TrustPack: a Decentralized Trust Management Framework

Amit Portnoy

TrustPack: a Decentralized Trust Management Framework

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Amit Portnoy

Submitted to the Senate of
the Technion — Israel Institute of Technology
Sivan 5772 Haifa May 2012

The research thesis was done under the supervision of Prof. Roy Friedman in the Computer Science Department.

I would like to thank my supervisor, Prof. Roy Friedman for his excellent guidance during my studies. I could not have done this work without his constructive comments, encouragement, and attention to detail.

I would also like to thank my parents, Shela and Yekutiel Portnoy for their love and support throughout the years.

The generous financial support of the Technion and the Israel Science Fund (ISF grant 1247/09) is gratefully acknowledged.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

With the rise of the internet and its applications, it has become more and more common for unfamiliar parties to interact with each other. In order to help and encourage such interactions *trust management* systems were introduced. Those systems try to alleviate mistrust by gathering and processing statistical information about previous events.

This thesis describes TrustPack, a trust management framework that provides trust management as a service, i.e., TrustPack is separated from any specific application usage. TrustPack is unique in that it does not provide a central service. Instead, it is run by many autonomous services. This design enables it to alleviate privacy concerns, as well as potentially provide better personalization and scalability when compared with current centralized solutions.

During the development of TrustPack, we found that providing easy access to trust related information was an especially interesting challenge. Such access is not trivial because, by design, TrustPack services do not necessarily have access to all relevant information.

To enable such access we have created GraphPack, a generic framework that deals with management and processing of graphs that are distributed by a network of autonomous services. We note that TrustPack is able to use GraphPack because trust related information can be trivially represented by a graph. In addition, we note that GraphPack was built as a separate framework because 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.

Prototypes for TrustPack and GraphPack were implemented as part of this work. Their source code and full documentation are located at `http://code.google.com/p/trustpack/` and `http://code.google.com/p/graphpack/` correspondingly.

List of Acronyms

BFS	Breath First Search
BNF	Backus-Naur Form
DFA	Deterministic Finite Automaton
DFS	Depth First Search
DSL	Domain Specific Language
GUI	Graphic User Interface
JDO	Java Data Object
P2P	Peer-to-Peer
PGP	Pretty Good Privacy
PKI	Public key Infrastructure
PSN	Pocket Switched Network
TM	Trust Management
TMF	Trust Management Framework
TS	Trust Statement
YRP	Yahoo Reputation Platform

1. Introduction

1.1. Trust and Trust Management

Trust Management (TM) is a facility that helps with decision making in the presence of uncertainty [35]. TM became extremely common with the proliferation of the internet and its applications [1, 21]. It is used by a variety of applications, such as: Peer-to-Peer (P2P) networks (e.g., BitTorrent [8], Tribler [30]), e-commerce sites (e.g., eBay[1], Amazon[2]), web-based social communities (e.g., CouchSurfing[3], Facebook[4]), sites with a collaborative content management (e.g., Wikipedia[5], YouTube[6], Yelp[7]), etc.

To explain what TM is and how it works, we must first introduce a few relevant concepts. First of all, 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 in 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 was 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.

[1]http://www.ebay.com
[2]http://www.amazon.com
[3]http://www.couchsurfing.com
[4]http://www.facebook.com
[5]http://www.wikipedia.com
[6]http://www.youtube.com
[7]http://www.yelp.com

1.2. Trust Management Frameworks

Currently TM is usually developed together with the application that uses it. Yet, recently it has been noticed [15, 17, 23] that the concept of a trust graph is generic and the methods for its processing are common between many applications. This enables for 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.

While current TMFs are a step forward, and they possess many of the advantages that we have mentioned above, we think they also have a few crucial problems. Those TMFs manage a global trust graph, which is the union of the trust graphs of all the applications that use it. They also provide a central *TM service*, which is a service that enables applications to execute trust related tasks with 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 not a reasonable assumption since some applications and users might use (and trust) different services.

1.3. Our Contributions

In this work, we present a prototype for 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 found that providing easy access to the trust graph was 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 some 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 [6] and Safebook [10]). 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 facilities that we have mentioned in the previous paragraph and to increase TrustPack's flexibility, we provide a second 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.

1.4. Thesis Structure

In Chapter 2, we present GraphPack and demonstrate how it can be used. In Chapter 3, we present TrustPack and show how it can alleviate the concerns that we have mentioned while still providing usable TM for a variety of applications. In Chapter 4, we presenet a case study of an application that uses TrustPack. Finally, in Chapter 5, we present future work and closing remarks.

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 2.1 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.

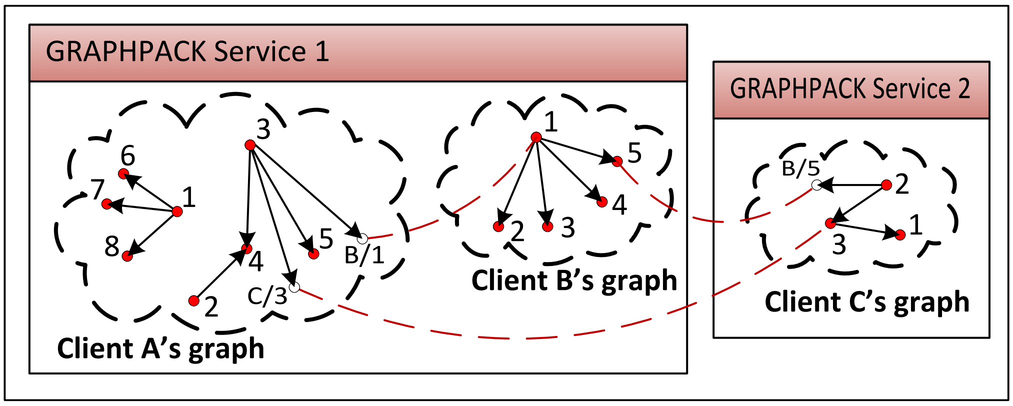


Figure 2.1.: An example of GraphPack services with a few graphs

We note that when considering the unification of all the clients' graphs, GraphPack may be viewed as a type of decentralized graph database that manages a single *property graph*, which is a directed graph where unstructured data (i.e., properties) can be attached to edges. Property graphs are commonly used at the core of many graph databases (this is demonstrated by Blueprints[1], which is a project that uses the property graph model in order to provide a common interface for graph databases). In GraphPack, we use property graphs because they are very generic. 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 by a property graph as follows:

[1]https://github.com/tinkerpop/blueprints/wiki/

Node properties can be modeled with a special edge that stores the node's properties and is directed from that node to itself. Undirected edges can be modeled by two opposing directed edges.

The next sections are ordered as follows: In Section 2.2, we present GraphPack's architecture and components. In Section 2.3, we introduce a new language for graph traverses. In Section 2.4, we explain how traverses are being executed. In Section 2.5, we present a few GraphPack usage examples. In Section 2.6, we discuss our GraphPack implementation. Lastly, in Section 2.7, we discuss a few works that are related to our GraphPack framework.

2.2. GraphPack Service Architecture

The components of a GraphPack service (Figure 2.2) can be divided into three groups: *infrastructure*, *processing* and *libraries*.

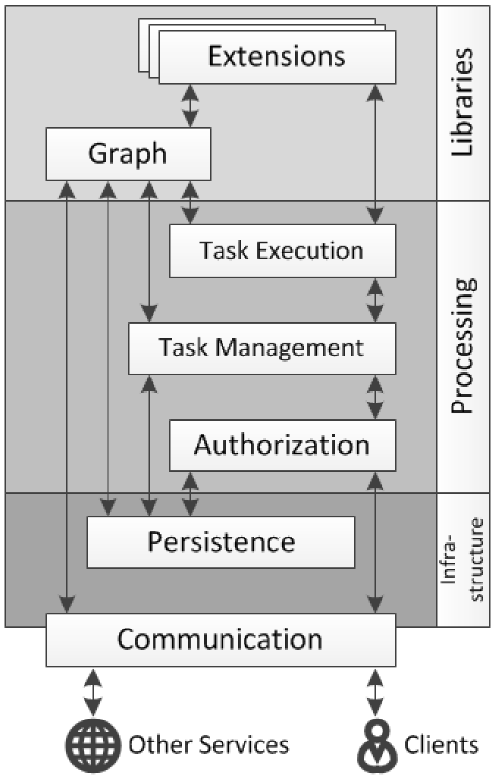


Figure 2.2.: GraphPack service architecture

In the *infrastructure* group there are components that we consider as common facilities required by the GraphPack service. This group includes the *communication* and *persistence* 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 Section A.1). Lastly, this component also includes parts that can be used by clients when communicating with services.

Persistence provides a facility for simple storing and retrieving of objects (for the complete specification, see Section 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 *authorization*, *task management* and *task execution* 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 persistence component for the persistence 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 persistence component for the persistence of tasks and their configurations.

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 *graph* and *extensions* components:

Graph provides graph related operations that are detailed in Table 2.1. 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 Section 2.3. The traverse process is discussed in Section 2.4.

Extensions are a set of optional libraries that provide executing tasks with extra data structures and utilities. One such extension is the TM extension that is discussed in Chapter 3.

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

Table 2.1.: Graph operations

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 developed a graph traverse language, nicknamed PackCypher. PackCypher is inspired by Cypher, an emerging language that is part of the Neo4j graph database[2]. In this section we provide an overview of the PackCypher language, for the full syntax specification see Appendix B.

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

[2]http://neo4j.org/

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 the help 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*[3]. 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 edge. For example, "`-[X]->`" will capture as 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 2.3 demonstrates a few path expressions that use capturing groups. Those expressions are applied on node 'a' in Figure 2.3.

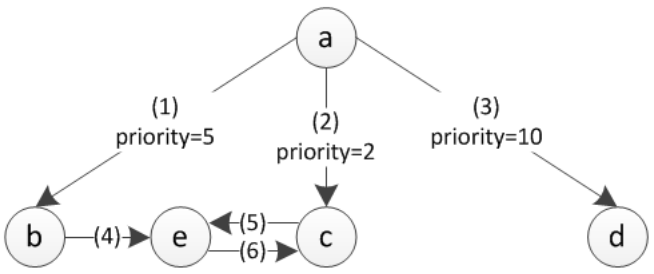


Figure 2.3.: An example graph

PackCypher also provides a method for specifying predicates that should be checked during the traverse. Predicates are specified by appending an edge or a

[3]http://www.regular-expressions.info/named.html

PackCypher expression	Interpretation	Result
`-->X`	capture as X all nodes that are the target of some outgoing edge	*X={b,c,d}*
`-[X]->`	capture as X all outgoing edges	*X={1,2,3}*
`(-->-->)=X`	capture as X all paths with two edges	*X={-[1]->b-[4]->e, -[2]->c-[5]->e}*
`-[X]->Y`	capture as X all outgoing edges capture as Y all their target nodes	*(X,Y)={(1,b),(2,c),(3,d)}*
`-->X-->Y-->X`	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	*(X,Y)={(c,e)}*
`(-->X-->)=Y`	capture as Y all paths with two edges capture as X the nodes that are at the middle of those paths	*(X,Y)={(c,-[2]->c-[5]-e), (b,-[1]->b-[4]-e)}*

Table 2.2.: Capturing groups examples

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 2.3, this path expression will capture *X={1,3}*.

An additional important feature of PackCypher 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 PackCypher:

- 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 PackCypher expression with an *environment*, which is a mapping from variable names to objects. Those objects can then be referenced in the PackCypher 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 2.1, which is a common traverse algorithm used in graph databases. It uses an object called *processor,* which is derived from the PackCypher 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 2.1 *traverse*(*processor, path*)

```
1: $results \leftarrow \varnothing$
2: for all $e \in path.lastNode.readEdges()$ do
3:     $newPath \leftarrow$ path appended with $e$
4:     $status \leftarrow processor.process(newPath)$
5:     $results \leftarrow results \cup status.getResults()$
6:     if $status.canContinue()$ then
7:         $results \leftarrow results \cup traverse(processor,\ newPath)$
8:     end if
9: end for
10: return $results$
```

Notice that Algorithm 2.1 requests the *readEdges* operation from every node along all matching paths. This means that all the edges that are encoutered 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 2.2. Algorithm 2.2 calls the *readEdges* operation locally and *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 2.2 and Algorithm 2.1 is that the processor object used in Algorithm 2.2 is more advanced than the one used in Algorithm 2.1. 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 [2, 28]. 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 2.2 *traverse*(*processor, state*)

```
1: $results \leftarrow \varnothing$
2: for all $e \in readEdges()$ do
3:     $(status,\ newState) \leftarrow processor.process(state,\ e)$
4:     $results \leftarrow results \cup status.getResults()$
5:     if $status.canContinue()$ then
6:         $results \leftarrow results \cup e.getTarget().traverse(processor,\ newState)$
7:     end if
8: end for
9: return $results$
```

Algorithm 2.3, which represents what is currently implemented on GraphPack, further improves Algorithm 2.2. The processor object it uses does not process entire paths. Instead, it *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, and as indicated in [29], even the initial processor is likely to be smaller in size than a state based processor, like the ones used in Algorithm 2.2. 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* [5] (rediscovered in [29]). 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 2.3 introduces a few new operations: ∂_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 2.3 *traverse*(*processor*)

```
1: $results \leftarrow \varnothing$
2: for all $e \in readEdges()$ do
3:    $(status, derivedProcessor) \leftarrow \partial_e processor$
4:    $results \leftarrow results \cup status.getResults()$
5:    if $status.canContinue()$ then
6:       $newResults \leftarrow newResults \cup$
            $(status.getPartialResults() \otimes e.getTarget().traverse(derivedProcessor))$
7:    end if
8: end for
9: return $results$
```

2.5. Usage Examples

In this section, we present three simple examples that demonstrate how GraphPack may be used. Those examples correspond to three different network types: unstructured P2P network, P2P social network and Pocket Switched Network (PSN). We note that the TrustPack framework that is presented in Chapter 3, can also be viewed as a 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 Gnuettela 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 [6, 10].

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 want 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 [13].

GraphPack can be used by PSNs to concisely implement simple forwarding schemes. For example, the following code:
```
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 [19].

[4]http://en.wikipedia.org/wiki/Gnutella

2.6. Implementation Notes

In this section, we give a high level description of the GraphPack service prototype implementation. Most of the code in the prototype is written in Java[5]. It was chosen mainly because it is a widely adopted language, making it more likely that the prototype's code will be used in the future. In addition, Java has a very simple and powerful syntax, many useful packages and it works on a virtual machine that enables it to run on many operating systems and processor architectures.

The different components are implemented in the prototype as follows:

Communication is based on RMI[6], which is the default Java framework for making remote procedure calls.

Persistence has two implementations. The first is an in-memory implementation, i.e., objects are stored in basic data structures such as Map[7] or List[8], 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*[9], 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 presistency components. The traverse operation is the exception. It includes an advanced implementation of Algorithm 2.3 and a parser for PackCypher path expressions. The parser is implemented in Scala[10], 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 prototype implementation has a few crucial 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.

[5]http://en.wikipedia.org/wiki/Java_(programming_language)
[6]http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
[7]http://docs.oracle.com/javase/6/docs/api/java/util/Map.html
[8]http://docs.oracle.com/javase/6/docs/api/java/util/List.html
[9]http://quartz-scheduler.org/
[10]http://www.scala-lang.org/

2.7. 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[11], FlockDB[12], and Apache Hama[13]). Additionally, P2P data management systems usually concern with raw data or relational data, but not with graph structures. The two systems that we found to be similar to GraphPack are HyperGraphDB [20] and Plasma [31].

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 support 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.

[11]http://neo4j.org/
[12]https://github.com/twitter/flockdb
[13]http://incubator.apache.org/hama/

3. The TrustPack Framework

3.1. Overview

In this chapter, we present the details of the TrustPack framework. 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 senarios, as demonstrated in Section 2.5. Figure 3.1 illustrates how applications, users (e.g., through a browser) and TM services interact in current TMFs versus how they interact in TrustPack.

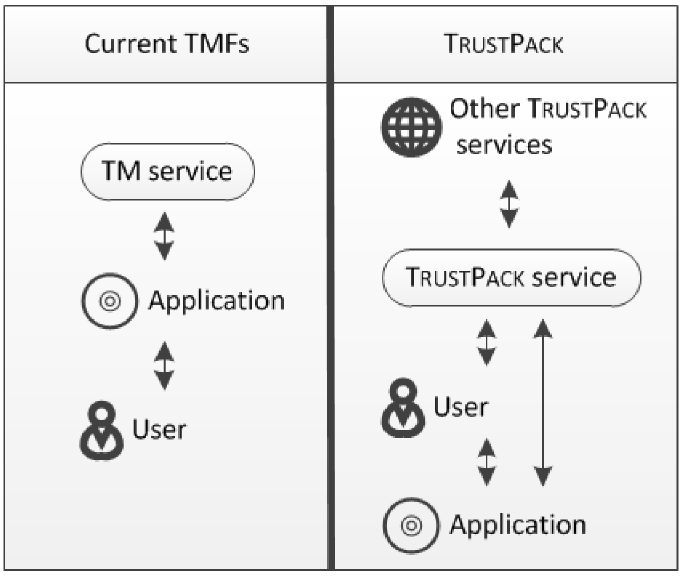


Figure 3.1.: Application's interaction in current TMFs versus in TrustPack

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 Section 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 [15]. 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 [17, 23], currently, there is no such solution that is being used in practice.

The rest of this chapter is organized as follows: Section 3.2 presents the basic TM building blocks provided by TrustPack services, section Section 2.5 demonstrates a few typical TrustPack usage scenarios, and Section 3.4 presents related work.

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 3.1 shows a few examples of such extensions.

Base	Extension	Description
Entity	User	A user of some application
	Restaurant	A restaurant, this type of entity may only be a target of trust
	TS	In some applications, TSs themselves may be the targets of other TSs
Evaluation	Binary	Contains two values: “good”, which represents a positive experience, and “bad”, which represents a negative experience
	Free text	Contains plain text with no structure. The actual sentiment of the evaluation may be apparent only to human readers
TS	Recommendation	Indicates a positive sentiment towards some entity
	Credibility	Indicates the level of trust that should be given to some other entity’s TSs
	Karma	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

Table 3.1.: 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 3.2 show a few examples of such extensions.

Base	Extension	Description
map	linear normalization	Calculates a normalized value given some minimum and maximum bounds i.e., $normalizedValue \leftarrow \frac{originalValue - minimum}{maximum}$
aggregate	sum	Sums up all the values in some set given some function that adds two values together
group	filter	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

Table 3.2.: Extensions of common operators

3.3. Usage Examples

In this section, we present examples that demonstrate how TrustPack can be used in different scenarios and how its 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

In the first example, we 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 3.1. We note that in this scenario, it is also possible to use more advanced methods (e.g., TidalTrust [16] or advogato [26]).

Algorithm 3.1 *personalizedAggregatedRatingTask*(*emp*)

```
1: $results \leftarrow traverse($‘(-[:credibility]->*0..3)=p-[r:rating]->{emp}’$)$
2: return $\frac{\sum_{(p,r)\in results} MIN(p)r}{\sum_{(p,r)\in results} MIN(p)}$ $//MIN(p)$ is the minimal creadibility score in the path that led to $r$
```

3.3.2. Example 2: Trust Sharing Among Online Stores

In the second 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 [9], 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[1] 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.

Algorithm 3.2 and Algorithm 3.3 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 3.2 *queryAboutClient*(*client*)

```
1: $results \leftarrow traverse($‘-[:partner]->store-[h:History]->{client}’$)$
2: for $(store, \_) \in results$ do
3:     $reciprocity[store] \leftarrow reciprocity[store] + 1$
4: end for
5: return $\left(\sum_{(\_,h)\in results} h.successful, \sum_{(\_,h)\in results} h.unSuccessful\right)$
```

[1]http://en.wikipedia.org/wiki/Tit_for_tat

Algorithm 3.3 *queryRespond*(*requestingStore*, *client*)

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

3.3.3. Example 3: Fairness in a P2P File Sharing Network

In this example, we describe a P2P file sharing network similar to BitTorrent [8]. 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 [32], 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 [22]). 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 [14].

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 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).

3.4. 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 [12] 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 [34] relies on *Certification Authorities* (CAs), which can issue and revoke certificates [24] 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-priory. PGP [38] 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 [3, 4] 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 [7] 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 [36, 37] 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 [17] and uTrust projects [23] 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) [15] 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 particularity 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 In this section, we reviewed selected works that show the evolution of TM services. We note that although TrustPack lacks some important capabilities that are found in those works (to be discussed in Section 5.1), it is considerably different in design and goals, and as such it presents a unique contribution. 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 them. This enables for increased privacy and personalization, and may provide a solution to the scalability problems encountered in large TMFs such as YRP.

4. Case Study

4.1. Overview

In this chapter, 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.

4.2. Implementation

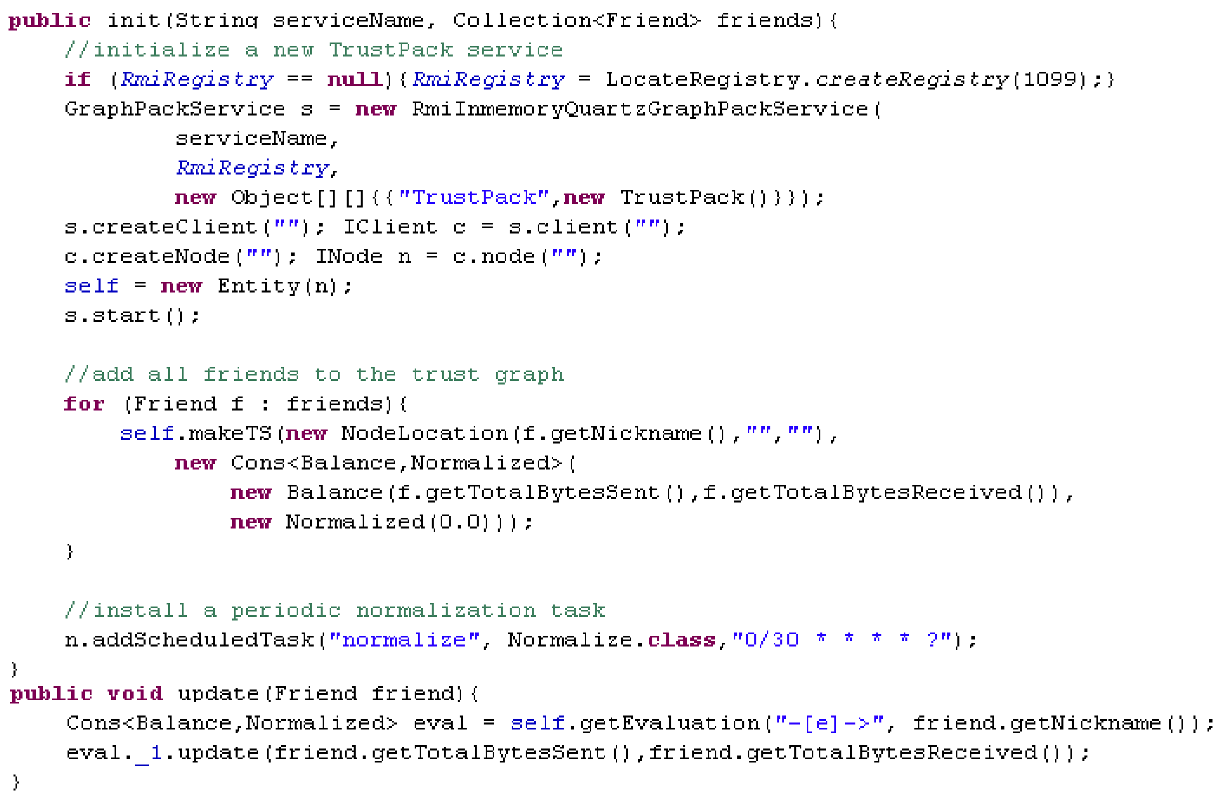
Our code is based on Alliance version 1.2. Our first change in the code initializes a local TrustPack service during the Alliance start-up process. This initialization includes opening the service for incoming messages, adding outgoing edges that represent the peer's friends and installing a custom task 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 4.1 shows the code added to the Alliance program in order to initialize the TrustPack service and update it when new information is gathered. Figure 4.2 shows the code of the

[1]http://www.alliancep2p.com/

three trust related operations that we have described, and in Figure 4.3 there is a screen-shot of the modified GUI, which displays the new trust metrics.



```
public init(String serviceName, Collection<Friend> friends){
    //initialize a new TrustPack service
    if (RmiRegistry == null){RmiRegistry = LocateRegistry.createRegistry(1099);}
    GraphPackService s = new RmiInmemoryQuartzGraphPackService(
            serviceName,
            RmiRegistry,
            new Object[][]{{"TrustPack",new TrustPack()}});
    s.createClient(""); IClient c = s.client("");
    c.createNode(""); INode n = c.node("");
    self = new Entity(n);
    s.start();

    //add all friends to the trust graph
    for (Friend f : friends){
        self.makeTS(new NodeLocation(f.getNickname(),"",""),
            new Cons<Balance,Normalized>(
                new Balance(f.getTotalBytesSent(),f.getTotalBytesReceived()),
                new Normalized(0.0)));
    }

    //install a periodic normalization task
    n.addScheduledTask("normalize", Normalize.class,"0/30 * * * * ?");
}
public void update(Friend friend){
    Cons<Balance,Normalized> eval = self.getEvaluation("-[e]->", friend.getNickname());
    eval._1.update(friend.getTotalBytesSent(),friend.getTotalBytesReceived());
}
```

Figure 4.1.: TrustPack Alliance extension: initialization and update

4.3. Evaluation

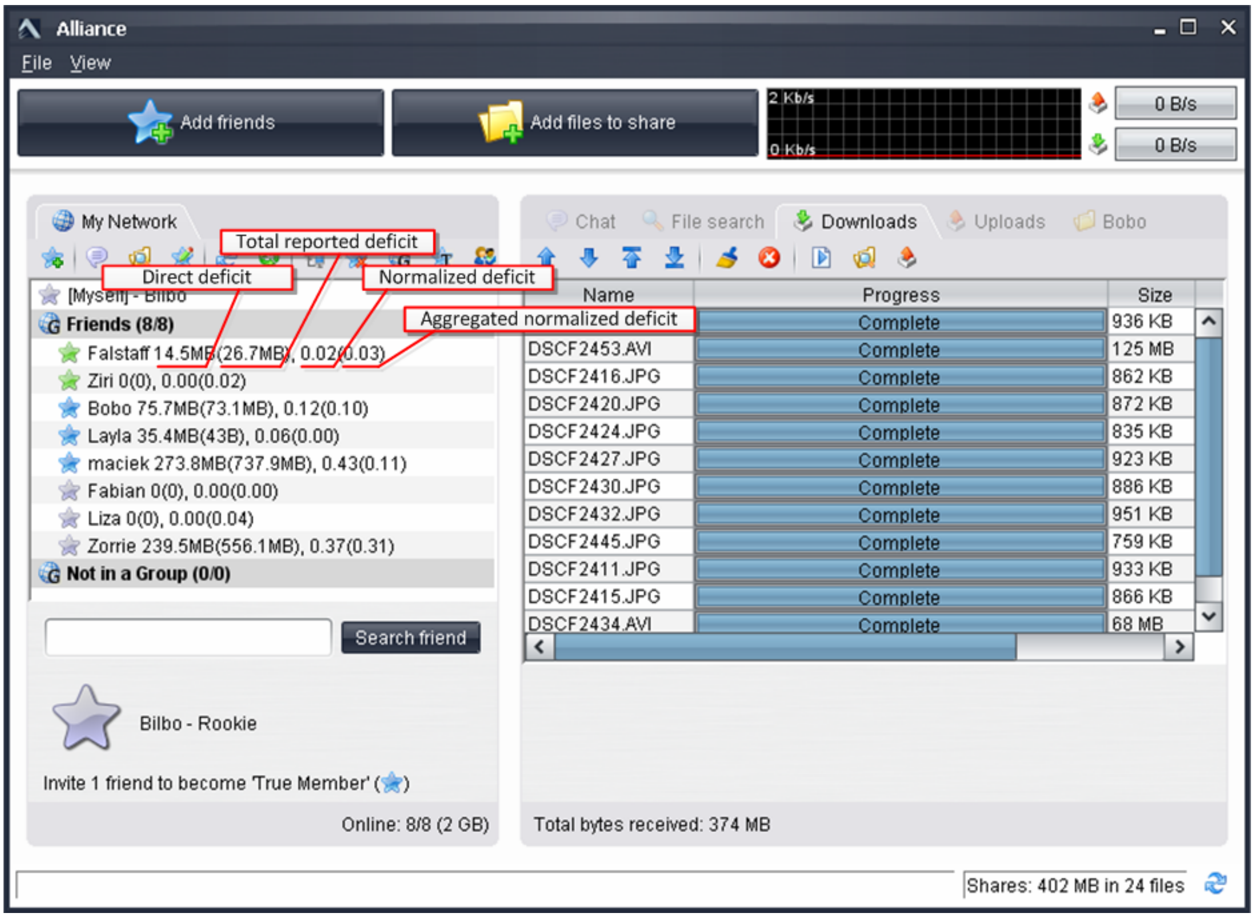
Configuring Alliance to work with TrustPack was a fairly simple task that consisted mostly of finding the appropriate entry points in the Alliance code. We note that our code uses a prototype but, as mentioned in the Future Work section (Section 5.1), a complete implementation of TrustPack would include standard trust processes and a service administrator tool, which should greatly simplify the initialization code.

In addition, we have checked that the new enhanced Alliance works reasonably 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.

```
/** a periodic normalization task: <br/>
 *  1 for each v: positiveV := MAX(0,v) <br/>
 *  2 sum := sum over all positiveV <br/>
 *  3 for each v: normalizedV := positiveV/sum <br/> */
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.aggr(
                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.aggr(
            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: <br/>
 *  for each reporter R: ReportR := <R Score> * <P's Score according to R> <br/>
 *  P's weighted normalized score := sum over all ReportR */
public double WeightedReports(String friendName){
    ResultSet $ = self.traverse("-[e1]->-[e2]->",friendName);
    double total = 0.0;
    for (Result 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 4.2.: TrustPack Alliance extension: trust related operations



[Myself] - Bilbo
Friends (8/8)
Falstaff 14.5MB(26.7MB), 0.02(0.03)
Ziri 0(0), 0.00(0.02)
Bobo 75.7MB(73.1MB), 0.12(0.10)
Layla 35.4MB(43B), 0.06(0.00)
maciek 273.8MB(737.9MB), 0.43(0.11)
Fabian 0(0), 0.00(0.00)
Liza 0(0), 0.00(0.04)
Zorrie 239.5MB(556.1MB), 0.37(0.31)
Not in a Group (0/0)

Invite 1 friend to become 'True Member'

Online: 8/8 (2 GB)

Name	Progress	Size
	Complete	936 KB
DSCF2453.AVI	Complete	125 MB
DSCF2416.JPG	Complete	862 KB
DSCF2420.JPG	Complete	872 KB
DSCF2424.JPG	Complete	835 KB
DSCF2427.JPG	Complete	923 KB
DSCF2430.JPG	Complete	886 KB
DSCF2432.JPG	Complete	951 KB
DSCF2445.JPG	Complete	759 KB
DSCF2411.JPG	Complete	933 KB
DSCF2415.JPG	Complete	866 KB
DSCF2434.AVI	Complete	68 MB

Total bytes received: 374 MB

Shares: 402 MB in 24 files

Figure 4.3.: A sceen-shot of Alliance with the TrustPack extension

5. Conclusions

5.1. Future Work

In this section, we suggest future research directions and review aspects that are missing from the prototype implementation of the GraphPack and TrustPack frameworks.

5.1.1. 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 [25] 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 [27]* that specifies their structure and allows for safe typing. The concept of *views [33]* can apply to edges that are dynamically created upon request. *Triggers [11]* 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 [18]* may also be investigated.

5.1.2. 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.

5.1.3. Management

In Section 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.

5.1.4. 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 persistence 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).

5.1.5. 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.

5.2. Closing Remarks

This thesis has presented two frameworks, GraphPack and TrustPack. GraphPack provided an architecture for 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, extended 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.

Implementing of the GraphPack and TrustPack frameworks was a challenge that was also addressed in this work[1]. Our GraphPack and TrustPack prototypes correspondingly support the functionality that is presented in Chapter 2 and Chapter 3. Additionally, both frameworks feature a modular design that enables customization and extension. Those prototypes validate the applicability of our design and provide references for future implementations.

[1]Source code and full documentation are located at http://code.google.com/p/graphpack/ and http://code.google.com/p/trustpack/ correspondingly

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 Section 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 expression[1].

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

[1]http://en.wikipedia.org/wiki/Cron#CRON_expression

A.2. Persistence

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

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 B.1 presents the BNF for the basic primitives used in PackCypher. Table B.2 specifies how a predicate can be constructed from those primitives. Finally, Table B.3 specifies PackCypher's path expressions.

`<value> ::=`	`<entity> | <string> | <number> | ‘true’ | ‘false’ | ‘null’`
`<entity> ::=`	`<identifier> | <entity> ‘.’ <entity>`
`<string>`, `<number>`, `<whole-number>` and `<identifier>`	specified by Java syntax[1]

[1] http://docs.oracle.com/javase/specs/

Table B.1.: PackCypher BNF: basic primitives

`<predicate> ::=`	`<or>`
`<or> ::=`	`<and> | <or> ‘||’< and>`
`<and> ::=`	`<equality> | <and> ‘&&’ <equality>`
`<equality> ::=`	`<relational> | <equality> ‘==’ <relational> | <equality> ‘!=’ <relational>`
`<relational> ::=`	`<additive> | <relational> ‘<’ <additive> | <relational> ‘>’ <additive> | <relational> ‘<=’ <additive> | <relational> ‘>=’ <additive>`
`<additive> ::=`	`<mult> | <additive> ‘+’ < multiplicative > | < multiplicative > ‘-‘<mult>`
`<multiplicative> ::=`	`<unary> | <multiplicative > ‘*’ <unary> | <multiplicative > ‘/’ <unary>`
`<unary> ::=`	`<value> | ‘!’ <unary>`

Table B.2.: PackCypher BNF: predicates

`<path> ::=`	`<edge> | <path> <edge> | ‘(‘<path>‘)’ [‘?(’ <predicate> ‘)’] [’*’[<whole-number>‘..’]<whole-number>]`
`<edge> ::=`	`‘-‘[<identifier>]‘->’[<identifier>][‘.’<identifier>‘()’][‘?(’ <predicate> ‘)’] [’*’<whole-number>‘..’<whole-number>]`

Table B.3.: PackCypher BNF: paths

Bibliography

[1] Donovan Artz and Yolanda Gil. A survey of trust in computer science and the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):58–71, June 2007.

[2] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, December 1986.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[4] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. pages 254–274. Springer, 1998.

[5] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.

[6] Sonja Buchegger, Doris Schiöberg, Le H. Vu, and Anwitaman Datta. PeerSoN: P2P social networking: early experiences and insights. In *SNS '09: Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52, New York, NY, USA, 2009. ACM.

[7] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. Referee: Trust management for web applications. *World Wide Web Journal*, pages 2:127–139, 1997.

[8] Bram Cohen. Incentives build robustness in BitTorrent, May 2003.

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

[10] L.A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *Communications Magazine, IEEE*, 47(12):94 –101, dec. 2009.

[11] Umeshwar Dayal, Eric N. Hanson, and Jennifer Widom. Active database systems. In *Modern Database Systems*, pages 434–456. ACM Press, 1994.

[12] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[13] Christophe Diot, Richard Gass, James Scott, C Augustin Chaintreau, Pan Hui, and Jon Crowcroft. Pocket switched networks: Real-world mobility and its consequences for opportunistic forwarding. Technical report, 2005.

[14] John Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer Berlin / Heidelberg, 2002.

[15] Randy Farmer and Bryce Glass. *Building Web Reputation Systems.* Yahoo Press, 1 edition, March 2010.

[16] Jennifer Ann Golbeck. *Computing and applying trust in web-based social networks.* PhD thesis, College Park, MD, USA, 2005. AAI3178583.

[17] Tyrone Grandison. *Trust Management for Internet Applications.* PhD thesis, Imperial College London, July 2003.

[18] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[19] Pan Hui and Jon Crowcroft. How Small Labels Create Big Improvements. pages 65–70, March 2007.

[20] Borislav Iordanov. Hypergraphdb: a generalized graph database. In *Proceedings of the 2010 international conference on Web-age information management*, WAIM'10, pages 25–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Audun Josang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, March 2007.

[22] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. Working Paper 2002-56, Stanford InfoLab, 2002.

[23] Tomasz Kaszuba, Krzysztof Rzadca, Adam Wierzbicki, and Grzegorz Wierzowiecki. A blueprint for universal trust management services. Technical report, Polish-Japanese Institute of Information Technology, Warsaw, Poland, 2008.

[24] L.M. Kohnfelder. *Towards a Practical Public-key Cryptosystem.* M.I.T., Department of Electrical Engineering and Computer Science, 1978.

[25] P.J.M. Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Mathematics and Its Applications. D. Reidel, 1987.

[26] Raph Levien, Alex Aiken, Raph Levien, and Alexander Aiken. Attack resistant trust metrics for public key certification. In *In 7th USENIX Security Symposium*, 1998.

[27] D. Maier. *The theory of relational databases.* Computer software engineering series. Computer Science Press, 1983.

[28] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960.

[29] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.

[30] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, February 2008.

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

[32] Alex Sherman, Jason Nieh, and Clifford Stein. Fairtorrent: bringing fairness to peer-to-peer systems. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 133–144, New York, NY, USA, 2009. ACM.

[33] Oded Shmueli and Alon Itai. Maintenance of views. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 240–255. ACM Press, 1984.

[34] International Telecommunication Union. ITU-T Recommendation X.509 | ISO/IEC 9594-8: "Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks". Technical report.

[35] Adam Wierzbicki. *Trust and Fairness in Open, Distributed Systems*, volume 298 of *Studies in Computational Intelligence*. Springer, 2010.

[36] Walt Yao. Fidelis: A policy-driven trust management framework. In Paddy Nixon and Sotirios Terzis, editors, *iTrust*, volume 2692 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2003.

[37] Walt Yao. Trust management for widely distributed systems. Technical Report UCAM-CL-TR-608, University of Cambridge, Computer Laboratory, November 2004.

[38] Philip R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

להלן מבנה התזה: פרק 1 הינו הקדמה לתזה. בפרקים 2 ו-3 אנו מבצעים סקירה של מערכות GraphPack ו-TrustPack בהתאמה. בפרק 4 אנו מציגים חקר מקרה בו אנחנו מרחיבים מערכת העברת קבצים קיימת (AlliancePP[1]) עם יכולות ניהול אמון. לבסוף, בפרק 5 נדון בעבודות המשך ונסכם את התזה.

[1]http://www.alliancep2p.com/

בעבודה זו אנחנו מציגים אב-טיפוס של מערכת ניהול אמון חדשה המכונה TrustPack. TrustPack הינה מערכת ניהול אמון מבוזרת (decentralized) אשר רצה בעזרת מספר שירותים (services) עצמאיים במקום שירות אחד מרכזי. ביזור זה מאפשר שליטה מלאה לעושי ההצהרות עצמם, אשר יכולים לבחור את השירות שבו יישמרו ההצהרות שלהם ואת האופן שבו ייעשה שימוש בהצהרות אלו. שליטה זו מאפשרת למעשה לכל משתמש לטפל ישירות במידע שלו ובכך לשמור על פרטיותו אם ירצה.

האתגר המרכזי עימו התמודדנו היה לספק יכולת עיבוד מידע למרות הביזור של המערכת. בעיה זו אינה טריוויאלית, שכן לאף אחד מהשרותים לא מובטחת גישה לכלל ההצהרות או אפילו לכלל ההצהרות של אפליקציה כלשהי.

כדי לספק יכולת כזו, יצרנו מערכת כללית אשר מספקת כלים לעיבוד של גרף מבוזר. מערכת זו נקראית GraphPack. נשים לב כי זה טבעי להתייחס להצהרות אותם מנהלים ב-TrustPack בתור גרף המכוון מגורמים שעשו הצהרות למושאי ההצהרות ומכאן GraphPack מהווה בסיס מתאים ל-TrustPack.

חשוב לציין, GraphPack פותחה כמערכת נפרדת מכיוון שהבעיה שתיארנו אינה ייחודית ל-TrustPack. קיימות מערכות נוספות שנדרשות לעבד מידע מבוזר בעל אופי דומה. לדוגמא, רשתות חברתיות מבוזרות, רשתות מבוזרות להעברת קבצים (Peer-to-Peer file sharing) ורשתות אד-הוק (Ad-Hoc).

GraphPack למעשה מספקת פיתרון כללי ושימושי ובכך מהווה חלק חשוב מהתרומה של עבודה זו. GraphPack כוללת בתוכה כלים לניהול דינאמי של משימות ומאפשרת ביצוע של מעברים על הגרף (graph traverses) בעזרת שפת תיכנות חדשה הנקריאת PackCypher. שפה זאת מאפשרת ביטוי (expression) של מעברים מורכבים בצורה הצהרתית (declarative) ומתומצתת. ביטויי PackCypher מתורגמים לתהליכים מורכבים שיכולים להתבצע במספר שירותים שונים באופן שקוף למשתמש.

שתי המערכות המוצגות בעבודה זאת מומשו בעזרת שפות התכנות Java ו-Scala. בכתובות `http://code.google.com/p/trustpack/` ו-`http://code.google.com/p/graphpack/` ניתן למצוא את קוד המקור והתיעוד המלא של TrustPack ו-GraphPack בהתאמה.

תקציר

עם העלייה בפופולריות של האינטרנט, נוצר מצב בו לעיתים קרובות גורמים זר-ים צריכים לבצע פעולות גומלין ביניהם. לדוגמא, eBay.com הינו אתר המא-פשר לגולשים לקנות מוצרים מבלי להכיר את המוכר. דוגמא נוספת היא מערכת Wikipedia.org אשר מאפשרת לכל גולש לכתוב תוכן ישירות באתר. כדי להתמודד עם הבעיה, נוצרו מערכות לניהול אמון (trust management). מערכות אלה מנסות לעזור למשתמשים ולאפליקציות לבצע פעולות מושכלות במצב של חוסר ודאות. הם עושות זאת בעיקר על ידי שמירה ועיבוד של הצהרות בין גורמים שונים, כאשר כל הצהרה היא רשומה בה גורם כלשהו מביע דעה או מדווח על ניסיון שהיה לו עם גורם אחר. רשומה כזאת, יכולה למשל להכיל ביקורת שמשתמש באתר קניות כתב עבור מוצר כלשהו, או הצהרה על חברות ברשת חברתית.

נכון לזמן כתיבת עבודה זו, מערכות לניהול אמון בדרך כלל מפותחות ביחד עם האפליקציות שמשתמשות בהן. לדוגמא, באתר eBay.com, פיתחו מערכת משל עצמם שמדרגת את הקונים והמוכרים. רק לאחרונה הוצעו מערכות שמספקות ני-הול אמון בתור שירות (service) נפרד מאפליקציה כלשהי [14, 16, 22]. לגישה זאת ישנם מספר יתרונות. קודם כל, שימוש חוזר בקוד מאפשר פיתוח מהיר יותר של אפליקציות איכותיות יותר. דבר שני, הפרדת מערכת ניהול האמון מהאפליקציות מאפשר גמישות ומעודדת שיתוף מידע בין אפליקציות שונות.

בעוד שפיתוח מערכות ניהול אמון בתור שירות הוא צעד קדימה, המערכות שהוצעו עד כה מספקות שירותים ריכוזיים בלבד. תכונה שיש עימה מספר בעיות. קודם כל, למערכות אלה שליטה מלאה על כל ההצהרות שנעשו על ידי כל הגורמים, דבר שיכול לפגוע בפרטיות של משתמשים במקרים מסויימים. דבר שני, הסילומיות (scalability) של השירות יכולה להיפגם, שכן שירות כזה יכול להידרש לעבד מידע רב ולענות בזמן אמת על מיליוני שאילתות (Yahoo! Reputation Platform [14] הינה דוגמא למערכת בקנה מידה כזה).

המחקר נעשה בהנחיית פרופסור רועי פרידמן בפקולטה למדעי המחשב.

אני מודה לטכניון ולקרן הלאומית למדע (מענק 1247/09) על התמיכה הכספית הנדיבה בהשתלמותי.

טרסטפק (TrustPack):
מערכת מבוזרת לניהול אמון

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

עמית פורטנוי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
סיוון התשעב חיפה מאי 2012

טרסטפק (TrustPack): מערכת מבוזרת לניהול אמון

עמית פורטנוי