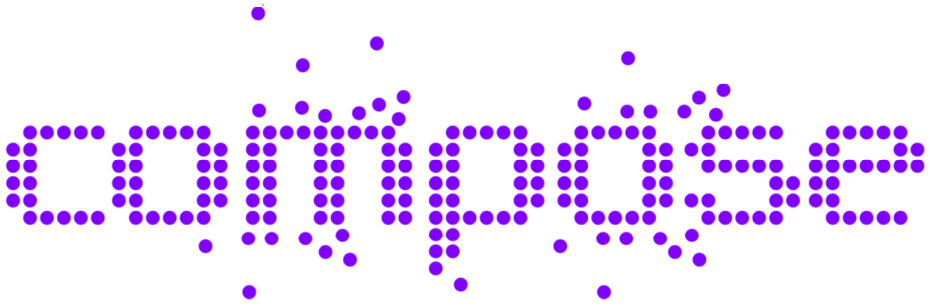

Collaborative Open Market to Place Objects at your Service

D3.1.2.2

Service recommender – Final prototype

Project Acronym	COMPOSE	
Project Title	Collaborative Open Market to Place Objects at your Service	
Project Number	317862	
Work Package	WP3.1 Service Management	
Lead Beneficiary	OU	
Editor	Luca Panziera	OU
Reviewer	Dave Raggett	W3C
Reviewer	Benny Mandler	IBM
Dissemination Level	PU	
Contractual Delivery Date	30/04/2015	
Actual Delivery Date	30/04/2015	
Version	V1.0	

Abstract

The service recommender is a subcomponent of the service management infrastructure. The aim of this module is to assess which are the best services or service objects according to several criteria that matches users' needs. By taking a service description as input, the recommendation is implemented through a four-step process composed of similarity evaluation, filtering, scoring and ranking activities. The recommender returns a ranked list of services, which are similar to the input one. Top-ranked services are the best according to recommendation criteria.

For the final prototype of the recommender, filtering and scoring are implemented for evaluating the following criteria: (i) service trust, (ii) service usage and (iii) sensor status of service objects.

The main new feature of the final prototype is a similarity engine, which provides support for content-based recommendation of services. In addition, criteria evaluation techniques have been improved in terms of both effectiveness and efficiency.

This document describes the high level architecture of the recommender and the novel algorithms and techniques that implement the overall recommendation process for the final prototype.

Document History

Version	Date	Comments
V0.1	03/03/2015	Deliverable skeleton
V0.2	09/04/2015	First draft
V1.0	29/04/2015	Final version

Table of Contents

List of Figures

Acronyms

Acronym	Meaning
COMPOSE	Collaborative Open Market to Place Objects at your Service
NFP	Non-functional property
QoS	Quality of Service

1 Introduction

The service management work package is based on three main components (see Figure 1):

1. An advanced linked services discovery engine, whose job is to discover distributed and heterogeneous COMPOSE entities. The service discovery engine is layered on top of a service registry, which exploits information retrieval and semantic search and storage technologies.
2. An advanced service recommender system, which is in charge of suggesting new relevant services based on users' previous interactions, similarity between services, and other non-functional properties such as performance, trust, etc.
3. An assisted service composition engine, which is meant to help users create new composite services by (semi) automatically combining existing services to obtain the desired functionality.

Both the service recommender and the service composition engines leverage the service discovery engine, as shown in Figure 1.

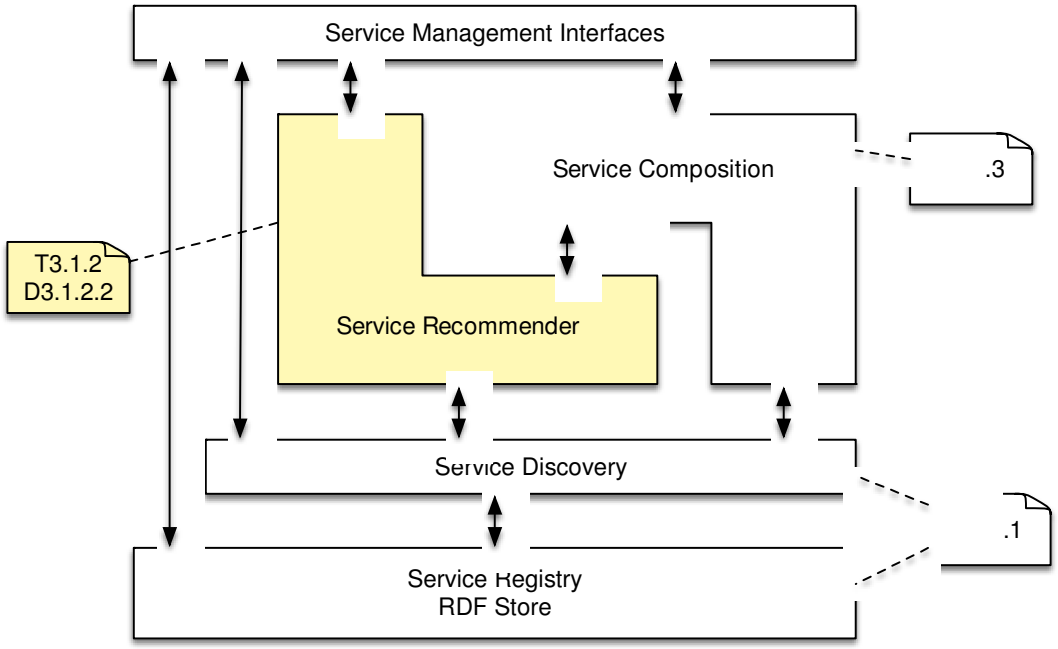

Figure 1: A simplified logical view of service management components

This deliverable provides the final prototype architecture and implementation of the service recommender. The objective of the service recommender is to provide functionalities for suggesting the best service according to criteria that are relevant for users. The service recommendation implements techniques for computing service similarity, on a functional

prospective, and ranking and filtering based on non-functional properties (NFPs) of services. The final prototype of the recommender will evaluate the following NFPs of services:

- trust and reputation (as result of WP5);
- service usage;
- sensors status associated with a service.

The recommender has been designed through a modular and flexible architecture that allows COMPOSE platform providers to remove and integrate service evaluator in a seamless way. As well as the composition engine, the recommender is managed as a plugin by the discovery engine based on iServe (details in D3.1.1.2).

2 Service recommender design

This section provides details about the design of the service recommender. The design is based on the following approach. As initial step, actors that interact with the recommender are identified. Then, functionalities and interfaces of the recommender are defined. Finally, an overall architecture of the software component is provided.

2.1 Actors and interactions

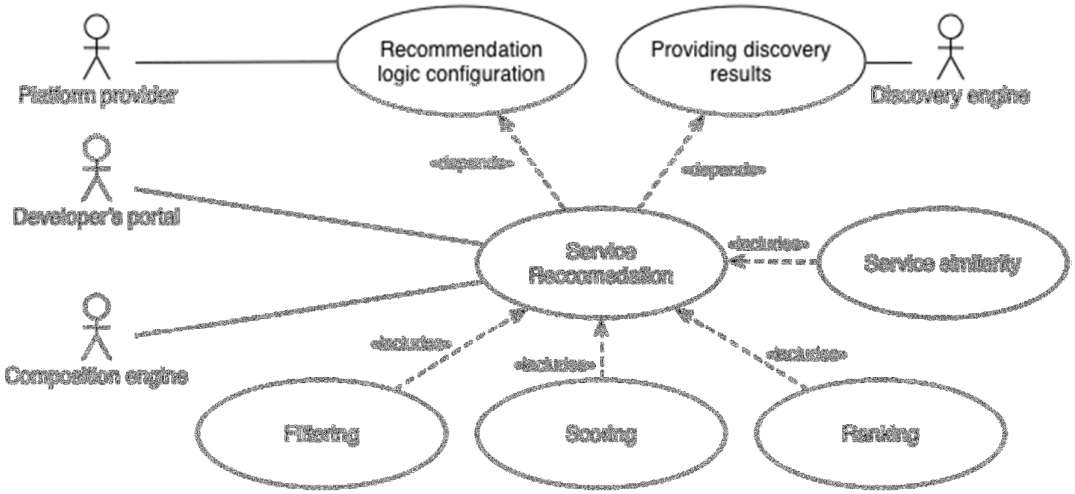

Figure 2: Actors and interactions with the recommender

Four actors that interact with the service recommender are identified, as show in Figure 2: developer's portal, composition engine, discovery engine and platform provider. The developer's portal (result of WP6) provides a graphical user interface (GUI) for recommendation functionalities. The composition engine (T3.1.3) uses recommendation functionalities for evaluating NFPs of candidate services for compositions.

Service recommendation includes four activities: service similarity filtering, scoring and ranking. Given a service description, service similarity returns services that implement the same or similar functionalities. Filtering excludes services that do not guarantee particular criteria (e.g., reputation). Scoring will assign a numeric value to each service, which assesses the service quality according to specific criteria (e.g., usage). Finally, the ranking returns an order services based on scoring results. The configuration of recommendation logic is defined by the provider of the COMPOSE platform.

The role of the discovery engine is to provide functionalities to support service similarity by exploiting text search and semantic matching techniques.

2.2 Architecture

The architecture of the service recommender is composed of three layers, as shown in Figure 3. The first layer provides the interface to the component, which is implemented through REST and APIs of the language adopted for the implementation. On the one hand, the RESTful interface provides access to the recommendation functionalities to third party software as a service. On the other hand, the interface based on native APIs allows COMPOSE platform modules a more efficient communication with the service recommender.

The second layer is the executor of the recommendation logic. This logic implements service similarity, filtering, scoring and ranking according to the configuration defined by the provider of the COMPOSE platform.

Filter plugin manager and scorer plugin manager are the components of the third layer. The first module manages filters. Filters are components with a common interface. Each filter assesses if a service meets specific criteria (e.g., trust), therefore if the service must be excluded by the recommendation results. The latter manager runs scores. Similar to filter, scores share a common interface. Each scorer evaluates a characteristic of each specific service (e.g., provider popularity). The result of the evaluation is a score that represents the quality of the service characteristic. Scores are combined in a global score that will be used to rank services.

Both mangers will invoke, respectively, filters and scores according to the recommendation logic defined by the platform provider. The management of filters and scorers as plugins has several advantages:

- to improve the efficiency and effectiveness of the recommender by adding or substituting filtering and scoring strategies;
- to adopt different recommendation approaches based on the service domain;
- to tune the recommendation in order to fulfil better user needs.

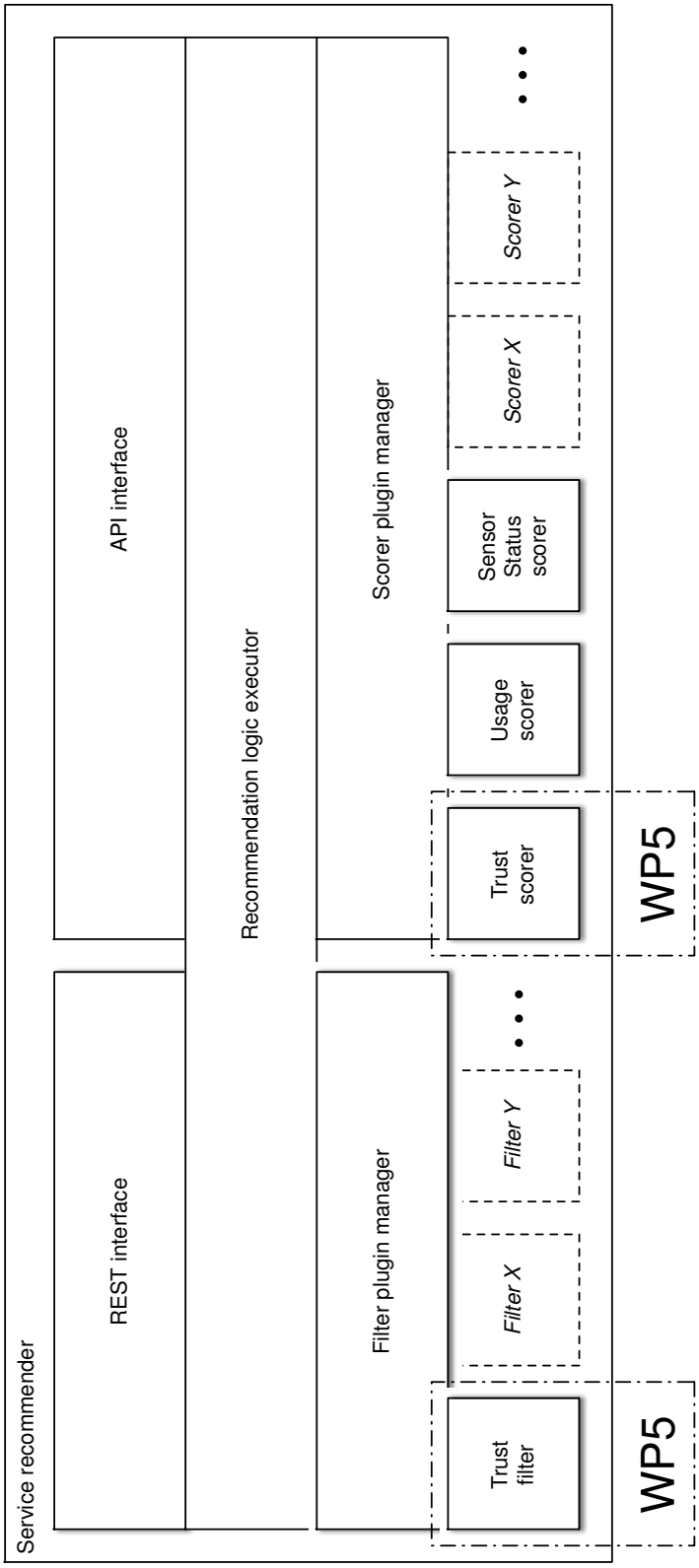

Figure 3: Service recommender architecture

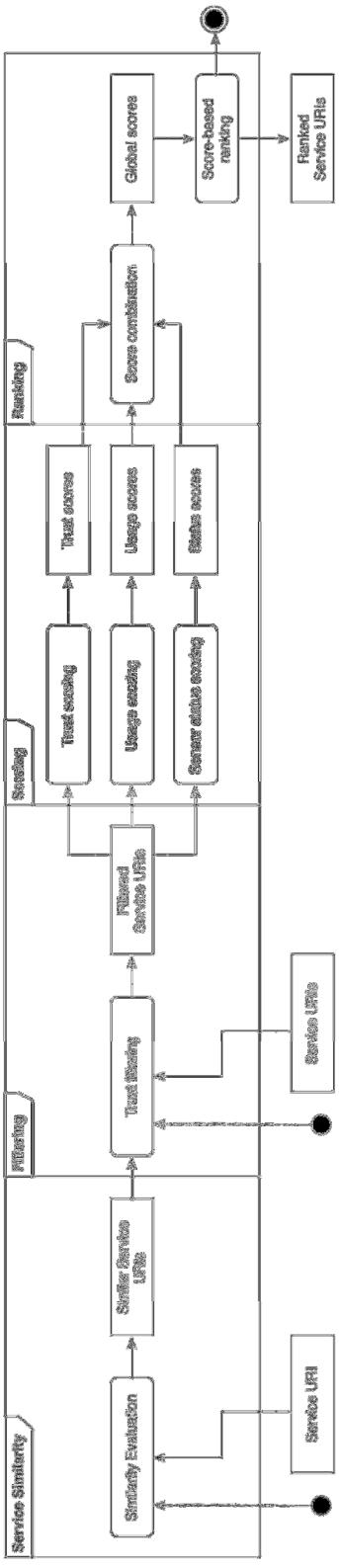

Figure 4: Recommendation process

2.3 Recommendation process

The recommendation process is composed of four main activities, as introduced in section 2.1: service similarity evaluation, filtering, scoring and ranking. These activities are sequential and composed of sub-activities as shown in Figure 4. The process can start with the service similarity evaluation or filtering phase. The service similarity evaluation receives a service URI as input, and then it returns a list of similar services as output. The filtering phase can receive a list of service URIs as input, which can be the result of similarity evaluation or a list defined by the user. The second phase performs trust filtering, which returns a filtered URI list of services that fulfil some trust criteria. The third phase performs three different scoring strategies. For each filtered service, scores are returned for evaluating (i) service usage, (ii) status of sensors as service objects and (iii) trust. The ranking phase combines all the scores computed for each service through an average in order to produce a global score for each service. Each global score is exploited for ranking services.

2.4 Functionality implementation

The service recommender is implemented in Java by exploiting Guava libraries. Guava provides efficient manipulation functions of data structures therefore it allows an efficient implementation of the recommendation process.

2.4.1 Java API and RESTful interfaces

The recommender functionalities are available through the `ServiceRecommender`. The first class provides the `recommend` method that receives a recommendation request as input. The method output is a `SortedMap`, which contains service URIs of recommended services as map key and a numeric rank score as map value. The map keys are sorted according the rank score.

The recommendation request is specified through a JSON model composed of four parts:

- Services – it specifies a list of services;
- Filtering – it specifies the filtering strategy;
- Scoring – it specifies the scorers list necessary for the recommendation ranking
- Ranking – it specifies the ranking strategy.

The high level structure of the recommendation request is provided below:

```
{
    "services": [ ... ],
    "filtering": [ ... ],
    "scoring": [ ... ],
    "ranking": " ... "
}
```

The services part is a JSON array that contains a set of service URIs. If the array contains a single URI, service similarity evaluation will be performed, otherwise the list of URIs will be passed directly to the filtering phase.

Filtering strategy is defined as JSON array of "filter" objects, as shown in the example below:

```
"filtering":[
 {
  filterClass":"com.inn.trusthings.integration.TrustFilterByThreshold",
  "parameters":{
    "attributes": [
      {
       "type":"ProviderWebReputationBy3rdParty",
       "importance": 1,
       "value": "0"
      }, ...
 ]
```

Each "filter" object has two attributes:

- `filterClass` (mandatory): defines the Java class that implements the filter;
- `parameters` (optional): defines the configuration parameters of the filter as JSON object.

The "parameters" object model depends on the filter implementation. The sequence of "filter" objects defines the sequence of applied filters.

The scoring is specified as a JSON array of "scorer" objects. Each scorer returns a score according to a service property (e.g., popularity, reputation). "Scorer" objects are similar to "Filter" objects as provided below:

```
"scoring":[
 {
  scorerClass":"com.inn.trusthings.integration.TrustScorer",
  "parameters":{
    "attributes": [
      {
       "type":"ProviderWebReputationBy3rdParty",
       "importance": 1,
       "value": "0"
      }, ...
 ]
```

The ranking is specified as a string, which can assume two values:

- `standard`: it ranks services by decreasing order according to the scores;
- `inverse`: it ranks services/operations by increasing order according to the scores.

If ranking is not specified, standard ranking is performed by default.

The following listing shows an example of recommender invocation through Java APIs.

```
String request = "{\"service\":[ ... }\";

ServiceRecommender recommender = new ServiceRecommender();
SortedMap<URI,Double> result = recommender.recommend(request);
```

The same functionality is also available through a RESTful interface. The interaction is implemented through a HTTP POST that receives as input the JSON request.

2.4.2 Filters and Scorers

Filters and scores are the main modules that implement the recommendation logic. These components are implementations, respectively, of the `Filter` and `Scorer` Java interfaces. The following listing shows the two interfaces.

```
public interface Filter {
  public boolean apply(URI serviceId);
  public boolean apply(URI serviceId, Object parameters);
}

public interface Scorer {
  public Double apply(URI serviceId);
  public Double apply(URI serviceId, Object parameters);
}
```

Each Filter implements its filtering logic through the `apply` method. This method returns `false` if a service must be excluded because does not meet some criteria (e.g., an acceptable trust level), otherwise it returns `true`.

A scorer apply method implements the algorithm that assess a particular service characteristic (e.g., provider popularity). The method returns a value between 0 and 1. The higher the returned value, the higher is the assessment of the characteristic. For instance, a scorer that assesses popularity of service providers returns a value close to 1 for services with very popular providers. Instead, the scorer returns a value close to 0 if a service has a provider with low popularity.

3 Final prototype features

This section describes the features implemented by the final prototype. The description of the approach for computing Service similarity is provided in section 3.1. Then, section 3.2 describes techniques for trust evaluation. Section 3.3 shows the approach for evaluation service usage. Finally, algorithms for evaluating status of sensor, registered as service objects, are available in section 3.4.

3.1 Service similarity evaluation

The technique to evaluate service similarity is an application of the state-of-the-art approach available in [1] on MSM descriptions, the formalisms adopted to describe services by the COMPOSE platform (see deliverable D1.3.2).

The following pseudo-code algorithm describes the service similarity approach.

```
function ServiceSimilarity(service) {
  R = emptySet();
  C = getComponents(service);

  foreach component in C {
    A = getServicesWithSimilarTermsInComponent(component);
    B = getServicesWithSimilarModelReferenceOfComponent(component);
    R = union(R,A);
    R = union(R,B);
  }

  foreach <uri,score> in R {
    score = score / (sizeOf(P) + 1);
  }
  return R;
}

function union(X,Y){
  for <uri,scoreX> in X {
    if(uri in Y) {
      <uri,scoreY> = getPair(uri);
      scoreY += scoreX;
    } else {
      Y <- <uri,scoreX>;
    }
  }
  return Y;
}
```

According to MSM, each *service* has *operations* and each *operation* has *message contents* as *inputs* or *outputs*. *Operations*, *inputs* and *outputs* are components of the *service*. For each service component, two functions are applied.

The first function returns a set *A* of pairs <*URI*,*score*> in which *URI* is the URI of a service that has the same kind of component described with similar terms and *score* is a similarity score in range [0,1]. For instance, if an operation input described by the label "temperature" is the input component, the function put in the result set the URI of services that have an input labelled as "local temperature" with a score 0.5. The similarity score is computed according to [1].

Instead, the second function computes a semantic similarity of a component. The result set B contains pairs <*URI*,*score*>, where *URI* is the URI of a service that has a subsumption path between the model references of the input component and one of its own component of the same kind and *score* is similarity score value between 0 and 1. For example, let's assume that Schema.org is used to define model references of service components. If the input of the function is an operation with Activate Action[1] as model reference, the URI of a service with an

[1] http://schema.org/ActivateAction

operation that has Control Action[2] will be in the result set, because there is a subsumption relation between the two model references. The similarity score will be 0.66 according to the length of the subsumption path, as shown in [1].

The result sets for each component are combined through the *union* function in order to compute an overall similarity score for each service that has at least a component similar to the input service. Finally, the last *for* loop normalise the overall score of each similar service between 0 and 1.

In order to optimise the efficiency of the similarity evaluation the provided algorithm is developed through a multithread implementation with caching strategies of partial results.

3.2 Trust evaluation

In the first prototype of the service recommender we have introduced an approach for the trust evaluation of services in the COMPOSE platform and provided an initial implementation of the main functional aspects, namely trust filtering and scoring. In addition, a trust-based ranking was developed. The initial prototype demonstrated the main functional aspects, however, without the demonstration of integration with the service recommendation logic and without the deployment of the trust engine into the first integrated COMPOSE cloud platform (shown at M24). These integration tasks were completed after the initial prototype.

From the perspective of COMPOSE service discovery and recommendation, trust is contemplated as a subjective, multidimensional, and context-dependent concept. That actually means that the trust expectations (i.e. trust-related attribute of services) may be of different relevance in different contexts, or be different in the same context for different users, given the users past experiences and skills. In our approach, when the service's trust guarantees match the service consumer's trust expectations, in the service consumer context, then the service is considered as trustworthy.

The trust expectation, or trust criteria, is a set of weighted trust dimensions (i.e. service NFPs related to the trust) and their values expected by a service consumer. We refer a reader to the deliverable D3.1.2.1 for the details about the trust scoring and filtering approach taken in COMPOSE, while in the following we highlight the main redesign changes and new features introduced in the final version of the prototype.

Changes and new features in the final prototype

Specifically, these are the changes and new features added:

- At the back-end side of the trust evaluation engine, we prototyped a data store (MySQL) where service trust profiles (i.e., trust-related attributes and their values) can be centrally kept for the purpose of the trust evaluation. The trust profiles could be inserted into there by a service developer (via an appropriate GUI or REST API) or the data store can be populated by the trust information collectors (e.g. by a Reputation data collector).
- Trust evaluation engine is refactored into a web service, using a standard and portable JAX-RS API Jersey RESTful Web Services framework, in order to integrate it into the prototype COMPOSE architecture and to foster future re-use of the module. The web service has been integrated into the COMPOSE Cloud Foundry (at M24) as a java application running in an Apache Tomcat container, bound to a MySQL service.

[2] http://schema.org/ControlAction

The web service exposes the following operations:

(1) POST http://trusthings.147.83.30.133.xip.io/trust/score - for the trust scoring

Request Header: Content-Type:application/json

Request Body: JSON

```
{
    "resources": [{ <resourceURI>, <resourceURI>,
<resourceURI>…}],
    "parameters": {
        "attributes": [ <type, value, minvalue, maxvalue,
importance>, <type, value, minvalue, maxvalue, importance>…] },
    "strategy": <strategy>
}
```

A field "resources" is an array of URIs that identify services, which are input for trust scoring. The "attributes" is a specified trust criterion, as an array of desired trust-related attributes, their desired values and weights. Then, the "strategy" field tells the trust engine which trust scoring strategy to apply. It may be either a 'standard" (weighted sum model) or "topsis" strategy, as we have introduced in the D3.1.2.1. We will explain the JSON-based syntax of trust criteria ("attributes" field) in the next bullet - it is a new feature added to the final prototype.

Response - Content-Type:application/json; Status code: 200

```
{
"success": "true",
  "result": [ <resource score>, <resource score>]
}
```

where "result" is an array of <resource score> attributes that contain a service URI (field "resourceURI"), its trust score (field "index") and its rank (field "rank"). E.g.,

```
{
   "success": "true",
   "result": [{
      "resourceURI": "http://example.com/weather-service",
      "index": 0.9,
      "rank": 1
    },
{
     "resourceURI":"http://example.com/myWeatherservice",
     "index": 0.8,
     "rank": 2
    }
]
}
```

(2) POST http://trusthings.147.83.30.133.xip.io/trust/filter/threshold - for the trust filtering using a trust score threshold (set at 0.5)

Request Header: Content-Type:application/json

Request Body: same as the above, but without the "strategy" attribute.

Response - Content-Type:application/json; Status code: 200

```
{
  "success": "true",
  "result": [ <resourceURI>]
}
```

where "result" is an array of <resourceURI> attributes that are URIs of services evaluated as trusted by the filter. E.g.

```
{
  "success": "true",
  "result": [
      {
     "resourceURI":"http://example.com/weather-service"
      },
      {
      "resourceURI":"http://example.com/myWeather-service"
       }
    ]
}
```

(3) POST http://trusthings.147.83.30.133.xip.io/trust/filter/exclusion - for filtering out services whose trust profiles do not satisfy at least one of the trust-related attributes specified in a trust criteria

Request Header: Content-Type:application/json

Request Body: same as the above, also without the "strategy" attribute.

Response - Content-Type:application/json; Status code: 200 – JSON same as the JSON response from POST /trust/filter/threshold

In a case of an error, the trust web service responds with a JSON message:

Response: Content-Type:application/json; charset=UTF-8, Status Code: 500

```
{
  "success" : "false",
  "message" : "error message text here"
}
```

- A JSON-based syntax for specifying trust criteria

```
{"attributes":
      [
      {"type":<trust attribute type>,
      "importance":<weight>,
       "value":< numerical value | a value on some custom scale | an
      expression>,
      "minvalue":<min threshold>,
      "maxvalue": <max threshold>}
      | {"or":<or expression>}
      ]
}
```

The "attributes" is a specified trust criteria as an array of desired trust-related attributes, their desired values and their importance (weight 0 to 1). A vocabulary for the attributes is defined in the COMPOSE Trust ontology, and "type" refers to concepts in that ontology (e.g. http://www.compose-project.eu/ns/web-of-things/trust#Reputation, http://www.compose-project.eu/ns/web-of-things/trust#CertificateAuthorityAttribute, or http://www.compose-project.eu/ns/web-of-things/trust#SecurityGuarantee). For each attribute, its desired value, if needed, can be specified in a field "value". In a case of measurable, quantified attribute, a value is in a metric of that attribute. A "minValue" and "maxValue" thresholds on quantified attributes can be also specified (see the next bullet). In a case of descriptive attributes, such as CertificateAuthority or SecurityGuarantee attribute, a desired value is an expression, as illustrated below:

```
{
"type": "http://www.compose-project.eu/ns/web-of
things/trust#CertificateAuthorityAttribute",
"value":
  {"certificateauthority": [{"type": "http://www.compose-project.eu/ns/web-
of-things/security#EU-Based"}]},
  "importance":1
}
```

or,

```
{
    "type": "http://www.compose-project.eu/ns/web-of-
things/trust#SecurityGuarantee",
"importance": 1,
    "value": {
        "securitygoal": [
            {"type": "http://www.linked-usdl.org/ns/usdl-
sec#Confidentiality"}],
        "securitymechanism": [
            {"type": "http://www.linked-usdl.org/ns/usdl-
sec#Cryptography"}],
        "securitytechnology": [{"type": "http://www.compose-
project.eu/ns/web-of-things/security#TSL"}]
    }}
```

- A "minValue" and "maxValue" thresholds on quantified trust-related attributes are introduced into a trust criteria specification. In this way, a service consumer may express trust expectations such as "I trust to services with popularity greater than 0.5", "I trust services that have more than 100 users", or "I trust services whose response time is not more than 1s". For example, in the JSON-syntax a specification of the trust to services that have popularity index greater than 0.5 can be expressed like:

```
{
"type": "http://www.compose-project.eu/ns/web-of-
things/trust#Popularity",  "importance": 1,  "minValue":"0.6"
}
```

- The trust evaluation algorithm is extended to support an evaluation of trust criteria with a disjunction of descriptive trust attributes of the same type (e.g. Certificate Authority or Security Guarantee attribute). Attributes in the disjunction group can be assigned different weights to state their importance and among them the algorithm finds the best match (best score) to the trust criteria. For example, a service consumer trusts payment services that have certificate authority datum, but trusts more EU-issued certificates than US-issued certificates (importance = 0.5). The evaluation will assign a higher score to the services that have a certificate issued in Europe, regardless if they have or have not also a certificate issued in the US. JSON syntax is

```
"or":{ "importance":<weight> , "attributes":[ < type, value, importance>,
<type, value, importance>…]}
```

E.g.
```
{
 "attributes": [
   {
      "or": {
       "importance": 1,
       "attributes": [{
```

```
            "type": "http://www.compose-project.eu/ns/web-of-
things/trust#CertificateAuthorityAttribute",
            "importance": 0.5,
            "value": {
              "certificateauthority": [
                {
                  "type": "http://www.compose-project.eu/ns/web-of-
things/security#US-Based"
                }]}},  {
            "type": "http://www.compose-project.eu/ns/web-of-
things/trust#CertificateAuthorityAttribute",
            "importance": 1,
            "value": {
              "certificateauthority": [
                {
                  "type": "http://www.compose-project.eu/ns/web-of-
things/security#EU-Based"
                }]}}]}]}
```

- Integration with WP5 Reputation server (PopularIoTy) - the final prototype of the trust engine has a collector for reputation data, which are centrally collected and managed by the Reputation Server (WP5 component). More specifically, the Reputation Server exposes a RESTfull API endpoint (http://docs.popularioty.apiary.io/) to provide on demand, in a batch request manner, the aggregated reputation information including the popularity index, activity monitoring index, count of user feedback, count of service usages, and the overall reputation score. All those attributes are considered as trust-related, therefore, are now supported within the trust evaluation. When needed, the trust engine sends HTTP request to the Reputation Server to get the aggregated reputation information for further trust evaluation.

  In that same manner, the trust-related service monitoring (QoS) attributes, such as an average availability or average response time, could be added into trust profiles. A Monitoring Collector is not prototyped, however, to that end, we have defined a `Collector` Java interface with the `collectInformation(List<URI> resources, Map<URI, Model> map)` signature. Implementation of this signature should populate or update a trust profile (`Model`) with trust-relevant attributes for each service identified with a URI. Implementations of the collector must be registered into the trust engine, simply by adding it into a `collectors.json` file. COMPOSE provides a Monitoring Component (WP3.2) that answers whether a service is available or not. Simply, an index of average availability could be obtained by storing the availability status over the time, and then by calculating a probability distribution of the availability, for example by applying a beta density function for probability of binary events [2].

- Trust scorer and filter Java API implement the final version of scoring and filtering interfaces. The filter and scorer can be invoked like this:

```
Set<URI> set = Sets.newHashSet();
set.add(new URI("http://example.com/weather-service”));
set.add(new URI("http://example.com/myWeather-service”));
//trust criteria is in a json file and we read it from there
InputStream is = getResourceAsStream("myTrusCriteria.json")
String criteria = CharStreams.toString(new InputStreamReader(is));
//invocation of trust filtering
```

```
Set<URI> filteredSet = new TrustFilterByThreshold().apply(set, criteria);
// invocation of trust scoring
Map<URI, Double> scoreMap = new TrustScorer().apply(filteredSet,
criteria);
```

Internally, `TrustFilterByThreshold` and `TrustScorer` make a HTTP POST request to the operations exposed by the trust web service. Importantly, there is only one, a batch HTTP request, for all the service URIs that are subjects to the trust scoring or filtering. The initial prototype performed a single scoring/filtering request per single given service URI, and that lead to a long response time. By implementing a HTTP batch request in the final prototype, we significantly improved the response time.

Finally, all the source code is published to the GitHub source code management and issues tracking system. The repository is located at https://github.com/vujasm/trusthings-compose. In addition to the source code, the repository contains examples of trust criteria and how-to instructions (e.g. invocations of trust filter and scorers, deployment into Cloud Foundry).

3.3 Evaluation of service usage

The novel approach implemented by the first prototype of the recommender was based on the exploitation of Web data that allows the estimation of criteria that drives service usage by developers. Through the measurement of these criteria, we estimate usage.

In order to figure out these criteria, a questionnaire has been submitted to 56 Web developers. Survey results show that the two main criteria are (i) popularity of the provider and (ii) quality of community that support developers' issues. The first criterion highlights that users prefer Web APIs with popular providers. The second one shows that users prefer Web APIs, which provide communities, through forums or mailing lists that better support developers.

The provider popularity is measured as the number of daily visitors to the providers' websites. Our assumption is that the more a provider is popular, the more users visit their website. Alexa[3], a service that monitors network traffic of websites, is used for evaluating provider popularity.

The quality of community support is measured as the number of daily active community members. This metric, which we call "community vitality" for short, is defined on the base of an additional survey. The users of the former survey evaluated the utility of eight metrics proposed in the literature for measuring the health of a community [3]. Survey participants assessed that community vitality is the most useful metrics for assessing support quality. For these measurements, data have been collected from 110 Web API support forums available on Google Groups.

To verify that provider popularity and community vitality can estimate usage, correlation metrics have been computed. On data collected between 2005 and 2013, the correlation evaluations of measurements between the two metrics and usage, computed as number of mashups, show that:

[3] http://www.alexa.com

- Provider popularity is a good usage estimator for Web APIs that are not provided by organizations which offer services in multiple domains (e.g., Google, Yahoo!, Microsoft, Amazon, etc.);
- Community vitality is a good estimator of usage for most Web APIs.

On the base of the correlation results, the usage u_a of a Web API a is computed as follows:

$$u_a = \begin{cases} \frac{v_a}{\max(V)} & \text{if } v_a \in V \\ \frac{p_{\rho_a}}{\max(P)} & \text{if } p_{\rho_a} \in P \\ 0 & \text{otherwise} \end{cases}$$

where v_a is the vitality of the community that support a and p_{ρ_a} is the popularity of the provider ρ of a. V is the set of community vitality measurements and P is the set of popularity measurements for providers that focus on a single domain. Both metrics are exploited for estimating usage because communities' data is not always available or collectable due to licensing restrictions. In addition, Alexa offers data for almost all of Web API provider websites available.

Each measurement of u_a is defined as follows:

$$v_a = \left|U_{c_a,t}\right| \text{ and } p_{\rho_a} = \left|\beta_{\rho_a,d}\right|.$$

$U_{c_a,t}$ is the set of users that posted a message on the forum or mailing list provided by the Web API a in the time period t. $\beta_{\rho_a,t}$ is the set of visitors on ρ_a's website during the day d. The size of t is 60 days and d represents the most recent day in t. Vitality is measured on large time intervals because the amount of daily active community users has a big variance, therefore measurement are affected by noise. By evaluating vitality in a big interval, the noise is smoothed, therefore the measurement better figures out community behaviours. The size of t has been defined empirically based on the Google Groups dataset. We observed that the maximum delay between two posts in the dataset is 27 days. Therefore, the t size has been defined at least twice this period in order to have a statistically representative measurement composed of at least one active user of the community.

Compared to deliverable D3.1.2.2, the main change on the evaluation of usage is not on the algorithms but on the software engineering prospective. The efficiencies of the approach have been improved by adopting the architecture shown in figure 5. A data crawler periodically extracts raw data for the Web sources used to compute service usage. The crawled raw information is pre-computed and stored in a repository in order to avoid on-the-fly computation of scores each time the recommender is invoked. In addition, a caching mechanism is implemented to provide a more efficient return of scoring results.

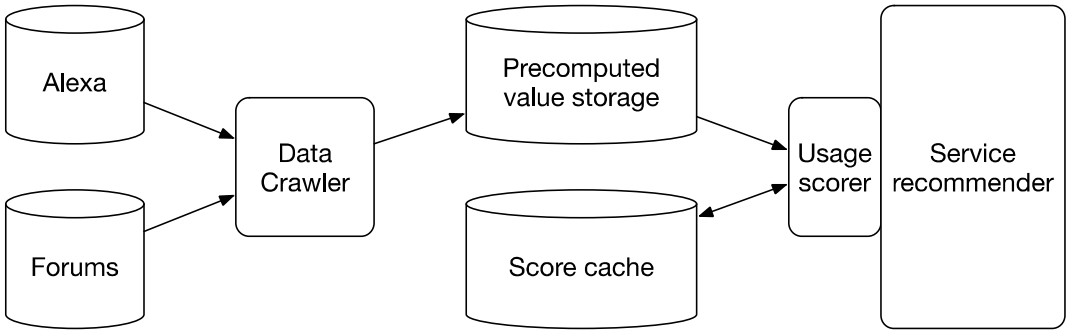

Figure 5: Architecture for efficient usage scoring

3.4 Sensor status evaluation

As described in deliverable D3.1.2.1, a component providing a quantitative evaluation of the sensor quality has been integrated within the system architecture. The evaluation is based on a function measuring quality of the source dataset as a result of the joint computation of the regularity in the frequency of measurements and the basic statistic features of the data time-series.

Extended efforts have been submitted to the task within the scope of the iteration to the final prototype. On the one hand, evaluation against additional volumes of test data has been performed in order to confirm the relevance of the algorithmic solution. Together with that, additional approaches on time-series quality have been analyzed looking for a benchmark on the cost efficiency of the different solutions. Specifically, a quality evaluation approach based on anomaly detection techniques has been favored as the most promising solution in terms of effectiveness. Within the domain of the Temporal Data Mining, existing implementations of statistically-based anomaly detection techniques have been evaluated. The method uses the median from a neighborhood of a data point and a threshold value to compare the difference between the median and the observed data value. Whilst such additional methods have not been finally included in the final prototype implementation in order to avoid potential robustness leaks, both have been tested in exploratory environment and a significantly good performance has been seen.

References

[1] Plebani, P., & Pernici, B. (2009). URBE: Web service retrieval based on similarity evaluation. *Knowledge and Data Engineering, IEEE Transactions on*, *21*(11), 1629-1642.

[2] Commerce, Bled Electronic, Audun Jøsang, and Roslan Ismail. "The beta reputation system." *In Proceedings of the 15th Bled Electronic Commerce Conference*. 2002.

[3] Rowe, M., Alani, H.: What makes communities tick? community health analysis using role compositions. In: proc. of International Conference on Social Computing (SocialCom). pp. 267–276. IEEE (2012)