A Generic Decentralized Trust Management Framework

Roy Friedman  Amit Portnoy
Computer Science Department
Technion
Haifa 32000, Israel
{roy,amitport}@cs.technion.ac.il
November 12, 2012

Abstract

This paper describes TRUSTPACK, a decentralized trust management framework that provides trust management as a generic service. TRUSTPACK is unique in that it does not provide a central service. Instead, it is run by many autonomous services. This design enables TRUSTPACK to alleviate privacy concerns, as well as potentially provide better personalization and scalability when compared with current centralized solutions.

A major component of TRUSTPACK is a generic decentralized graph query processing framework called GRAPHPACK, which was also developed as part of this work. GRAPHPACK consists of a decentralized graph processing language as well as an execution engine, as elaborated in this paper. The paper also presents several examples and a case study showing how TRUSTPACK can be used to handle various trust management scenarios, as well as its incorporation in an existing third party P2P file sharing application.

Prototypes of TRUSTPACK and GRAPHPACK are available as open source projects at http://code.google.com/p/trustpack/ and http://code.google.com/p/graphpack/ respectively.

1 Introduction

1.1 Background

Trust is information that helps making a judgment about something. Trust Management (TM) is a facility that helps with trust related decision making in the presence of uncertainty [42]. TM became extremely common with the proliferation of the Internet and its applications [8, 28]. It is used by a variety of applications, such as: Peer-to-Peer (P2P) networks (e.g., BitTorrent [15], Tribler [37]), e-commerce sites (e.g., eBay, Amazon), web-based social communities (e.g., CouchSurfing, Facebook), sites with a collaborative content management (e.g., Wikipedia, YouTube, Yelp), etc.

Currently, TM is usually developed together with the application that uses it. Yet, recently it has been noticed [22, 24, 30] that the concept of trust management is generic and the methods for its processing are common between many applications. This enables the creation of systems that externally provide TM for several applications simultaneously. Such systems are called Trust Management Frameworks (TMFs).

We believe that in the future TMFs will become a common cost-effective middleware. This is because their approach has many advantages over ad-hoc implementations of TM. First, new applications rely on
a time-tested stable system instead of writing new code. Thus, they can be developed faster, and in better quality in terms of scalability, responsiveness and security. Second, a generic TM inherently enables more flexibility and interoperability. This facilitates sharing trust between applications and trust readjustment, resulting in a more informed decision making process, and hence a better TM.

1.2 Required Terminology

In order to set a common terminology for the discussion below, and in particular to analyze the benefits and limitations of existing TMFs, and correspondingly our contributions, we present the following definitions: Recall that for our purposes, trust is information that helps to make a judgment about something. The entity that makes such a judgment, called the source of trust or trustor, is usually either a user of an application (e.g., a user that chooses a restaurant based on restaurants reviews) or the application itself (e.g., a social site that bans users based on other users’ complaints). The entity that is being judged, called the target of trust or trustee, can be almost any type of object, for example: real world objects (e.g., restaurants), online content (e.g., YouTube videos, Wikipedia articles), other users (e.g., sellers on eBay), etc.

A record of such trust assignment (i.e., judgment making) is called a Trust Statement (TS). It contains the identities of the trustor and trustee, an evaluation of the trust (e.g., “the restaurant has 4 stars out of 5”), and other contextual information such as the time when the statement was made.

An application may maintain a repository of TSs that it considers valuable for its future decision making processes. Such a repository is called the trust graph of that application (the graph metaphor derives from viewing every TS as an edge that is directed from a trustor node to a trustee node). In this work, we consider the parts of an application that manage and process its trust graph as the TM of that application. A TM service is a service that enables applications to execute trust related tasks.

1.3 The Current Gap and Our Contributions

While current TMFs are an important step forward, they still suffer from some significant shortcomings. Specifically, existing TMFs manage a global trust graph, which is the union of the trust graphs of all the applications that use them. They also provide a central TM service, which has unlimited permissions for accessing the global trust graph (note that this service is logically central, i.e., it may be deployed on several machines that are controlled by a single authority).

We find this approach problematic for two reasons: First, the applications and the central TM service are assumed to be trusted with the TSs that were made by the users of the applications. We believe that those TSs should be considered sensitive personal information and that it is reasonable that a user may not trust the application or the central TM service with such information. Second, this approach assumes that the central TM service has access to the entire global trust graph. We believe that this is a limiting assumption since some applications and users might use (and trust) different services.

In this work, we present a prototype of a new TMF, nicknamed TRUSTPACK. In TRUSTPACK, there is no central TM service. Instead, it is run by a network of completely autonomous TM services, called TRUSTPACK services. Each of those services considers the applications’ users as its first-class clients. That is, users can choose which service they trust and directly control how their TSs are used by that service. This approach improves the users’ privacy and encourages extendability and interoperability by allowing services to be run and be developed independently.

During the development of TRUSTPACK, we have found that providing easy access to the trust graph is an especially interesting challenge. Such access is not trivial because, by design, TRUSTPACK services do not necessarily have access to the global trust graph or even the complete trust graph of a given application.
In addition, we found that this problem is not unique to TRUSTPACK. That is, there are other facilities that require access to a graph that is distributed among autonomous services. For example, there are some social networks where the social relationships are distributed among the users because of privacy concerns (e.g., PeerSoN [13] and Safebook [17]). Other more inherently distributed examples are ad-hoc networks and P2P networks. In those examples, the network itself is the graph that is distributed.

To enable the above mentioned facilities and to increase TRUSTPACK’s flexibility, we provide another, even more generic framework, nicknamed GRAPHPACK. The GRAPHPACK framework deals with management and processing of graphs that are distributed by a network of autonomous services, called GRAPHPACK services. We note that TRUSTPACK services are actually GRAPHPACK services that are extended with TM capabilities.

GRAPHPACK consists of a distributed graph query language, nicknamed PACKCYPER, as well as a distributed execution engine. PACKCYPER is inspired by CYPER, an emerging language that is part of the Neo4j graph database [5]. Hence, the advantages of GRAPHPACK is that it enables decentralized query processing over distributed graphs in which the queries can be expressed in a graph query language. The benefits of using such a language over general purpose query languages is simplicity for the user. The benefit of the distributed execution engine is that it hides the details of networking and network interaction from the user.

In addition, the paper includes several examples and a case study of applying TRUSTPACK in order to capture the TMs of several realistic scenarios, as well as an incorporation of a trust incentive mechanism into an existing third party open source P2P file sharing application. These exemplify the simplicity of using TRUSTPACK on one hand, and its generality on the other hand.

Paper roadmap We start by presenting GRAPHPACK in section 2. In section 3, we describe TRUSTPACK and show how it can alleviate the concerns that we have mentioned while still providing usable TM for a variety of applications. In section 4, we present a case study of extending an existing application using TRUSTPACK. Related work is discussed in section 5. Finally, we conclude with a discussion and future work in section 6.

2 The GRAPHPACK Framework

2.1 Overview

At the core of GRAPHPACK there is a decentralized network of services, called GRAPHPACK services. Each of those services provides clients with methods for managing their own graph. This graph is a directed graph where edges can be attached with unstructured data and the targets of those edges may refer to nodes that are managed by other clients. Figure 1 shows an example of several graphs and their connections. In that example, “GRAPHPACK service 1” manages the graphs of clients A and B, while “GRAPHPACK service 2” manages client C’s graph. The filled circles represent nodes that are managed by the client that own the graph and the empty circles represent nodes that refer to other graphs.

We note that the unification of all client’s graphs can be viewed as a property graph, which is a directed graph where unstructured data (i.e., properties) can be attached to edges. The reason for using property graphs in GRAPHPACK is their generality. That is, they can be used to express many types of graphs. For example, a graph with node properties and undirected edges can be modeled as follows: Node properties can be modeled with a self loop edge that stores the node’s properties. Undirected edges can be modeled by two opposing directed edges.
2.2 GRAPHPACK Service Architecture

The components of a GRAPHPACK service, as illustrated in Figure 2, can be divided into three groups: infrastructure, processing and libraries.

In the infrastructure group there are common facilities required by the GRAPHPACK service. This group includes the following components:

**Communication** provides a naming service that can create unique identifiers and locate other services based on such identifiers. It also provides facilities for sending and receiving a set of pre-known message types (the complete list appears in subsection A.1). Lastly, this component also includes parts that can be used by clients when communicating with services.

**Persistency** provides a facility for simple storing and retrieving of objects (for the complete specification, see subsection A.2).

The components of the processing group are responsible for the management and execution of tasks, which are blocks of code. Each client of a GRAPHPACK service has a few pre-installed tasks that he can control and execute. In addition, clients can also request the execution of other clients’ tasks and to install their own custom tasks. This enables the flexibility that is expected from a generic service. This group includes the following components:

**Authorization** enforces access control rules and provides clients with a method for managing those rules. The rules are simple execution permissions for tasks. This component uses the persistency component for the persistency of clients’ permissions.

**Task Management** provides facilities for dynamically installing, configuring and calling tasks. Tasks can be configured to be executed on schedule (e.g., “every Sunday at 21:00”), or just wait for execution requests (from a client, another GRAPHPACK service or another task). In addition, tasks can be configured to execute with permissions that are lower than the permissions of the client that installed them. This capability is useful for when a client installs a task that was received from a source that it does not completely trust. This component uses the persistency component for the persistency of tasks and their configurations.

Figure 1: An example of GRAPHPACK services with a few graphs
**Task Execution** loads and executes tasks upon request by the task manager. The tasks are executed securely and with proper permissions. That is, with the permissions of the client that installed it, or with lower permissions, if specified by that client. In addition, the task execution component enables the executing tasks to access the libraries components.

The components of the *libraries* group provide executing tasks with data structures and utilities that they can use during their execution. Since tasks may arrive from clients, those library components are designed to be simple and extendable. This group includes the following components:

- **Graph** provides graph related operations that are detailed in Table 1. The operations are divided into *scopes*, which determine where each operation is present. That is: *client*, *node* and *edge* scoped operations are provided for every *client*, *node* and *edge*, respectively. We note that because of our decision to use autonomous services, we cannot easily provide *globally scoped operations* (e.g., “get all edges”). Graph operations are automatically wrapped by tasks that enable them to be accessed remotely. Most of the operations are implemented trivially by delegating to the persistence component. The only exception is the *traverse* operation, which uses a Domain Specific Language (DSL) and has a complex process. The traverse DSL, called PACKCYPHER, is discussed in subsection 2.3. The traverse process is discussed in subsection 2.4.

- **Extensions** are a set of optional libraries that provide executing tasks with extra data structures and utilities. In particular, TRUSTPACK is an example of such an extension (as detailed in section 3).
Table 1: Graph operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>addNode</td>
<td>adds a node to the graph</td>
<td>client</td>
</tr>
<tr>
<td>readNodes</td>
<td>iterates over the nodes that were added by this client</td>
<td></td>
</tr>
<tr>
<td>deleteNode</td>
<td>deletes this node</td>
<td>node</td>
</tr>
<tr>
<td>addEdge</td>
<td>adds an edge that originates at this node</td>
<td></td>
</tr>
<tr>
<td>readEdges</td>
<td>iterates over the edges that originate at this node</td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td>(see subsection 2.3 and subsection 2.4)</td>
<td></td>
</tr>
<tr>
<td>deleteEdge</td>
<td>deletes this edge</td>
<td>edge</td>
</tr>
<tr>
<td>getTarget</td>
<td>returns the target of this edge</td>
<td></td>
</tr>
<tr>
<td>getSource</td>
<td>returns the source of this edge</td>
<td></td>
</tr>
<tr>
<td>getEdgeData</td>
<td>returns the data that is attached to this edge</td>
<td></td>
</tr>
<tr>
<td>updateEdgeData</td>
<td>updates the data that is attached to this edge</td>
<td></td>
</tr>
</tbody>
</table>

2.3 PACKCypher - a Language for Graph Traverses

Since many graph related algorithms require complex traverses over the graph, it is important to demonstrate that such complex traverses can be easily specified using our graph library. For this reason, we have developed a graph traverse language, nicknamed PACKCypher. As mentioned above, PACKCypher is inspired by Cypher, an emerging language that is part of the Neo4j graph database [5].

Cypher is a graph query language, i.e., a language for retrieving data from a graph. PACKCypher borrows from Cypher its declarative pattern matching syntax, which enables for a very concise and simple expression of traverses. In addition, PACKCypher extends Cypher with support for specifying tasks that should be executed by nodes that are visited during the traverse. This capability enables PACKCypher to be used not only for specifying data retrieval but also for specifying complex communication processes between the nodes of the graph (e.g., gossip).

When the traverse operation is called on some node, it needs to specify the paths (i.e., sequences of edges where the source of every edge is either the initial node or the target of the previous edge) that should be taken, the information that should be gathered, and the remote tasks that should be called. All of those can be specified with PACKCypher. In the rest of this section, we describe only the core of PACKCypher features. For the complete syntax specification see [Appendix B](#).

In PACKCypher, the main string that specifies the traverse is called a path expression. The most simple path expression is called the edge string, which represents any outgoing edge and is written as “-->”. Several edge strings can be concatenated together to specify longer paths. For example: “-->-->” represents any path with exactly two edges, “-->-->-->” represents any path with exactly three edges, and so forth. We can also specify paths with bounded repetition by writing an asterisk (‘*’) followed by upper and lower bounds that are separated by “..”. For example: “-->*1..20” represents any path with 1 to 20 edges.

For specifying complex patterns in the paths we use a type of named capturing groups[^1]. Named capturing groups allow for parts of the paths that are matched by the path expressions to be captured for later reference. It is possible to capture target nodes, outgoing edges and paths: Nodes are captured by writing an identifier after an edge. For example, “-->X” will capture as X all nodes that are the target of some outgoing edge. Edges are captured by writing an identifier surrounded by brackets in the middle of an

[^1]: http://www.regular-expressions.info/named.html
edge. For example, “−[X]−>” will capture X all outgoing edges. Paths are captured by surrounding a path in parentheses and appending them with the equals symbol (“=”) followed by an identifier. For example, “(−>−>−>)=X” will capture as X all the paths with two edges. Table 2 demonstrates a few path expressions that use capturing groups. Those expressions are applied on node ‘a’ in Figure 3.

<table>
<thead>
<tr>
<th>PACKYPHER expression</th>
<th>Interpretation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>−&gt;X</td>
<td>capture as X all nodes that are the target of some outgoing edge</td>
<td>X={b,c,d}</td>
</tr>
<tr>
<td>−[X]−&gt;</td>
<td>capture as X all outgoing edges</td>
<td>X={1,2,3}</td>
</tr>
<tr>
<td>−[X]−&gt;Y</td>
<td>capture as X all outgoing edges, capture as Y all their target nodes</td>
<td>(X,Y)={(1,b),(2,c),(3,d)}</td>
</tr>
<tr>
<td>−&gt;X−&gt;Y−&gt;X</td>
<td>capture as X all nodes that are both at edge distance of 1 and also at edge distance of 3, capture as Y nodes that are the between the nodes matched by X</td>
<td>(X,Y)={(c,e)}</td>
</tr>
<tr>
<td>(−&gt;X−&gt;−&gt;)=Y</td>
<td>capture as Y all paths with two edges, capture as X the nodes that are at the middle of those paths</td>
<td>(X,Y)={(c,−[2]−&gt;c−[5]−&gt;e), (b,−[1]−&gt;b−[4]−&gt;e)}</td>
</tr>
</tbody>
</table>

Table 2: Capturing groups examples

PACKYPHER also provides a method for specifying predicates that should be checked during the traverse. Predicates are specified by appending an edge or a path, with the ‘?’ symbol followed by a predicate expression surrounded by parentheses. Those predicate expressions provide basic comparisons and Boolean operations that are applied on properties, which are accessed by using the ‘.’ symbol. For example: “−[X]−>? (X.priority>3)” specifies all outgoing edges with a priority property that is higher than 3. When applied to node ‘a’ in Figure 3, this path expression will capture X={1,3}.

Figure 3: An example graph
An additional important feature of PACKYPHER is that it provides a method for requesting the execution of tasks. Such a request is specified by appending an edge with the ‘.’ symbol followed by the name of the task that is requested and the task’s arguments surrounded by parentheses. For example, the expression “-->receiveGossip('Madonna directed a new movie’)” will request from all the nodes that are at the target of some outgoing edge, to execute a task called “receiveGossip” with the parameter string: “Madonna directed a new movie”.

Lastly, there are a few additional small features that are offered by PACKYPHER:

- The source, edge and target of the last edge that was matched can be referred by using the “_s”, “_e” and “_t” identifiers correspondingly. For example: “-->X?(_e.priority>3)” captures nodes as X, but runs the predicate on the edges without matching them.

- Since “type” is expected to be a commonly used edge property, it can be easily checked for equality by specifying the expected type after a colon inside the brackets that match the edge. For example, “-[[:friend]->]” is the same as “-->?(_e.type=='friend’)”.

- It is possible to attach a PACKYPHER expression with a mapping from variable names to objects. Those objects can then be referenced in the PACKYPHER expression by using their corresponding variable name in braces. For example, the expression “-->?(_e.date>{currentDate})” will look for a mapping from “currentDate” to some object. It will then replace “{currentDate}” with that object.

2.4 The Traverse Process

As we have mentioned before, in GRAPHPACK every client controls only its own partition of the graph. This means that if some client wishes to process parts of the graph that are in another client’s partition, it has to send a task request to that client’s service. Such a task request may be very expensive due to the fact that it may be performed remotely and may require some security checks. For this reason, in the traverse process, we try to minimize the number of such task requests and the amount of information that is transferred by those tasks.

Our first attempt with a traverse process was Algorithm[1] which is a common traverse algorithm used in graph databases. It uses an object called processor, which is derived from the PACKYPHER expression by straight-forward parsing. The processor object can process paths (via the process function) and return a status object, which returns results that were matched by the processor via the getResults function (it may return the empty set) and indicates if it is possible to continue processing along that path via the canContinue function.
Algorithm 1 \textit{traverse}(\textit{processor}, \textit{path})

1: \textit{results} \leftarrow \emptyset
2: \textbf{for all} \ e \in \textit{path.lastNode.readEdges()} \textbf{ do}
3: \quad \textit{newPath} \leftarrow \textit{path} appended with \( e \)
4: \quad \textit{status} \leftarrow \textit{processor.process} (\textit{newPath})
5: \quad \textit{results} \leftarrow \textit{results} \cup \textit{status.getResult}()
6: \quad \textbf{if} \ \textit{status.canContinue}() \textbf{ then}
7: \quad \quad \textit{results} \leftarrow \textit{results} \cup \textit{traverse}(\textit{processor}, \textit{newPath})
8: \quad \textbf{end if}
9: \textbf{end for}
10: \textbf{return} \ \textit{results}

Notice that Algorithm 1 requests the \textit{readEdges} operation from every node along all matching paths. This means that all the edges that are encountered during the traverse are being sent to the node that initiated the traverse, including edges that are not part of any matching path. In order to avoid the transfer of such edges, we have developed Algorithm 2. Algorithm 2 calls the \textit{readEdges} operation locally and \textit{delegates} the traverse to nodes that should be further processed. This reduces the number of task requests as well as the amount of information that is transferred in each task request.

Another difference between Algorithm 2 and 1 is that the processor object used in Algorithm 2 is more advanced than the one used in Algorithm 1. The new processor works incrementally. That is, at every edge along a path, its process function receives the current edge and a state that tells it what processing has already been done. Such a processor can be created from a PACKCYPHER expression by using algorithms that create Deterministic Finite Automaton (DFA) from regular expressions [9, 35]. We consider this processor to be preferable since it saves computation at each processing step (only simple state transitions are performed). It also better preserves the privacy of the client that initiated the traverse because it does not require that every traversed node will see the exact path that led to it (although the processor and the state do reveal some information about that path).

Algorithm 2 \textit{traverse}(\textit{processor}, \textit{state})

1: \textit{results} \leftarrow \emptyset
2: \textbf{for all} \ e \in \textit{readEdges}() \textbf{ do}
3: \quad (\textit{status}, \textit{newState}) \leftarrow \textit{processor.process}(\textit{state}, e)
4: \quad \textit{results} \leftarrow \textit{results} \cup \textit{status.getResult}()
5: \quad \textbf{if} \ \textit{status.canContinue}() \textbf{ then}
6: \quad \quad \textit{results} \leftarrow \textit{results} \cup e.get\textit{Target}().\textit{traverse}(\textit{processor}, \textit{newState})
7: \quad \textbf{end if}
8: \textbf{end for}
9: \textbf{return} \ \textit{results}

Algorithm 3, which represents what is currently implemented on \textit{GRAPHPACK}, further improves Algorithm 2. The processor object it uses does not process entire paths. Instead, it \textit{derives} new processors from the immediate edges. A processor that is derived by some edge, processes paths in the exact same manner as the original processor would have processed those paths if they were prepended with that edge. In other words, a derived processor is a processor for the remainder of the traverse. By using processor derivation, we can constantly decrease the size of the processor that is sent on each traverse request. Further, as indicated
in [36], even the initial processor is likely to be smaller in size than a state based processor, like the ones used in Algorithm 2. In addition, privacy is better preserved since the clients that respond to the traverse requests only receive the information that is necessary for the continuation of the traverse.

Our derivation method is based on Brzozowski’s derivation rules [12] (rediscovered in [36]). In his paper, Brzozowski investigates a method for matching a string to a regular expression. His method is based on a set of derivation rules, which specify how to derive a new regular expression from a given regular expression and a symbol that is matched against its start. More formally, when given a regular expression $R$ and a symbol $a$, he specifies how to calculate the $a$-derivative of $R$, i.e., $\partial_a R$. $\partial_a R$ is a regular expression that will match a string $s$ if and only if $R$ matches $s$ when prepended with $a$. It is important to note that in order to fully support pattern matching, our implementation extends Brzozowski’s derivation rules with support for capturing groups.

Algorithm 3 introduces a few new operations: $\partial_e$ is an operation that applies on a processor and returns its $e$-derivative. The status object’s new `getPartialResults` function returns partially matched results that were not required for the operation of the derived processor. $\otimes$ designates a Cartesian product over result sets. It is being used in order to merge every partially matched result with every non-conflicting result returned by the next traverse operation.

**Algorithm 3 traverse(\text{processor})**

1: $\text{results} \leftarrow \emptyset$
2: \textbf{for all} $e \in \text{readEdges}()$ \textbf{do}
3: \hspace{1em} $(\text{status, derivedProcessor}) \leftarrow \partial_e \text{processor}$
4: \hspace{1em} $\text{results} \leftarrow \text{results} \cup \text{status.getResults}()$
5: \hspace{1em} \textbf{if} \hspace{0.5em} \text{status.canContinue}() \hspace{0.5em} \textbf{then}
6: \hspace{1.5em} $\text{newResults} \leftarrow \text{newResults} \cup$
7: \hspace{2em} $(\text{status.getPartialResults}() \otimes e.getTarget().\text{traverse(\text{derivedProcessor})})$
8: \hspace{1em} \textbf{end if}
9: \textbf{end for}
10: \textbf{return} \text{results}

### 2.5 Usage Examples

Below, we present three simple usage examples that correspond to three different network types: unstructured P2P network, P2P social network and Pocket Switched Network (PSN). We note that our TRUSTPACK framework, as presented in section 3, is a more complex GRAPHPACK usage example.

**Unstructured P2P networks** are networks that do not have any central control and do not have any fixed layout of the peers within the network. The classic example of such a network is the Gnutella network [4].

GRAPHPACK can be used in those networks to immediately add support for an advanced search feature that uses PACKCYpher expressions. Complex (flood) searches become easily expressible. For example, the expression:

```
-[:neighbor]->*0..4-[[:book]]->X?(X.author == ‘H.G. wells’)
```

returns all the books that are written by ‘H.G. wells’ and are in a distance of up to four network hops.

**P2P Social networks** are social networks that distribute their social graph, usually for reasons of privacy and anti-censorship. Such social networks gained much interest in recent years [13][17].
GRAPHPACK can enable advanced search and sharing options that align with the privacy requirements of such social networks. For example, for every client of the social network, it is possible to add a share task that enables clients to customize the items that they wish to be shared with. That share task can be called by other clients using a PACKCypher expression, such as:

```
-->*0..1-->{hobbies include 'ski'}.share({ski related news item})
```

which finds friends and friends of friends that listed 'ski' as one of their hobbies and then sends them a 'ski related news item' by sending it as a parameter to their share task.

PSNs are opportunistic networks that make use of human mobility and local forwarding in order to distribute data [20].

GRAPHPACK can be used by PSNs to concisely implement simple forwarding schemes. For example, the following code:

```java
for (i=0;i<4;i++)
-->{label=={target’s label}}*0..3-->.receiveMessage({msg})?(id=={target’s id})
```

implements a label based forwarding scheme that is presented in [26].

### 2.6 Implementation Notes

Most of the code of GRAPHPACK is written in Java. The different components are implemented in the prototype as follows:

**Communication** is based on Java RMI, which is the default Java framework for making remote procedure calls.

**Persistency** has two implementations. The first is an in-memory implementation, i.e., objects are stored in basic data structures such as Map or List, and they do not persist beyond a single execution of the prototype application. The second implementation is based on Java Data Object (JDO), a standard Java model for object persistence.

**Authorization** wraps tasks with a simple check that verifies the requesting client is included in a permitted access list.

**Task Management and Execution** are based on the quartz scheduler [6], which is a set of Java packages that provide facilities for executing dynamic tasks that can be triggered on a specified schedule, or by request.

**Graph** is mostly implemented by delegating to the communication and persistency components. The traverse operation is the exception. It includes an advanced implementation of Algorithm 3 and a parser for PACKCypher path expressions. The parser is implemented in Scala [7], which is a Java compatible language that has powerful parsing facilities.

**Extensions** are basic immutable objects that can be loaded by the GRAPHPACK service when it is initialized and then passed to tasks during their execution.

It can be noticed that the current prototype implementation has a few limitations. Most notably, it does not have any security guarantees, and it relies on a central service (Java RMI uses a central service, called
registry). In spite of those limitations, we believe that our prototype demonstrates the applicability of our service design. This is because it is apparent in many real world applications that such issues are solvable, and also because our prototype has modular design, which enables the components to be easily updated or replaced.

3 The TRUSTPACK Framework

3.1 Overview

As mentioned in the Introduction, TRUSTPACK is based on a network of services that extend GRAPHPACK’s autonomous services with support for trust graphs management and flexible TM processing. This extension is implemented by using GRAPHPACK’s generic graph management and dynamic task processing facilities. Additionally, the fact that GRAPHPACK’s services only allow their clients to manage edges that originate at their own nodes is used to ensure that clients only control their own TSs.

Unlike current TMFs, TRUSTPACK services are designed to be used by applications as well as users. That is, while currently an application that uses a TM service needs to manage its own users and to tunnel their feedback to its TM service, in TRUSTPACK, users can communicate directly, e.g., via a browser, with their own TRUSTPACK service. This enables a large set of usage scenarios, as demonstrated in subsection 2.5. Figure 4 illustrates how applications, users (e.g., through a browser) and TM services interact in current TMFs versus how they interact in TRUSTPACK.

It is important to note that TRUSTPACK does not force a universal TM solution, i.e., it does not specify how TM should be performed. Instead, TRUSTPACK only specifies a few basic TM related building blocks, which are described in subsection 3.2. This is derived from a couple of reasons: First, a minimal specification coupled with TRUSTPACK services’ ability to dynamically manage tasks increases TRUSTPACK’s flexibility. We believe that such flexibility is necessary because TM requirements can be considerably different across applications, as elaborated on in [22]. Second, our focus is on the architecture level and we
find that specifying a universal TM constructs and methodology is an orthogonal problem that is outside the scope of this work. In particular, while there were a few attempts to define a universal TM [24, 30], currently, there is no such solution that is being used in practice.

### 3.2 TM Building Blocks

**TRUSTPACK** services have an extension library that provides executing tasks with a few basic building blocks. Those building blocks include a few data structures that correspond to the following basic TM concepts:

**Entity** represents a node in the trust graph, i.e., it represents trustors and trustees.

**Evaluation** represents trust stripped from any contextual information. For example, an evaluation can specify that something is “good”, but does not specify what is that something, who made that claim, when it was said, etc.

**TS** represents a record of trust assignment. It contains an evaluation and contextual information. The contextual information includes the identities of the trustor and trustee entities, and may also include additional contextual information.

**TRUSTPACK** allows additional data structures to be defined by extending the basic data structures mentioned above. [Table 3] shows a few examples of such extensions.

<table>
<thead>
<tr>
<th>Base</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>User</td>
<td>A user of some application</td>
</tr>
<tr>
<td></td>
<td>Restaurant</td>
<td>A restaurant, this type of entity may only be a target of trust</td>
</tr>
<tr>
<td></td>
<td>TS</td>
<td>In some applications, TSs themselves may be the targets of other TSs</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Binary</td>
<td>Contains two values: “good”, which represents a positive experience, and “bad”, which represents a negative experience</td>
</tr>
<tr>
<td></td>
<td>Free text</td>
<td>Contains plain text with no structure. The actual sentiment of the evaluation may be apparent only to human readers</td>
</tr>
<tr>
<td>TS</td>
<td>Recommendation</td>
<td>Indicates a positive sentiment towards some entity</td>
</tr>
<tr>
<td></td>
<td>Credibility</td>
<td>Indicates the level of trust that should be given to some other entity’s TSs</td>
</tr>
<tr>
<td></td>
<td>Karma</td>
<td>Indicates the amount of contribution that was made by some entity. A contribution by itself may be interpreted differently between applications. For example, a file sharing application may consider the amount of uploads made by some user as a measurement of contribution while a wiki site may consider the number of edits as a valid measurement</td>
</tr>
</tbody>
</table>

Table 3: Extensions of basic data structures

In addition to the basic data structure, the **TRUSTPACK** extension library also includes a few common operators that can process TSs or evaluations:

**Map** transforms an object (i.e., maps the object to some value). It can also be applied to a set, in which case it will transform each member of the set.
**Aggregate** transforms a set of objects into a single value.

**Group** divides a set of objects into a few non-overlapping groups.

As with the data structures we have mentioned, those operators are expected to be extended. Table 4 lists a few examples of such extensions.

<table>
<thead>
<tr>
<th>Base</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>linear normalization</td>
<td>Calculates a normalized value given some minimum and maximum bounds i.e., [normalizedValue \leftarrow \frac{originalValue - minimum}{maximum}]</td>
</tr>
<tr>
<td>aggregate</td>
<td>sum</td>
<td>Sums up all the values in some set given some function that adds two values together</td>
</tr>
<tr>
<td>group</td>
<td>filter</td>
<td>Given some predicate, splits a set into two sets: a set of objects that have passed the predicate and a set of objects that have not passed the predicate</td>
</tr>
</tbody>
</table>

Table 4: Extensions of common operators

### 3.3 Usage Examples

Below, we present a few usage examples. In particular, they demonstrate how TRUSTPACK’s design contributes to interoperability and user privacy. It is important to note that since we are mainly interested in how TRUSTPACK is used and not in the TM itself, the TM algorithms presented in these examples are purposely simple.

#### 3.3.1 Example 1: A Privacy Preserving Employment Guide

Consider a site that helps users search for employers and rate them. In this site, users can give each employer a rating between 0 (a very bad employer) and 5 (a very good employer) as well as assign other users with a credibility score that ranges from 0 (a completely un-credible rater) to 10 (a highly credible rater).

Because users’ employment record and their opinions on their employers is considered sensitive information, the site does not directly handle the employers’ ratings and users’ credibility scores. Instead, those are handled distributively by the users’ TRUSTPACK services. In addition, when a user queries about an employer, the site requests that in order to get a personalized aggregated rating of that employer the user will execute a task in its TRUSTPACK service. This task receives an employer as a parameter and searches the graph projected from the credibility scores for users who have rated that employer. Then the task returns an average of the ratings found, weighted by the minimal credibility score found in the path that led to each rater. The pseudocode for this task is presented in Algorithm 4. We note that in this scenario, it is also possible to use more advanced methods (e.g., TidalTrust [23] or advogato [33]).

**Algorithm 4** personalizedAggregatedRatingTask(emp)

1. \( results \leftarrow traverse(’(-[:credi\text{\textunderscore}bility]-*0..3)=p-[r:rating]\rightarrow\{\text{emp}\})’)
2. \( \text{return } \sum_{(p,r)\in results} MIN(p)r \sum_{(p,r)\in results} MIN(p) \) \( //MIN(p)\) is the minimal credibility score in the path that led to \( r\)
3.3.2 Example 2: Trust Sharing Among Online Stores

In this example, we consider small scale online stores. Such stores may not have a long transaction history. This can make them vulnerable to bad client behavior, such as clients with bad credit or clients that regularly return products.

To solve this vulnerability, those sites can use TRUSTPACK services in the following manner: Every time a store makes a transaction, it eventually decides if the transaction was successful or not and notifies its TRUSTPACK service, which maintains a TS that records the sum of successful and the sum of unsuccessful transactions. When facing a new client, the store will query partner stores about that client and will aggregate the results to find the total sum of successful and the total sum of unsuccessful transactions. Those values will then be used by the store to determine if a client is trustworthy. We note that more advanced methods, such as the beta reputation system [16], may also be used in this scenario.

Additionally, to account for the fact that different stores may not have an incentive to collaborate, a simple Tit-for-tat [15] reciprocity mechanism is added: Every pair of stores, $i$ and $j$, keep a reciprocity score towards each other, where the reciprocity score of $i$ towards $j$ is the amount of queries $j$ answered $i$ subtracted by the amount of queries $i$ answered $j$. When $i$’s reciprocity score towards $j$ is below some threshold, $i$ will discontinue answering $j$’s queries.

Algorithms 5 and 6 present pseudocode of the tasks that are performed when a store queries about some client and when a store responds to such a query respectively.

Algorithm 5 \texttt{queryAboutClient(client)}

1: $results \leftarrow \text{traverse}('\text{-[:partner]}->\text{store}-[h:History]->\{\text{client}\}')$
2: \textbf{for} ($\text{store,}_\ldots \in results$ \textbf{do})
3: \hspace{1em} $reciprocity[\text{store}] \leftarrow reciprocity[\text{store}] + 1$
4: \textbf{end for}
5: \textbf{return} \hspace{2em} (\sum_{(\text{h})\in results} h.successful, \sum_{(\text{h})\in results} h.unSuccessful)

Algorithm 6 \texttt{queryRespond(requestingStore,client)}

1: \textbf{if} \hspace{1em} $reciprocity[\text{requestingStore}] \leq \text{lowThreshold}$ \textbf{then}
2: \hspace{1em} \textbf{return} $\emptyset$ //no results
3: \textbf{else}
4: \hspace{1em} $res \leftarrow \text{getEdge}('\text{-[h:History]}->\{\text{client}\}')$
5: \hspace{1em} \textbf{if} \hspace{1em} $res \neq \emptyset$ \textbf{then}
6: \hspace{2em} $reciprocity[\text{requestingStore}] \leftarrow reciprocity[\text{requestingStore}] - 1$
7: \hspace{1em} \textbf{end if}
8: \hspace{1em} \textbf{return} $res$
9: \textbf{end if}

3.3.3 Example 3: Fairness in a P2P File Sharing Network

Here, we describe a P2P file sharing network similar to BitTorrent [15]. In this network, peers only benefit from downloading files and not from uploading files. In order to add incentives for file uploading, we first require that every peer keeps track of the amount of bytes other peers owe it, i.e., it records for each peer, the amount of bytes that were sent to that peer subtracted by the amount of bytes that were received from that peer.
We use these records similarly to the way they are used in FairTorrent [39], where each peer uploads only to peers that owe the lowest amount of bytes among all the peers it interacted with. While FairTorrent presents a reasonable approach, it does have some problems, most notably, peers only rely on their local records, which means that peers do not have incentives to upload to peers that cannot reciprocate them directly. This can happen, for example, between two peers that only one of them has a file that the other wants. To avoid this problem, every peer is associated with a TRUSTPACK entity that records in TSs what other peers owe it. The TRUSTPACK services are then loaded with tasks that run a distributed algorithm that aggregates those TSs into a global reputation (e.g., EigenTrust [29]). Peers then upload only to peers with the highest global reputation values.

3.3.4 Example 4: Real Person Validation

Validating if a given entity represents a real person is important in many applications, including reputation systems and social networks. In those applications, fake identities can hurt the stability and quality of the application [21].

In order to implement a real person validation service using TRUSTPACK, we can actually use the previous examples: First, Examples 1 and 3 are changed so their services add a TS indicating that a person is real if it frequently uses the system and has a minimal history length. Second, Example 2 (subsubsection 3.3.2) is changed to add a TS indicating that a person is real after it received some minimal amount of payment from a transaction with that person.

After those changes are implemented, any application that requires real person validation can execute a task in their TRUSTPACK service that will search for real person TSs made by other pre-known entities (such as the entities from Examples 1-3).

4 Case Study

In this section, we describe our experience from incorporating TRUSTPACK in a real P2P file sharing application, called Alliance [1]. In Alliance, each peer joins a private network together with a group of other peers. The members of that private network, called friends, can share with each other files from their personal computers.

In our enhanced implementation of Alliance, each peer uses a TRUSTPACK service to report the amount of bytes it received from and sent to other friends. A TRUSTPACK task then calculates each friend’s relative contribution and presents the results graphically. Additionally, in order to demonstrate TRUSTPACK’s flexibility, we use several different tasks for estimating the relative contribution.

4.1 Implementation

Our code is based on Alliance version 1.2. Our first change in the code initializes a local TRUSTPACK service during the Alliance start-up process. This initialization includes opening the service for incoming messages, adding outgoing edges that represent the peer’s friends and installing a custom task that will be used for periodically calculating a normalized trust value. A second change is that Alliance peers now report to their TRUSTPACK service the amount of bytes received from or sent to any individual friend. A third and last change includes processing the information gathered by TRUSTPACK and displaying the results using Alliance’s Graphic User Interface (GUI).
The code includes three trust related operations: The first, aggregates the reported amount of uploaded bytes subtracted by the reported amount of downloaded bytes. The second, periodically produces a normalized score for each friend, based on local experience. The third, calculates an average of the reported scores weighted by the score of each reporter.

Below are a few figures that show the changes that we have made: Figure 5 shows the code added to the Alliance program in order to initialize the TRUSTPACK service and update it when new information is gathered. Figure 6 shows the code of the three trust related operations that we have described, and in Figure 7 there is a screen-shot of the modified GUI, which displays the new trust metrics.

4.2 Evaluation

Configuring Alliance to work with TRUSTPACK was a fairly simple task that consisted mostly of finding the appropriate entry points in the Alliance code. In addition, we have checked that the new enhanced Alliance works well. In order to do so, we have created a test network with 18 peers, established data by downloading several files between the peers, and then validated that the trust values are calculated and displayed properly in the GUI. During this check, the application worked correctly and without any issues.

Lastly, we conclude that the TRUSTPACK prototype has been successfully incorporated into this P2P file sharing application. We have demonstrated that it requires a relatively simple configuration, that tasks can flexibly encapsulate trust related processes, and that its architecture is generally applicable.
public static class Normalize implements ITask {
    @Override
    public void execute(INode node, Map<String, Object> ExtendedNodeMap, Object... params) {

        // get the TrustPack extension
        List<Cons<Balance,Normalized>> evals = ((Entity)ExtendedNodeMap.get("TrustPack")).getEvaluations();

        // sum all positive balance
        final Double sum = AggregateOperator.Sum.aggz(evals,
                new MapFunc<Cons<Balance,Normalized>,Long>(){
                    @Override
                    public Long evaluate(Cons<Balance,Normalized> o) {
                        return Math.max(0, o._1.balance);
                    }
                });

        // normalize each positive balance by dividing it by the previous sum
        for (Cons<Balance,Normalized> eval : evals){
            eval._2.val = ((double)Math.max(0, eval._1.balance))/sum;
        }
    } //

    /**
     * sums balance reports received from friends about a specific peer */
    public long totalBalanceReports(String friendName){
        List<Cons<Balance,Normalized>> evals = self.getEvaluations("-\-[e]\-","friendName");
        final Double total = AggregateOperator.Sum.aggz(evals,
                new MapFunc<Cons<Balance,Normalized>,Long>(){
                    @Override
                    public Long evaluate(Cons<Balance,Normalized> o) {
                        return o._1.balance;
                    }
                });

        return total.longValue();
    }

    /**
     * calculate a weighted normalized value of received reports about some peer P:
     * for each reporter R: ReportR := <R Score> * <P's Score according to R> 
     * P's weighted normalized score := sum over all ReportR */
    public double WeightedReports(String friendName){

        ResultSet r = self.traverse("-[e1]\-\-[e2]\-","friendName");
        double total = 0.0;
        for (ResultSet r : r.elementSet()){
            double normalized1 = ((Cons<Balance,Normalized>r.get("e1"))._2.val;
            double normalized2 = ((Cons<Balance,Normalized>r.get("e2"))._2.val;
            total += normalized1*normalized2;
        }
        return total;
    }
}
Figure 7: A screen-shot of Alliance with the TRUSTPACK extension

5 Related Work

5.1 GRAPHPACK Related Work

The GRAPHPACK framework closely relates to graph management systems and P2P data management systems. Although there are many examples of both types of systems, we only found a couple of examples of systems that are similar to GRAPHPACK. This is because graph management systems do not generally enable distribution of the graph that they manage, and most of the graph systems that do enable distribution, require a central service for coordination (e.g., Neo4j [5], FlockDB [3], and Apache Hama [2]). Additionally, P2P data management systems usually deal with raw data or relational data, but not with graph structures. The two systems that we found to be similar to GRAPHPACK are HyperGraphDB [27] and Plasma [38].

HyperGraphDB is a graph database that can run as a service, which can send and receive messages. The structure of those messages and the methods that handle their delivery are designed for customization. This enables implementation of protocols that can share data between the services. But, unlike GRAPHPACK, HyperGraphDB’s nodes cannot refer to other services, clients cannot specify a traverse that can be transparently distributed over the unified graph and it does not separate between clients and services (i.e., every
service manages a single graph).

Similarly to HyperGraphDB, the Plasma framework enables independent services to maintain their own graph and communicate with each other. In addition, it allows nodes in the graph to refer to other services and it has a query language that can be used to execute transparently distributed queries. While Plasma and GRAPHPACK have a similar general design, they do have a few differences: First, Plasma only supports queries (i.e., data retrieval) while GRAPHPACK supports dynamic task management and the execution of complex traverses that can use those tasks. Second, like with HyperGraphDB, in Plasma every service corresponds to a single graph.

5.2 TRUSTPACK Related Work

Distributed access control (i.e., distributed authentication and authorization) can be viewed as a specialized type of distributed TM. Those systems rely on public key cryptography [19] and a Public Key Infrastructure (PKI), which is a general term for mechanisms that are used for distributing public keys and verifying their validity. Two commonly used PKIs are X.509 and Pretty Good Privacy (PGP). X.509 [41] relies on Certification Authorities (CAs), which can issue and revoke certificates [31] that are used for authentication and for certifying other CAs. An identity is considered authenticated in X.509 if there is a path of certificates from that subject to a CA that is trusted a-priori. PGP [45] does not rely on CAs like x.509. Instead, in PGP users personally validate and manage public keys based on their own experience. In addition, users can vouch for the authenticity of public keys and specify how to treat vouches made by other users.

PolicyMaker [10, 11] is an access control system that introduces a type of programmable credentials called assertions. An assertion is basically a statement that some entity trusts another entity if certain conditions apply. Those conditions are specified by a Boolean function that is attached to the assertion. A set of assertions that is available to an application is called a policy. Upon receiving some request, PolicyMaker will search local policies for a path of satisfied assertions (i.e., assertions that return ‘true’ in the context of the current request) from a trusted root to the requesting entity. REFEREE [14] is a similar system that allows policies to include references to remote policies and assertions. This enables REFEREE to recursively download and use additional assertions and policies during its request evaluation process.

The Fidelis system [43, 44] represents an important step towards general purpose TM. In Fidelis, a credential may include relevant properties and not just the identities of the issuer and the subject. For example, a movie recommendation issued by some user may include properties such as a rating given by that user and the time and location in which the movie was first viewed by that user. The fact that credentials now include properties allows for the specification of complex policies using a DSL called the Fidelis Policy Language. For example, a policy may specify that a movie is recommended only if two different persons recommended the same movie and gave it a rating higher than four stars. Another important feature of Fidelis is that it can run as an autonomous service and that it allows for credentials to be pushed or pulled from one such service to another.

The SULTAN [24] and uTrust projects [30] separately try to standardize trust related processes. Both specify how trust, credibility, recommendation, risk and experience should be represented, measured, stored and processed. Yet, while in SULTAN this specification includes specific trust related algorithms, administrator tools, a DSL for specifying trust, and the layout of the servers and databases that should be used, uTrust does not offer much details beyond the conceptual level.

The Yahoo! Reputation Platform (YRP) [22] is perhaps the only documented example of a working large scale TMF. The YRP is a centralized service used by Yahoo! online applications to gather and process millions of implicit and explicit trust evaluations per day. In YRP every application loads its own event based trust model into the service. This trust model consists of code blocks that can be triggered by external
events (e.g., a new trust evaluation made by some user), read previous reputation values, make some simple calculations, update reputation values, and trigger new events. There are two design choices made in YRP that are particularly interesting: First, the code blocks used by the trust models are not pre-built and are written in a general purpose language, as opposed to a trust specific scripting language. In their words, this decision was made because “every context is different, so every model is also significantly different”. Second, queries do not trigger complex processes, but only simple data retrieval. This specifically means that since YRP needs to handle about half a billion registered users, it cannot support personalized values, such as ‘your friends rated this entity 4.5’. This is because such queries will require it to pre-calculate and store an unfeasible amount of aggregations that are likely to be redundant. Yahoo!’s engineers made this choice because of responsiveness requirements. That is, if they would have allowed queries to trigger complex processes, then the service simply would not reasonably handle the large amount of requests coming from the applications that use it.

Summary  TRUSTPACK is considerably different than the systems and mechanisms surveyed here in both its design and goals. Most notably, TRUSTPACK is the only framework that allows for applications, as well as users, to dynamically choose and control autonomous TM services that can easily share data between themselves. This enables for increased privacy and personalization, and may provide a solution to the scalability problems encountered in large TMFs such as YRP.

6 Conclusions

This paper has presented two frameworks, GRAPHPACK and TRUSTPACK. GRAPHPACK provides an architecture for the management of decentralized graphs. Its novelty lies mainly in its support for transparently distributed graph traverses that can be concisely specified using a new DSL, called PACKCYPHER. The second framework, TRUSTPACK, extends GRAPHPACK’s functionality with support for TM data structures and operators. With GRAPHPACK at its core, TRUSTPACK features a new kind of TMF architecture that lets clients control their own TSs. This architecture helps to deal with privacy, personalization and scalability concerns encountered in other TMFs.

Our code is open source. The code and full documentation of GRAPHPACK and TRUSTPACK are located at [http://code.google.com/p/graphpack/](http://code.google.com/p/graphpack/) and [http://code.google.com/p/trustpack/](http://code.google.com/p/trustpack/), respectively. Both frameworks feature a modular design that enables customization and extension. Those prototypes along with the use case examples we have presented validate the applicability of our design and provide references for future implementations.

6.1 Future Work

6.1.1 The Graph Library

Currently, GRAPHPACK’s graph library only has a built-in support for Depth First Search (DFS) traverses along outgoing edges. Using incoming edges, basic search strategies, such as Breath First Search (BFS) and IDDFS, as well as more complex search strategies, such as random walk, simulated annealing [32] and global search, may also be investigated. Additionally, it may be useful to provide built-in support for queries that require sorted or top-k results.

Another issue is that currently, successful traverses are being backtracked along the paths in which they were found, and this backtracking only assembles the results. It may be interesting to allow for additional
predicate checks, task executions and result capturing, to be performed during those backtracks. Additionally, we may avoid backtracking entirely by using a tail-recursive traverse implementation. We note that such an implementation will have to sacrifice some privacy, since it requires that all the results will be sent to the initial node from the nodes at the end of the traverse.

Current databases provide many additional ideas for capabilities that can be incorporated in our framework. Most notably, nodes and edges may be attached with a schema that specifies their structure and allows for safe typing. The concept of views can apply to edges that are dynamically created upon request. Triggers can be used to automate actions that should be taken upon certain changes in the graph (e.g., new edge creation). Lastly, the use of transactions may also be investigated.

6.1.2 The TM Library

Our current TM library is a Proof-of-Concept, and a real TM library is yet to be developed. Such a library should include many trust related algorithms, configurable tasks, data structures and operators. It should support measuring of real world usage patterns in order to eventually standardize common trust related processes.

Additionally, it should provide tools for trust alignment, which is a process that tries to mediate between two parties that wish to share trust related data but use different evaluation types or have a different interpretation of the same evaluation types. For example, two movie rating sites may want to share rating data between them, but one site manages a five star rating system and the other manages a rating system that uses real values from one to ten. A trivial trust alignment in this case is to perform a simple linear transformation from one rating system to the other.

6.1.3 Management

In subsection 2.2, we have described how our services let clients directly manage dynamic tasks, but we should also consider a more automatic form of task management. That is, the services should include configurable protocols for distributing, updating and revoking tasks. It is interesting to note that such protocols may themselves use TM in order to verify the legitimacy of task related requests.

Additionally, services should provide administrative tools for managing software updates, extensions, client management, monitoring and load balancing, as well as provide clients with tools for managing their own data, tasks and authorizations. We note that, as with task management, it is possible that such tools will use TM themselves. For example, a load balancing tool may measure the number of requests made by some client and report that information to other services.

6.1.4 Performance and Security

Our prototype implementation lacks basic performance and security constructs. For performance, at least a trivial implementation of caching, compression, pre-fetching and buffering techniques should be added to both the communication and persistency components. For security, we need to first enable authentication of clients, services, TSs and tasks. Second, enable encrypted communication. Third, ensure a secure execution of tasks. Fourth, enable for detection, reporting and sanctioning of malicious behavior (e.g., a task that tries to hack into the services).
6.1.5 Privacy

Privacy issues should also be further investigated. The fact that we feature a decentralized environment contributes greatly to our ability to maintain basic privacy, but if strict privacy is required then additional measures should be taken. Communicating parties can be hidden using layers of indirection, identities can be hidden using pseudonyms, and queries and results can be obfuscated in order to minimize the amount of information that is revealed.

References


[38] Jeff Rose and Antonio Carzaniga. Plasma: a graph based distributed computing model.


A Interface Specification

A.1 Communication

A.1.1 Service Scope

Create Client parameters: clientName
create a new client account in the service
A.1.2 Client Scope

Create Node parameters: nodeName
create a new graph node for this client

Connect parameters: targetService
connect to another service using this client credentials

A.1.3 Node Scope

Add Outgoing Edge parameters: target, payload
add a new outgoing edge to this node

Get Outgoing Edges no parameters
get this node’s outgoing edges

Traverse parameters: pathMatcher
start a traverse at this node (see subsection 2.4)

Add Task parameters: taskName, task
associate a new task with this node

Add Scheduled Task parameters: taskName, task, cronExpression
add a new task and execute it according to a schedule specified by a cron expression²

Call Task parameters: taskName, taskParameters
call a task that is associated with this node using given task parameters

A.2 Persistency

A.2.1 Service Scope

Store Client parameters: clientName, clientRef
store a client

Contains parameters: clientName
check if a specific client is stored by this service

Get Client parameters: clientName
retrieve a previously stored client

A.2.2 Client Scope

Store Node parameters: nodeName, nodeRef
store a new outgoing edge

Contains parameters: nodeName
check if a specific node is stored by this service

Get Node parameters: nodeName
retrieve a previously stored node

²http://en.wikipedia.org/wiki/Cron#CRON_expression
A.2.3 Node Scope

**Store Outgoing Edge** parameters: target, payload
store a new outgoing edge that originates at this node

**Get Outgoing Edge** no parameters
retrieve this node’s outgoing edges

B PACKCypher Language Specification

In this appendix, we present the complete PACKCypher syntax specification in Backus-Naur Form (BNF). Table 5 presents the BNF for the basic primitives used in PACKCypher. Table 6 specifies how a predicate can be constructed from those primitives. Finally, Table 7 specifies PACKCypher’s path expressions.

---

<table>
<thead>
<tr>
<th>&lt;value&gt; ::=</th>
<th>&lt;entity&gt;</th>
<th>&lt;string&gt;</th>
<th>&lt;number&gt;</th>
<th>'true'</th>
<th>'false'</th>
<th>'null'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;entity&gt; ::=</td>
<td>&lt;identifier&gt;</td>
<td>&lt;entity&gt; '.' &lt;entity&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>specified by Java syntax1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| <string>, <number>, <whole-number> and <identifier> |

---

| <predicate> ::= | <or> |
|                | <and> | <or> '||' <and> |
| <or> ::= | <equality> | <and> '&&' <equality> |
| <and> ::= | <equality> '==' <equality> | <equality> '!=' <equality> |
| <equality> ::= | <additive> | <multiplicative> |
| <relational> ::= | <multiplicative> | <unary> |
| <additive> ::= | <multiplicative> | <value> | '!' <unary> |

---

Table 5: PACKCypher BNF: basic primitives

Table 6: PACKCypher BNF: predicates
<table>
<thead>
<tr>
<th>path ::=</th>
<th>edge</th>
<th>path edge</th>
<th>'(' path ')'</th>
<th>'?(' &lt;predicate&gt; ')'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'*' [whole-number] '..' [whole-number]</td>
</tr>
</tbody>
</table>

Table 7: PACKCypher BNF: paths