SEVENTH FRAMEWORK PROGRAMME
Challenge 1
Information and Communication Technologies

Trusted Architecture for Securely Shared Services

Document Type: Software Deliverable

Title: **Behavioural Trust Management Engine**

Work Package: WP5

Deliverable Nr: D5.2

Dissemination: Public

Preparation Date: December 31, 2009

Version: 1.0

The TAS³ Consortium

	Beneficiary Name	Country	Short	Role
1	KU Leuven	BE	KUL	Coordinator
2	Synergetics NV/SA	BE	SYN	Partner
3	University of Kent	UK	KENT	Partner
4	University of Karlsruhe	DE	KARL	Partner
5	Technische Universiteit Eindhoven	NL	TUE	Partner
6	CNR/ISTI	IT	CNR	Partner
7	University of Koblenz-Landau	DE	UNIKOL	Partner
8	Vrije Universiteit Brussel	BE	VUB	Partner
9	University of Zaragoza	ES	UNIZAR	Partner
10	University of Nottingham	UK	NOT	Partner
11	SAP Research	DE	SAP	S&T Coord.
12	EIfEL	FR	EIF	Partner
13	Intalio	UK	INT	Partner
14	Risaris	IR	RIS	Partner
15	Kenteq	NL	KETQ	Partner
16	Oracle	UK	ORACLE	Partner
17	Custodix	BE	CUS	Partner
18	Medisoft	NL	MEDI	Partner
19	Symlabs	PT	SYM	Partner

Contributors

	Name	Organisation
1	Christian Hütter	KARL
2	Klemens Böhm	KARL
3	Jerry den Hartog	TUE
4		
5		

Contents

1 Executive Summary

One approach to build user trust in service providers is behavioural trust management (BTM). Here, users give feedback on the services they have consumed. Based on this feedback, the BTM engine dynamically computes and updates the reputation of service providers. Users can define behaviour-based trust policies which refer to these reputation values in order to identify trustworthy service providers. Besides feedback provided by end-users, the trust feedback service also gathers feedback based on auditing results.

In this deliverable, we define a trust policy language for behavioural trust management and describe the implementation of the BTM engine. Since trust is a very subjective issue, each user has individual policies on how to derive trust from feedback. Our trust policy language is flexible enough to support such subjective trust policies. Rather than using a fixed calculation schema, the BTM engine offers customizable calculation rules to combine feedback into reputation values. The reputation values in turn are used to identify trustworthy service providers. This approach enables users to define trust policies that meet their individual notion of trust.

1.1 Reading Guide

This document is structured as follows. In Section 2 we describe the purpose and scope of Behavioural Trust Management as well as the functionality provided by the BTM engine. Section 3 gives installation and configuration instructions and explains how to run the regression tests. Section 4 describes the role of the BTM engine in the TAS^3 scenario and the usage of the BTM engine by defining its trust policy language. In Section 5 we explain the position of the BTM engine in the trust management architecture as well as its internal architecture. Section 6 provides the public interface (API) and Section 7 the license information of the BTM engine. Finally, Section 8 concludes this deliverable.

This is the first iteration of the BTM engine. A second iteration of this deliverable is scheduled for PM27. In the second iteration, the BTM engine will be optimized to scale well with large volumes of feedback data. In addition, the trust feedback service will interface the authorization services to verify that users are authorized to give feedback.

2 Introduction to the Software

Services in the employability and e-health setting rely heavily on the exchange of personal information. Since people disclose private information to service providers and count on the correctness of information obtained from these providers it is crucial to identify trustworthy service providers in such settings. One approach to build user trust in service providers is behavioural trust management (BTM). Here, users give feedback on the services they have consumed. Based on this feedback, the BTM engine dynamically computes and updates the reputation of service providers. Users can define behaviour-based trust policies which refer to these reputation values in order to identify trustworthy service providers.

2.1 Purpose

The main purpose of the BTM engine is to enable trust decisions based on the behaviour of service providers. The engine evaluates trust policies which are based on behavioural information gathered by the Trust Feedback service. This service provides an interface both for users giving feedback on service providers and for auditing services which provide feedback based on the inspection of logs. All feedback data is stored in a relational database, which we call *Trust Database*.

Since trust is a highly subjective issue, each user has individual policies on how to derive trust from feedback. Let us consider the following example policies:

- *Alice:* “I let a service provider access my personal data if its average feedback is positive.”
- *Bob:* “I want to request service *X* from a provider which has got no negative feedback within the last 24 hours.”
- *Carol:* “I will only interact with service providers if the *k* most reputable users recommend it.”
- *Dave:* “I only request services from providers if their performance regarding *complex tasks* has been satisfactorily.”

These examples show that there are significant differences in the way how the trustworthiness of service providers should be derived. Therefore, a flexible language is required to specify when a particular service provider is trusted. Please note that trust policies may require complex operations as well a complete history of feedback data which is not available to individual users.

Our trust policy language described in section 4.2.2 is flexible enough to support such subjective trust policies. Rather than using a fixed trust calculation schema, the BTM engine offers customizable calculation rules to combine feedback into reputation values. The reputation values in turn are used to identify trustworthy service providers. This approach enables users to define trust policies that meet their individual notion of trust.

2.2 Scope

The scope of this deliverable is to describe the implementation of the BTM engine. The BTM engine is part of the trust tool set [D5.4], which implements the trust management architecture. The trust management architecture [D5.1] is designed to support different sources of trust information, which are provided by dedicated trust services. For communicating with the TAS³ infrastructure the trust tool set implements a standardized XACML interface. A trust policy decision point (PDP) manages the dedicated trust services and facilitates their interaction. One of the core trust services is the RTM service, which is basically an interface to the BTM engine.

2.3 Functionality

In behavioural trust management, the decision whether to trust a service provider depends on the previous behaviour of that provider. When a user interacts with a service provider, he gives feedback on that interaction and makes the feedback publically available. Other users can refer to this feedback in their trust decisions. Since users are not solely dependent on their individual experiences anymore, reputation trust management is especially useful in situations where service providers are not known beforehand.

Centrality measures [WF05] are a well-established approach for analyzing network structures such as feedback graphs. They are used to compute the importance of a vertex based on the graph structure. In the TAS³ setting, users and service providers are represented as vertices and the feedback as edges. Centrality measures can then be used to determine which service provider has a high reputation. The intuition is that providers with high reputation values are considered trustworthy. Users can refer to the reputation values in their trust policies, such as “I trust a service provider if it is among the top five of the most reputable providers“.

However, centrality measures are not directly supported by any existing database management system (DBMS). Computing centrality measures directly inside the database allows for a seamless integration in existing query processing as well as a flexible pre/post-processing of the data. The goal of this deliverable is therefore the implementation of an extensible operator for the computation of centrality measures directly inside the trust database. The BTM engine is based on PostgreSQL [P09], a state-of-the-art database management system which is released under a BSD-style license. We have decided for PostgreSQL because of its strict compliance with the ANSI SQL standards.

2.4 Available Releases and Components

This is the first iteration of the BTM engine. A second iteration of this deliverable is scheduled for PM27. The BTM engine consists of two components which we describe in this document: The Reputation Trust Management Service (T3-TRU-RTM) and the Trust Feedback Service (T3-TRU-FB).

Reputation Trust Management Service (T3-TRU-RTM)

Version	Description	Date
1	BTM engine with customizable trust calculation rules.	PM24
2	Optimize the engine to scale well with large numbers of users and large volumes of feedback data.	PM27

Trust Feedback Service (T3-TRU-FB)

Version	Description	Date
1	Trust information collection point to gather behavioural information.	PM24
2	Interface the authorization services (WP7) to verify that users are authenticated and authorized to give feedback.	PM27

3 Installation Guidelines

In this section we describe the hardware and software prerequisites of the BTM engine, give installation and configuration instructions, and explain how to run the regression tests. Parts of these instructions have been taken from the official PostgreSQL documentation [PD09].

3.1 Hardware and Software Prerequisites

A platform (i.e., the combination of hardware and software) is considered supported by the BTM engine if it has been verified to build and pass its regression tests on that platform. Since the BTM engine is based on PostgreSQL, we expect that most of the platforms supported by PostgreSQL will also be supported by the BTM engine.

PostgreSQL can be expected to work on these CPU architectures: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K, and PA-RISC. Code support exists for M32R, NS32K, and VAX, but these architectures are not known to have been tested recently.

PostgreSQL can be expected to work on these operating systems: Linux (all recent distributions), Windows (Win2000 SP4 and later), FreeBSD, OpenBSD, NetBSD, Mac OS X, AIX, HP/UX, IRIX, Solaris, Tru64 Unix, and UnixWare. Other Unix-like systems may also work but are not currently being tested.

The following software packages are required for building PostgreSQL, which have been released under an open source licence (GPL or BSD):

- GNU make is required; other make programs will not work. GNU make is often installed under the name gmake; this document will always refer to it by that name. (On some systems GNU make is the default tool with the name make.) It is recommended to use version 3.76.1 or later.

- An ISO/ANSI C compiler is required. Recent versions of GCC are recommendable, such as version 2.96.

- tar is required to unpack the source distribution in the first place, in addition to either gzip or bzip2.

- The GNU Readline library (for simple line editing and command history retrieval) is used by default. If you don't want to use it then you must specify the `--without-readline` option for configure.

- The zlib compression library will be used by default. If you don't want to use it then you must specify the `--without-zlib` option for configure. Using this option disables support for compressed archives in `pg_dump` and `pg_restore`.

- OpenSSL if you want to support encryption using SSL.

- GNU Flex 2.5.4 or later and Bison 1.875 or later are required. Other yacc programs can sometimes be used, but doing so requires extra effort and is not recommended. Other lex programs will not work.

Also check that you have sufficient disk space. You will need about 65 MB for the source tree during compilation and about 15 MB for the installation directory. An empty database cluster takes about 25 MB, databases take about five times the amount of space that a flat text file with the same data would take. If you are going to run the regression tests you will temporarily need up to an extra 90 MB.

3.2 Installation and Configuration Instructions

Installing the BTM engine consists of three basic steps which we describe next: (1) Installing PostgreSQL, (2) creating the trust database, and (3) deploying the centrality operator.

3.2.1 Installing PostgreSQL

1. Configuration

The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter:

```
$ ./configure
```

This script will run a number of tests to guess values for various system dependent variables and detect some quirks of your operating system, and finally will create several files in the build tree to record what it found.

The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under `/usr/local/pgsql` by default.

2. Build

To start the build, type

```
$ gmake
```

The build will take a few minutes depending on your hardware. The last line displayed should be

```
All of PostgreSQL is successfully made. Ready to install.
```

3. Installing the Files

To install PostgreSQL enter

```
$ gmake install
```

This will install files into the directories that were specified in step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you could create the target directories in advance and arrange for appropriate permissions to be granted.

3.2.2 Creating the Trust Database

As with any other server daemon that is accessible to the outside world, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data that is managed by the server, and should not be shared with other daemons. In the following, we will work with the user account `postgres`.

Before you can do anything, you must initialize a database storage area on disk which is called a *database cluster*. A database cluster is a collection of databases that is managed by a single instance of a running database server. In file system terms, a database cluster will be a single directory under which all data will be stored. This is called the *data directory*. It is completely up to you where you choose to store your data. In this document, we will use the directory `/usr/local/pgsql/data`.

To initialize a database cluster, use the command `initdb`, which is installed with PostgreSQL. Note that you must execute this command while logged into the `postgres` user account. `initdb` will attempt to create the directory you specify if it does not already exist.

```
$ initdb -D /usr/local/pgsql/data
```

Before anyone can access the database, you must start the database server, again from the database user account. PostgreSQL must know where to find the data it is supposed to use. This is done with the -D option. Use the following command to start the server in the background and put the output into the named log file:

```
$ pg_ctl start -D /usr/local/pgsql/data -l logfile
```

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user. To create a new database, in this example named `trustdb`, you use the following command:

```
$ createdb trustdb
```

Once you have created a database, you can access it by running the PostgreSQL interactive terminal program, called `psql`, which allows you to enter, edit, and execute SQL commands. It can be activated for the `trustdb` database by typing the command:

```
$ psql trustdb
```

In `psql`, you will be greeted with the following message:

```
Welcome to psql 8.3.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
```

```
        \? for help with psql commands
        \g or terminate with semicolon to execute query
        \q to quit

trustdb=>
```

The last line printed out by `psql` is the prompt, and it indicates that `psql` is listening to you and that you can type SQL queries into a work space maintained by `psql`. Try out this command:

```
trustdb=> SELECT version();
                            version
----------------------------------------------------------------
 PostgreSQL 8.3.1 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)
```

3.2.3 Deploying the Centrality Operator

In the examples that follow, we assume that you have created a database named `trustdb` and have been able to start `psql`.

First, we will create the scripts and compile the C files containing the centrality function `centrality_func`. This function is used by the centrality operator to compute the centrality measures. To compile the centrality operator, change to the following directory and run `make`:

```
$ cd contrib/centrality
$ gmake
$ gmake install
```

After you have installed the code you need to register the new objects in the database system by running the SQL commands in the supplied .sql file:

```
psql -d trustdb -f centrality.sql
```

To create the required tables and fill them with some sample data, do the following:

```
$ cd sql
$ psql -f setup.sql trustdb
```

3.3 Running the Tests

The regression tests are a comprehensive set of tests for the BTM engine. They test standard SQL operations as well as the extended capabilities of the centrality operator.

The regression tests can be run against an already installed and running server, or using a temporary installation within the build tree. Furthermore, there is a “parallel” and a “sequential” mode for running the tests. The *sequential* method runs each test script in turn, whereas the *parallel* method starts up multiple server processes to run groups of tests in parallel. Parallel testing gives confidence that inter-process communication and locking are working correctly.

To run the regression tests after building but before installation, type:

```
$ cd src/test/regress
$ gmake check
```

This will first build several auxiliary files, such as some sample user-defined trigger functions, and then run the test driver script. At the end you should see something like

```
======================
 All 100 tests passed.
======================
```

or otherwise a note about which tests failed.

Because this test method runs a temporary server, it will not work when you are the root user (since the server will not start as root). Alternatively, run the tests after installation.

To run the tests after installation, initialize a data area and start the server, as explained in Section 3.2.2, then type

```
gmake installcheck
```

or for a parallel test

```
gmake installcheck-parallel
```

The regression tests for the centrality function can be used only against an already-installed server. To run the tests for the centrality function, type

```
$ cd contrib/centrality
$ gmake installcheck
```

once you have a PostgreSQL server running. Note that `gmake check` is not supported. You must have an operational database to perform these tests and you must have built and installed the centrality function first.

4 How to Use the Software

In this section we first illustrate the role of the BTM engine in the employability integration test scenario described in [D9.1]. We then describe the usage of the BTM engine by defining the representation of behavioural information and our trust policy language.

4.1 Employability Integration Test Scenario

The employability integration test scenario has been designed specifically to show the integration of the TAS[3] technical components. This scenario has been mapped to a business process called “Nottingham process” in [D3.3]. We now describe the relevant steps of this process to illustrate the role of behavioural trust management.

1. **Registration**
   Learner Alice seeking a job placement registers with a placement coordinator which can provide suitable programmes.

2. **Programme selection**
   Alice is presented a list of programmes for which she is eligible and selects the one she prefers.

3. **Data submission**
   Alice provides more specific registration information, e.g. her CV.

4. **Policy selection**
   Alice sets requirements on how her data is used by specifying trust policies. She is presented with a set of default policies from which she can choose. Alice’s trust policies require that service providers have a good reputation amongst students.

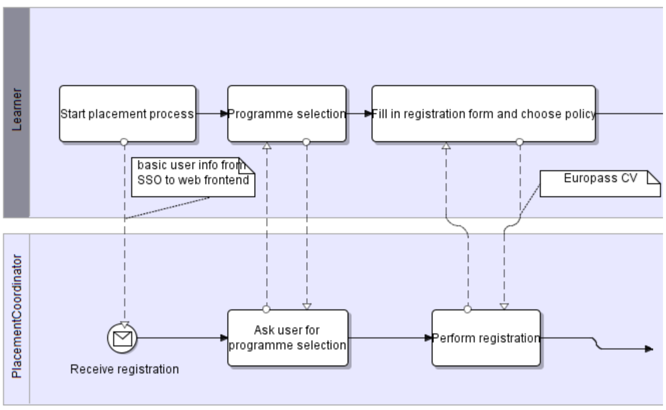


Figure 4-1: Beginning of the process (steps 1 - 4)

5. **Service discovery**
   Suitable service providers are retrieved and ranked according to the trust policies chosen during registration. If the policies depend on behavioural information the trust database is accessed to retrieve user feedback about service providers.

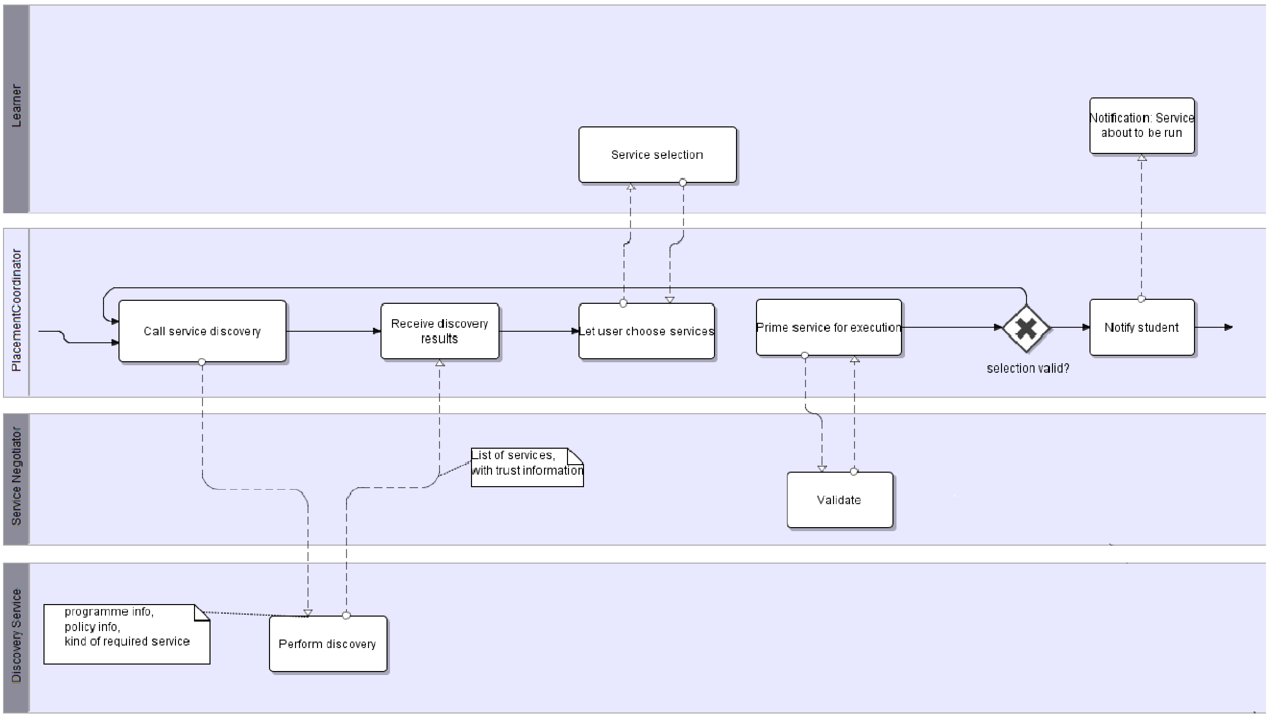


Figure 4-2: Service discovery and selection (steps 5 - 6)

6. **Service selection**
Alice is presented with a list of suitable service providers. She also has the option to renegotiate the level of trust up or down in order to increase or decrease the number of results. Alice selects a service provider and approves that her personal data is released to the provider.

7. **Service execution**
The service is executed and a list of matching vacancies is returned to Alice.

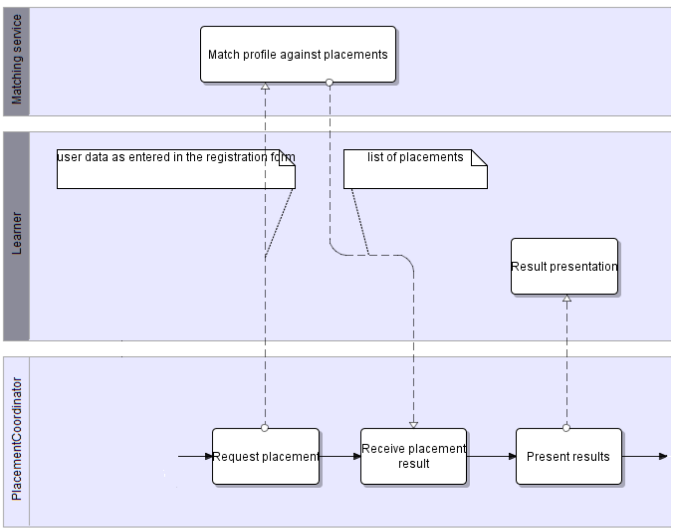


Figure 4-3: Service execution (step 7)

8. **Completion of process**
Alice applies for the placements she is interested in. If she does not receive any matches or rejects the choices offered the process can be repeated with Alice changing her trust policies.

9. **User feedback**
After the process is completed Alice has the opportunity to give feedback on the service. If Alice decides to do so the system shows a page which allows her to rate different aspects of the service such as speediness or quality. The business process execution engine than submits the feedback to the BTM engine using the Trust Feedback service.

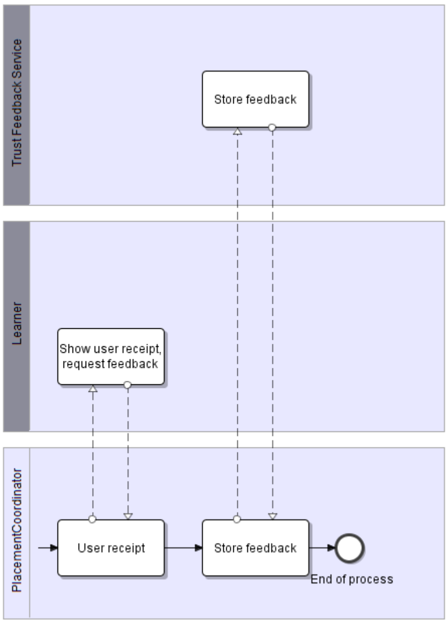


Figure 4-4: End of the process (step 9)

10. **Monitoring and auditing**
The TAS³ monitoring and auditing services are operational throughout the whole process. If the auditing service detects compliance validations by the service provider, it automatically submits negative feedback to the Trust Feedback service though a secure connection.

4.2 Usage of the BTM Engine

This section describes how behavioural information is represented in the trust database and which behaviour-based trust policies the BTM engine provides. In a nutshell we store the feedback graph representing behavioural information in a relational database. Based on this graph, we identify “trustworthy” service providers by calculating centrality measures directly in the database.

4.2.1 Representation of Behavioural Information

A basic concept of behavioural trust management is that users rate each other; they give feedback on their interactions. The reputation of one user is computed from the feedback given by other users. The resulting reputation value can then be used to determine the privileges of that user. In the following, we call instructions on how to derive the trustworthiness of users a *trust policy*.

There are different types of behavioural information such as feedback, recommendation, and reputation. The basic concern of behavioural trust management is how to make trust decisions based on feedback. We gather feedback from various sources such as feedback forms and auditing results. Since trust is a very subjective issue, each user has individual policies to derive trust from feedback. Thus it is important that our trust policy language is flexible enough to support subjective trust metrics.

We represent behavioural information as a weighted, directed graph G(V, E) with a set of vertices V, a set of edges E, and edge weights w(e). The representation of feedback data as graph structure is straight-forward: Users are represented as vertices, feedback as edges between users and feedback values as edge weights.

Feedback data has several characteristics, which we call *aspects*:

- **Feedback Value** $v \in [-1, 1]$. Continuous valuation allows for a finer granularity.
  Alice: "Bob's last service execution has been fairly good (~0.6)."
- **Context** c. Allows to distinguish between different situations where entities can interact.
  Alice: "Bob is good regarding computations of type X, but his performance wrt. services of type Y has been poor."
- **Facet** f_c of context c. Allows to distinguish between different perspectives of a context.
  Alice: "The last service invocation has been very satisfying but also very slow."
- **Timestamp** t. Allows to emphasize the impact of current knowledge. *Alice: "Bob's early service executions were satisfactory but recent ones were poor."*
- **Certainty** • $\in [0, 1]$. Allows to quantify the certainty of an assessment. *Alice: "I am absolutely sure (~1.0) that Bob's last performance was good."*
- **Effort** $e \in [0, 1]$. Allows to quantify the perceived complexity of an interaction.
  Alice: "Bob performed simple (~0.2) computations quite well but complex ones (~0.9) very poorly."

The feedback data is stored in a relational database called *trust database*. The following relations are used to represent the feedback graph:

- `Feedback(rater, ratee, value, context, facet, timestamp, certainty, effort)` contains feedback data.
- `Entity(id)` contains the unique identifiers of the entities.
- `Situation(context, facet)` contains all possible combinations of contexts and facets.

Example 4.1: Relational representation of feedback

Alice: "I am quite sure that the quality of service *S* by *Bob* was good. It was a complex problem."

This informal rating has to be translated into the formal representation described above. The resulting feedback tuple could look as follows:

Rater	Ratee	Value	Context	Facet	Time	Certainty	Effort
Alice	Bob	0.9	S	Quality	12:09:45	0.75	0.8

4.2.2 Trust Policy Language

We propose an algebra-based language for the formulation of behaviour-based trust policies. Such a language has the advantage that it supports the definition of arbitrary user-defined trust policies. The relational representation of feedback data allows for a straightforward implementation based on standard database technology.

We use the *Relational Algebra* (RA) as basis for our trust policy language. It defines a set of operators to be applied on relations. The relations are closed under operators which allows for nesting of operators to complex algebra expressions. A trust policy is then an algebra expression over the relational representation of behavioural information.

The usage of centrality measures has been proposed to determine the reputation of users based on feedback data [KSG03, YAI+04]. Centrality measures are graph algorithms which quantify the importance of vertices according to the graph structure [WF05]. They compute a numerical value for each vertex, the *centrality score*, which allows for a ranking of the vertices. The intuition is that a service provider with a high centrality score is considered as trustworthy.

Our approach is to compute centrality measures directly inside the trust database. This allows for seamless integration in existing query processing as well as flexible pre/post processing of the data. The computation of centrality measures is very time-consuming and resource-intensive. Thus, centrality computation is usually the most costly part of the evaluation of trust policies. Various optimizations have been proposed to improve the computation of centrality measures – see [LM03] for an overview. These optimizations will be addressed in the next iteration of this deliverable.

Until now no database management system (DBMS) directly supports the computation of centrality measures. In order to compute centrality measures the graph data has to be extracted such that the algorithms can be computed outside of the database. However, this approach is both expensive and inflexible. First, data pre-processing such as the selection of certain entities is often desired. Second, the possibility to post-process the results is important. For example, it might be desirable to use the resulting centrality values as input for some other query.

Example 4.2: Pre- and post-processing

Alice: “I trust only service providers which belong to the *k* most reputable entities based on feedback given by students.”

For evaluating this example trust policy, the BTM engine first pre-processes the input data to consider only feedback given by students. Second, it post-processes the results to identify the *k* entities with the highest centrality scores.

We extended PostgreSQL, a state-of-the-art DBMS, to arrive at the expressiveness desired. Our approach is to define a new operator, the *centrality operator* [WB06], which allows for the computation of centrality measures directly inside the database. The centrality operator can than be used to formulate arbitrary behaviour-based trust policies. In the next section, we will define an equivalent SQL statement.

The centrality operator has two design targets:

1. *Support of various centrality measures.* The desired measure is passed as parameter to the centrality operator such that the operator can easily be extended with other measures.

2. *Flexible specification of the graph structure.* Make the representation of the graph explicit by passing a list of attributes which specify the source, destination, and weight of the edges.

Definition of the centrality operator:

```
CENTRALITY[Name, Vertex, Source, Destination, Value, Measure]
(Vertices, Edges)
```

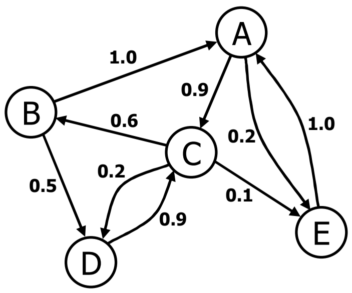
In the following we will explain the input parameters of the centrality operator. The attribute list specifies explicitly the structure of the feedback graph:

- `Name` determines how the result column in the output relation is called
- `Vertex` specifies the column of the relation `Vertices` containing the vertices
- `Source` specifies the column of the relation `Edges` containing the source vertices
- `Destination` specifies the column of the relation `Edges` containing the destination vertices
- `Value` specifies the column of the relation `Edges` containing the edge weights
- `Measure` specifies the centrality measure to be used
- `Vertices` Relation containing the vertices
- `Edges` Relation containing the edges

The output of the centrality operator is a relation consisting of two attributes `Vertex` and `Name`. That is, each tuple of the output relation consists of a vertex identifier (e.g. name of the user) and the score obtained for the vertex after the specified centrality measure has been applied.

Example 4.3: Usage of the centrality operator

The following figure gives an example for the centrality operator. We use the entities stored in table `Users` as vertices and the feedback data stored in table `Feedback` as edges. The column `ID` contains the vertices, while the columns `Rater`, `Ratee` and `Value` contain the source and destination vertices as well as the edge weights. The result column should be called `Score` and we decide for the `PageRank` centrality measure.



Feedback

Rater	Ratee	...	Value
A	C	...	0.9
A	E	...	0.2
B	A	...	1.0
B	D	...	0.5
C	B	...	0.6

Participants

ID
A
B
C
D
E

Nodes / Destination of edges / Centrality measure

CENTRALITY[Score, ID, Rater, Ratee, Value, **PageRank**]
(Participants, Feedback)

Set of nodes / Set of edges / Source of edges / Weight of edges

ID	Score
A	0.23
B	0.21
C	0.31
D	0.15
E	0.1

Figure 4-5: Example usage of the centrality operator

Note that `Vertices` and `Edges` can be arbitrary relations, i.e., they can be either tables or relations resulting from other operations such as select statements. This flexible definition of the graph structure allows for a powerful pre-processing of the input data. Because of the closure property of the relational algebra, the result of the centrality operator is another relation. The resulting relation can be used as input into the next expression, allowing for a post-processing of the output data.

Example 4.4: Behavioural trust policy

Alice: "I trust service providers if their average feedback value from the k most reputable users exceeds a specific threshold t. Use the PageRank centrality measure to rank the entities."

Algebra expression of that policy:

```
SELECT Ratee
FROM Feedback JOIN
      (SELECT Id
       FROM CENTRALITY[Score, Id, Rater, Ratee, Value, PageRank]
             (Users, Feedback)
       ORDER BY Score DESC
       LIMIT k) ON (Rater = Id)
GROUP BY Ratee
HAVING AVG(Value) > t
```

5 Architecture

In this section we define the architecture of the BTM engine. First we describe its position in and interface to the trust management architecture [D5.1]. Next we explain the internal architecture of the BTM engine.

5.1 Position in the Trust Architecture

The trust management architecture described in D5.1 implements a trust policy decision point (PDP) which is called by the master PDP of the TAS[3] architecture. The trusted infrastructure allows users to build trust on different types of information sources. These sources can range from subjective and informal, such as a user's opinion after using a service, to objective formally defined information, such as a credential certifying that an entity is a health-care professional. The trust management architecture consists of so-called *trust services* which provide different sources of trust. The Trust PDP manages the dedicated trust services and facilitates their interaction. The trust services available in the current iteration of D5.1 are illustrated in Figure 5-1.

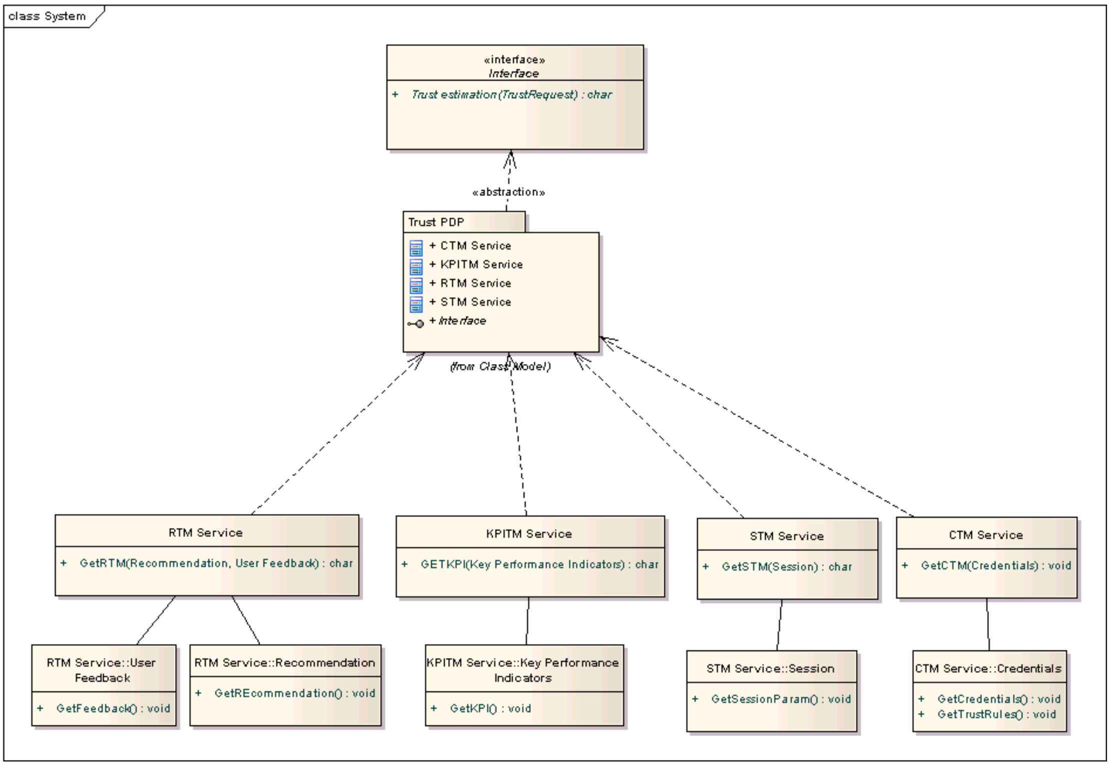


Figure 5-1: Trust management architecture (Source: [D5.1])

One of the core trust services is the RTM Service, which is basically an interface to the BTM engine. It calculates reputation values based on feedback data which is gathered from various sources over time. Rather than using a fixed calculation schema for reputation values, the BTM engine allows users to choose their own method by formulating behaviour-based trust policies. That is, a trust policy describes how feedback is combined to calculate reputation values. Besides feedback provided by end-users, the BTM engine also gathers feedback from auditing services based on the inspection of logs.

Besides the type of trust used, the trust management architecture also distinguishes between internal and external trust services.. While a call to an *internal* trust service is evaluated on the Trust PDP itself, *external* trust services evaluate trust policies outside of the PDP. The reputation in behavioural trust management is a key example of this; to get a reliable reputation score one needs to combine the feedback from many parties. The mechanism would be severely limited if each service or user could only work with the feedback provided to them.

Example 5.1: External trust service

Alice: "I trust only service providers which belong to the *k* most reputable entities. Compute the `PageRank` centrality measure based on feedback provided along with the policy."

```
SELECT Id
FROM (CENTRALITY Score, Id, Rater, Ratee, Value, PageRank
      FROM '(VALUES (1),(2),(3)) AS Entity (Id)',
           '(VALUES (1, 3, 0.5),(3, 2, 0.8),(2, 1, -0.3))
             AS Feedback (Rater, Ratee, Value)') AS c)
ORDER BY Score DESC
LIMIT k
```

An external trust service can be provided by a trusted third party running a feedback and RTM service gathering feedback on a myriad of services. For example, a reputation trust service which gathers feedback and provides reputation values on vacancy providers, employment offices, training programs etc., can provide a job seeker with more meaningful reputation than employment offices separately evaluating vacancy providers. A second advantage of this approach is that behaviour-based trust policies do not have to be evaluated on resource constrained devices, e.g., a job seekers PDA. Evaluation of BTM policies can be computation intensive, but these calculations can be offloaded to the BTM engine.

5.2 Internal Architecture

In this section we describe the implementation of the centrality operator. For a better understanding we first explain the query processing in PostgreSQL. Next we discuss the design decisions we made and define a SQL statement of the centrality operator. Finally we describe how the BTM engine recognizes and executes centrality statements.

5.2.1 Query Processing

Before we describe the implementation of the centrality operator, it is important to understand how a query is processed in a database management system such as PostgreSQL. Each query has to pass four successive stages which we outline in this section:

1. The Parser
2. The Rewriter
3. The Query Optimizer / Planner
4. The Executer

The *parser* verifies whether the query string which arrives as plain ASCII text is syntactically correct. If the syntax is correct an internal representation of the query string, the parse tree, is built up and handed back. Otherwise an error is returned.

The query *rewriter* is a module that exists between the parser stage and the planner/optimizer. PostgreSQL allows the user to define rewrite rules which have to be executed in case of an event, such as a SELECT statement. It takes as input the query tree from the parser and, if a rule applies to the query in question, it rewrites it corresponding to the rule body. The rule system is used in PostgreSQL for the implementation of views. The output of this stage is zero or more rewritten query trees.

The query *optimizer/planner* has the task of creating an optimal execution plan. To do that, it first combines all possible ways of scanning and joining the relations that appear in the query. The optimizer then estimates the execution cost of each created path in the query tree to find the cheapest one. In case of a single-relation query, the optimal path corresponds to the cheapest access path of the relation.

The *executer* takes as input the plan created by the planner optimiser. This plan tells the executer by which method and in which order the base relations have to be accessed and joined. The executer always starts processing at the top node, the root of the plan tree. It first passes through the tree in order to make various initialisations such as initialisation of internal execution states and allocation of memory for tuple storing. Each parent node recursively triggers the initialisation of its child nodes. Then a second pass is done in which the actual execution of the plan is done. A final pass through the plan is done in order to do a general clean-up such as deallocation of memory.

5.2.2 Centrality SQL Statement

PostgreSQL is pipeline oriented. This means that a node in the operator tree pulls a tuple from his children only to immediately process it. However, centrality measures need all the tuples at the same time in order to compute a result. Another issue is that the centrality operator needs to be easily extendable by other centrality measures.

We took these issues into consideration in the implementation of the centrality operator. More specifically, we defined a new operator called CENTRALITY inside PostgreSQL. The SQL statement of the centrality operator is as follows:

```
CENTRALITY  Name,  Vertex,  Source,  Destination,  Value,  Measure
FROM Vertices, Edges
```

The attributes `Name`, `Vertex`, `Source`, `Destination`, `Measure` are of type `varchar`. The attribute `Value` is of type `real`. `Vertices` and `Edges` are either base relations or `SELECT` statements.

The functionality of the operator, i.e. the routines that have to be executed for centrality computation, was implemented as a user-defined function in the contribution folder of the PostgreSQL distribution. That is, the execution of the operator is actually a function call. After the parser has checked the number, types, and order of the input parameters, the parameter list is passed to the centrality function `centrality_func`. This function calls the dynamically linked library which contains the centrality algorithms. The library first gets all required tuples, pre-processed according to the desired centrality algorithm, from the base relations via the Server Programming Interface and stores them into a `tuplestore`. It then processes the tuples according to the centrality algorithm and stores the result into a vector which is passed back to the database.

5.2.3 Definition of New Structures

In PostgreSQL each SQL statement is represented internally as a tree with a vertex for each structure (clause). The first implementation step is to create a new vertex for the CENTRALITY statement. This is done in `parsenodes.h`:

```
typedef struct CentralityStmt
{
     NodeTag type;
     List *targetList; /* the target list (of ResTarget) */
     List *fromClause; /* the FROM clause*/
} CentralityStmt;
```

Each node has a `NodeTag` that specifies its type. This allows for type casting. The CENTRALITY statement consists of two lists: the `targetList` which contains all the parameters between the keywords CENTRALITY and FROM, and the `fromClause` which contains the tables Feedback and Entity.

Because CENTRALITY defines a new command type we need to add it to the enumeration of command types in `nodes.h`. Command types are the operation types represented by a Query or Planner statement.

Now we need to implement the input and output functions for the CENTRALITY node. An output function reads the internal representation of a node and writes it to a serialized string.

```
static void _outCentralityStmt(StringInfo str, CentralityStmt
*node)
{
     WRITE_NODE_TYPE("CENTRALITY");
     WRITE_NODE_FIELD(targetList);
     WRITE_NODE_FIELD(fromClause);
}
```

The read functions for all statements are represented by the `_readQuery(void)` function, so no changes are needed here. We also defined the functions `copy` and `equality`. They are not mandatory, but they offer the functionality needed to copy a CENTRALITY node and two compare two CENTRALITY nodes for equality.

```
static CentralityStmt *
_copyCentralityStmt(CentralityStmt *from)
{
	CentralityStmt *newnode = makeNode(CentralityStmt);
	COPY_NODE_FIELD(targetList);
	COPY_NODE_FIELD(fromClause);
	return newnode;
}

static bool
_equalCentralityStmt(CentralityStmt *a, CentralityStmt *b)
{
	COMPARE_NODE_FIELD(targetList);
	COMPARE_NODE_FIELD(fromClause);
	return true;
}
```

5.2.4 Parsing

After defining the new CENTRALITY structure the grammar file `gram.y` has to be adapted. Two new nodes are defined, one for the centrality statement and one for the centrality clause:

```
%type <node> CentralityStmt
%type <node> centrality_clause
```

This distinction allows a flexible redefinition of the structure of the statement, making it possible to add alternative structures. For example the centrality statement can be surrounded by any number of brackets „( )“.

Next, CENTRALITY has to be added in the key words list, which is sorted in alphabetical order in order to allow quick finding of keywords. The Centrality statement is also added as an alternative for a possible statement:

```
stmt:
...
	| CentralityStmt
...
```

Then the centrality clause and the actions to be taken when a centrality statement is identified are defined. In the following it will only be described what happens when a centrality statement is identified, since the code is too large to be displayed here.

Whenever a CENTRALITY statement

```
CENTRALITY Name, Vertex, Source, Destination, Value, Measure
FROM 'Entity', 'Feedback'
```

is identified, it will be transformed in the equivalent statement:

```
SELECT *
FROM centrality_func(Name, Vertex, Source, Destination, Value,
Measure, Entity, Feedback)
```

That is, the equivalent SELECT statement can be used instead of the CENTRALITY statement. However the CENTRALITY statement is not only semantic sugar: It makes sure that the parameters of the operator are valid before the actual execution is started. Note that the Entity and Feedback relations from the CENTRALITY statement have to be passed as strings, hence the quotes. This is needed to make sure that the parser does not automatically process the SELECT statements that could replace the table names in the statement.

Last but not least CENTRALITY is added to the definition of the reserved keywords. Doing so makes sure that the word centrality cannot be used for variable, type or function names. For consistency reasons this has to be done in the file `keywords.c` as well.

The file `analyze.c` is responsible for the transformation of the parse tree into the query tree. The following modifications have to be done. A new function `transformCentralityStmt` had to be defined. It takes as input parameters the parse state and the original CENTRALITY statement and returns a query tree.

```
static Query *transformCentralityStmt(ParseState *pstate,
CentralityStmt *stmt);
```

The function is called by the `transformStmt` function when the respective statement has the CENTRALITY tag attached.

```
case T_CentralityStmt:
     result = transformCentStmt(pstate,(CentralityStmt *) stmt);
     break;
```

It resolves the FROM clause by calling `transformFromClause`. This function processes the FROM clause and adds items to the query's range table, join list and namespace. It checks whether the relation names in the FROM clause are known to the system and creates for every relation present in the system catalogues an RTE (Range Table Entry) node containing the relation name, the alias name (if given) and the relation id, which will be used to refer to the relation. Then the `transformTargetList` function is called: It checks that the attribute names used in the statement are contained in the relations given in the query. For each attribute a TLE (Target List Entry) node containing the attribute information is created.

In case the query string is not passed to the server through the `psql` interface, but by another application, it has to be pre-processed by a different grammar. This grammar is found in `preproc.y`. The changes we had to make here are similar to those in the grammar from `gram.y`. CENTRALITY has to be added to the keywords token list, `CentralityStmt` and `centrality_clause` must be declared of type string and `CentralityStmt` added as a possible statement. The grammar can then transform the centrality statement passed as a query into a string to be returned.

```
CentralityStmt: centrality_clause {$$=$1;};
centrality_clause:
     CENTRALITY ColLabel ',' target_el ',' target_el ','
     target_el ',' Sconst ',' Sconst
     FROM Sconst ',' Sconst
{$$ = cat_str(16, make_str("centrality"), $2, make_str(","), $4,
make_str(","), $6, make_str(","), $8, make_str(","), $10,
make_str(","), $12, make_str("from"), $14, make_str(","), $16);
};
```

Here, CENTRALITY has to be added to the list of reserved key words, as well in the `keywords.c` file from the same folder.

Before we can move on to the execution code of the centrality statement, one more issue has to be addressed. Since the transformation of the CENTRALITY statement into an equivalent SELECT statement is done in the parsing stage, an attempt to rewrite the centrality query tree would raise an error. To ensure this does not happen we have to tell the traffic cop not to attempt to rewrite a CENTRALITY statement:

```
if (query->commandType == CMD_UTILITY ||
    query->commandType == CMD_CENTRALITY)
{
     /* don't rewrite utilities or CENTRALITY statements*/
     querytree_list = list_make1(query);
}
```

The `tcop` (traffic cop) is the module that dispatches requests to the proper module. It contains the PostgreSQL backend main handler, as well as the code that makes calls to the parser, optimizer, executor and commands functions.

5.2.5 Execution

The Executer has to do exactly the same for the CENTRALITY statement as it does for the SELECT statement, namely it returns a tuple unchanged to the caller:

```
static  void  ExecCentrality(TupleTableSlot  *slot,  DestReceiver
*dest, EState *estate)
{
     /* do same thing like select, i.e. give tuple back */
     (*dest->receiveSlot) (slot, dest);
     IncrRetrieved();
     (estate->es_processed)++;
}
```

This is due to the fact that the processing is done further down the tree in a function call node. This node executes the code of the user-defined function `centrality_func`, which is defined in file `centrality.c`.

The centrality function is the main handler for centrality requests. It is implemented as a self-materializing function, i.e., the result of the function is written to a table. It takes as input all the parameters from the centrality statement (the result column name, the column name of the vertices, the names of the source, destination, and weight, columns as well as the name of the algorithm to be used) and two table names. The function returns a `tuplestore` representing the output table of the centrality measure, i.e. a ranking table.

The centrality function first performs some basic checks if the caller supports a `tuplestore` as returning value and if the `tuplestore` to be returned has the right form. It then calls the `get_tuplestore` function which uses the SPI (Server Programming Interface) to collect the tuples of the base tables from the database. The tuples are retrieved from the database in form of an adjacency list. In order to compute the centrality algorithm, the name of centrality measure is first read and then an according SQL statement is sent to the server for execution. A structure check of the returned list then takes place before it is processed according to the centrality algorithm specified by the query. If the algorithm is unknown (i.e., not implemented) an error is returned. Currently four algorithms are supported: PageRank [BP98], Hubs and Authorities [K99], and Proximity [L76].

In order to compute the centrality measures, the adjacency list is first transformed into an adjacency matrix. This transformation is the most cost expensive one since the matrix can be very large. Then the matrix along with other parameters needed is passed to the function that actually implements the centrality measure. There are also help functions implemented, like function for power iteration, vector normalization and output functions (printing of vectors and matrices).

Adding new centrality measures is quite easy. All that has to be done is the implementation of a new function which computes the new measure. This function has to be added to the `centrality.c` file.

6 API and Library Information

The BTM engine has two interfaces: one to the Trust PDP, in form of the RTM Service, and one to the business process execution engine, in form of a Trust Feedback service. A service requestor can connect to the RTM Service though the Master PDP, which chains the service request to the Trust PDP that finally invokes the RTM Service. The business process execution engine calls the Trust Feedback service if the business process provides an opportunity to give feedback on the service.

A service request to the BTM engine must be formulated in the trust policy language defined in Section 4.2.2. When evaluating a behaviour-based trust policy (i.e., a SQL query), the BTM service computes its decision either based on the feedback data provided by the feedback service or based on additional data provided by external trust services. Thus, not all users have to rely on or agree upon the data in the feedback repository, and can provide their own feedback data.

The Trust Feedback service gathers behavioural information and makes it available to the BTM engine. It provides an interface both for users giving feedback on service providers and for auditing services based on the inspection of logs. All feedback data is stored in the relational database management system PostgreSQL.

When receiving feedback from a user, the feedback service will invoke the Authentication Service to verify the link between this user and an actual interaction thus authorizing the user to give feedback on the interaction. Feedback based on auditing results will be received though a secure connection from the auditing service and is considered trustworthy. Thus, all information stored in the trust database can be assumed to be authentic. Please note that we cannot make statements about the validity of feedback, i.e. users can still lie about someone else's performance.

6.1 RTM Service

The RTM service has a rather simplistic interface which is required by the Trust PDP. This interface is implemented by a JDBC client which connects to the BTM engine, executes the trust metric, and returns the reputation value of the requester.

```
/**
 * Computes a trust metric for the given requester.
 *
 * @param metric
 * @param requester
 * @return Reputation value of the given requester
 */
public float evaluate(String metric, String requester);
```

6.2 Trust Feedback Service

The Trust Feedback service also has a straightforward interface which provides methods for giving and retrieving feedback about a service. This interface is implemented by an Axis web service. The method `giveFeedback` is called by the business process execution engine when a user gives feedback about a service. The method `getFeedback` may be called during service discovery to determine a ranking of service providers based on user feedback.

```
/**
 * Give feedback about a service.
 *
 * @param userID User giving feedback.
 * @param serviceEPR Endpoint of the service for which feedback
 *        is given.
 * @param rating Feedback value (in the range [-1,+1]).
 * @param feedbackFacet Facet (like speediness, correctness,
 *        quality, ...) which the feedback refers to.
 * @param serviceType Type of the service (semantically specifying
 *        the function of the service, like "matching", "training"
 *        or "diagnosis").
 * @throws Exception If something goes wrong.
 */
public void giveFeedback(String userID, String serviceEPR,
            String serviceType, String feedbackFacet,
            double rating) throws Exception;

/**
 * Get average feedback about a service.
 *
 * @param serviceEPR Endpoint of the service to get feedback for.
 * @param serviceType Type of the service (semantically specifying
 *        the function of the service, like "matching", "training"
 *        or "diagnosis").
 * @param feedbackFacet Facet (like speediness, correctness,
 *        quality, ...) which the feedback refers to.
 * @return Average feedback value (in the range [-1,+1]).
 * @throws Exception If something goes wrong.
 */
public double getFeedback(String serviceEPR, String serviceType,
              String feedbackFacet) throws Exception;
```

7 License Information

The Software is provided under a BSD style license given in Section 7.1. This code uses PostgreSQL code to which the license given in Section 7.2 applies.

7.1 Behavioural Trust Management Engine

7.2 PostgreSQL Data Base Management System

8 Roadmap and Conclusions

In this deliverable, we have defined a trust policy language for Behavioural Trust Management (BTM) and described the implementation of the BTM engine. Since trust is a very subjective issue, each user has individual policies on how to derive trust from feedback. Our trust policy language is flexible enough to support such subjective trust policies. Rather than using a fixed calculation schema, the BTM engine offers various centrality measures to combine feedback into reputation values. The reputation values in turn are used to identify trustworthy service providers.

The BTM engine is based on PostgreSQL, a state-of-the-art database management system which is released under a BSD-style license. We have extended PostgreSQL with a centrality operator that allows calculating centrality measures directly in the database. A corresponding SQL statement for the centrality operator has also been defined. This allows users to formulate behaviour-based trust policies simply as SQL queries. We have also shown how to extend the operator by additional centrality measures.

However, there are some shortcomings in the current version of the BTM engine. Currently, no automatic query optimization is done by the DBMS. There is also no caching of the centrality measures computed. That is, the computation of the adjacency matrix, which is the most expensive part of the entire centrality computation, is done for each call of the centrality function. Further iterations of this deliverable will address these shortcomings; especially query optimization and caching.

In the next iteration, the trust feedback service will interface the authorization services [D7.1] to verify that users are authorized to give feedback on a particular interaction. Besides feedback provided by end-users, the BTM engine will also gather feedback based on auditing results from the inspection of logs. This feedback will be received though a secure connection from the auditing service and has already been verified. Auditing and logging guidelines needed for this are part of the work planned in Task 5.3, which is scheduled together with the next iteration of this deliverable for PM27.

References

TAS³ Deliverables

[D3.3] Deliverable 3.3: Integration with TAS³ trust applications of employment and eHealth processes.

[D5.1] Deliverable 5.1: Trust Management Architecture Design.

[D5.4] Deliverable 5.4: Trust Tool Set.

[D7.1] Deliverable 7.1: Design of Identity management, Authentication and Authorization Infrastructure.

[D9.1] Deliverable 9.1: Pilot Specifications and Use Case Scenarios.

Literature

[BP98] S. Brin and L. Page, ”The anatomy of a large-scale hypertextual Web search engine“, Computer Networks and ISDN Systems, pages 1–7, 1998.

[K99] J. M. Kleinberg, ”Authoritative sources in a hyperlinked environment“, Journal of the Association for Computing Machinery, pages 1–7, 1999.

[KSG03] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina, ”The Eigentrust Algorithm for Reputation Management in P2P Networks“, In Proc. 12th International Conference on World Wide Web, pages 640–651, ACM Press, 2003.

[L76] N. Lin, ”Foundations of Social Research“, McGraw-Hill, 1976.

[LM03] Amy N. Langville and Carl D. Meyer, ”Deeper Inside PageRank“, Internet Mathematics Vol. 1, No. 3: 335-380.

[P09] The Official PostgreSQL Web Page, http://www.postgresql.org (01.12.2009).

[PD09] The PostgreSQL Global Development Group, “PostgreSQL 8.3 Documentation”, 2009.

[WF05] Stanley Wasserman and Katherine Faust, ”Social network analysis: methods and applications“, Cambridge Univ. Press, 2005.

[WB06] Christian von der Weth and Klemens Böhm, ”A Unifying Framework for Behavior-based Trust Models“, Proceedings of Cooperative Information Systems (CoopIS), 2006.

[YAI+04] A. Yamamoto, D. Asahara, T. Itao, S. Tanaka, and T. Suda, “Distributed pagerank: A distributed reputation model for open peer-to-peer networks“, In SAINT-W ’04 (SAINT ’04 Workshops), Washington, DC, USA, 2004.

Amendment History

Ver	Date	Author	Description/Comments
0.1	2009-12-01	CH, KB	First complete version.
0.2	2009-12-10	CH, KB	Implemented review comments by Slim Trabelsi (SAP).
0.3	2009-12-15	CH, KB	Proofreading by Klemens Böhm (KARL).
0.4	2009-12-22	CH, KB	Updated TAS^3 scenario (Section 4.1)
1.0	2009-12-31	CH	Final version released to public.