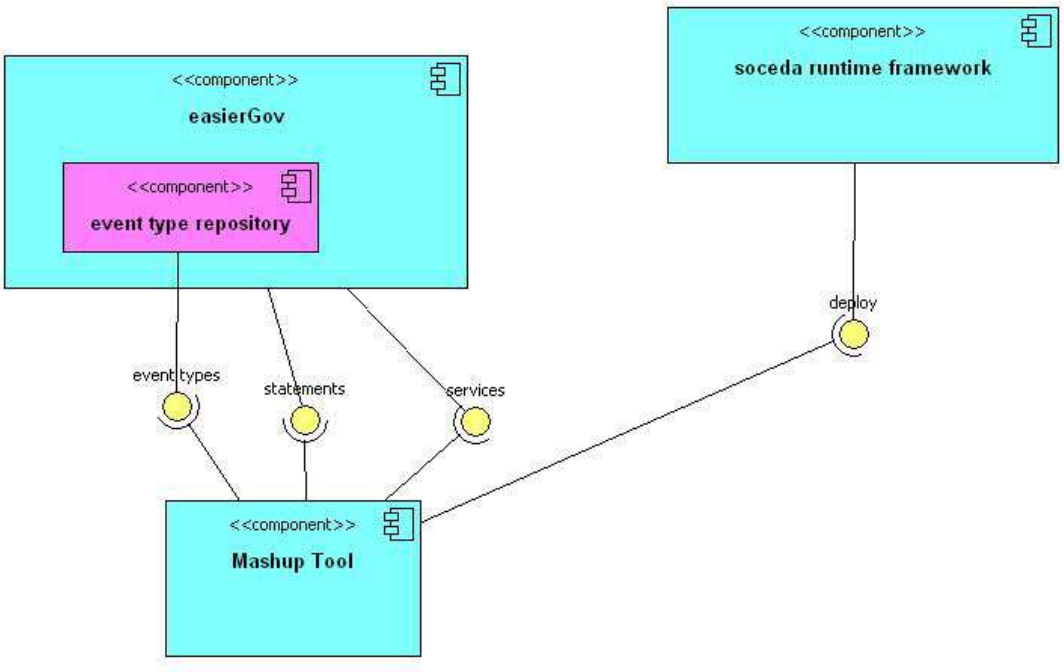
To retrieve the current available event types, the Mashup tool needs to be linked with the event type repository from the EasierGov component. This link will be created through an interface that will return the current available event types.

In order to retrieve the running EPL statement, the mashup tool will use an interface of the EasierGov. The available services should also be retrievable through a EasierGov interface.

3.1.1.2. Technical Requirements

There is no technical restriction as long as the four needed interfaces are reachable either via webservices or any other remote protocol.

3.1.2. Components



3.1.3. Interfaces

Event type: this interface will be used by the CEP Editor in order to retrieve all the available event types' description.
Statements: this interface will be used to retrieve the current running statements.
Services : an interface to retrieve the available services, already included in the platform.
Deploy: this interface will be used to push the link between the running statement and the services to trigger.

3.2. CEP Editor

3.2.1. Requirements

In order to make the EPL rule design aspect easier, the CEP editor is a drag & drop tool with an interface to an event type repository. Basically, it will allow designing diagram representation of rules in order to push EPL queries within an ESPER instance without having to interact with any IDE. Clearly, the CEP Editor is strongly linked with the Mashup tool at design time.

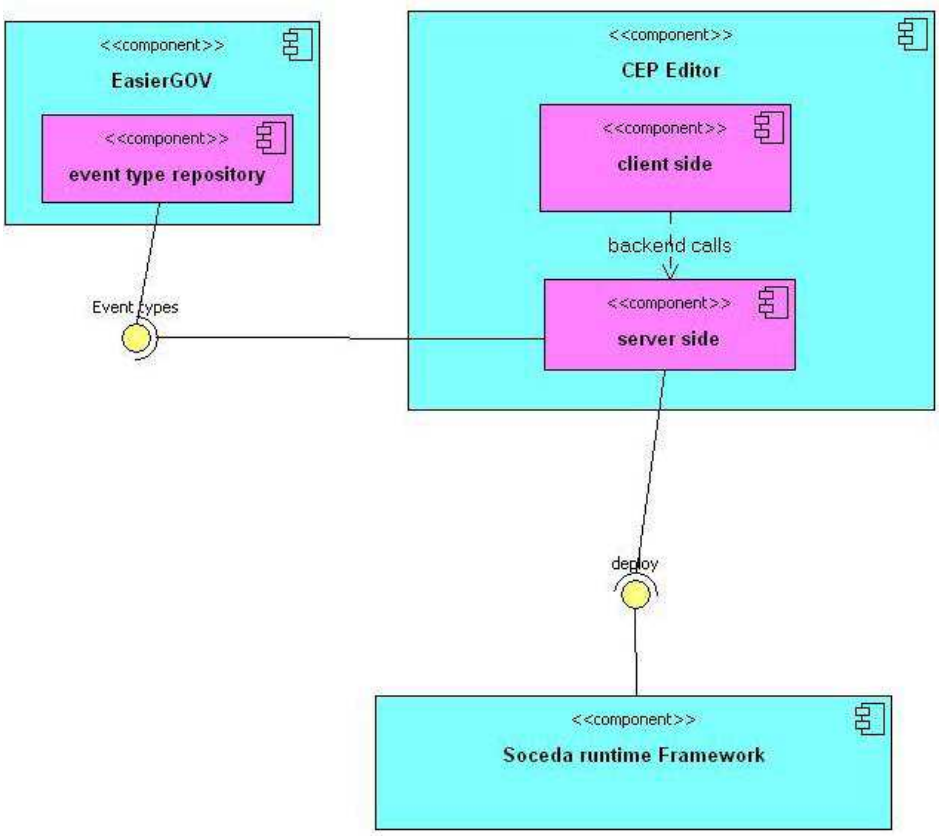
3.2.1.1. Functionality

To retrieve the current available event types, the CEP Editor needs to be linked with the event type repository from the EasierGov component. This link will be created through an API that will return the current available event types. In order to push the generated EPL rules in the DCEP environment or the Soceda runtime Framework, the CEP Editor has to be bound with it.

3.2.1.2. Technical Requirements

There is no technical restriction as long as the two needed interfaces are reachable either via webservices are any other remote protocol. As described in the next section those calls are done from the backend which makes it transparent from the user point of view.
On the client side the user uses a javascript interface which makes asynchronous remote call to the backend in order to retrieve the event types but also in order to push the generated EPL rules.
The backend is then using any available remote method to be linked with the EasierGov component and the Soceda runtime framework.

3.2.2. Components



3.2.3. Interfaces

Event type: this interface will be used by the CEP Editor in order to retrieve all the available event types' description.
Deploy: interface used to push the generated EPL rules within the Social runtime Framework that does contain the running DCEP environment.

3.3. Petals BPM

3.3.1. Requirements

3.3.1.1. Functionality

Petals BPM is an open source, cloud-enabled graphical business process designer. One can design BPMN 2.0 processes and translate them into executable BPEL processes. The main functionalities of Petals BPM are:

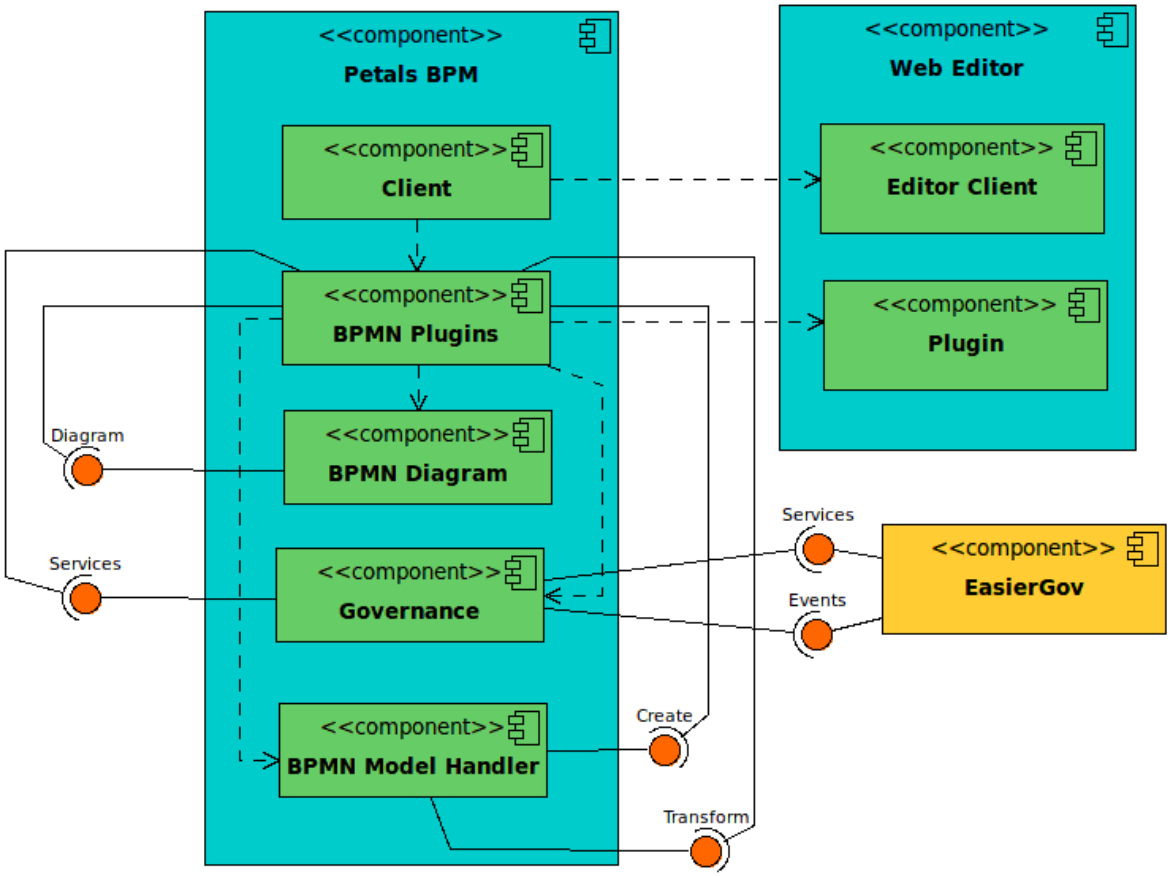
- Allow users to create and edit BPMN 2.0 diagrams.
- Edit high-level elements of the BPMN Definitions (interfaces, operations, message etc..).

- Import and export of the created diagrams to a BPMN 2.0 XML file.
- Import and export of the created diagrams to a XPDL 2.1 XML file.
- Export of the created diagrams to a BPEL 2.0 file.
- BPMN syntax validation
- The ability to connect with a governance API
- Automated transformation of the services retrieved from the governance API into BPMN 2.0 services

3.3.1.2. Technical Requirements

PetalsBPM is deployed as a WAR application on tomcat 5 or lighter

3.3.2. Components



Petals BPM is based on WebEditor, a GWT framework developed by PetalsLink to ease the creation of web based editors. WebEditor framework defines a default client (Editor Client component) upon which some plugins can be added to extend the core functionalities provided by WebEditor.
Therefore, Petals BPM top level components (Client, BPMN Plugins), mainly consists of specialized WebEditor components (Web Editor Client and Plugin) in order to add the required features. These features are mainly provided by the lowest level components: BPMN Diagram, Governance and BPMN Model Handler. These components are respectively described:

- BPMN Diagram : UI library that allows to design / load BPMN 2.0 diagrams.
- Governance: Defines an interface between the implementation of Petals BPM and any governance tool (EasierGov for exemple) in order to retrieve some services with some criteria
- BPMN Model Handler: Mainly use to handle bpmn model creation and transformation into other formats (BPEL/XPDL)

3.3.3. Interfaces

External Interface Name	Operations List	Used by component

Diagram	- BPMNDiagram create(); - void load(BPMNDiagram diagram)	BPMN Plugins
Services	– Map<String,String> searchServices(Criterion[] criteria); – String getWSDLAsString(String id);	BPMN Plugins
Create	– XMLDefinitions createXMLDefinitions(Definitions bpmnDefinition) – Definitions loadDefinitions(XMLDefinitions xmlDefinitions);	BPMN Plugins
Transform	– File BPMN2BPEL(Definitions definitions) – File BPMN2XPDL(File bpmnFile) – Definitions XPDL2BPMN(File xpdlFile)	BPMN Plugins

3.4. EasierGOV

The Soceda Governance platform uses (and extends) the EasierGov[1] tools developed by PetalsLink in order to provide a complete Event/Stream/Service governance solution.
The governance platform contains all the necessary information in order to describe services and events producers and consumers. EasierGov has been designed in order to support service description according to the USDL[2] (Unified Service Description Language) standard currently under development.

3.4.1. Requirements

3.4.1.1. Functionality

The core functionality of the governance platform is to provide ways to describe services/events actors by using standards. It also offers functionalities to describe service interactions, agreements between parties and access rights.
The actors descriptions will use efficient standards as basis: WSDL for services, WS-Topics for event producers and consumers. One added value of an advanced governance platform is the capability to describe agreements between consumers and providers. To achieve this goal, the governance platform contains SLA/ELA ‘templates’ described with the help of WS-Agreements and which can be retrieved with a search engine provided by the governance platform. By using patterns, a simple process can be described as:

[1] http://research.petalslink.org/display/easiergov/EasierGov+Overview

[2] http://www.w3.org/2005/Incubator/usdl/

- Each service provider will be able to define SLA template based on objectives it can provides. This template is stored in the governance platform and linked with the service provider (or topic in the case of event provider).
- A consumer will be able to search providers based on some objectives. Once found, it will create an agreement instance between parties (there is a negotiation phase here which needs to be defined). For example, the consumer can search all the service providers providing latency lower than 1 second, get a list and choose one to create an agreement.
- The Agreement instance is deployed at the monitoring level.. The agreement should express the constraints/objectives (Service Level Objective, Event Level Objective) that the monitoring facility may verify. Consequently the possible set of SLO/ELO has to be defined.
- The monitoring platform will get monitoring information coming from the runtime platform and analyse it. On agreement violation, alerts can be raised using properties previously defined at the agreement level. The alert definition is defined in the WS-Agreement contract at design time and stored in the governance platform. Alerts are also events and so are stored into the event cloud.

3.4.1.2. Model for Event Level Agreements (ELA)

Event Level Agreement (ELA) extends and reuses Service Level Agreements (SLA) to express policies at the Event producers and consumers levels.
It is important to highlight what is an event producer and an event consumer at the Soceda level. Event Level Agreements and Event Level Objectives may be defined at several levels between producers and consumers. We identified four cases:

1. A Soceda platform user is a consumer; The Soceda platform is the provider.
2. Business services connected to the platform are event providers; The Soceda platform is the consumer.
3. Business services connected to the platform are event providers; A Soceda platform user is a consumer
4. Business services connected to the platform are event providers; some others business services are event consumers.

In order to be able to implement Event Level Agreement templates, a first list of agreements has been created within the project.

Table 1: Possible Event Level Agreements

#	Property	Alterable	Description
1	Maximum number of notifications received over time	Yes	It may be possible for an event consumer to define the maximum number of notifications it can received in a window period. For example : 'at most 10 messages / second'
2	Minimum number of notifications received over time	Yes	It may be possible for an event consumer to define the minimum number of notifications it want to receive per unit of time. For example : 'at least 10 messages / second'
3	Replication Rate	No	The number of time a data that is published is replicated into the network (for **Critical Events**)

4	Delivery Order	No	The delivery order: FIFO (per event cloud), unordered
5	Delivery Speed	No	Priority playcement/routing (for **Priority Events**)
6	Maximum number of different event sources	No	What is the limit of the platform in terms of listening to different event sources?
7	Maximum lag between subscription time and start time of receiving notifications	Yes	What is the maximum time that it is needed in order to start receiving notifications after you have been registered to an event source?
8	Delivery Order for Past Events	No	Delivery Order when querying for Past Events FIFO (per event cloud), unordered
9	Privacy	No	Restrict the ability of a user to subscribe to a stream (directly or for a user-specified event pattern). Should not be alterable later because we only enforce privacy a subscribe-time.
10	Min/Max number of False Positives / Fals Negatives per time unit	tbd	Restrict/Define the data quality in events, such as FPs, FNs, noise in number of events. This might not be applicable in PLAY since we deal with 100% confidence in events, nothing probabilistic. Still this should be mentioned in the appropriate deliverable. :)

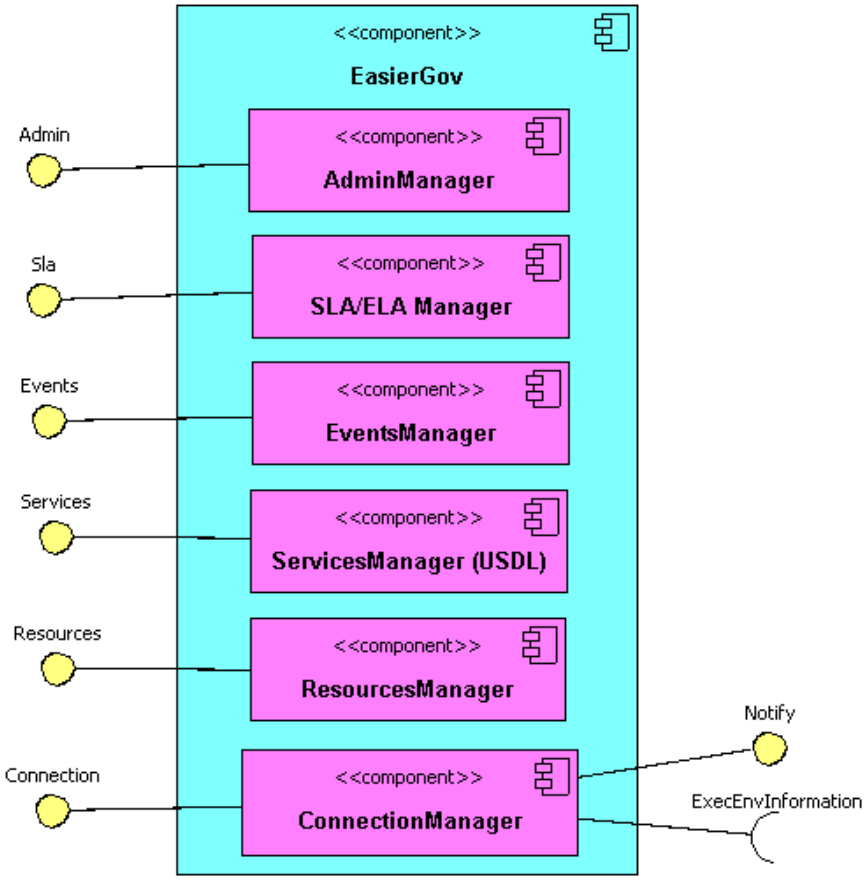
3.4.1.3. WS-Agreement

The goal of this section is to show how the WS-Agreement standard can be used without any extension to define ELA between event consumers and providers.
According to the 4 cases defined above, all actors, services or events ones are finally clients and services. WS-Agreement has been initially defined for services where agreements are created between service clients and service providers. By applying this standard communication pattern the use cases previously detailed, a natural result is to apply WS-Agreement to define not only Service Level Agreements, but also Event Level Agreements.

3.4.1.4. Technical Requirements

The EasierGOV needs Java 6 or higher to run. It must be connected to an infrastructure of services (PEtALS DSB or EasyESB) to work correctly.

3.4.2. Components



3.4.3. Interfaces

EasierGov exposes several interfaces:
Admin api: This interface allows us to know all services provided by EasierGov.
SLA api: This interface allows us to store and find SLA or ELA contract.
Events api: This interface allows us to find specific Event producer on the service infrastructure.
Services api: This interface allows us to find specific service on service infrastructure.
Resources api: This interface allows us to publish, get or remove a resource (ie, XML document).
Connection api: This interface allows us to connect EasierGov to a service infrastructure.

The table below summarizes all api:

External Interface Name	Operations List	Used by external component

Resources	- void publish(URL address, ResourceType type); - Document getResource(String resourceId) ; - SubscribeResponse subscribe(EventType event) ;	PetalsBPM
Services	- FindServicesResponse findServices(FindServices parameters); - FindTechnicalInterfacesResponse findTechnicalInterfaces (FindTechnicalInterfaces parameters) throws Fault;	PetalsBPM
Events	- FindEventProducersResponse findEventProducers (FindEventProducersRequest body) throws Fault; - ServiceType getEventProducer(QName idEventProducer) throws GetEventProducerFault; - List<EPR> findAddressesOfEventProducers(List< String> topicExpressions) throws Fault;	Event Proxy Manager
SLA	- Not defined for the moment	
Admin	- QName getInformation(); - GetServicesResponse getServices(GetServices parameters);	EasiestDEMO
Connection	- void synchronize() throws SynchronizeFault; - ExecutionEnvironmentInformationType connectToEnvironment(String endpointAddress) throws Fault; - ExecutionEnvironmentInformationType unconnectToEnvironment(String endpointAddress) - List<ExecEnvironmentInformationType> getAllEnvironments() throws GetAllEnvironmentsFault;	EasiestDEMO
Notify	- void notify(NotifyRequest request);	ESB (to prevent that new endpoint are been created)

All these interfaces are modeled in WSDL documents:

For Resources: https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easiergov/resources-api/src/main/resources/resources-api.wsdl
For Events: https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easiergov/events-api/src/main/resources/events-api.wsdl
For Services: https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easiergov/services-api/src/main/resources/services-api.wsdl
For SLA: Not defined for the moment
For Admin: https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easiergov/admin-api/src/main/resources/admin-ws.wsdl
For Connection:
https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easyresources/ws-binding-resources/src/main/resources/execution-environment-synchronizer-impl-1.0.wsdl

For Notify: https://svn.petalslink.org/svnroot/trunk/research/dev/experimental/easiergov/execution-environment-connection-api/src/main/resources/execution-environment-connection-api-1.0.wsdl

3.5. Social Editor

3.5.1. Requirements

3.5.1.1. Functionality

The Social Editor provides a user interface to manually declare social relationships between services. In order to declare a social relationship between two services, the user specifies the *source_node* and the *target_node*. The parameter *source_node* is the identifier of the service which has a social relationship towards the service identified by the parameter *target_node*. The relationship is asymmetric, therefore a social relationship in the opposite direction requires a separate declaration. The other parameters to describe a social relationship that can be specified in the Social Editor are listed in the following table.

Element	Data Type	Comments
Role	Role ID	
interaction_count	{0,1,2,3,…}	Number of interactions between the two nodes
amount_of_data_exchanged	{0,1,2,3,…}	Amount of data exchanged between the two nodes. The data unit can be considered as kilobytes, megabytes, etc.
time_of_last_interaction	Timestamp	The time of the latest interaction between the two nodes
Trust	[0,1]	A value representing the amount of subjective trust that the source node holds in the target node. 0: weak, 1: strong. The context of trust is the general integrity of the target node.
time_of_first_interaction	Timestamp	The time of the earliest interaction between the two nodes
mutual_nodes_count	{0,1,2,3,…}	The count of nodes that have relationships with both the source and the target nodes

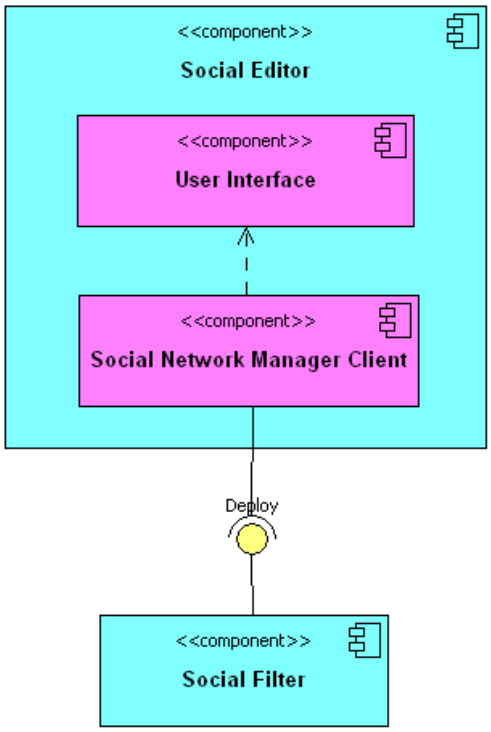
3.5.1.2. Technical Requirements

The Social Editor uses the web services technology to interact with other components in the

architecture.

3.5.2. Components

The Social Editor is composed of the following two sub components: 1) User Interface, and 2) Social Network Manager Client. The User Interface component provides the user with an interface to declare social relationships between services. The Social Network Manager Client component uses the external Deploy interface of the Social Filter component to push the declared social relationships.



3.5.3. Interfaces

Deploy: interface used to push the social relationships declared by the user to the Social Filter component.

4. Runtime

4.1. Query Decomposer

4.1.1. Requirements

4.1.1.1. Functionality

The Query Decomposer is the frontal of Soceda RunTime Framework. It allows to client to subscribe to a specific producer, to add CEP rules or deploy BPEL process.

4.1.1.2. Technical Requirements

The Query Decomposer needs Java 6 or higher to run. On some node deployments, specific network ports needs to be open to enable node to node communication.

4.1.2. Components

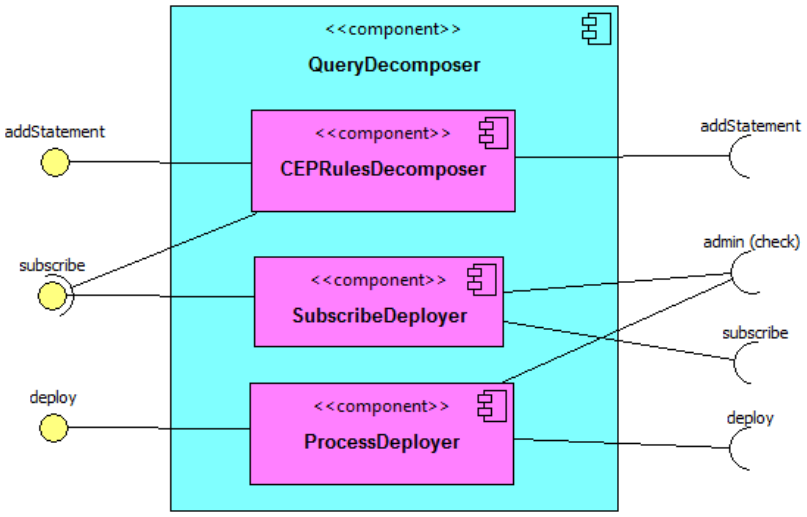


Figure 2: QueryDecomposer Architecture

The Query Decomposer is composed of 3 components:
- ***SubscribeDeployer***: This component is in charge of realizing a subscription from a client request. To do this, it must realize a check on Event Cloud to get the identifier of valid Event Cloud Node in a first time and then send the subscribe request on this valid Event Cloud Node.
- ***CEPRulesDecomposer***: This component is in charge to decompose the CEP rules sent by clients to realize subscribe on primitive topic used in the rules. This subscription are sent on Event Cloud Node corresponding to the topic.
- ***ProcessDeployer***: This component received the BPEL process sent by clients and analyze it to change the location of subscription towards the valid Event Cloud Node.

4.1.3. Interfaces

QueryDecomposer exposes by default these following interfaces:

External Interface Name	Operations List	Used by component
addStatement	- `ListAllStatementsResponse listAllStatements();` - `GetStatementByIdResponse getStatementById(GetStatementById request);` - `AddStatementResponse addStatement(AddStatement requets);` - `UpdateStatementResponse updateStatement(UpdateStatement request);` - `DeleteStatementResponse deleteStatement(DeleteStatement request);`	CEP Editor
subscribe	- `SubscribeResponse subscribe(SubscribeRequest);`	Adaptation Service, any clients (Prefet)
deploy	- `DeployReportResponse deploy(DeployRequest request)`	any clients (Prefet)

4.2. DSB

This chapter and the following chapters describe one component each in detail. The descriptions follow a unified structure to ease the reading. At first the component will be introduced, followed by a requirements analysis. Then each component will be described using UML 2.0 component diagrams showing decomposition into subcomponents and their links. After that the interfaces of the component will be described in more detail, divided into provided and consumed interfaces according to the directionality of the links.
The Distributed Service Bus (DSB) developed within the Soceda project is an extension of the open source Entreprise Service Bus provided by PetalsLink called Petals ESB. The distributed feature will be described later in this section and will highlight the need of such feature at the infrastructure level.
The DSB provides the SOA and EDA infrastructure for the so-called platform components and for end user services. The DSB aims to provide connectivity between services providers, services consumers, event consumers and event providers, potentially distributed over distinct

administrative domains, in a completely transparent way on the user point of view. The core features of the DSB are listed below:

- SOA: The DSB provides a core SOA layer which will be the basis of all other DSB components. On the client side, the SOA layer provides a way to invoke services without any knowledge on the final service location and transport protocol. It is up to the service bus to route the message to the right service and to send back the response to the right consumer. All the location and routing knowledge is located at the SOA layer level.
- Service Binding: The DSB provides the core functionality to bind external services and to expose internal service with the help of binding components. All the transport and transformation logic to address specific protocols and final endpoints are located at this service binding level.
- Standards compliant: The DSB extensively uses and implement the OASIS and W3C standards to provide most of its core functionality
    - WSDL: The Web Service Description Language is used in the DSB to describe all the services which are used and available
    - OASIS WSN: The Web Service Notification standard is implemented in order to provide the core EDA feature.
    - OASIS WSDM: The Web Service Distributed Management is used to monitor services invocations.
    - WS-Agreement: Used to define Service Level Agreements (SLA) and extended to define Event Level Agreements (ELA).
- EDA: The Event Driven Architecture feature allow to connect event producers and event consumers to the service bus infrastructure.
- Polling Engine: The goal of this core component is to easily connect standard request/response services and to transform them as event producers. The component will be in charge of polling services and to wrap the service responses into WS-Notification messages.

4.2.1. Requirements

4.2.1.1. Functionality

The core functionality needed from the DSB are

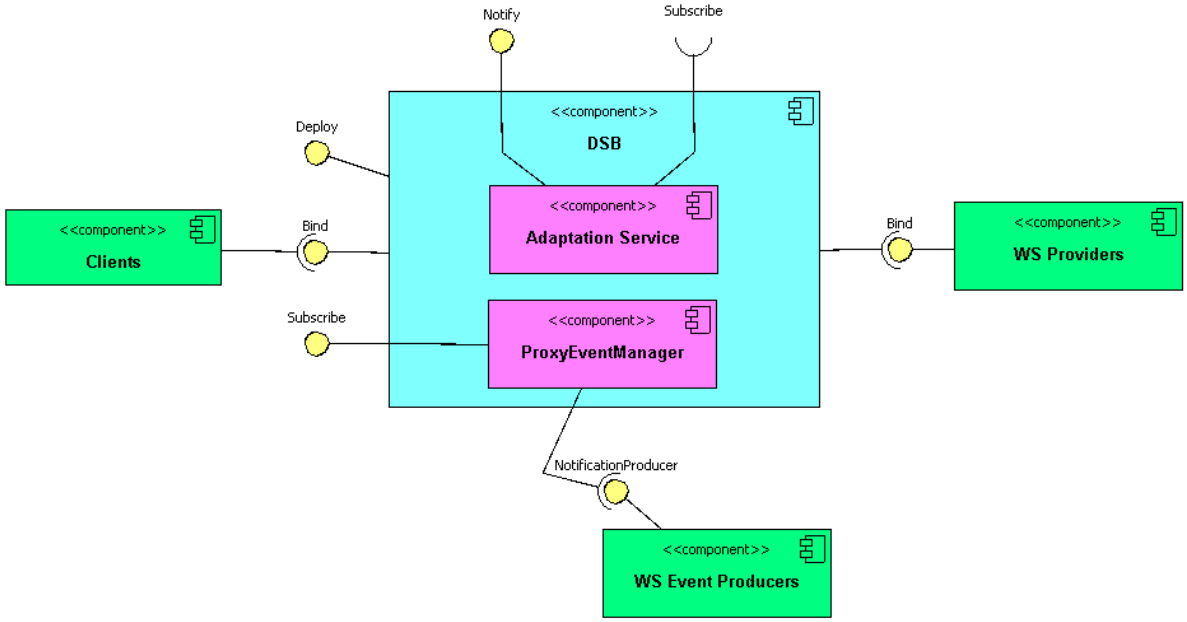
- Services binding: The SOA layer will provide this functionality to connect business services to the platform.
- Event consumer/producer binding: Based on the SOA layer, the EDA layer will be able to connect event producers and consumers to the platform. The protocol and message transformation between the event environment and the service one will be taken in charge by the event component.
- Distributed: Addressing high number of producers and consumers will need a distributed feature.
- Federated across domains: The service bus will need to be able to connect producers and consumers distributed across private and public domains.

- Internet compliant: The service bus needs to be create communication links over the Internet.
- Monitorable: The service bus needs to provide some monitoring sensors which will be gathered by the governance tool to enforce agreements (SLA and ELA ones).
- Manageable: APIs to manage to service bus must be provided.
- Extensible: New providers and consumers must be connected to the platform without the need to restart and to loose connectivity to other services actors.

4.2.1.2. Technical Requirements

The Distributed Service Bus needs Java 5 or higher to run. On some node deployments, specific network ports needs to be open to enable node to node communication.

4.2.2. Components



4.2.3. Interfaces

The service bus provides platform level interfaces which will be used by other platform components to connect, monitor and manage it.
The service level interface provides support for request/response services. The service interface is composed of several operations:

- bind(service): Bind s a service to the service bus. Once connected to the DSB, the service will be reachable from all the DSB nodes.
- invoke(service, message): Services can be invoked directly by clients. The DSB will route the service call to the right service provider.

4.3. Proxy Event Manager

4.3.1. Requirements

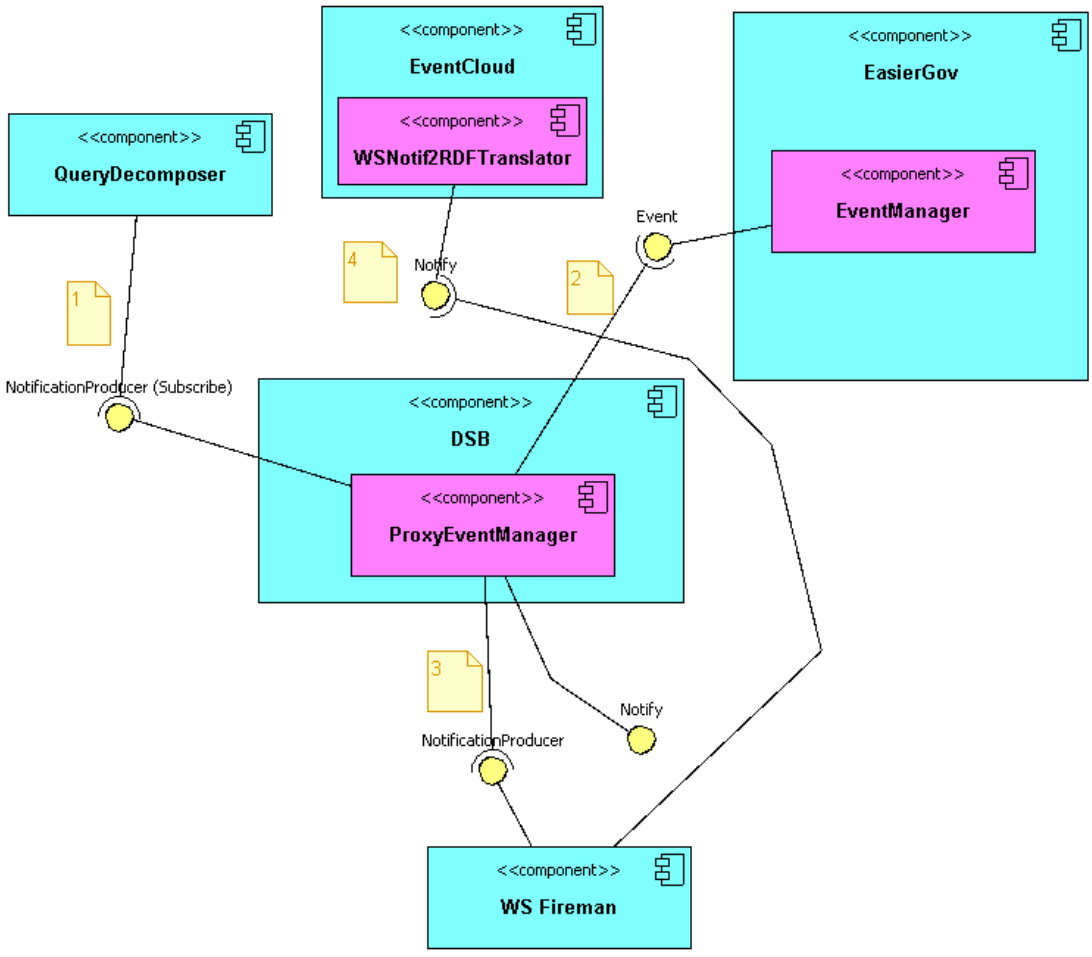
4.3.1.1. Functionality

The Proxy Event Manager is able to maintain the list of Event Producers for a given topics. It can be considered as a broker for Event Producer.

4.3.1.2. Technical Requirements

The Proxy Event Manager is a native service directly integrated in PEtALS DSB.

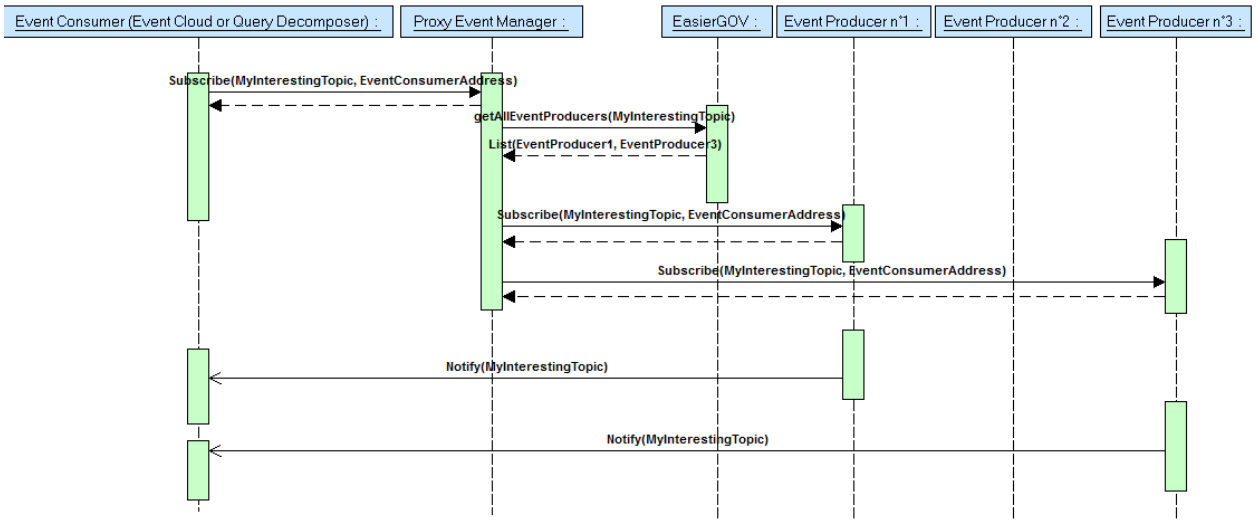
4.3.2. Components



The functional description of this component is described in the sequence diagram below. An

event consumer send a subscribe request containing the topic and the consumer address to receive notification. When the PEM receives this request, it interrogates the governance tool (EasierGOV) to get the list of Event Producers that are able to produce this topic. This list is regularly updated to add new event producers. This list allows PEM to subscribe to business event producers for the event cloud.

When a event producer notify an event, it sent this event directly to the event cloud.



4.3.3. Interfaces

As a broker, the PEM component exposes the standard API of WS-BaseNotification of OASIS

External Interface Name	Operations List	Used by component
Subscribe	- SubscribeResponse subscribe(SubscribeRequest request)	Query Decomposer, Event Cloud
Notify	– void notify(NotifyRequest request)	Event Producer (Not necessary for the moment)

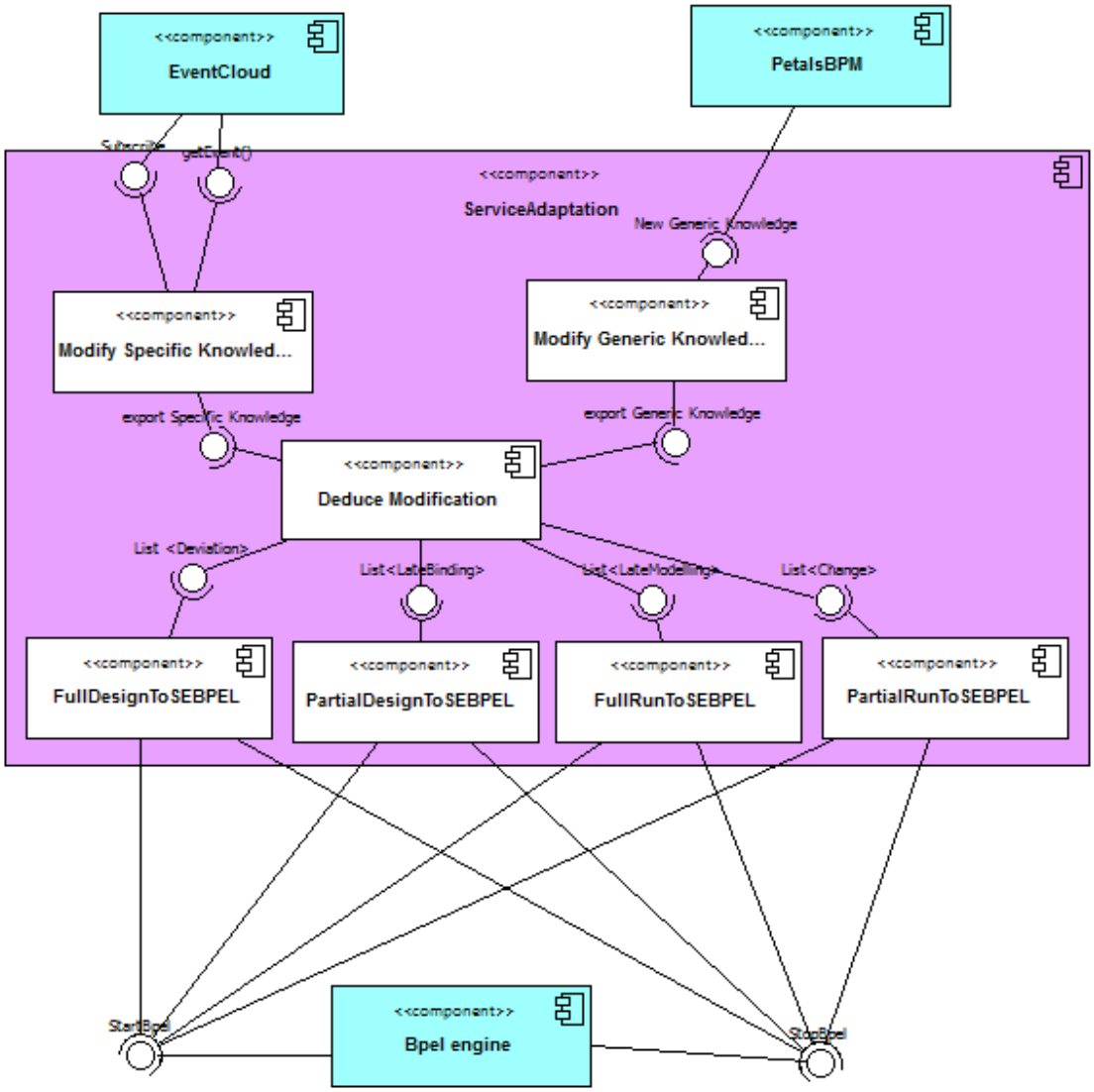
4.4. Adaptation Service

4.4.1. Requirements

4.4.1.1. Functionality

4.4.1.2. Technical Requirements

4.4.2. Components



4.4.3. Interfaces

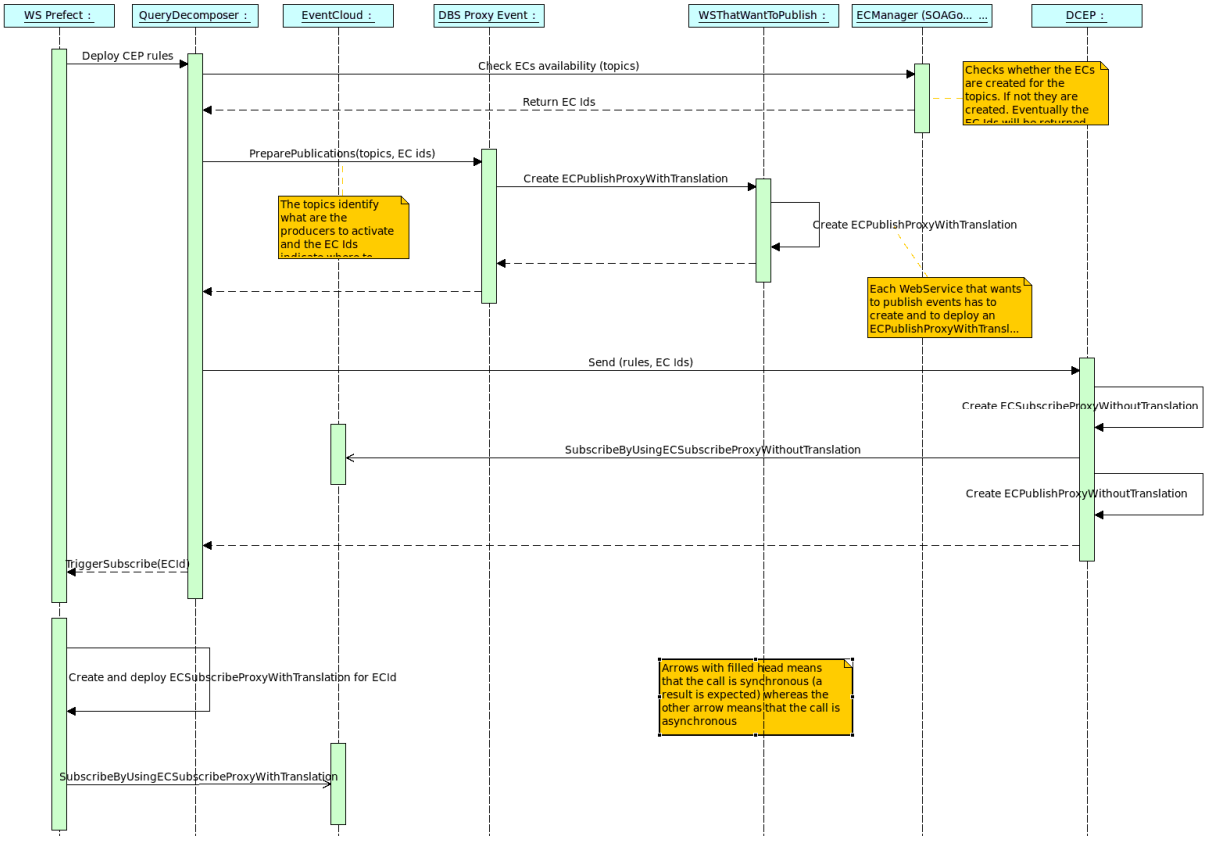
4.1. Event cloud

4.1.1. Requirements

4.1.1.1. Functionality

The Event Cloud aims to provide functionalities for enabling large-scale event storage, retrieval and dissemination capabilities by offering a publish/subscribe API on top of a peer-to-peer overlay network harnessing hosting nodes from possibly multiple - so heterogeneous - clouds at once.

The following figure illustrates the Event Cloud functionality in the SocEDA platform. To be able to communicate with the event cloud each user must creat a client Proxy (for publisher and subscriber).

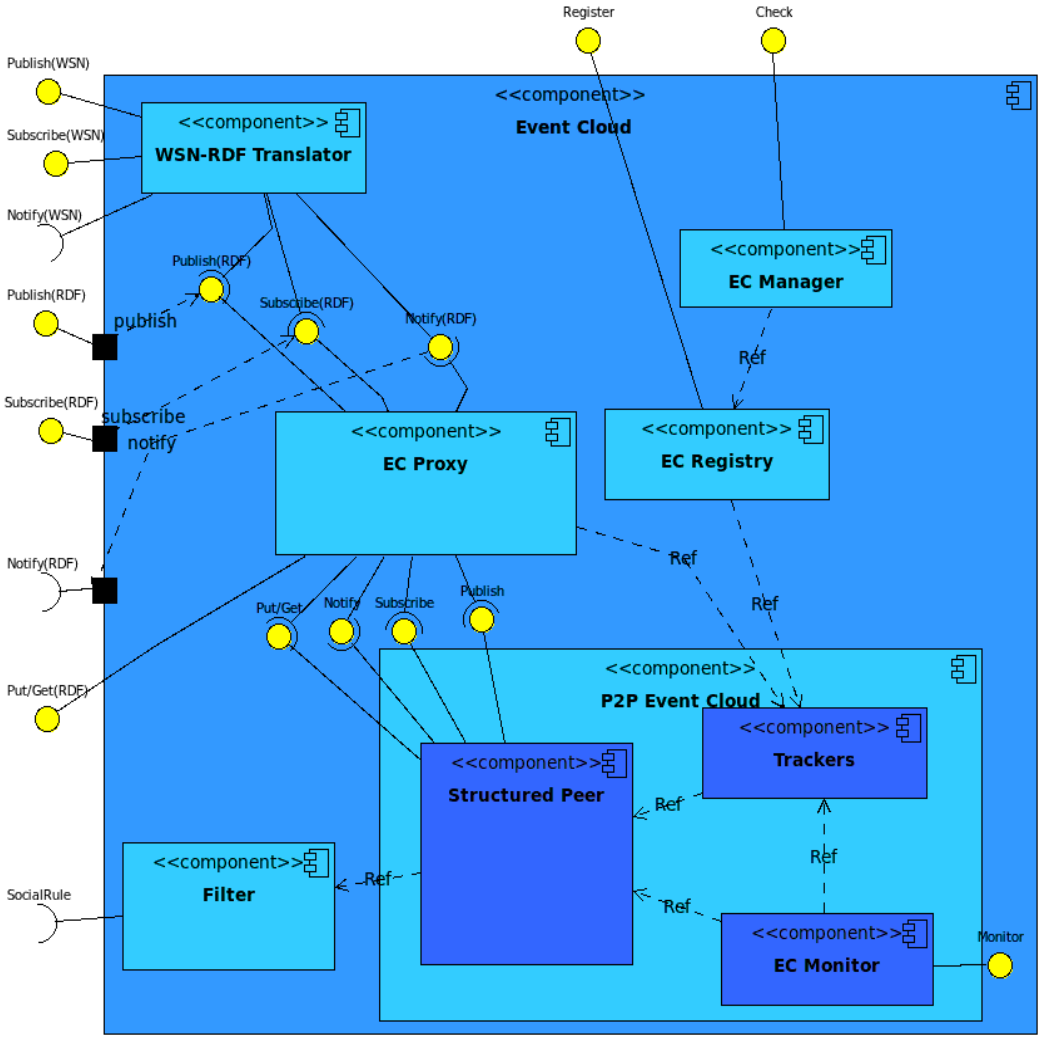


4.1.1.2. Technical Requirements

The Event Cloud requires Java 6 or higher to run. On some node deployments, specific network ports needs to be open to enable communication with external entities. It requires the following functionalities:

- Event storage (temporary and/or permanently) through:
  - Put/Get API to populate “synchronously” the store with, for example, data from a given database
  - Publish/Subscribe API to allow higher scalability
- Quality of Service corresponding to Event Level Agreements
  - Number of notifications delivered to subscribers per time unit
  - Event delivery ordering (or at least out-of-order detection) in order to guarantee fairness regarding a notification point of view, events should better be delivered in a preserved order
  - Event delivery guarantee even in case of non-robust or malicious nodes
- Flexibility, elasticity, portability, easiness for someone to deploy and to use.

4.1.2. Components



4.1.3. Interfaces

The Event Cloud exposes the following interfaces:

- ***Publish(WSN):*** a web service which proposing the publish functionality to the Event Cloud. The client can bind to this web service to send a notification event using the standard WS-Notification message. The client of this web Service could be the DSB, the DiCEP or an external Web Service.
- ***Subscribe(WSN)***: a web service which exposes the subscribe functionality to the Event Cloud. The client subscribe to a stream using the standard WS-Notification subscription message. The Client is either the DSB, the DiCEP or and external Web Service.

- ***Publish/Subscribe(RDF)***: Two Interfaces that offer the functionality of publish/subscribe of events to the Event Cloud using the format RDF of the Events. The using of this interface allow the client to get an Event Cloud proxy directly without a translation process.
- ***Put/Get(RDF)***: Two Interfaces that offer the functionality of storing and retrieving events synchronously. This is also used to retrieve historical events.
- **Event Cloud Managemet Interfaces**: Manage the event clouds (e.g. to create a stream, to configure the elasticity of a stream, etc.). This includes: the ***Register*** that offer the registry point of the Event Cloud that a client bind into; the ***Check*** which offers the check of the availability of Event Cloud Identifier and the ***Monitor*** which Manages the specified stream in non-functional ways such as monitoring data (hardware like the memory consumption but also virtual like the number of subscriptions).

4.2. WSNotif2RDF

4.2.1. Requirements

4.2.1.1. Functionality

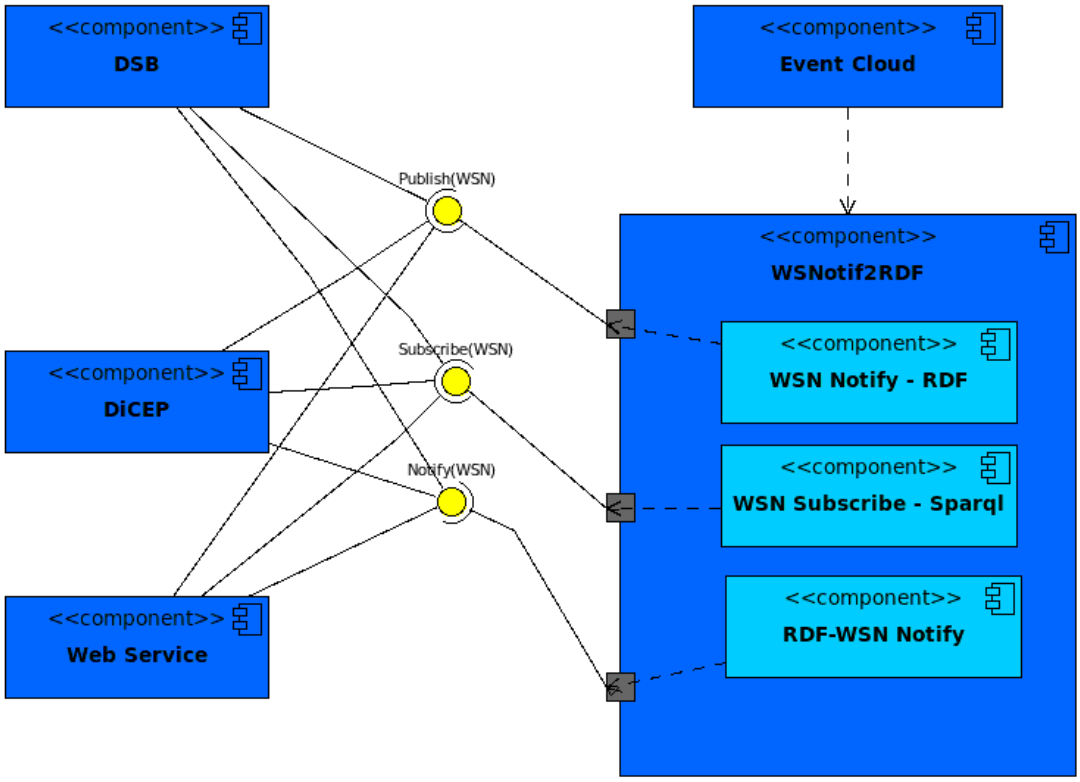
The WSNotif2RDF is a component that translates the WS-Notification standard event payload into the corresponding RDF event elements (quadruples) that can be treated directly by the Event Cloud. It is doing the following translations:

- From a WS-Notification notification payload to an Event Java Object (which is composed of quadruples)
- From a Java Event Object (which has been generated from the previous translation) to a WS-Notification notification payload
- From a WS-Notification subscription to a SPARQL query

4.2.1.2. Technical Requirements

Regarding the algorithms, all is based on the initial WS-Notification notification payload to Event translation. The idea for this translation is to translate the XML tree into RDF quadruples by keeping the tree into the quadruples predicate component and by outputting a quadruple for each leaf of the XML tree. Then from the set of quadruples it is possible to construct an Event.

4.2.2. Components



4.2.3. Interfaces

The WSNotfi2RDF component provides the following Interfaces:

- ***TranslateEventToWsNotifNotification***: Translates an Event to a WS-Notification notification XML payload.
- ***TranslateWsNotifNotificationToEvent***: Translates a xmlPayload standing for a WS-Notification notification to an Event where the literal values associated to the quadruple contained by the event are annotated by using the xsdPayload.
- ***TranslateWsNotifSubscriptionToSparqlQuery***: Translates a xmlPayload standing for a WS-Notification subscription to a SPARQL query as String.

4.3. Filters

4.3.1. Requirements

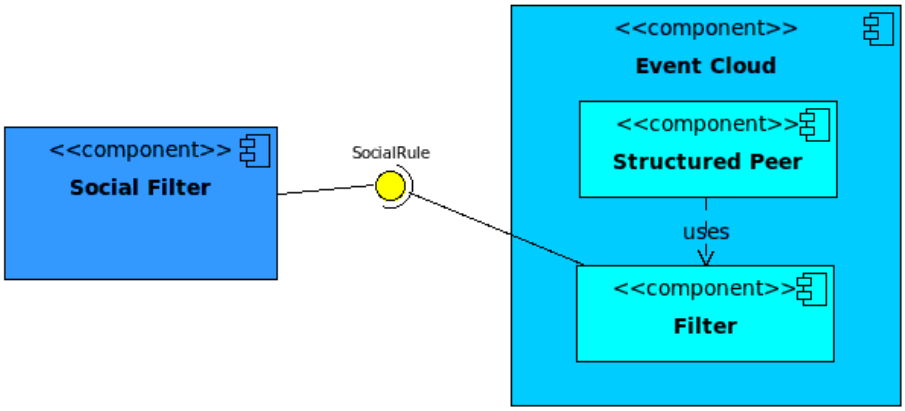
4.3.1.1. Functionality

The Filters of the Event Clouds are used in association with the matching process of Events. When a match process need to determine if the matching is OK so it needs to verify if the rule that has been supplied by the Social Filter allows such event (for example a trusted notifier).

4.3.1.2. Technical Requirements

The Filter Component requires a binding to the social filter component. The social filter component provides the social rule interface through a Web service. In this case the Filter of the Event Cloud has to instantiate a client of this Social Filter Component web Service.

4.3.2. Components



4.3.3. Interfaces

The Filter Component of the Event Cloud requires the Interface provided by the Social Filter component which is the ***SocialRule***.

4.4. Social Event Filter

4.4.1. Requirements

4.4.1.1. Functionality

The Event Cloud component (or any other external component) submits the IDs of two services to the Social Filter. The IDs are given as *source_node* and *target_node*. The Social Filter computes the strength of the relationship of the given *source_node* in the *target_node*. The Relationship Strength Engine component of the Social Filter uses techniques such as trust recommendation and propagation for inferring the relationship strength. The engine operates on social network data which is maintained in a local database. The local social network data is updated by the Social Network Manager component. The social network manager interfaces with external components such as the Social Editor to receive updates on the social relationships between services.

The Social Filter can also take as input the identities of a source node and a list of target nodes. The Social Filter then operates on the social network data and computes the following results:

1) The ranking of the target nodes in terms of the amount of trust that the source node holds in them
2) The set of top *k* target nodes in terms of the amount of trust that the source node holds in them, where *k* is a predefined constant less than the size of the list of target nodes
3) The quantified trust of the source node in each of the target nodes
4) A binary value corresponding to each target node suggesting whether the source node should trust that target node or not.

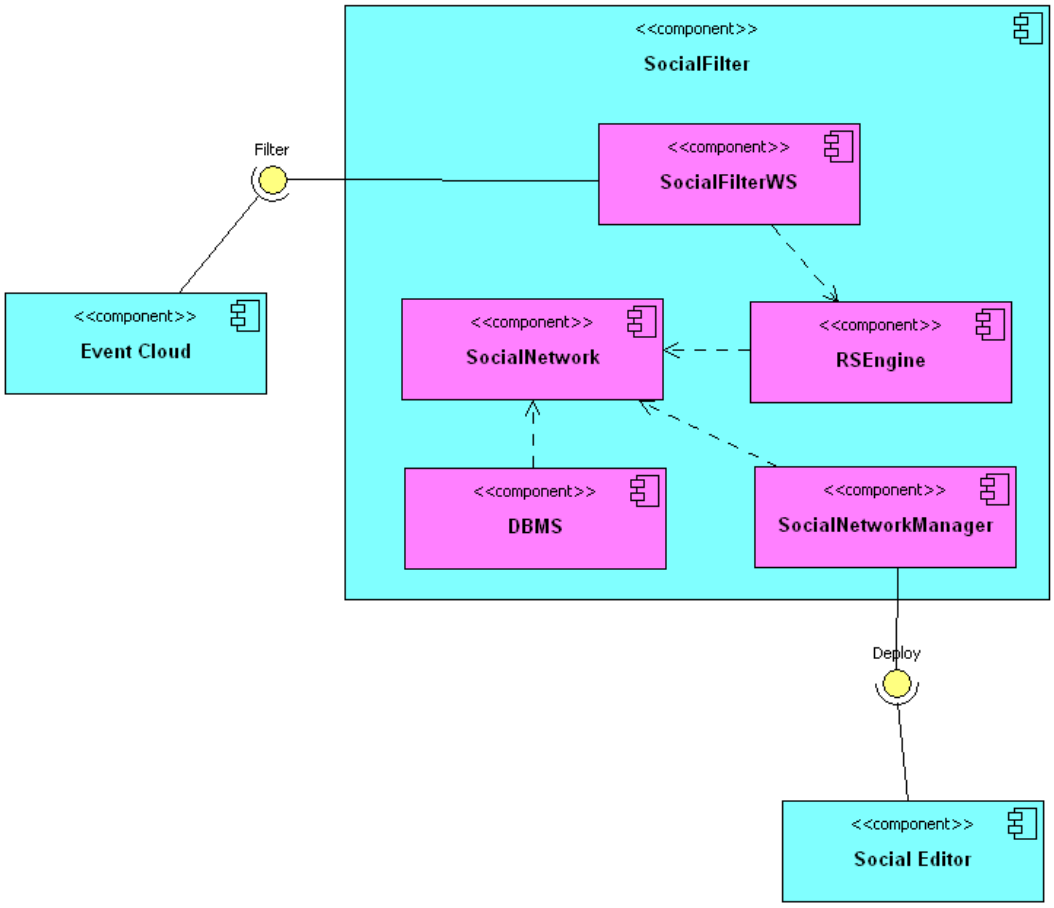
The trust from a source node to a target node can be considered as the strength of the relationship.

4.4.1.2. Technical Requirements

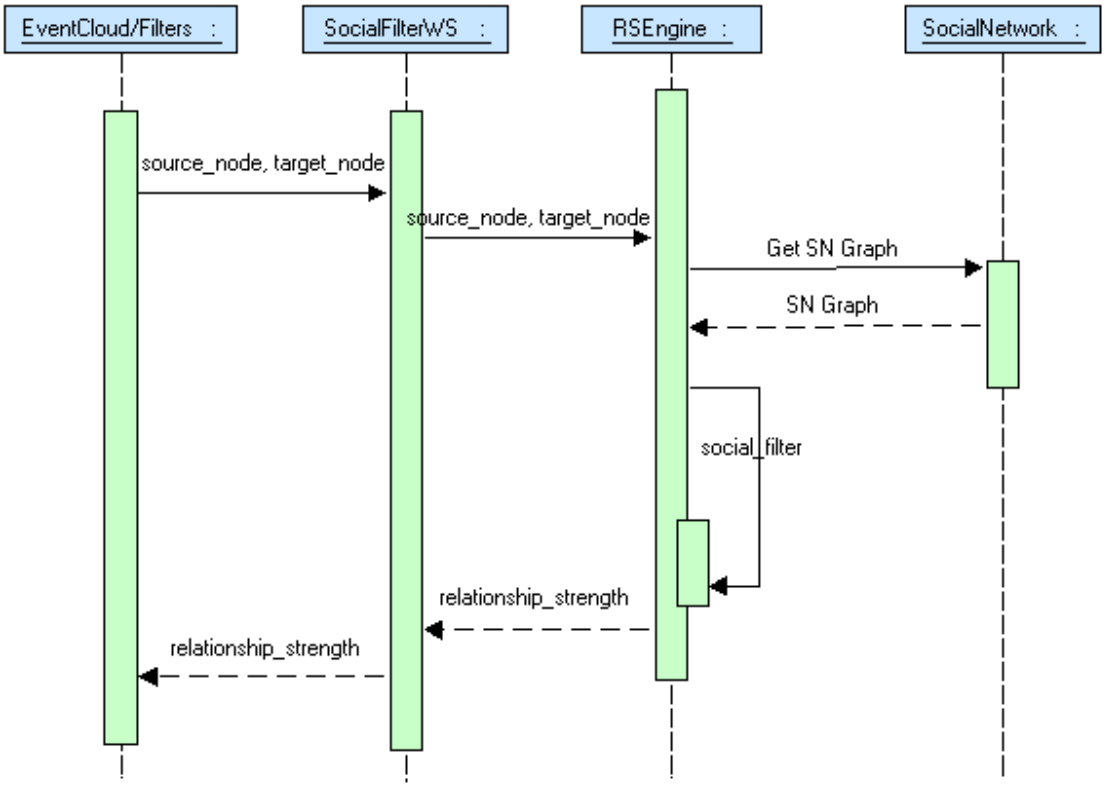
The Social Filter provides web service based interfaces for external components.

4.4.2. Components

The Social Filter component has a sub component called the Social Filter Web Service which exposes a web service interface to external components. An external component submits the IDs of a *source_node* and a *target_node* and obtains the social relationship strength between the two nodes through this interface. The Social Network Manager component updates the locally maintained social network data. The Social Network Manager provides an interface which external components can use to push updates about social relationships between nodes. The function of the other sub components is as discussed in the previous section.



The following figure shows a sequence diagram for the interaction of the *SocialFilterWS* component with the external *EventCloud/Filters* component. The *EventCloud/Filters* invokes the *SocialFilterWS* by submitting the IDs of a source node and a target node. The *SocialFilterWS* calls the *RSEngine*. The *RSEngine* in turn retrieves related social network information from the *SocialNetwork* component and runs the social filter algorithms to compute the strength of the relationship between the source and the target node. The *RSEngine* returns the relationship strength to the *SocialFilterWS*, which conveys it to the *EventCloud/Filters*.



4.4.3. Interfaces

Deploy: interface used by external components to push updates about social relationships between nodes.

Filter: an external component submits the IDs of a *source_node* and a *target_node* and receives information about the social relationship strength between the two nodes through this interface.

4.5. DiCEPE

4.5.1. Requirements

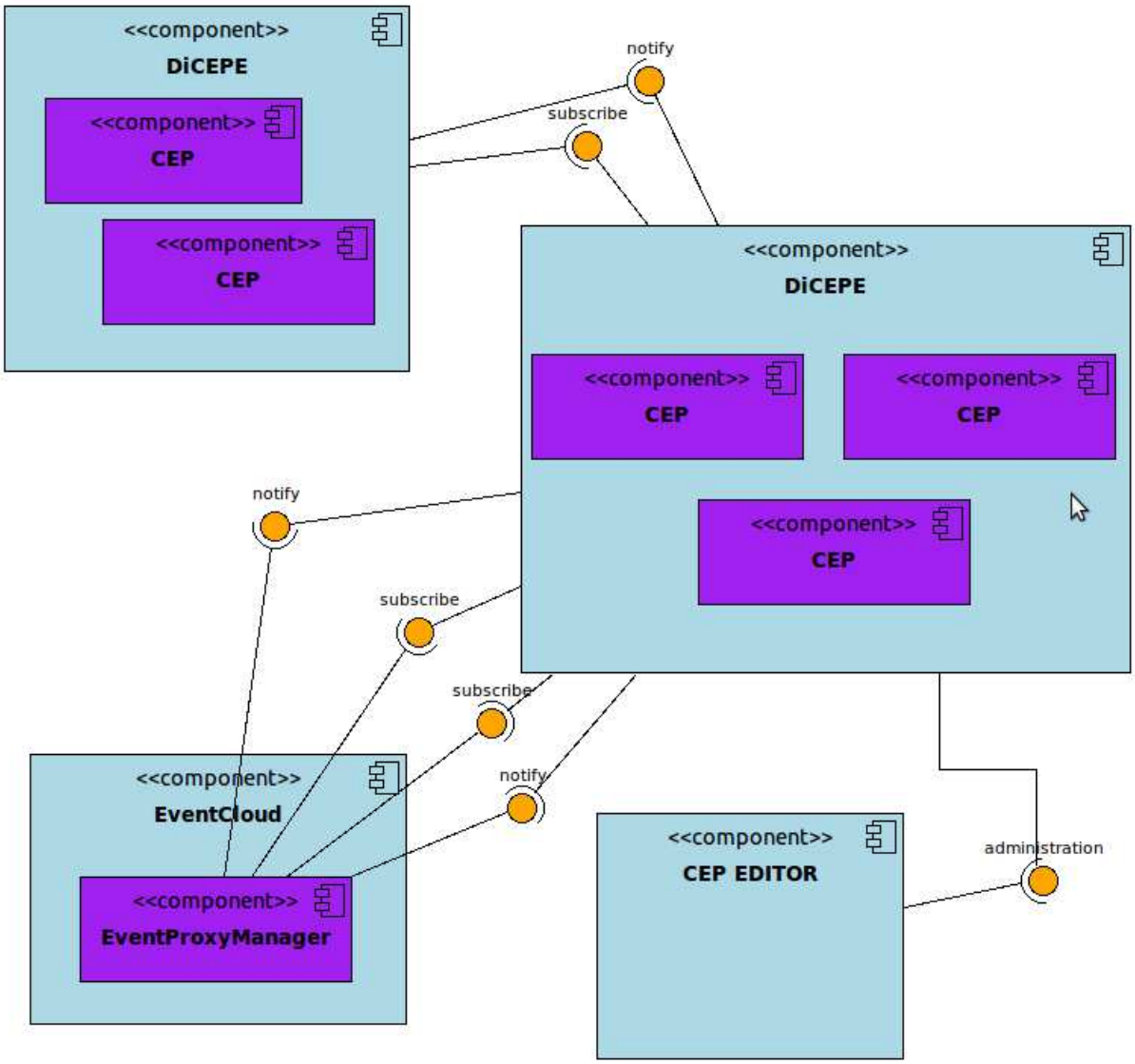
Events from various sources such as federated SOAs, and other can become very complex and difficult to process with centralized CEP architecture, which requires greater bandwidth and computational capability and Lacks of robustness and scalability because of single point failure or network break. The DiCEPE (Distributed Complex Event Processing Engine) spreads centralized CEP tasks load over multiple communicating stations by a space-based communication paradigm over multiple CEP. It subscribes to events published by Event Cloud and produces complex events to the workflow engine. DiCEPE is managed trough the CEP EDITOR which interacts with the administration interface to Create/Read/Update/Delete statements.

4.5.1.1. Functionality

DiCEPE subscribes to a topic which provides events, this connection is done by the EventProxyManager. When DiCEPE produces events, it notifies them to the EventCloud. DiCEPE supports many protocols of communication, such as HTTP, JMS, and SOAP.

4.5.1.2. Technical Requirements

Knowledge of web service, in fact the DiCEPE is designed as SCA components with FraSCAti. Then the interfaces are exposed as web service and support many communication protocols such as HTTP, JMS, and SOAP.



4.5.2. Interfaces

Subscribe : DiCEPE subscribe send a subscribe message to a NoticationProducer in order to register the interest of a NotificationConsumer for NotificationMessages related to one or more topics.

Notify : DiCEPE send notifies message to the NotificationConsumer in order to deliver one or more NotificationsMessage(s).

Administration : Allow the CEP EDITOR to manage the DiCEPE.

administration	
Methods	Description
listAllStatements	This interface return the list of all statements currently loaded in DiCEPE .
getStatementById	This interface retrieves the statement associated to a given identifier.
addStatement	This interface is used to upload an new statement in DiCEPE.
updateStatement	This interface is used to update a statement from DiCEPE.
deleteStatement	This interface is used to delete a statement from DiCEPE.

4.6. Monitoring

4.6.1. Requirements

4.6.1.1. Functionality

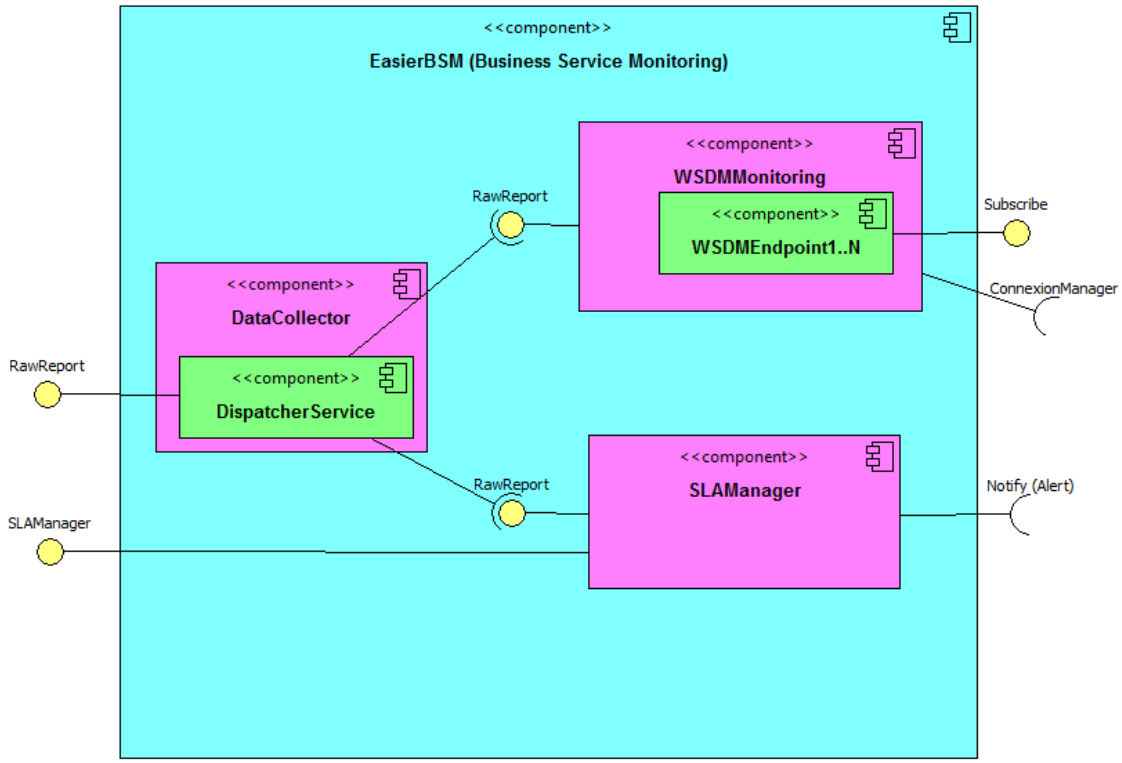
EasierBSM (for Business Service Monitoring) is a framework able to monitor services deployed on a insfractucture of services such as PEtALS-DSB.

It is also able to control the agreements negociated between clients and providers.

4.6.1.2. Technical Requirements

EasierBSM needs Java 5 or higher to run. On some node deployments, specific network ports needs to be open to enable node to node communication.

4.6.2. Components



EasierBSM is composed of three main components:

- The Data Collector is in charge of collect all reports sent by the service infrastructure.

- The WSDM Monitoring create a WSDM non functional endpoint when a functional endpoint is created on the service infrastructure. To do this, it is connected with ESB using connexionManager interface. Then, it is possible to subscribe on each WSDM endpoints to get non functional current properties like the latency for example.

- The SLA Manager is able to send alert (using notify interface) when a business exchange in the ESB is in violation with the agreement loaded.

4.6.3. Interfaces

EasierBSM exposes by default these following interfaces:

External Interface Name	Operations List	Used by component
SLAManager	- LoadAgreementResponse loadAgreement(LoadAgreementRequest parameters); - LoadAgreementsResponse loadAgreements(LoadAgreementsRequest parameters); - LoadAgreementResponse load(LoadAgreementFromUrlRequest parameters);	Monitoring Client
RawReport	- void addNewReportList(ReportListRequest);	PEtALS DSB
Notify	- void notify(NotifyRequest request)	Monitoring Client

5. Conclusion

In conclusion, this delivrable exposes the first architecture of Soceda Framework. This framework is able to connect easily producers and consumers between them and so to deploy large scale EDA applications. The goal is to create a big market place on which it is possible to realize specific subscriptions based on non-functional properties such as the trust on the source of events using social filter. From primitive events, more complex events can be produced owing to DiCEPE (DIstributed Complex Event Processing Engine). To implement this kind of framework, producers and consumers must be easily connected and identified. It is the goal of a service infrastructure like PEtALS DSB and a governance tools like EasierGOV.