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: PU

Preparation Date: June 30, 2010

Version: 2.0

The TAS³ Consortium

	Beneficiary Name	Country	Short	Role
1	KU Leuven	BE	KUL	Proj. Coord.
2	Synergetics NV/SA	BE	SYN	Partner
3	University of Kent	UK	KENT	Partner
4	Karlsruhe Institute of Technology	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.

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, its internal architecture, and how users are authorized to give feedback. Section 6 describes the optimization techniques implemented. Section 7 provides the public interface (API) and Section 8 the license information of the BTM engine. Finally, Section 9 concludes this deliverable.

This is the second iteration of the BTM engine. A first iteration of this deliverable was provided in PM24. In this iteration, the BTM engine was optimized to scale well with large volumes of feedback data. In addition, the trust feedback service interfaces 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. People disclose private information to service providers and count on the correctness of information obtained from these providers. Thus, it is crucial in such settings to identify trustworthy service providers. 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 for users giving feedback on service providers. 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 *some complex task* 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 subjective 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.

Since centrality measures are typically based on complex algorithms, their computation can take a long time, depending on the size of the feedback graph. Thus, it is not feasible to recalculate the centrality values with each policy evaluation. Several approximation algorithms have been proposed to accelerate the computation of centrality algorithms. But even an improvement of the complexity by one order of magnitude (e.g., from days to hours) might not be enough to evaluate trust policies immediately.

To solve this problem, we propose the combination of approximation algorithms with a caching system. The BTM engine accesses cached centrality values for the evaluation of trust policies to provide immediate trust decisions. The cached centrality values are recalculated periodically, e.g., at night or once a week. To guarantee the timeliness of the centrality values between the periodic recalculations, the BTM engine continually updates the cached values using

approximation algorithms. While the approximation algorithms might not compute exact centrality values, they still provide a correct ranking of the entities. We consider this as acceptable because behavioural trust policies typically use relative instead of absolute reputation values.

2.4 Available Releases and Components

This is the second iteration of the BTM engine. A first iteration of this document was delivered in PM24.

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.	PM30

Trust Feedback Service (T3-TRU-FB)

Version	Description	Date
1	Trust information collection point to gather behavioural information.	PM24
2	Use the authorization infrastructure to verify that users are authenticated and authorized to give feedback.	PM30

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 passes 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 -f centrality.sql trustdb
```

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

```
$ cd contrib/centrality/sql
$ psql –f setup.sql trustdb
```

Finally, we will enable the optimizations of the centrality computation. Run the following SQL script to create the required tables and functions:

```
$ cd contrib/centrality/optimization/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 TAS³ technical components. This scenario has been mapped to a business 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.

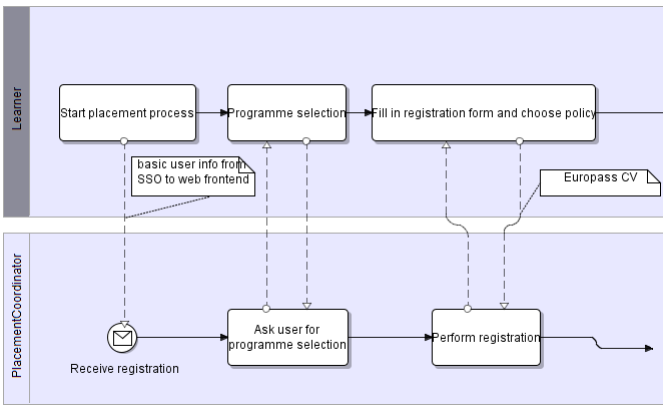


Figure 4-1: Beginning of the process (steps 1 - 4)

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.

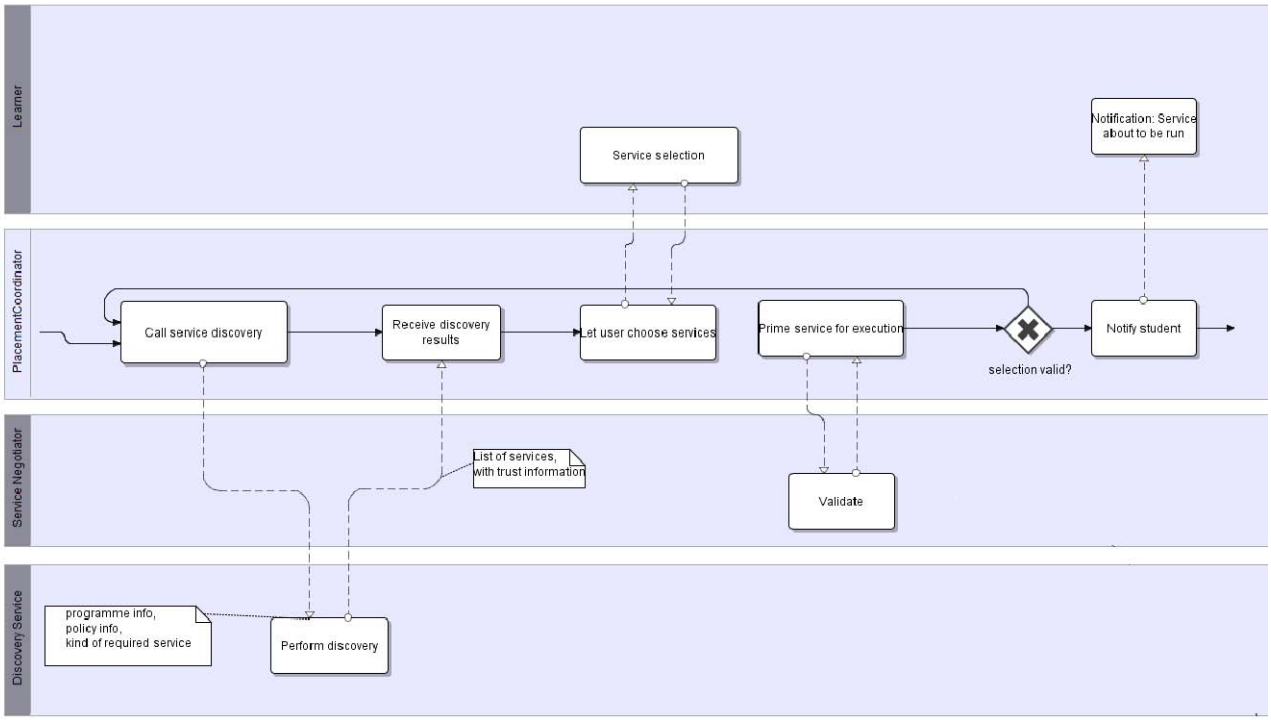


Figure 4-2: Service discovery and selection (steps 5 - 6)

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.

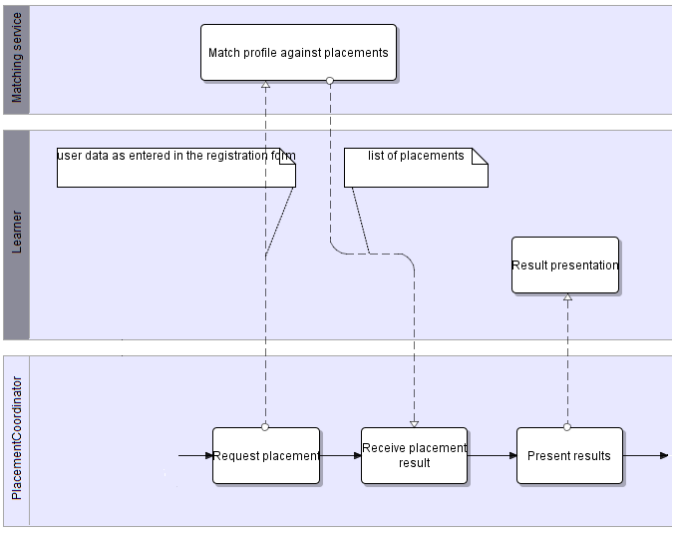


Figure 4-3: Service execution (step 7)

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 then submits the feedback to the BTM engine using the Trust Feedback service. This service verifies that Alice is authorized to give feedback.

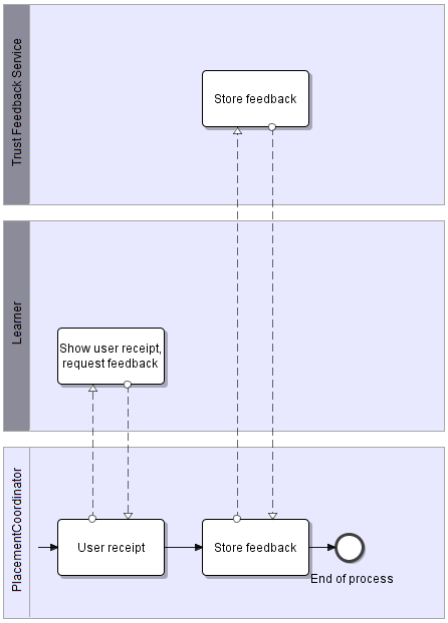


Figure 4-4: End of the process (step 9)

10. **Monitoring and auditing**
    The TAS³ monitoring and auditing services are operational throughout the whole process. If the online compliance testing (OCT) engine detects compliance validations by the service provider, it automatically reduces its trust rating.

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 gathered from the users. 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 distinguishing 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 distinguishing between different perspectives of a context.
  Alice: "The last service invocation has been very satisfying but also very slow."
- **Timestamp** t. Allows emphasizing the impact of current knowledge.
  Alice: "Bob's early service executions were satisfactory but recent ones were poor."
- **Certainty** $\sigma \in [0, 1]$. Allows quantifying 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 quantifying 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 Centrality Measures

In the literature, the usage of centrality measures has been proposed to determine the reputation of entities 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, the *centrality score*, for each entity which allows for a ranking of the entities. The intuition is that a service provider with a high centrality score is considered as trustworthy.

Centrality measures can be categorized into the following categories according to their algorithmic complexity:

- **Local measures** take into account only direct neighbours of a node. Examples are *Indegree* (the sum of the weights of incoming edges) and *Outdegree* (the sum of the weights of outgoing edges). Local measures have linear computational complexity.

- **Eigenvector-based measures** not only use the incoming edges, but also the centrality values of the neighbours. Intuitively, nodes may 'pass' their score to their neighbours. Examples are *PageRank* [BP98] and *HITS* (Hubs and Authorities) [K99]. Eigenvector-based measures have quadratic computational complexity.

- **Distance-based measures** compute the shortest paths between all nodes. Examples are *Closeness* [S66] and *Proximity* [L76]. Distance-based measures have cubic computational complexity.

4.2.3 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 *Structured Query Language* (SQL) as basis for our trust policy language, which in turn is based upon the Relational Algebra (RA). The RA 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.

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. These optimizations will be addressed in Section 6.

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 c*entrality operator* [WB06], which allows for the computation of centrality measures directly inside the database. The centrality operator can then 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. Thus, the operator can easily be extended with other measures.

2. *Flexible specification of the graph structure.* The structure of a graph is defined only implicitly in a relational database. We 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.

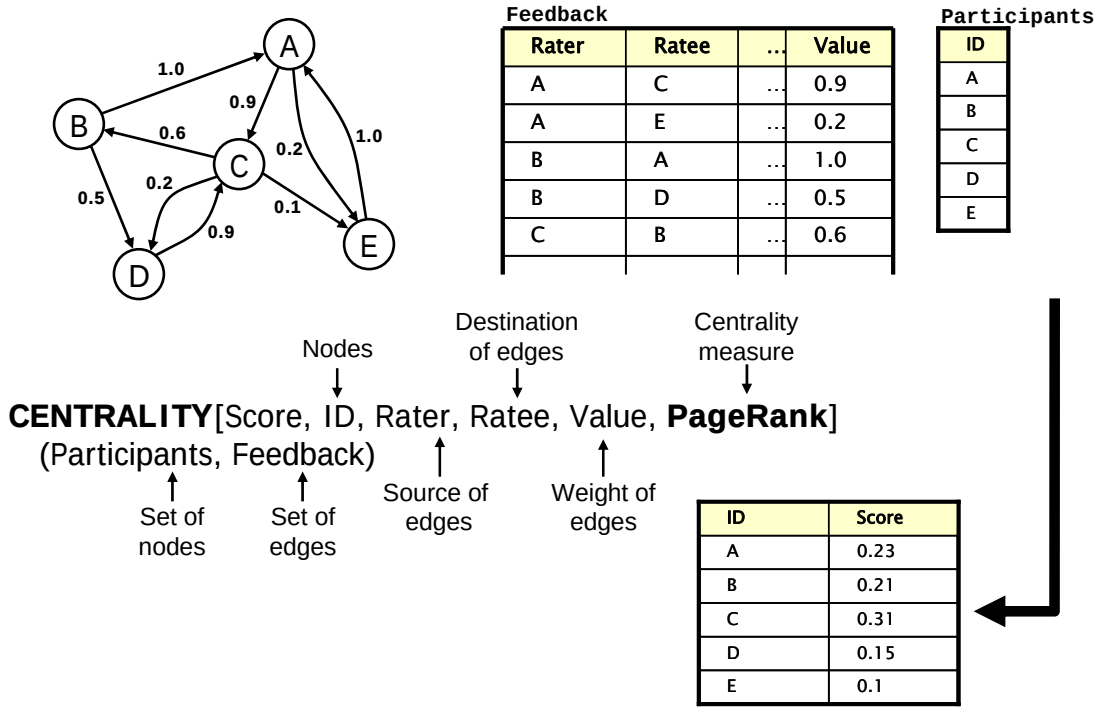


Figure 4-5: Example usage of the centrality operator

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 TAS³ 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.

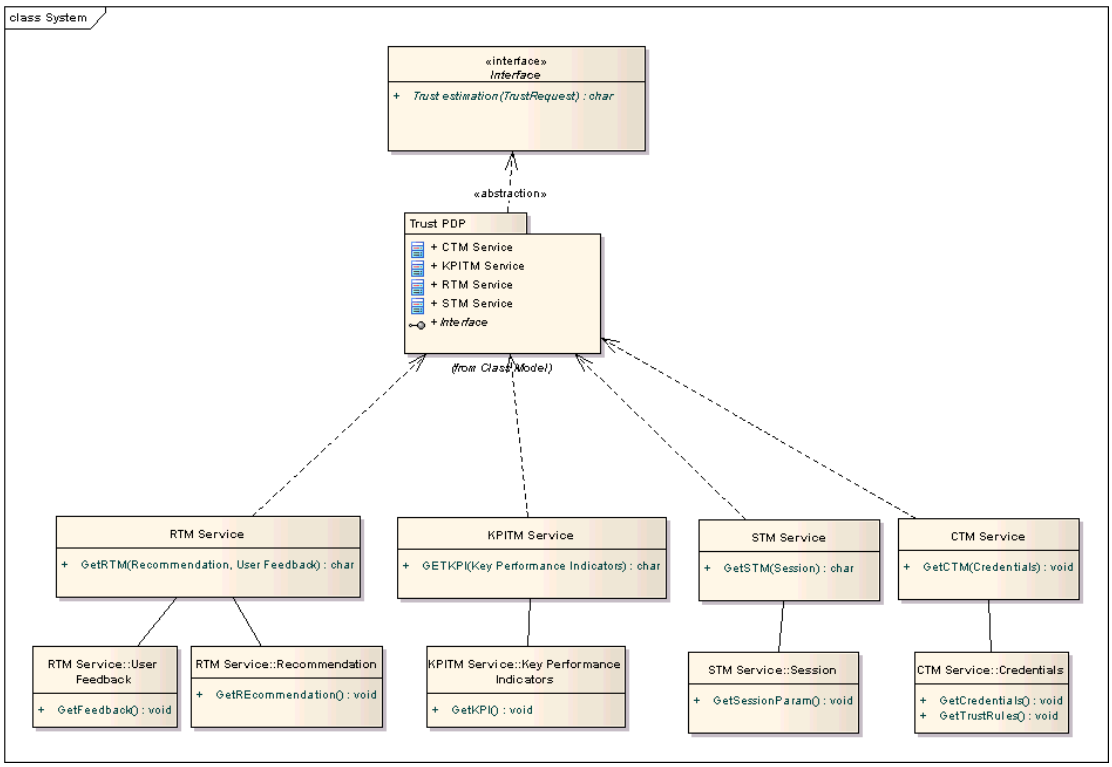


Figure 5-1: Trust management architecture (Source: [D5.1])

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 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., 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.

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.

5.3 Authorization of the Feedback

As described in Section 4.1, the student placement process implements the Employability Integration Test Scenario. At the end of the process, the user has the opportunity to give feedback on the degree of satisfaction with the execution of the service. The business process engine sends the feedback to the BTM engine using the Trust Feedback service. Since users must be authenticated to execute the process and only trusted components may call the feedback service, only authorized users can give feedback.

5.3.1 Single Sign-On

TAS^3 uses a common single sign-on (SSO) framework based on SAML 2.0. Whenever a user logs in to a service provider, an identity provider (IdP) is involved to assert the user's identity. All components which interact with the user have to support SSO.

In the student placement process, the business process manager (T3-BP-MGR) serves as user interface. Before the user can execute the business process, he has to log in to the business process manager which provides a module for authorization and authentication.

The SSO capabilities are provided by ZXID (T3-ZXID-LINUX-X86), which handles the entire protocol exchange. The servlet allows the user to choose an IdP for logging in. Once the user is authenticated, ZXID stores the authentication information in the session of the servlet container. This information can be accessed by other components such as the business process which run in the same servlet container.

5.3.2 Operational Sequence

During the registration for the process, the user has to define a policy for his trust requirements. The trust policy is used to discover appropriate services which fulfill the user's trust requirements. This functionality is provided by the

T3-PEP-RQ component, which uses the ZXID function `zxid_get_epr()` to discover all available services of a given service type. The Trust PDP then checks these services to determine whether they fulfill the user's trust policy. If so, the value of the service's trust ranking is returned.

At the end of the process, the user has the possibility to give feedback on the service (see Figure 5-2) using a web form. The Business Process Execution Engine (T3-BP-ENGINE-ODE) submits the feedback to the Trust Feedback Service (T3-TRU-FB) via a web service call. It uses the ZXID function `zxid_call()` which encrypts and signs the SOAP message before performing the call.

The feedback service validates the SOAP request by using the ZXID function `zxid_wsp_validate()`. If the validation succeeds, ZXID returns the target name identity of the request. Since the user has been authenticated by the business process and only trusted components may execute ZXID calls, a successful validation indicates that the user is authorized to give feedback.

The TAS^3 architecture provides a federated identity management (FIDM) which lets users use different, non-linkable identities at different services. It also provides a linking service with which users can combine attributes provided by different identity providers. The BTM engine uses identity mapping to maintain persistent identities. Identity mapping converts an identity from one domain to an identity valid in another domain. The conversion is made by an IdP that trusts the starting domain and is trusted by the ending domain.

When a user submits feedback for an interaction, ZXID maps the provided (transient) identity of the user to the persistent identity. The BTM engine only knows that this user has had an interaction with the given service provider, but does not get to know the other identities of the user. Vice versa, the service provider with which the user has interacted will not get to know the persistent identity used by the BTM engine. The feedback service uses the persistent identity of the user as source of the feedback (*rater*) and the endpoint reference (EPR) of the service as destination (*ratee*).

The Audit Bus serves as common logging facility for all TAS^3 components. A so-called Audit Trail Viewer presents the relevant logging information on the portal. The Feedback Service logs all feedback events to the Audit Bus. If a trusted component allows unauthorized users to give feedback, this incident can be discovered by inspecting the audit trail. As a consequence, the component might be excluded from the circle of trust.

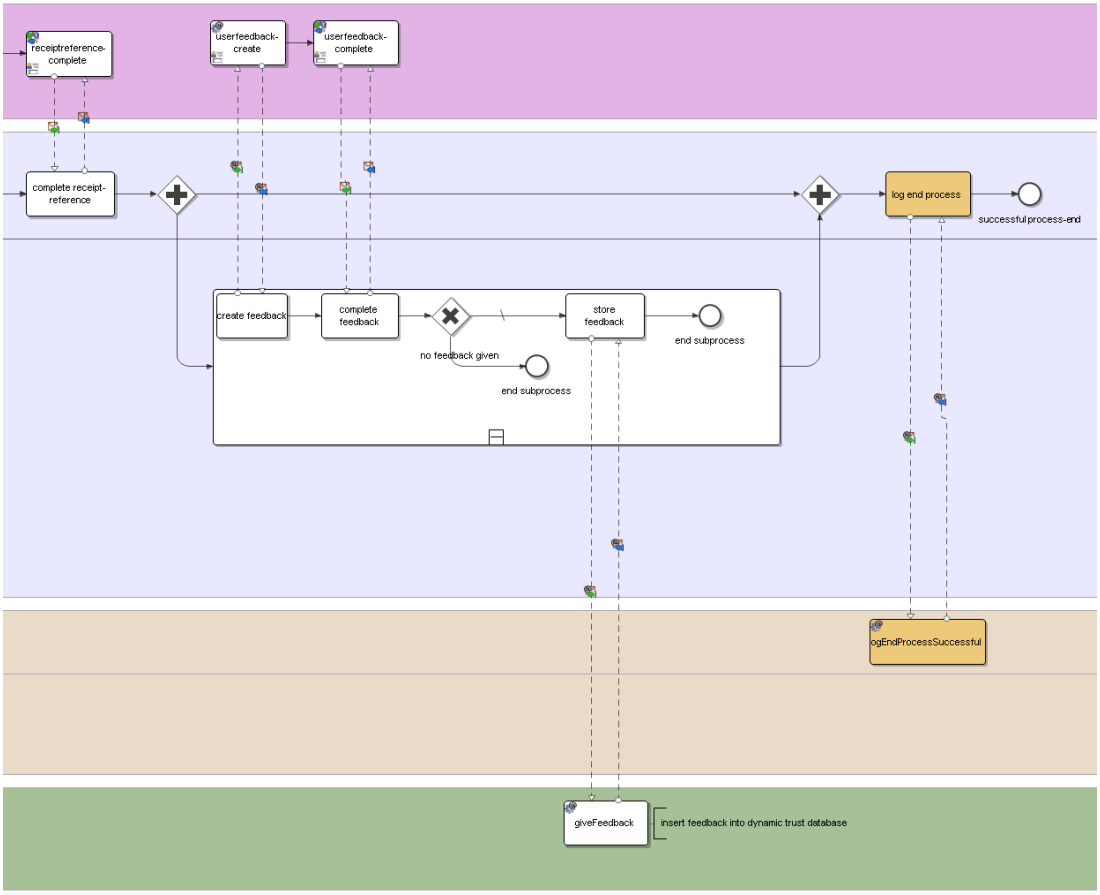


Figure 5-2: Feedback phase of the process [D3.3]

6 Optimizations

The computation of centrality measures may take a long time (e.g., days or weeks for large graphs) because of their algorithmic complexity. In this section, we describe various optimizations to improve the computation of centrality measures. The BTM engine currently supports four centrality measures: PageRank [BP98], Hubs and Authorities [K99], and Proximity [L76]. The optimizations presented in this section will focus on these measures.

6.1 Optimization Algorithms

Research has proposed various optimization algorithms which accelerate the computation of centrality algorithms. These algorithms can be categorized as follows:

- **Exact algorithms** enhance existing algorithms by reducing their run times.
- **Approximation algorithms** reduce the algorithmic complexity, but do not compute exact centrality values.

However, optimization algorithms are not directly applicable to behavioural trust management. First, even an improvement of the complexity by one order of magnitude (e.g., from days to hours) might not be enough for our setting. It is not feasible that a user has to wait hours until his trust policies are evaluated – policies must be evaluated immediately. Second, approximation algorithms have the disadvantage that they might not compute exact results. This is problematic for the evaluation of trust policies if the approximation algorithms do not guarantee the quality of their results.

To solve these problems, we propose the combination of approximation algorithms with a caching system for centrality values. To provide immediate trust decisions when evaluating trust policies, the BTM engine accesses cached centrality values. The cached values are recalculated periodically, e.g., at night or once a week. The BTM engine updates the cached values continually to consider new feedback which is given between the periodic recalculations. We use approximation algorithms with lower complexity for the updates.

In the literature, several algorithms have been proposed for the optimization of centrality measures. However, most of these optimizations are not feasible for the scenario of trust management. First, approaches based on the reduction of the number of iterations decrease the run times only marginally. While *Aitken* and *Quadratic Extrapolation* [KHM03] are useful only for uncommonly large damping factors, the *Step-length Calculation* approach [SN06] does not guarantee correct rankings of the entities. Second, approaches based on the reduction of the complexity of the iterations are also not applicable. While the *Adaptive PageRank* algorithm [KHG03] does not guarantee the convergence of the approximations, the *Two Phase* algorithm [LGZ03] is only useful for graphs with a large number of sinks (nodes without outgoing edges).

An optimization for Eigenvector-based centrality measures we consider feasible is the *PageRank Update* algorithm [CDK+04]. This algorithm determines the subgraph which is affected by new edges and recalculates the PageRank values

only for this subgraph. That is, this algorithm considers only those parts of the graph which are likely to change when new feedback is given. We use the resulting subgraph for both PageRank and HITS. As damping factor d we chose the default value of 0.85 and as threshold δ we chose 10^{-7}.

To reduce the complexity of distance-based centrality measures, we have adapted the sampling algorithm *RAND* [EW04] to the Proximity centrality measure. This algorithm gives guarantees for the maximal error in the approximated centrality values. As sampling size we used $2 \ln(n) / \varepsilon^2$ with $\varepsilon = 0.9$, resulting in a maximal error of $\varepsilon \cdot \text{diam}(G)$. We implemented the algorithm as new measure in the centrality operator. This allows users to use the heuristic directly in their trust policies:

```
CENTRALITY Rank, ID, Rater, Ratee, Value, ProximityApprox
FROM Participants, Feedback
```

Note that the implemented approximation algorithms might not compute exact centrality values. We evaluated the algorithms to test if they still provide a correct ranking of the entities. We consider this as acceptable because behavioural trust policies typically use relative instead of absolute centrality values.

6.2 Caching of the Centrality Values

The BTM engine follows a centralized approach for the evaluation of trust policies. Thus, both the storage of the data and the centrality calculation take place in a single database. Users are interested in instant trust decisions whether a given service provider is trustworthy or not. To evaluate trust policies instantly, the BTM engine relies on cached centrality values.

6.2.1 Required Tables

When new feedback is given through the feedback service, the BTM engine stores it both in the table `feedback` and in an additional table called `newFeedback`. This table represents a materialized view of the feedback table and thus has the same schema. The newFeedback table stores feedback which has been given since the last calculation of the centrality measures. The BTM engine flushes the table after each update by the heuristics and each complete recalculation.

For each feedback of user *A* on service provider *B*, the feedback service inserts a new tuple into the feedback table. We use the following trigger to copy new feedback tuples automatically into the newFeedback table. The trigger calls the function copy_feedback() after each insert statement:

```
CREATE TRIGGER copy_feedback AFTER INSERT ON Feedback
FOR EACH ROW EXECUTE PROCEDURE copy_feedback();
```

As illustrated in Figure 6-1, the exact and approximated centrality values are stored in tables with the columns `Id` and `Rank`. The column `Id` contains the unique identifiers of the entities and the column `Rank` contains the centrality values. The tables have the same name as the centrality measures to allow for a simple processing of cached centrality statements in the query parser. We prefer

separate tables for each centrality measure over one table with several columns because it allows us to flush the tables with the exact values entirely with each periodical recalculation.

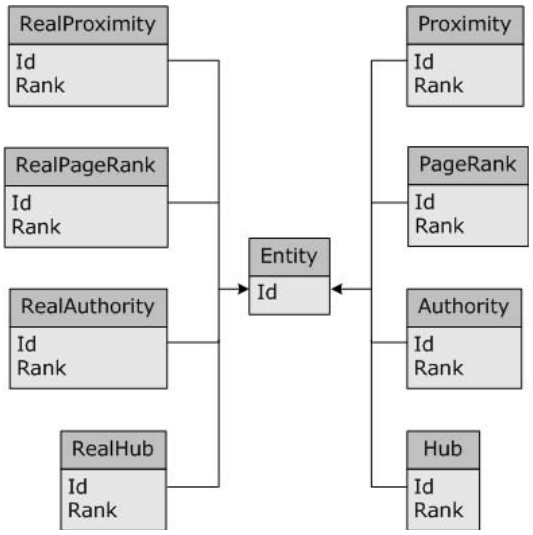


Figure 6-1: Tables for the cached centrality values

6.2.2 Cached Centrality Operator

For the convenience of the users, we have created a special centrality operator to access the cached values. The parser of the BTM engine redirects centrality statements to the tables which contain the cached values. For example, the parser translates the centrality statement

```
CENTRALITY CACHE rankCol, idCol, centralityMeasure
```

into the following SQL statement:

```
SELECT id AS idCol, rank AS rankCol FROM centralityMeasure
```

If the user prefers a complete recalculation of the centrality values, he can still use the original centrality operator or call the centrality function directly:

```
SELECT id, rankValue FROM centrality_func(RankValue, ID, Rater,
Ratee, Value, CentralityMeasure, Participants, Feedback)
```

6.3 Heuristics

The caching of the centrality values allows the BTM engine to evaluate trust policies instantly. However, we also have to consider feedback which was given since the last recalculation of the centrality measures. To keep the centrality values as current as possible, the BTM engine updates the centrality values continually by using the approximation algorithms presented in Section 6.1.

For the calculation of the heuristics, further tables are needed (see Figure 6-2). The table `nodeStat` stores the indegree and outdegree values for each node in the graph. Thus, these values must be computed only once and may easily be updated when new feedback arrives.

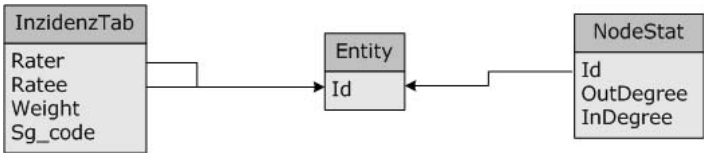


Figure 6-2: Tables for the calculation of the heuristics

Recall that the feedback graph is a multigraph, i.e., there might be several edges between a given pair of nodes. Since centrality algorithms are defined on simple graphs only, we have to transform multiple edges to a single edge. We use the *Beta* transformation which we implemented in SQL as follows (see procedure `fillInzidenzTab`).

```
SELECT rater, ratee,
SUM(CASE WHEN value > 0 THEN value ELSE 0 END)/SUM(ABS(value))
FROM Feedback
WHERE rater IN (SELECT * FROM Entity) AND ratee IN (SELECT * FROM
Entity)
GROUP BY rater, ratee
HAVING SUM(ABS(value)) > 0;
```

The transformed edges are stored in the table `inzidenzTab`. Note that the weights of the transformed edges might change when new feedback is given. Thus, the weights of the edges which are affected by the new feedback must be recomputed before the heuristics can be applied again.

To update the edge weights in the table `inzidenzTab` when new feedback is given, we use the pl/pgSQL-function `doUpdatesByNewFeedback()`. If the new edge does not yet exist, the function inserts a new tuple. If the new edge already exists, the edge weight is updated according to the Beta transformation. Finally, the function updates the in- and out degree in table `nodeStat`.

6.3.1 The Subgraph

The table inzidenzTab contains the attributes `rater`, `ratee`, and `weight` to store edges. Furthermore, the attribute `sg_code` describes the affiliation of each edge to the subgraph. This attribute is computed during the update of the PageRank values. The attribute can take one of the following values:

- '0' if both rater and rate belong to the super node;
- '1' if only the ratee belongs to the super node;
- '2' if only the rater belongs to the super node;
- '3' if neither rater nor ratee belongs to the super node.

We use this coding for the calculation of the edge weights of the subgraph. These weights are needed because we approximate both PageRank and HITS with the PageRank Update algorithm. This allows us to reuse the information which nodes and edges belong to a super node. Just the weights of the edges in the subgraph must be determined separately.

To enable the calculation of PageRank and HITS on the subgraph with the centrality operator, we store the edges and weights of the subgraph in the table `subgraphTab` (see Figure 6-3). This table does not correspond to a materialized

view of the relation inzendenzTab because it contains additional edges between the nodes of the subgraph and the super node as well as recursive edges from the super node to itself. The table `subgraphEntities` stores the identities of the nodes of the subgraph as well as the identity of the super node (`-1` in our case).

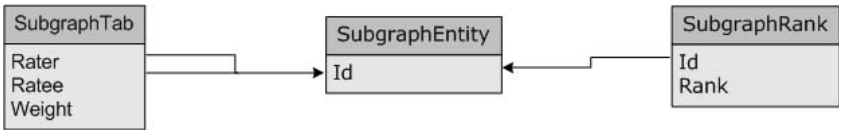


Figure 6-3: Tables for the subgraph

The centrality values of the subgraph must also be cached to determine the global centrality values for the whole graph. For each centrality measure which is computed based on the subgraph, a new table is required with the columns `id` and `rank` (see Figure 6-3). This allows us to add new centrality measures dynamically.

6.3.2 Execution of the Heuristics

In general, a new feedback edge has only very little influence on the global centrality values. It is not feasible to execute the heuristics for updating the centrality values with each new feedback edge. Thus, the heuristics are executed only if a certain amount of new feedback is available. We use the following trigger to call the function `new_feedback_added()` when a new tuple is inserted into the newFeedback table:

```
CREATE TRIGGER new_feedback_added AFTER INSERT ON newFeedback
FOR EACH ROW EXECUTE PROCEDURE new_feedback_added();
```

The function `new_feedback_added()`monitors the amount of new feedback and calls the function `updateRankings()` for updating the centrality values if the fraction of new feedback exceeds a certain threshold (3% in our case). After the updates are completed, the function clears the table newFeedback. Note that new feedback which arrives during the update procedure is preserved and not deleted.

6.3.3 Periodical Recalculations

Both the insertion of tuples into the `newFeedback` table and the execution of the heuristics are controlled by triggers. In addition to the continuous updates, we propose a complete recalculation of the centrality measures in periodical intervals, e.g. at night or once a week. This may be done automatically by an external program (e.g., a shell script) or manually through the SQL interface of the BTM engine.

We propose to use a cron job to trigger the recalculation of the centrality measures periodically. This cron job uses the PostgresSQL user interface `psql` to execute the SQL statements for the recalculation. For each centrality measure, the table used to store the exact values must be cleared and filled with new values. We combined the required SQL statements into the SQL script computeRankings.sql, which may be executed with the following command:

```
psql -f /mySQLScripts/computeRankings.sql trustdb
```

6.4 Evaluation

We have tested the implemented heuristics on different graphs to evaluate their performance and quality. In this section, we describe the graphs and the error measures used in the tests as well as the experimental setup. The results obtained are presented and discussed in the next section.

6.4.1 Test Graphs

We tested the heuristics on two graphs of different size (see Table 6-1). We used real-world graphs of the social networking sites Advogato and Epinions to obtain reliable results.

Advogato (www.advogato.org) is a blogging community for developers of freely available software. The blogs are created and managed by users, which form the nodes of the graph. Each user can rate the blogs of other users. The ratings form the edges of the graph.

Epinions (www.epinions.com) is a website on which users can rate products. The users form the nodes of the graphs. Each user can select other users whom he trusts. The trust relations form the edges of the graph.

	# nodes	# edges
Advogato	7 382	57 572
Epinions	49 290	487 183

Table 6-1: Size of the test graphs

6.4.2 Error Measures

We use two error measures to evaluate the heuristics. The first one measures the absolute error between the actual and the approximated values. The second one measures the difference in the ranking, i.e., the relative error.

For the evaluation of behaviour-based trust policies, a correct ranking is often more important than exact centrality values. The reason is that behaviour-based trust policies often refer to the ranking of the entities. A simple example of such a policy is: "I trust only service providers which are in the top 10 of the most reputable entities."

The *L1-error* measures the absolute distance between two vectors. It is defined as the sum of the differences between the actual and the approximated values. This measure can be used to determine the accuracy of the approximated values. However, it is possible that small differences in the L1-norm cause large differences in the ranking.

The *Spearman Footrule distance* [S06] is an appropriate measure for comparing two rank vectors. It is defined as the sum of the square differences between the actual and the approximated values. If the difference exceeds a threshold value $\delta \geq 0$, it is added to the error. This threshold value allows ignoring small

differences in the ranking. The errors are normalized to enable the comparison of rank vectors of different lengths.

In its original form, calculation of the Spearman Footrule distance takes place on the whole vector. However, in the example policy above, only the ranking of the 10 most reputable entities is relevant. For this purpose, the *Top-k Spearman Footrule distance* considers only the top *k* entities for the calculation of the error.

6.4.3 Experimental Setup

We implemented the heuristics in the relational database management system PostgreSQL 8.3.1. The evaluation took place on a machine with four AMD Opteron 2600 MHz dual core processors and 24 GB main memory. The machine was running Red Hat Linux 3.4.6-11.

Our evaluation is based on the scenario that users give feedback on other users. The feedback is given randomly and distributed uniformly over the network. After a certain amount of new feedback (1%, 3%, and 5%), the calculation of heuristics is triggered.

To implement this scenario, we took a random sample of 1%, 3%, and 5% of the edges from both graphs. The edges of the sample were then deleted from the original graph. The name of the test graphs depends on the percentage of new feedback: *Advogato1, Advogato3, Advogato5* and *Epinions1, Epinions3, Epinions5*. As reference values, we calculated the centrality measures on the original graph (with all edges).

We evaluated the heuristics for Eigenvector-based centrality measures on both the Advogato and the Epinions graph. However, the calculation of the reference values for distance-based centrality measures would take a very long time (several weeks) on a graph of the size of the Epinions graph. Thus, we limited the evaluation of the heuristics for distance-based centrality measures to the Advogato graph.

We used three error measures to evaluate each graph. *L1Error* measures the accuracy (i.e., the absolute error) of the approximated values. *TopnSFD* measures the relative error in the ranking of all entities. We used a threshold value of 0 to determine all errors in the ranking. *TopkSFD* measures the relative error in the ranking of the entities that belong to the top 10% of the most reputable entities.

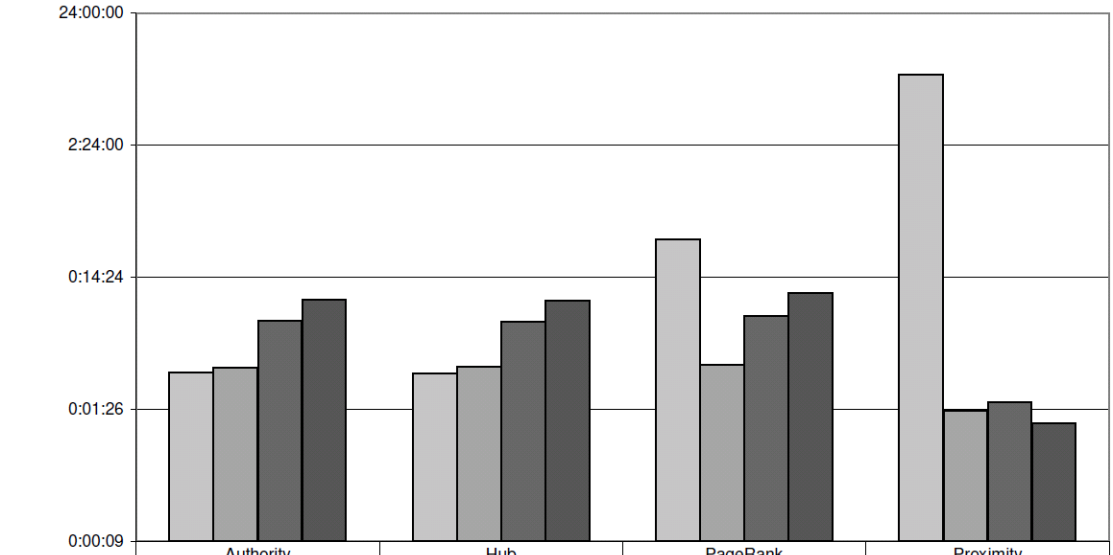
6.5 Results

In this section, we present the results of the evaluation. First, we compare the run-times of the heuristics with those of the exact algorithms. Then we determine the approximation error of the heuristics.

6.5.1 Performance

First, we determine how much the updates of the centrality values by the heuristics are faster than the recalculation by the exact algorithms. Note that we measured CPU times to exclude waiting times for I/O and resource sharing. Figure 6-4 and Figure 6-5 show the run-times of the heuristics measured on the

different graphs. The ordinate axes of the figures are scaled logarithmically. *RefTime* specifies the reference time required to recalculate the centrality values using the exact algorithm, which is referred to as 'standard algorithm' in the following. *ApproxTime* is the time required to approximate the centrality value using the respective heuristic.



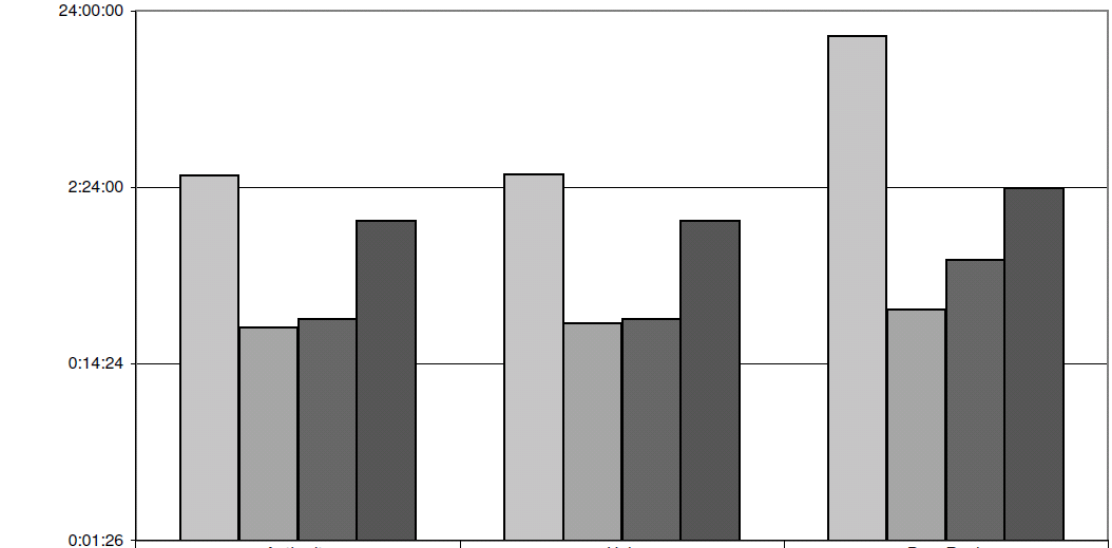
	Authority	Hub	PageRank	Proximity
RefTime	0:02:43	0:02:41	0:27:46	8:11:35
Advogato1	0:02:59	0:03:02	0:03:08	0:01:24
Advogato3	0:06:42	0:06:36	0:07:22	0:01:38
Advogato5	0:09:38	0:09:31	0:10:57	0:01:08

Figure 6-4: Run times of the heuristics on Advogato

The results on the Advogato graph (Figure 6-4) show that the heuristic for HITS (Authority and Hub) has longer run-times than the standard algorithm. While the difference is relatively small on Advogato1, the heuristics on Advogato3 and Advogato5 take almost four times longer than the standard algorithm. A possible explanation is that the number of iterations required until the heuristic converges is higher on sparsely populated graphs. We conclude that the heuristic for HITS does not pay off for small graphs such as Advogato.

The heuristic for PageRank is not affected by this phenomenon because the standard algorithm typically requires a high number of iterations. The heuristic for PageRank has a much smaller run-time than the standard algorithm. Depending on the amount of new feedback, the heuristic is three to nine times faster than the standard algorithm.

The approximation of the values by the Proximity heuristic takes very little time in comparison to the calculation of the exact values by the standard algorithm. Recall that this heuristic is based on a random sample of the nodes (and not of the edges). Thus, the run-times are independent of the amount of new feedback.



	Authority	Hub	PageRank
RefTime	2:49:29	2:51:27	17:13:06
Epinions1	0:23:18	0:24:18	0:29:27
Epinions3	0:25:50	0:25:54	0:55:55
Epinions5	1:33:25	1:33:14	2:22:21

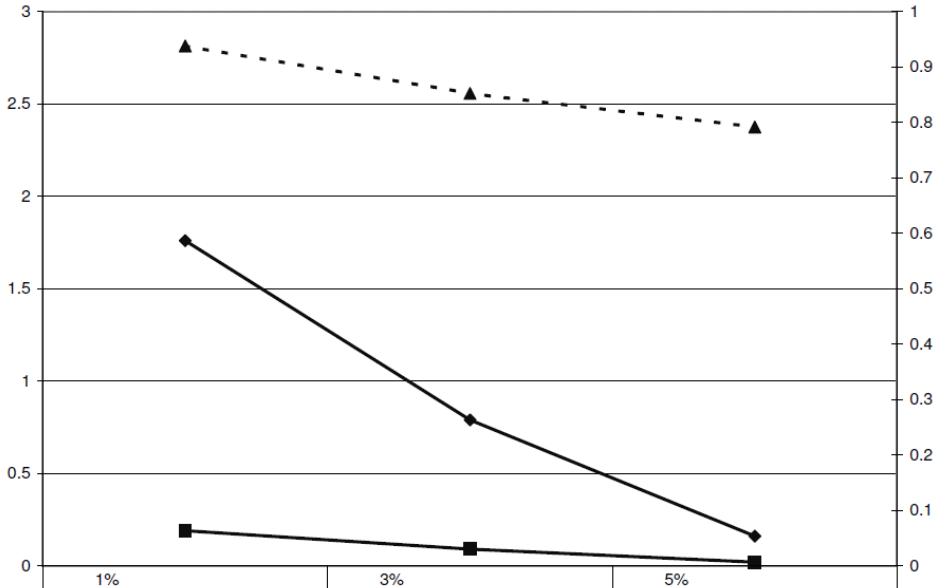
Figure 6-5: Run times of the heuristics on Epinions

In contrast to the Advogato graph, the heuristic for HITS pays off on the Epinions graph (see Figure 6-5). The standard algorithm for HITS takes seven times longer than the heuristic for 1% and 3% of new feedback and two times for 5% of new feedback. For PageRank, the gain of time is even higher. The heuristic is 7 to 35 times faster than the standard algorithm, depending on the amount of new feedback. Again, an increase in run-time of the heuristic is observed at 5% of new feedback.

6.5.2 Approximation Errors

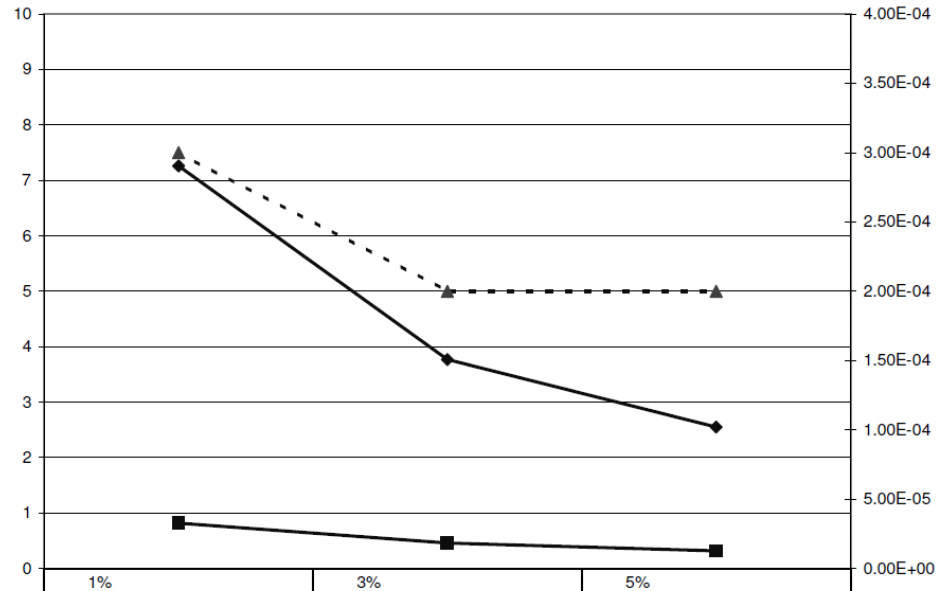
Reduced computation times in comparison to the standard algorithms are not sufficient for the heuristics to be suitable for behaviour-based trust management. The continuous updates of the centrality values must also guarantee small approximation errors, in particular for the rank order of the entities. Since the computation of the reference rankings by the standard algorithms takes a very long time for distance-based centrality measures, we limit this evaluation to the smaller Advogato graph.

Figure 6-6 shows the approximation error of the PageRank Update algorithm on the Advogato graph. On the primary axis, the TopkSFD and TopnSFD errors are shown in percent. On the secondary axis, the absolute value of the L1-error is plotted. Despite of large L1-errors, the errors in the rank order are small (under 2%). That is, the heuristic computes almost perfect rank orders, both for the top-k and for the whole ranking. We observe that both the L1-error and the TopkSFD and TopnSFD errors decrease with the amount of new feedback. The reason is that the subgraphs grow in size with an increasing amount of new feedback, allowing the heuristics to approximate the centrality values more precisely.



	1%	3%	5%
TopkSFD(%)	1.76	0.79	0.16
TopnSFD(%)	0.19	0.09	0.02
L1Error	0.9381	0.8526	0.7916

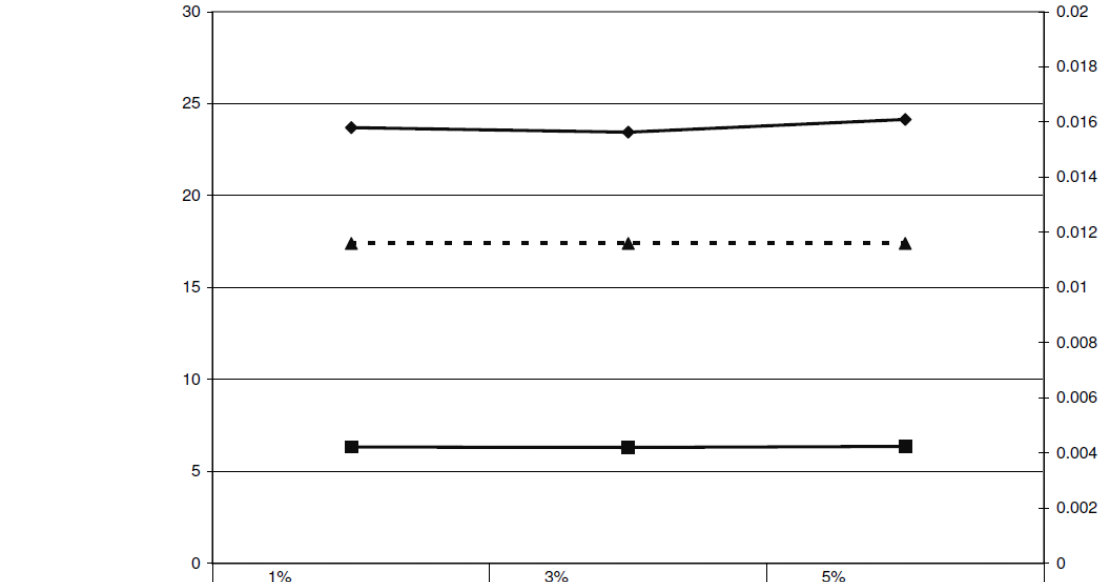
Figure 6-6: Errors for PageRank on Advogato



	1%	3%	5%
TopkSFD(%)	7.26	3.77	2.55
TopnSFD(%)	0.82	0.46	0.32
L1Error	0.0003	0.0002	0.0002

Figure 6-7: Errors for HITS on Advogato

Although the heuristic for HITS causes a small L1-error on the Advogato graph, the errors in the top-k rank order are rather large (see Figure 6-7). The TopkSFD error is about 7.3% for 1% of new feedback. With an increasing amount of new feedback the TopkSFD error falls to 2.55% for 5% of new feedback. The errors in the order of the whole ranking are almost negligible. We conclude that a small absolute error does not guarantee a correct rank order, especially in the top-k of the ranking where the values are very close to each other.



	1%	3%	5%
TopkSFD(%)	23.7	23.45	24.14
TopnSFD(%)	6.32	6.3	6.36
L1Error	0.0116	0.0116	0.0116

Figure 6-8: Errors for Proximity on Advogato

The sampling algorithm for Proximity causes a relatively small L1-error (see Figure 6-8). However, it causes a very large error in the rank order. While the error in the order of the whole ranking is still acceptable (about 6.3%), the error in the top-k ranking is very big (around 24%). Again, the amount of new feedback does not make a difference. The error in the whole ranking is much smaller than the error in the top-k ranking because many nodes in the bottom of the ranking have a Proximity value of 0. These are nodes that cannot be reached from any other node in the network, i.e., users without ratings.

6.6 Discussion

As expected, our evaluation has shown that the updates of the centrality values by the heuristics take much less time than a complete recalculation by the standard algorithms.

However, the sampling algorithm for Proximity causes a large error of almost 25% in the top-k ranking, despite of a relatively small L1-error. This shows that even a small L1-error can lead to large differences in the rank order. Due to the high error, we consider the heuristic for Proximity as not suitable for our setting. Since this heuristic is the only known algorithm for optimizing distance-based centrality measures, we discourage their usage in behaviour-based trust policies.

The PageRank Update algorithm has proven to be suitable for our setting. Despite its L1-errors, the heuristic provides a good approximation of the ranking order, both in the top-k and in the whole ranking. Thus, we consider this heuristic as suitable for behaviour-based trust management.

We recommend 3% of new feedback as threshold value to trigger the calculation of the heuristics. It is a good compromise between the size of subgraphs and the approximation errors, ensuring a good approximation of the rank order.

7 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 for users giving feedback on service providers. 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, authorizing the user to give feedback on the interaction. 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.

7.1 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);
```

7.2 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 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 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;
```

8 License Information

The Software is provided under a BSD style license given in Section 8.1. This code uses PostgreSQL code to which the license given in Section 8.2 applies.

8.1 Behavioural Trust Management Engine

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the Karlsruhe Institute of Technology nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE KARLSRUHE INSTITUTE OF TECHNOLOGY "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE KARLSRUHE INSTITUTE OF TECHNOLOGY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8.2 PostgreSQL Data Base Management System

9 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.

Because of their algorithmic complexity, the calculation of centrality measures may take days or even weeks for large feedback graphs. It is not feasible that the users have to wait such a long time for the evaluation of their policies. Thus, the centrality values have to be cached and recalculated periodically. To guarantee the timeliness of the cached values, we propose the usage of heuristics to continually update the centrality values between the periodic recalculations. While the heuristics might not compute exact centrality values, they still provide a correct ranking of the entities, what we consider as adequate.

An analysis of various optimization techniques from the literature showed that most heuristics are not suitable for the evaluation of trust policies. However, we have identified and implemented two heuristics for both Eigenvector-based and distance-based centrality measures. While the PageRank Update algorithm computes correct rankings, the sampling algorithm for Proximity causes errors in the ranking of the entities. Thus, this heuristic is not suitable for the evaluation of behaviour-based trust policies.

We verify that users are authorized to give feedback on a particular interaction by interfacing the authorization infrastructure. The authorization procedure is twofold: First, users must be authenticated by an identity provider. Second, users may give feedback only through trusted components such as the business process engine. Besides feedback provided by end-users, it is also conceivable to gather feedback based on auditing results from the inspection of logs. Auditing and logging guidelines needed for this are described in D5.3.

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.

[D5.3] Deliverable 5.3: Novel Trust Metrics.

[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.

[CDK+04] S. Chien, C. Dwork, R. Kumar, D.R. Simon, D. Sivakumar, "Link Evolution: Analysis and Algorithms", Internet Mathematics, 1:277–304, 2004.

[EW04] D. Eppstein, J. Wang, "Fast Approximation of Centrality", Journal of Graph Algorithms and Applications, 8:39–45, 2004.

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

[KHG03] S.D. Kamvar, T.H. Haveliwala, G.H. Golub, "Adaptive Methods for the Computation of PageRank", Technical report, Stanford University, 2003.

[KHM03] S.D. Kamvar, T.H. Haveliwala, C.D Manning, G.H. Golub, "Extrapolation Methods for Accelerating PageRank Computations", In Proceedings of the 12th international conference on World Wide Web, 1:261–270, 2003.

[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.

[LGZ03] C. P. Lee, G. H. Golub, S. A. Zenios, "A Fast Two-Stage Algorithm for Computing Pagerank and Its Extensions", Technical report, Stanford University, 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.

[S66] G. Sabidussi, "The Centrality Index of a Graph", Psychometrika, 31:581–603, 1966.

[SN06] K. Saito and R. Nakano, "Improving Convergence Performace of PageRank Computation Based on Step-Length Calculation Approach", Lecture Notes in Computer Science, Knowledge-Based Intelligent Information and Engineering Systems, 2006.

[S06] C. Spearman, "Footrule for Measuring Correlation", British Journal of Psychology, 2:89–108, 1906.

[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 TAS³ scenario (Section 4.1)
1.0	2009-12-31	CH	Final version released to public.
1.1	2010-05-21	CH, KB	Chapter on optimizations.
1.2	2010-05-31	CH, KB	Second complete version.
1.3	2010-06-18	CH, KB	Implemented review comments by Slim Trabelsi (SAP).
2.0	2010-06-30	CH	Created final version.