# TRUSTWORTHY PEER-TO-PEER INFRASTRUCTURE USING HARDWARE BASED SECURITY

by

TIEN TUAN ANH DINH

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
September 2010

We shall not cease from exploration

And the end of all our exploring

Will be to arrive where we started

And to know the place for the first time

---

T.S Elliot – Little gidding

If I have been able to see further, it was only because I stood on the shoulders of giants

---

Isaac Newton

Science is what we understand well enough to explain to a computer. Art is everything else we do

---

Donald Knuth

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them

---

Kevin Ryan, James M. Cooper

**Abstract**

Peer-to-Peer (P2P) infrastructure, thanks to its scalability, has been used for designing many large-scale distributed systems. Of the two types of P2P infrastructure: unstructured and structured, the latter has received a greater amount of research attention. Security is one of many challenging problems faced by existing systems based on structured P2P. Having trust in such the P2P environments can help mitigate many problems including security, because peers can choose to only interact with the ones that are deemed trustworthy. However, there exists numerous hurdles that need to be overcome before a reliable trust infrastructure (or trust system) can be implemented for P2P. This thesis investigates and seeks to improve the existing reputation metrics and feedback mechanisms which are important components of the trust system. It shows the limitations of existing reputations metrics and feedback mechanisms, then proposes new algorithms and protocols addressing these limitations. The new reputation metrics are more resilient to manipulations, and they take into account negative feedback. The new protocols are designed as parts of the feedback mechanisms, and they allow an honest peer in a structured P2P system to securely detect if another peer has misbehaved. The mechanism for detecting misbehavior has proved difficult in structured P2P, and has not been considered by existing feedback mechanisms.

The new protocols leverage hardware-based security which is in the form of trusted devices. Some protocols utilize the Trusted Platform Modules which are currently available at high-ends computers. The others make use of the newly proposed trusted hardware called Trusted Token Modules.

The protocols concerning the detection of misbehavior in structured P2P routing are analyzed in this thesis using both formal methods and simulations. CSP is used to model and verify the properties of these protocols. The performance of these protocols is then evaluated using the newly proposed, distributed simulation platform called dPeerSim.

# ACKNOWLEDGEMENTS

With greatest gratitude, I would like to dedicate this thesis to my family: my grandparents, my parents and my little sister. I thank my family for having brought me into this world and provided me with the nurturing environment that I am very lucky to have. I thank them for their unconditional love and support, without which I would not have had the necessary foundation and strength to pursue my dreams.

I am indebted to my supervisor, Professor Mark Ryan, who has served not only as a mentor to me, but also as a real life role model that I want to see myself become in the future. I am particularly grateful for his persistent encouragement, his incredible attention to details, and his perseverance in understanding and contributing to my thesis. One of the biggest lessons I have learned from working with Mark is the need to express ideas in the simplest, clearest but also most exact form. I always admire him for being considerate, and for the way he treats his students as peers. But above all, his charisma, his positive outlook on life, his kindness and his sense of humor are so infectious that I feel myself becoming a better person just by working with him.

I am sincerely grateful to my second supervisor, Dr Georgios Theodoropoulos. While working with Mark has taught me to be meticulous in research, the greatest skills I have learned from Georgios are how to take a step back, to look at the big picture, and to evaluate your work in context. I appreciate his conspicuous *Greekness*, his sense of humor, and the way he instills confidence into his students. I thank Georgios for his guidance, both technical and general, during the course of the thesis. I thank him for facilitating a lively distributed system group which consists of extremely smart and friendly people with whom I have had a number of successful collaborations.

Doing a PhD, as I have learned in almost four years, is not only about research, but also about meeting new people and making good friends. As I see old friends from undergraduate moved on with their jobs, I would have felt left out and sad if it was not for all the new friends, new colleagues that I have had the pleasure to know. They truly are exceptional characters that make my four-year PhD such a joyous experience. I would like to thank Rob Minson, Michael Lees for their early guidance and involvements in my first publication. Thanks also go to Tom Chothia and Rami Bahsoon for all the fruitful discussions, and for their valuable advice on doing research. I am grateful to Vinoth Suryanarayanan, Richard Price for their help in accessing the university cluster. I am thankful to have known people like Edward Robinson, Seyyed Shad, Peter Lewis, who always seem to have answers to all my technical, geeky problems. Last but not least, I would like to thank other colleagues such as Josef Baker and Ben Smyth, who have got me into running and kept on encouraging me with their great stories of achievements in running.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

The past decade has witnessed phenomenal developments in Peer-to-Peer (P2P) infrastructure, thanks to the vast number of systems based on P2P, and to the research in P2P systems being an increasingly active field. A P2P system can be defined as a network of computers (or peers) that communicate mainly with each other without the need for a centralized party. Compared to a traditional system in which one server deals with requests from many clients, the P2P system is more scalable and its lack of the centralization components means that there is no single point of failure.

Two types of P2P infrastructure exist, and they differ in the graphs formed by the connections among peers and the mechanisms for locating objects. In *unstructured* P2P, peers form a random graph, as a peer can be a neighbor of any other peer. For locating an object, the peer broadcasts the request to its neighbors which again forward the request to their own neighbors. In *structured* P2P, peers form a rigid topology, which allows for efficient mechanisms for locating objects.

Well-known examples of systems based on unstructured P2P are file-sharings: Napster, Limewire [42], eDonkey [55], Bittorrent [22], etc. Because of its scalability, structured P2P has been used for many large-scale applications. For instance, it has been used for an application-level multicast infrastructure that enables multicast routing without the need

for hardware support [17]. The performance of such the infrastructure can come very close to that of a hardware-supported multicast infrastructure. Other examples include P2PSIP [14] (a Voice-over-IP application) and PAST [81] (a network storage system). Even botnets — networks of compromised computers used for sending spams — are built on structured P2P [49].

The scalability property of structured P2P comes with a number challenges that need to be addressed before one could take full advantage of the infrastructure. One of the main challenges is security, which is inherent in P2P because peers are autonomous and there lacks a centralized authority that polices peers' interactions. An adversary could disrupt the mechanism for locating objects by redirecting the requests to another malicious party, or by impersonating the final nodes in the routing paths of the requests. It could also attack properties that are specific to the systems built on structured P2P.

Trust, a concept originating from human society, can help dealing with the security issues in structured P2P. In real-world societies, trust allows people to interact confidently with strangers. It also helps people identify and then avoid the untrustworthy. In the context of structured P2P, having trust helps the honest peers identify and avoid interacting with the adversaries — ones that cause harms to the system and are therefore considered untrustworthy. Having trust in the system also encourages peers to be trustworthy, since acting otherwise would impede their opportunities to interact with the others. In other words, the implication of having trust goes beyond security, as trust induces cooperation, which is important to systems that rely on peers sharing their resources.

To facilitate trust in P2P (both structured and unstructured), a trust infrastructure (or trust system) is needed. Such a system provides the information about one peer that can be used by others to decide whether to trust that peer. The trust system is usually based on reputation which has been shown to be a good indicator of trust. Particularly, the trust system comprises the *feedback mechanism* that allows peers to give each other ratings for their behavior, and the *reputation metric* that combines feedback into reputation values (or scores). A peer having a high reputation score is likely to be trustworthy and to be

subsequently trusted by others.

Implementing a trust system for P2P is challenging, since such the system requires two properties to be met: reliability and efficiency. The latter states that the system is efficient in terms of resources (memory, bandwidth, etc.) that it uses. The former means that the system can prevent honest peers from making incorrect trust decisions, even if it is given inaccurate feedback, and even in the presence of the so-called *Sybil manipulation attack* in which the attacker introduces new peers to the system and uses them strategically to boost its reputation. In other words, in a reliable trust system, feedback elicited by the feedback mechanism is combined together by the reputation metric to yield reputation values which are intuitive and resilient to manipulation. It means that high reputation values are given to well-behaved peers and low reputation values are given to the poorly-behaved.

## 1.2 Scope and Contributions

This thesis is concerned with the reliability property of trust systems for structured P2P. In particular, it seeks the answers to the following research questions:

1. To what extent are existing reputation metrics such as PageRank [73] vulnerable to Sybil manipulations? Can they be improved? Are the feedback models used by those metrics strong enough? Can they be made more realistic?

2. Regarding the existing feedback mechanisms for structured P2P applications, is it always possible for peers to leave feedback to each other after their transactions? If not, can we design mechanisms that overcome such the limitation?

This thesis focuses on the resilience of the reputation metrics against manipulations, and on the use of hardware-based security in feedback mechanisms. More specifically, for reputation metrics, it enhances the capability of being resilient against manipulation, and it adds the support for negative feedback. For feedback mechanisms, it uses hardware-

based security to design mechanisms that enable a peer to detect the misbehavior of others in the structured P2P environment. It carries out formal analysis to show that the proposed mechanisms allow the correct detection of misbehavior, and experimental analysis to evaluate the performance of these mechanisms when implemented in practice.

The detailed contributions of this thesis are as follows:

1. Regarding reputation metrics:

   (a) Extending the previous work on the resilience of the PageRank reputation metric [73] against Sybil manipulation [21], by considering a stronger Sybil strategy. The result indicates that PageRank has different degrees of sensitivity against different Sybil strategies.

   (b) Showing, through experimental analysis, that PageRank is more sensitive to Sybil manipulation when used for undirected graphs — which are commonly found in P2P environments — than for directed graphs.

   (c) Proposing a new reputation metric based on PageRank, called Cluster-based PageRank (or CPR). Experimental results indicate that compared to the original PageRank, CPR produces relatively intuitive reputation scores, and that CPR is more resilient to Sybil manipulation than PageRank when used for undirected graphs.

   (d) Arguing that negative feedback is important for P2P trust system, which then leads to the proposal of the new reputation metric that is based on PageRank and supports negative feedback. The new metric, called PRN, is evaluated through experiments. The results indicate that it produces intuitive reputation values and is resilient against Sybil manipulation.

2. Regarding feedback mechanisms:

   (a) Presenting two examples of structured P2P systems that demonstrate the difficulties for an honest node to securely detect the misbehavior of another. The

detection is necessary in order for the honest node to leave appropriate feedback to the other. One example is related to routing protocols, and the other is a marketplace built on top of structured P2P.

(b) Proposing two mechanisms that enable secure detection of misbehavior. One mechanism targets the routing layer (called DTR1), the other targets the P2P-based marketplace application (called DTA1). Both DTR1 and DTA1 rely on the Trusted Computing infrastructure, particularly on the Trusted Platform Modules (TPMs), to provide trusted operations involving monotonic counters. DTR1, in addition, relies on a centralized party called the Certificate Authority (CA).

(c) Proposing a new general-purpose security hardware called Trusted Tokens Module (TTM). Examples of using TTMs in access control systems are presented to demonstrate that TTMs can be useful in other areas of application beside P2P. Next, new protocols that leverage TTMs for detecting misbehavior for the two case studies are proposed. DTR2 and DTA2 both improve upon DTR1 and DTA1. In particular, the centralized party (the CA) in DTR1 is removed in DTR2.

3. Regarding analysis:

(a) Formalizing DTR1 and DTR2 in CSP, and verifying that they satisfy the property that implies the honest peers' abilities to securely detect misbehavior of other peers in the context of P2P routing. Data independence is used as a model abstraction technique that reduces the original CSP models of DTR1 and DTR2 to ones with smaller state spaces. For the small models with a limited number of nodes, automated proofs showing that they satisfy the desired property are generated by the model checker tool FDR [37]. The automated proofs are accompanied by general, hand-written proofs for the models of arbitrary numbers of nodes.

(b) Introducing a new condition that when satisfied by a data-independent process allows the data type to be reduced without introducing new traces to the model.

(c) Designing a new distributed simulation platform, called dPeerSim, by extending the open-source, application-level P2P simulator named PeerSim [78]. Experimental studies demonstrate that dPeerSim is capable of simulating large P2P systems in reasonable time scales.

(d) Using dPeerSim to evaluate the performance of DTR1 and DTR2 in practice. The preliminary results suggest that the CA in DTR1 is unlikely to be a performance bottleneck. Furthermore, the performance of DTR1 and DTR2 under churn are comparable, and they still leave much room for improvement.

## 1.3 Structure of the Thesis

Chapter 2 starts with the background on P2P systems, with particular focus on structured P2P. It explains why P2P is useful by describing a number of applications built on top of P2P. The chapter also introduces the concepts of trust and reputation in social sciences and how they have been used in computer science. The next and important part of the chapter contains the gap analysis of the current literature on P2P. It brings to the fore a number of currently challenging problems in structured P2P, especially security issues, and then explains why having a trust system can help mitigate those problems. Finally, it discusses the difficulties in implementing a reliable trust system for P2P, some of which are in the main focus of this thesis.

It should be noted here that Chapter 2 only serves as a *scene-setting* literature survey, whose goal is to clarify the problems addressed in this thesis, as well as to relate them to the other works in the field. Distributed over the following chapters are the literature surveys and related works that are relevant to the topic discussed in the chapters.

Chapter 3 presents the improvements made to the existing reputation metrics. It is a joint work with Mark Ryan, which elaborates on the contributions of this thesis regarding

reputation metrics: contribution 1.a to 1.d. It starts with a detailed model of reputation for P2P systems, and the model of the PageRank metric. It shows that the resilience of PageRank depends on the Sybil manipulation strategy, and that PageRank is less resilient against Sybil manipulation under undirected graphs than under directed graphs. Next, a new metric called Cluster-based PageRank is presented and demonstrated to be more resilient than PageRank under undirected graphs. The next part of the chapter focuses on motivating the integration of negative feedback into existing reputation metrics. It presents the new metric, called PRN, that includes negative feedback. PRN is then shown to produce intuitive reputation values and be resilient against Sybil manipulation.

Chapter 4 contains the details of the contribution 2.a and 2.b. It presents the results from collaborative work with Mark Ryan and Tom Chothia. The chapter starts by presenting two case studies in which the misbehavior of peers proves hard to detect. One example concerns the misbehavior in routing protocols, the other concerns the misbehavior specific to the P2P-based marketplace application. The chapter introduces the concept and the main features of Trust Computing and the TPMs. For each case study, the property stating that peers are able to securely detect the misbehavior of each other is introduced. For example, in the case study concerning routing protocols, the property is called Root Authenticity (or RA) property. Finally, the chapter presents the details of two mechanisms, called DTR1 (for the example with routing protocols) and DTA1 (for the example with the marketplace application), that aim to satisfy the desirable properties.

Chapter 5 presents the contribution 2.c and 2.d in more detail. It discusses the limitations of TPMs in the context of detecting misbehavior in P2P: inefficiency and having to rely on a centralized party. It then describes the design of a new hardware device (TTM) whose main operations include the creation, storage and transferring of tokens between devices in secure manners. Examples of access control systems using TTMs are presented as examples to illustrate that TTMs can be used in other applications beside P2P. Finally, new mechanisms using TTMs that are designed to satisfy the RA property for the two case studies are presented. DTR2 improves upon DTR1 by removing the need for the CA.

DTA2 proves more efficient and allows for more application features than DTR1 does.

Chapter 6 provides the formal analysis for DTR1 and DTR2. The analysis of DTR1 was done in collaboration with Mark Ryan. It essentially contains the contribution 3.a and 3.b. It first explains the reasons behind selecting CSP as the modeling language for DTR1 and DTR2. Next, it presents a model of DTR1, and the formalization of the RA property, both in CSP. Due to the original of DTR1 being rather large and complex, the chapter presents two model abstraction techniques: weakening adversary and data independence that are used to reduce the verification that a model satisfies the RA property to checking the property of another model with a smaller state space. The existing data independence technique is extended in the chapter with the new condition that when satisfied allows the data type to be reduced without introducing new traces to the model. The chapter includes the automated proofs that DTR1 and DTR2 satisfy the RA property for a small number of nodes, and the general, hand-written proofs for arbitrary numbers of nodes. The automated proofs are generated by the CSP model checker called FDR.

Chapter 7 provides further analysis of DTR1 and DTR2 by evaluating their performance through simulation. Its contributions to this thesis are the contribution 3.c and 3.d. The first part of the chapter presents the motivations and designs of a new simulation platform called dPeerSim. It shows that dPeerSim is a scalable platform for simulating large P2P systems in reasonable time scales. This work was carried out with a number of collaborators: Michael Lees, Georgios Theodoropoulos and Rob Minson. The second part of the chapter presents and analyzes the preliminary results from simulating DTR1 and DTR2 under a strict failure model and with a simple maintenance protocol. The results suggest that in DTR1, CA is unlikely to be a performance bottleneck under normal network conditions. The performance of DTR1 and DTR2 under churn are comparable, and they still leave much room for improvement.

Finally, Chapter 8 concludes the thesis and discusses avenues for future work.

## 1.4 List of Publications

Most parts of this thesis have been published in a number of papers and reports, as listed below.

1. Chapter 3:

   (a) Tien Tuan Anh Dinh and Mark Ryan. A Sybil-Resilient Reputation Metric for P2P Applications. In *3rd International Workshop on Dependable and Sustainable Peer-to-Peer Systems (DAS-P2P 2008), in conjunction with the 2008 International Symposium on Applications and the Internet (SAINT2008)*, pages 193-196, Turku - Finland, August 2008.

   This paper provides the contents for Section 3.1 and Section 3.2. It describes the need for, and the details of the Cluster-based PageRank.

2. Chapter 4:

   (a) Tien Tuan Anh Dinh and Mark Ryan. Verifying Security Property of Peer-to-Peer Systems Using CSP. In *2010 European Symposium on Research in Computer Security (ESORICS)*.

   (b) Tien Tuan Anh Dinh and Mark Ryan. Checking Security Property of P2P Systems Using CSP. Technical report. University of Birmingham, CSR-10-07. Accessible at `http://www.cs.bham.ac.uk/~ttd/files/technicalReport.pdf`

   (c) Tien Tuan Anh Dinh, Tom Chothia, and Mark Ryan. A Trusted Infrastructure for P2P-Based Marketplaces. In *9th IEEE International Conference in Peer-to-Peer Computing (P2P'09)*, pages 151-154, Seattle - WA, October 2009.

   The ESORICS paper and the technical report CSR-10-07 provide the materials for the first part of the chapter where DTR1 is presented. The details of DTA1 presented in the second part of the chapter are based on the P2P'09 paper.

3. Chapter 5:

(a) Tien Tuan Anh Dinh and Mark Ryan. Secure Hardware Abstraction for Distributed Systems. Technical report. University of Birmingham, CSR-10-08. Accessible at `http://www.cs.bham.ac.uk/~ttd/files/secureHardware.pdf`

The contents of the chapter: the design of the new hardware and details of DTR2 and DTA2 are based on the technical report CSR-10-08.

4. Chapter 6: the formal analysis of DTR1 is presented in the ESORICS paper in brief details, and in the technical report CSR-10-07 in full details. The analysis of DTR2 is included in the technical report CSR-10-08.

5. Chapter 7:

   (a) Tien Tuan Anh Dinh, Georgios Theodoropoulos and Rob Minson. Evaluating large scale distributed simulation of P2P networks. In *12th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT'08)*, pages 51-58, Vancouver - Canada, October 2008.

   (b) Tien Tuan Anh Dinh, Michael Lees, Georgios Theodoropoulos and Rob Minson. Large Scale Distributed Simulation of p2p Networks. In *2nd International Workshop on Modeling, Simulation, and Optimization of Peer-to-peer Environments (MSOP2P 2008)*, in conjunction with Euromicro PDP 2008, pages 13-15, Toulouse - France, February 2008.

   (c) Richard Price, Tien Tuan Anh Dinh and Georgios Theodoropoulos. Analysis of a Self-Organizing Maintenance Algorithm Under Constant Churn. In *3rd International Workshop on Dependable and Sustainable Peer-to-Peer Systems (DAS-P2P 2008), in conjunction with the 2008 International Symposium on Applications and the Internet (SAINT2008)*, pages 209-212, Turku - Finland, August 2008.

The first two papers describe dPeerSim and demonstrate its scalability through experiments with different P2P protocols. The details of dPeerSim being used for

studying maintenance protocols is included in the third paper. These papers provide the contents for the first part of Chapter 7.

# CHAPTER 2

# BACKGROUND AND GAP ANALYSIS

This chapter discusses how trust in general and trust based specifically on reputation are important for P2P environments, and how the latter comprises reputation metrics and feedback mechanisms. It outlines the problems faced by reputation metrics and feedback mechanisms in building a reliable trust system. The chapter sets the scene from which Chapter 3 extends an existing reputation metric. It links the work done in Chapter 4, 6 and 7, which are the secure mechanisms for detecting misbehavior using hardware based security, to the context of improving the feedback mechanism.

It should be stressed again that this chapter does not serve as the complete literature background for this thesis. The rest of the literature survey is distributed over the following chapters. This chapter starts with the background of structured P2P systems. The literature on trust and reputation is presented next. The final section discusses how the problems being addressed in this thesis are related to the others.

## 2.1 Structured P2P and Its Applications

The common understanding of P2P systems characterizes them as networks of computers (which could be referred to as peers or nodes) interacting directly with each other without centralized servers. This concept, however, fails to identify file-sharing systems like Gnutella [42], eDonkey [55] and Bittorrent [22] as P2P, because in this thesis, I define

P2P systems as networks of peers with following properties:

1. *Autonomous*. Peers are not under control of a central authority. They can join and leave the system at will.

2. *Heterogeneous*. The system can accommodate peers of different types, i.e. different operating systems, network connections, etc.

3. *Most traffic passes between peers*. A certain degree of centralization is allowed. Nevertheless, peers mainly interact with each other. This accounts for most of the system traffic. In P2P file-sharing applications, for example, bootstrapping servers are commonly used to help new peers join the systems.

P2P systems are *overlay* networks, because they run on top of another network (the Internet, for example). In the remaining, the term P2P, P2P *overlays*, P2P *networks*, P2P *systems*, P2P *infrastructures* are used interchangeably to refer to systems satisfying the three properties listed above.

An essential part of a P2P system is the routing protocol that implements the search (or the lookup) operation for a piece of data. Two types of P2P infrastructures exist in the literature: *unstructured* and *structured*. In the following, a brief survey of unstructured P2P is presented for completeness and comparison with structured P2P.

### 2.1.1 Unstructured

File-sharing applications such as Limewire [42], eDonkey [55], and Bittorrent [22] are built on top of unstructured P2P. This infrastructure has evolved through three generations.

#### 2.1.1.1 First generation — Napster

There exists a centralized indexing server that facilitates the sharing of data among peers. Search queries for a file, for example, are forwarded to the server. If the file exists, the server then responds with the contact information of the peers having the file.

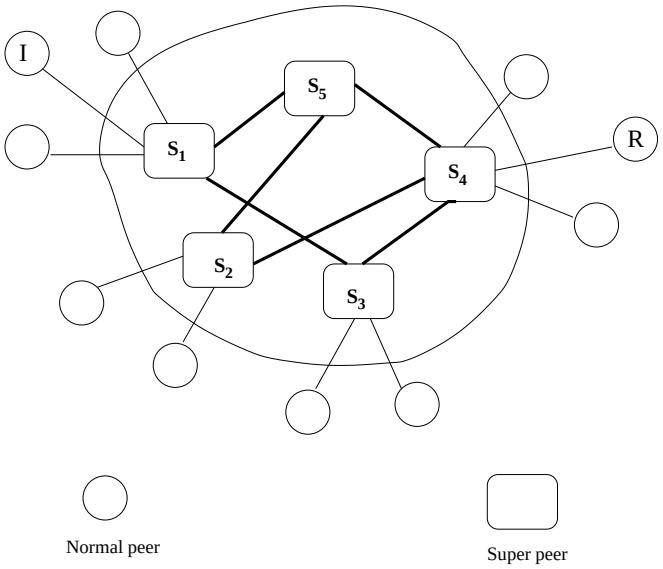



Figure 2.1.1: Gnutella architecture

#### 2.1.1.2 Second generation — Gnutella

The centralized index server is removed. A number of powerful peers (ones with high bandwidth, high up-time) form a network of superpeers. Ordinary peers connect and publish their shared data to at least one superpeer. Ordinary nodes forward their search queries to their superpeers. The superpeers then forward the queries to all of its neighbors, consequently flooding the network. For example, in Figure 2.1.1, peer $I$ searching for a file residing at peer $R$ first forwards its query $S_1$. The query is then broadcast to $S_3$, $S_5$ and finally to $S_4$ which returns the contact information of $R$.

#### 2.1.1.3 Third generation — Bittorrent

In the previous systems, peers from a single network to share their files. In Bittorrent [22], each file is associated with an independent network of peers interested in sharing the same file, called *swarms*. Each swarm has a centralized *tracker* that supplies information of the sharing peers and the data being shared. Bittorrent is a system of *multiple networks with one file per network*, as contrast to Napster and Gnutella — systems of *one network with multiple files*. The Bittorrent protocol consists of two key elements. First, the *rarest-first piece selection strategy* is used by a peer to select and download the rarest piece of data

in the swarm. Second, an uploading peer adopts the *tit for tat peer selection strategy* to select and upload data to the peer from which it has downloaded the most. This strategy provides a robust incentive for peers to share data, rather than to free-ride the network [57], a major problem in the early file-sharing systems [90].

#### 2.1.1.4 Summary

In unstructured P2P, a peer can be a neighbor of any other peer in the network. Therefore, the overlay topology approximates a random graph. In the early generation, the lookup is implemented by broadcasting the queries, which is unscalable. Limiting the number of hops that the queries propagate seems to mitigate this problem, but introduces nondeterminism. In particular, a negative result only means that the searched data may still exist but cannot be reached within the given number of hops.

### 2.1.2 Structured

Structured P2P systems are designed to address the weakness in the lookup mechanism of the early unstructured P2P. Searching in structured P2P is deterministic and more scalable. This comes at the expense of the system's robustness under frequent *churn* — the process of node joining and leaving the network. A number of structured overlays exist. The most popular ones can be divided into groups of different topologies: *ring*, *Plaxton graph*, *d-dimensional torus*, *etc.*.

In a structured P2P system, let $\mathcal{P}$ and $\mathcal{D}$ be the sets of peers and data in the system. A non-collision hash function is used to map each member of $\mathcal{P}$ and $\mathcal{D}$ to a distinct value in $\mathcal{I}$. Unless stated otherwise, $\mathcal{I} = [0, 2^m)$ where $m$ is the security parameter with the typical value of 160. A peer is uniquely identified by a *peer ID*. A data is uniquely associated with a *key*. The values of peer IDs and data keys come from the same set $\mathcal{I}$. A data $k$ is stored at the node called the *destination node* or *root node* of $k$. Let

$$root : \mathcal{D} \rightarrow \mathcal{P}$$

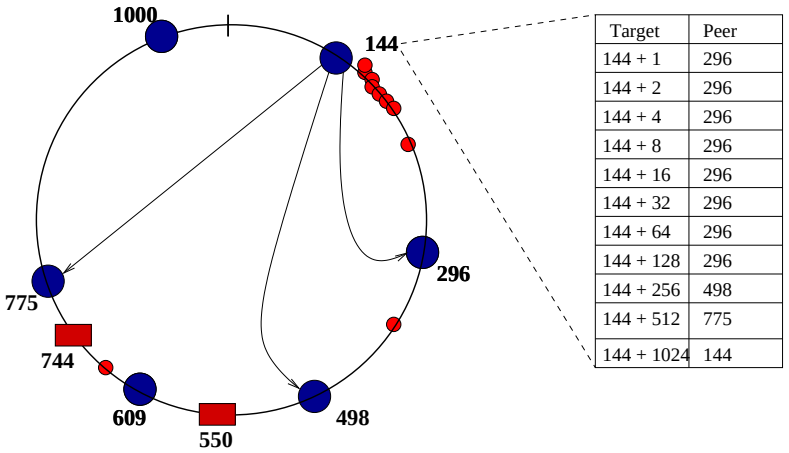



Figure 2.1.2: Example of a Chord overlay. $ID = [0, 2^{10})$, $\mathcal{P} = \{144, 296, 498, 609, 775, 1000\}$ and $D = \{550, 744\}$. The small circles represent the IDs that are used to construct the finger tables of peer 144.

be the function returning the root node of a given key in the set of nodes currently in the network. Structured overlays differ in the function *root* and the protocols that resolve the root nodes.

#### 2.1.2.1 Ring topology — Chord

Chord [88], one of the first structured P2P overlays, is well known for its simplicity and efficiency. In Chord, the ID space wraps around to make an ID ring. Each Chord node connects to the peer immediate on its left (called the predecessor) and another immediate on its right (called the successor). In Figure 2.1.2, the predecessor and successor of peer 144 are 1000 and 296 respectively.

Let $succ(k)$ be the function returning the peer immediate on the right of $k$ in the ID ring. For example, both $succ(300)$ and $succ(400)$ returns peer 498, whereas $succ(250)$ returns peer 296. In Chord:

$$root = succ$$

In Figure 2.1.2, data 744 is stored at peer 775. For efficient routing, every node maintains connections to other neighbors in a *finger table* of size $m$, in addition to the predecessor and successor. For a node $p$, the $i^{th}$ $(1 \leqslant i \leqslant m)$ entry in its finger table, $finger[i]$, is

defined as $succ(p \oplus 2^{i-1})$, where $\oplus$ is the addition operation in *modulo* $2^m$. In the above example, the $4^{th}$ and $9^{th}$ finger of peer 144 are peer 296 and 498 respectively.

To search for the destination node of $k$, i.e. to resolve $succ(k)$, the searching peer forwards its query to the neighbor (its successor, predecessor or a node in its finger table) that is furthest from it but is still on the left of $k$. This procedure is repeated at the new node until the current node is the closest on the left of $k$. In this case, the current node's successor is returned. In Figure 2.1.2, the routing path from node 144 for $k = 744$ is $144 \rightarrow 498 \rightarrow 609$. Finally, node 775 is returned as the destination of $k$. With $N$ nodes in the overlay, the average number of routing hop per query is $\frac{1}{2}.log_2(N)$.

When a new peer $p$ joins the network, it uses an existing node to initialize its successor and predecessor, namely $succ(p)$ and $succ(p)$'s predecessor. It then initializes its finger entries and notifies its existence to the other nodes that should have $p$ in their finger tables. When leaving gracefully, $p$ notifies the relevant neighbors, so that they can update their routing states. On average, the number of messages needed to maintain the network for a churn event is $O(logN)$.

#### 2.1.2.2 Plaxton mesh — Pastry

In Pastry [82], $root(k)$ returns the node whose ID is numerically closest to $k$ (if two nodes have the same distance to $k$, the one with smaller ID is returned). Data keys and peers IDs are are represented in base $2^b$ for a value of $b$ (usually $b = 4$). Pastry nodes form a Plaxton mesh topology [75], in which each node is the neighbor to ones sharing similar prefix with its own ID. In particular, the routing table contains $log_{2^b}N$ rows, where $N$ is the number of peers in the network. Each row has $2^b - 1$ entries. The entry at row $i$ ($1 \leqslant i \leqslant log_{2^b}N$), column $j$ ($1 \leqslant j \leqslant 2^b$) is a peer whose ID has the same first $(i-1)$ digits as the current node and its $i^{th}$ digit is $(j-1)$.

Figure 2.1.3 shows the routing table of node 10233102 in the Pastry overlay with $b = 2$. In addition to a routing table, each node also maintains a list of $l$ peers closest to its left and its right (on the circular $\mathcal{I}$ space), called the *leafset*. In the example in Figure 2.1.3,

| Leafset | | | |
|---|---|---|---|
| 10233000 | 10233001 | 10233033 | 10233021 |
| 10233120 | 10233122 | 10233230 | 10233232 |

| Routing Table | | | |
|---|---|---|---|
| 02212102 | **10233102** | 22301203 | 31203203 |
| **10233102** | 11301233 | 12230203 | 13021022 |
| 10031203 | 10132102 | **10233102** | 10323302 |
| 10200230 | 10211302 | 10222302 | **10233102** |
| 10230322 | 10231000 | 10232121 | **10233102** |
| 10233001 | **10233102** | 10233232 | |
| **10233102** | | 10233120 | |
| | | **10233102** | |

Figure 2.1.3: Routing neighbors of the Pastry node 10233102. This Pastry overlay uses $b = 2$ and $l = 8$. Therefore, all peers' IDs are in base 4, and each peer has 2*4=8 neighbors in its leafset

the leafset size is 8 and it contains 4 peers closest to the left and 4 peers closest to the right of 10233102, sorted by their distances to 10233102. The entry at row 3 column 3, for instance, contains peer 1022302, because it shares the prefix 102 (of size 3) with the current node, and its $4^{th}$ digit is 2. Some entries are empty and some refer to the current node.

The lookup protocol for the root node of a key $k$ consists of two steps. The query is first forwarded to the neighbor that is lexicographically closest to $k$. This neighbor is selected from the routing table as the node sharing the longest prefix with $k$. This step is repeated until the query arrives at a node whose leafset's range covers $k$. In this case, the node in the leafset (including the current node) that is numerically closest to $k$ is returned. The average number of hops per lookup operation is $O(logN)$, which is more efficient than Chord in practice since logarithm of base $2^b$ is used.

A new node joins the network by first asking a bootstrapping node to search for the root node of its own ID. The routing states can be initialized using most of the information from the other nodes in the routing path. The newly joined node then notifies its neighbors of its existence. When a node leaves the network gracefully, it also notifies the relevant neighbors. On average, the number of messages exchanged during a

churn event is $O(logN)$.

#### 2.1.2.3 XOR space — Kademlia

Kademlia [66] is a structured overlay based on XOR metric. The *XOR distance* between two value $i$ and $j$ in $\mathcal{I}$ is defined as $(i\ XOR\ j)$. $root(k)$ returns the node having the smallest XOR distance to $k$ (if two nodes have the same XOR distance to $k$, the one with smaller ID is returned). Each Kademlia node maintains $m$ $\kappa$-*buckets*. The $i^{th}$ bucket $(1 \leqslant i \leqslant m)$ contains $\kappa$ nodes whose XOR distance from the current node is in the range $[2^{i-1}, 2^i)$. For instance, with $m = 3$, the $2^{nd}$ 2-bucket of node 1110 can include node 1010, the $3^{rd}$ 2-bucket can contain node 0111 and 0001.

The lookup process is done iteratively and is highly parallel. For a key $k$, the searching peer forwards the query to the neighbors that are closest to $k$. Lists of peers that are closer to $k$ are returned. The query is then sent to nodes in these lists. This process is repeated until no closer node can be found. On average, the number of peers visited peer lookup operation is $O(logN)$. The joining and leaving process in Kademlia are very similar to those in Pastry. In particular, the number of maintenance messages exchanged during a churn event is also $O(logN)$.

#### 2.1.2.4 Multi-dimensional space — CAN

In previous systems, peers IDs and data keys are from the range $[0, 2^m)$. In Content Addressable Network (CANs) [77], they come from a $d$-dimensional torus. A peer ID is uniquely associated with a region, a key is uniquely mapped to a point in that space. The destination node of $k$ is the peer whose region contains $k$.

A peer's ID can be represented by the vector $((l_0, h_0), ..., (l_{d-1}, h_{d-1}))$ where $l_i, h_i$ is the low and high value in the $i^{th}$ axis. This region might be split when a new peer joins and merged with another region when another peer leaves. The peer ID can, as a consequence, change as the network evolves. The routing table of a node $p$ contains $O(d)$ nodes whose regions are adjacent to $p$'s. The lookup process involves forwarding

| Overlay | Topology | Routing state | Hop counts | Maintenance |
|---|---|---|---|---|
| Chord | Ring | $O(logN)$ | $O(logN)$ | $O(logN)$ |
| Pastry, Tapestry | Plaxton mesh | $O(logN)$ | $O(logN)$ | $O(logN)$ |
| Kademlia | XOR space | $O(logN)$ | $O(logN)$ | $O(logN)$ |
| Koorde | de Bruijn graph | 2 to $O(logN)$ | $O(logN)$ to $O(logN/loglogN)$ | - |
| CAN | d-dimensional torus | $O(d)$ | $O(d.N^{1/d})$ | $O(d)$ |

Table 2.1.1: Summary of structured P2P overlays

the query to adjacent regions until reaching the region containing the search key. For a system with $N$ nodes, the lookup operation takes $O(d.N^{1/d})$ steps on average. During churn, only $O(d)$ neighbors need to be informed.

#### 2.1.2.5 Other overlays

Tapestry [106] is similar to Pastry regarding the topology and searching protocol. In Tapestry, a node having data $k$ can *publish* this information by replicating it at all the nodes in the routing path from it to root node of $k$. A search for $k$ is likely to encounter a replica before reaching the root node.

Koorde [52] extends on Chord and uses a *de Bruijn* graph to achieve efficient lookup. It allows the number of neighbors per node to be tuned from 2 to $O(logN)$ to achieve hop counts ranging from $O(logN)$ to $O(logN/loglogN)$. An important theoretical result that comes out of Koorde is that:

> An $n$-node system with maximum degree $d$ requires at least $log_d n - 1$ routing hops in the worst case and $log_d n - O(1)$ on average

#### 2.1.2.6 Summary

Table 2.1.1 summarizes the properties of the previously described overlays.

### 2.1.3 Application of Structured P2P Overlays

Because of their decentralization, scalability and deterministic lookup mechanisms, structured P2P overlays have been used to build a wide range of large-scale distributed systems.

#### 2.1.3.1 Content delivery — Trackerless Bittorrent

In the original Bittorrent systems, there exist centralized servers providing tracker services for many swarms, such as The Pirate Bay [93] and Mininova [69]. The trackerless Bittorrent protocol [22] is proposed to eliminate such single points of failure. Peers form a structured overlay, namely Kademlia. Each peer is also a tracker server. A file being shared is mapped to a key $k$. The tracker service for file $k$ is provided by the root node of $k$.

#### 2.1.3.2 Application-level multicast

Multicast routing can be implemented at the software-layer. Such an application-level multicast infrastructure requires no hardware support, and it can be carefully designed to approximate the performance of the IP-level solutions. Structured P2P overlays have been used to support large multicast groups with low communication overheads.

Scribe [17] is built on top of Pastry. A multicast group is identified by an unique key $k$. The root node of $k$ becomes the rendezvous point, to which the multicast source sends its data to be broadcast to the members of the group. When a node $p$ wishes to join a group $\mathcal{G}$, it routes a Join message to the rendezvous point. Nodes in the routing path that have not seen messages belong to group $\mathcal{G}$ register an entry indicating that $p$ belongs to $\mathcal{G}$. The routing of the Join message stops when it arrives at a node that has registered an entry for $\mathcal{G}$. The data being broadcast is first transferred to the rendezvous node, which then sends the data to all the members of the group that it knows. Experiments showed that the median and maximal delay penalty per group are 1.7 and 4.26 respectively.

#### 2.1.3.3 Communication infrastructure

Voice-over-IP (VoIP) protocols also benefit from structured overlays. P2PSIP [14] implements the Session Initiation Protocol (SIP), an important part of the VoIP protocol, over the Chord overlay. VoIP clients form a Chord overlay, each is given an unique name, which is mapped to a key $k$. The connection details of the client, such as its IP address, are stored at the root node of $k$. When the details change, an update is sent to the root node. When another peer wishes to establish a connection with the client, it uses the latest information at the root node of $k$.

#### 2.1.3.4 Botnets

Botnets are networks of compromised machines (or bots) under the control of a single attacker. They are mostly used for sending spam. Holz *et al.* [49] have reverse-engineered Stormworm [86], a large and popular botnet, and discovered that bots were organized into a Kademlia overlay. The control command is uniquely identified with a key $k$ that is based on the current date and a random value known to all bots. The command is published at the root node of $k$. Periodically, bots fetch new commands (with the newly generated keys) and execute them, which could result in new commands being published.

#### 2.1.3.5 Others

A number of P2P network storage systems based on structured overlays have been proposed. PAST [81], for example, is a persistent global storage system built on top of Pastry. PAST supports only immutable files. It relies on the diversity of nodes (both geographically and politically) and the replication mechanisms to ensure high availability for data.

The Internet Indirection Infrastructure (I3) [87] used Chord to build a global rendezvous-based communication infrastructure. In particular, it supported unicast, multicast, anycast routing abstractions and host mobility. Application-level multicast and VoIP appli-

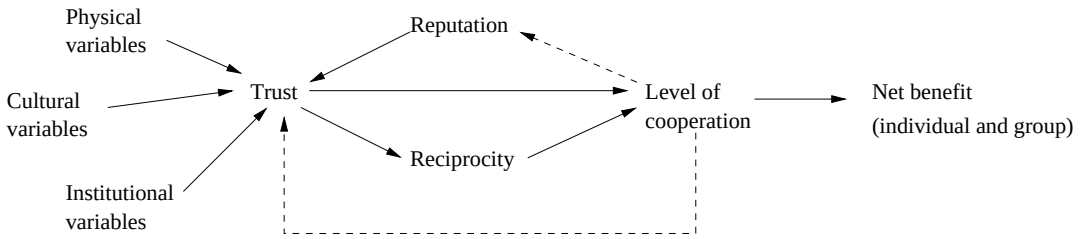



Figure 2.2.1: The core relationships between reciprocity, reputation and trust. Ostrom *et al.* [72]

cations could be built on top of this general infrastructure.

## 2.2 Trust and Reputation

Trust is a social phenomenon emerging from human interaction. Trust is important to society because it increases and reinforces the level of reciprocity and cooperation, as depicted in Figure 2.2.1. It is therefore not a coincidence that the concept of trust has been studied extensively in social sciences. This thesis uses the following definition by Hardin [46] (Chapter 2, page 11):

> To say we trust you means we believe you have the *right intentions* toward us and that you are *competent* to do what we trust you to do

From this definition, trust is *subjective* and *in relation to a given task*. It is important to understand the relationship between *trustworthiness* (and its adjective: *trustworthy*) and trust. Many authors in the literature failed to recognize the differences between these terms and labeled their works as about trust, even though were in fact about trustworthiness. Roughly speaking, the decision to trust depends on the assessment of trustworthiness (and perhaps other factors that are specific to the context in which the decision is made). While trust is a boolean-valued relation, there are degrees of trustworthiness. Once we have evaluated how trustworthy another person is, we have the knowledge to trust or distrust. Hardin went as far as arguing that trust is cognitive and therefore is not a choice.

A well known conception of trustworthiness is the *encapsulated interest* conception [46], in which one's trustworthiness is measured by one's interest or incentive to be trustworthy in the relevant context. Reputation, defined as record of the past deeds ([92], page 71), fits in this model because one with good reputation has the incentive to sustain that reputation, i.e. to behave in a trustworthy way.

### 2.2.1 Trust in computer security

In computer security, works in authentication implicitly make use of the notion of trust. However, they are not concerned with the question of how trust is formed or how trustworthiness is computed. Yahalom *et al.* [104] proposed three trust classes: *direct trust*, *indirect trust* and *recommendation trust*. In distributed environments where one client might not know another directly, a set of rules was proposed to derive indirect trust from the other classes of trust.

Maurer [65] extended the work in [104] to design a distributed public key infrastructure (PKI). Given a certificate graph and the trustworthiness values that peers give to each other, a set of derivation rules was used to determine the authenticity of a public key. This approach is similar to that in the Pretty Good Privacy (PGP) system. The PGP system represents trustworthiness as *fuzzy* values, as opposed to integer values in [65].

### 2.2.2 Trust based on Reputation

In sociology, reputation is studied in the encapsulated interest model [46]. A person with good reputation has the incentive to behave in certain ways that maintain the reputation. This is because a good reputation encourages others to choose him for future transactions that could be in his interest. This is important in society, because people increasingly have interactions with those with whom they might never have the occasion to interact again. In many cases, one can turn to the law of contracts to be protected from uncooperative behavior. Often enough, however, there is no such safeguard, especially for small-scale

exchanges.

As shown in Figure 2.2.1, reputation is an important factor contributing to trustworthiness. The other variables are physical, cultural and institutional variables. In online settings, both physical and cultural information are not available. Furthermore, many exchanges online are not protected by institutional safeguards. As the consequence, reputation is the main indicator of trustworthiness.

### 2.2.3 Model of Reputation

In computer science, research on computational models of reputation extends on the following model. Let $\mathcal{F}_i$ be the set of *feedback* given to agent $i$. Let $\mathcal{F}$ be the set of all feedback in the system, i.e. $\mathcal{F} = \bigcup_i \mathcal{F}_i$. The reputation value $R_i$ of $i$ is computed by applying a *reputation metric* (or *reputation function*) $\mu_i$ to the set of feedback. More specifically:

$$R_i = \mu_i(\mathcal{F}) \tag{2.2.1}$$

Xiong *et al.* [103], investigated a simple reputation system, in which feedback is the amount of satisfaction regarding the transaction. The reputation metric was a simple averaging function, i.e. $\mu_i(\mathcal{F}) = \frac{\sum \mathcal{F}_i}{|\mathcal{F}_i|}$. More discussion of other works on reputation models can be found in Section 2.3.

Another formal model of reputation exists and is based on game theory. Game theorists first noticed the reputation effects on repeated games [54] and how it drives the system's equilibrium to more socially desirable outcomes. Later studies focused on the nature of feedback in different environments. *Perfect information* (or *imperfect information*, conversely) means feedback is truthful (or contains noises). *Public monitoring* (or conversely *private monitoring*) indicates that the entire feedback history is (or is not) publicly available.

Aberer [4] analyzed the usefulness for game-theoretic models of reputations in online settings. The author argued that current computation models of reputation systems are

ad-hoc and rather intuitive. Game-theoretic models can predict how different strategies result in different equilibria, meaning that one can tune the system variables in order to achieve the desirable equilibrium (maximal overall profit, for example). In addition, one can also explain and predict how the users do exchanges with each other, for in the game theoretical framework players are rational utility maximizer and cannot do better than playing the equilibrium strategy. Nevertheless, the limitation of the game-theoretical approach lies in its complexity. In terms of feedback, online environments need to be modeled as imperfect information and private monitoring. Furthermore, investigating non-trivial aggregation strategies in this framework is difficult. Finally, many argues against how well game-theoretic models approximate the real world, especially since research has shown that humans do not normally act as rational economic agents.

## 2.3 Setting the Scene (or Gap Analysis)

This section discusses current issues in structured P2P and motivates the use of trust in P2P. The problems with establishing trust in P2P are discussed next, which servers as the motivation to the work done in this thesis.

### 2.3.1 Current Issues with Structured P2P

*Robustness under churn.* The main advantage of structured P2P overlays compared with its unstructured counterpart is the lookup efficiency. The trade-off, however, is the cost to maintain the rigid topology under churn. The metric used to assess this cost is the number of messages exchanged during the churn event, called the maintenance messages.

The original Chord protocol requires $O(logN)$ maintenance messages. In the extended version, each Chord node periodically runs the *stabilization* protocol, which prioritizes fixing successor and predecessor pointers over fingers. Liben-Nowell *et al.* [62] constructed a mathematical model of the evolution of Chord networks. A *half-life* is defined as the minimum period of time after that the network's size is doubled ($N$ new nodes joined) or

halved ($\frac{N}{2}$ nodes departed). Their model showed that the cost for correct routing (queries arrive at the correct destination) is $\Omega(logN)$ messages per half-life. For efficient routing (i.e. $O(logN)$ hops), the cost amounts to $\Omega(log^2N)$ rounds of stabilization protocols per half-life.

There exists a considerable number of works in the literature focusing on improving the performance of structured P2P under frequent churn. The node availability model consists of:

- *Session time*: the period during which the node stays in the network.
- *Life time*: the sum of the node's session times (the node can have multiple sessions, i.e. joining and then leaving for many times).

A high *churn rate* implies a short session time. Unstructured P2P applications perform reasonably well under high churn rate [58]. In structured overlays, however, the lookup performance drops significantly. Rhea *et al.* [79] showed via simulations that the lookup success falls well under 45% when the session time is as small as 23 minutes. Bamboo [79] is an improved version of Pastry and more resilient to churn. In particular, Bamboo introduces three extensions to Pastry. First, the *push-pull* mechanism for maintaining the leafset removes the need for newly joined node to announce itself to its leafset neighbors. Second, the global and local *tuning* protocol are executed periodically, which helps a node find better entries for its routing tables. Third, each node actively probes its neighbors to gather their up-to-date response time. If a neighbor has been unresponsive for longer than expected, it is considered as having departed and subsequent lookups can be forwarded to an alternative neighbor. Under the non-concurrent churn model with the median session time as short as 1.5 minutes, experiments showed that lookup success always exceeded 95%.

*Load balancing.* The load balancing problem in P2P systems arises because of the nodes' heterogeneity and the non-uniform distribution of data. The use of secure hash functions to generate data keys and nodes' IDs does not sufficiently address the problem,

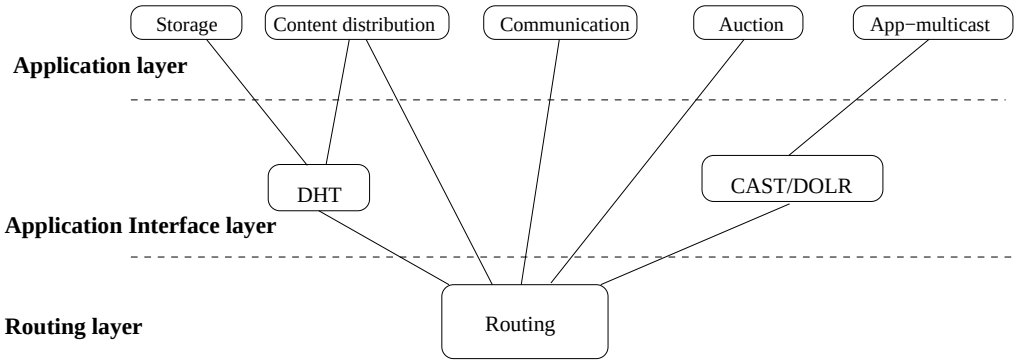



Figure 2.3.1: Abstraction levels of P2P applications. Adapted from [23]

because the hash function does not produce perfectly random values, especially when the number of of nodes and data is small.

One approach to mitigate this problem is to split an ordinary node into multiple *virtual nodes*. Godfrey *et al.* [43] used virtual nodes as load balancing units which can be transferred from a heavily loaded peer to a lightly loaded one. Simulation results showed that in a system with the utilization rate of 0.9, up to 99.9% of the nodes are still underloaded, and the overhead of virtual nodes transferring is small. However, under high churn rate, this overhead could become significantly large. Byers *et al.* [15] proposed another approach, in which at least two hash functions are used to map data to different places in the ID space. Among all the possible root nodes, the data is stored at the node with the lightest load. Compared with the approach using virtual nodes, experimental results showed that the system load can be more balanced.

*Security.* Security is an inherent problem in P2P, because of peers being autonomous and the lack of centralized authorities that police peers' interactions. As illustrated in Figure 2.3.1, a P2P system can be categorized as belonging to different groups corresponding to these layers. The hierarchy in these layers suggests that to achieve security at one layer, the lower layers need to be secure first.

1. *Routing layer*: The overlays discussed in Section 2.1 belong to this layer. They implement the lookup protocol that returns the root node of a given search key. An adversary targeting this layer can do the following:

+ No routing: routing queries are dropped. As the consequence, the network is partitioned into regions that cannot reach each other.

+ Redirection: queries are forwarded to other malicious nodes. They could also be forwarded to innocent nodes, in an attempt at a DDoS attack.

+ Impersonating the root node: claims to be root node in order to control traffic relating to the data key.

2. *Application interface layer*: this layer builds upon the lookup mechanism to implement the **store(k,data)** operation, which stores the tuple (k,data) at a node. In many systems, this tuple is placed at the root node of $k$. In some others, it is replicated at several nodes along the routing paths towards the root node of $k$. An adversary can compromise this layer by dropping the tuple at the destination nodes that it controls. It can also tamper with the data.

3. *Application layer*: this layer consists of application-specific protocols that utilize the lower levels. For instance, P2P communication systems make use of the routing layer, whereas P2P storage systems use the application interface layer to store data in a scalable way. The adversary at this layer attacks application-dependent properties, such as quality of services, availability, etc. For example, it could target replication, access control mechanisms or corrupt the application data.

Douceur [33] described another attack that is inherent to P2P environments, called the Sybil attack, in which the adversary gains multiple identities and appears as different peers in the network. Without proper identity management, Sybil attacks present real threats to P2P applications. In particular, the adversary can introduce Sybils in order to perform an attack at different places in the network, consequently increase the damage caused by the attack. For example, when attacking a system at the routing layer by dropping queries, the adversary can introduce additional Sybils so that more queries are routed through and subsequently dropped by it.

### 2.3.2 How Trust Could Help

The security problems described previously arise because honest nodes interact with the adversarial ones, which is what expected in a decentralized environment where peers do not know each other in advance. This model of interaction resembles human interactions in real-world societies. Trust in human society allows one to interact confidently with another stranger. It also helps one to identify and avoid the untrustworthy. In P2P, similarly, having trust encourages cooperation between nodes, which is important to systems relying on cooperation such as content delivery and network storage systems. More importantly, in a trust-enabled environment, an honest peer could identify the adversarial (i.e. untrustworthy) ones and avoid interacting with them. As a result, the adversaries are isolated and eliminated from the network.

In the context of security, untrustworthy peers are defined as ones with the intentions to cause harms to the system. This definition can be extended to include unintentional but frequent activities that disrupt the normal operations of the system. Thus, the role of trust could go beyond security and into the realm of robustness. For example, peers with high churn rates that leave ungracefully can be considered as untrustworthy neighbors, because they induce high maintenance overhead and an increase in lookup failure. In this case, being trustworthy means being available and reliable for a long period of time.

Another implication of having trust is that it opens up new areas of applications for P2P. E-commerce systems, for instance, are currently based on n-tier client-server architecture because of the lack of a trust enabling infrastructure among the participants. Similarly, current implementations of anonymity systems (Onion routing [91], for example) are unscalable and inefficient, for they have to consider the interactions between honest and dishonest nodes.

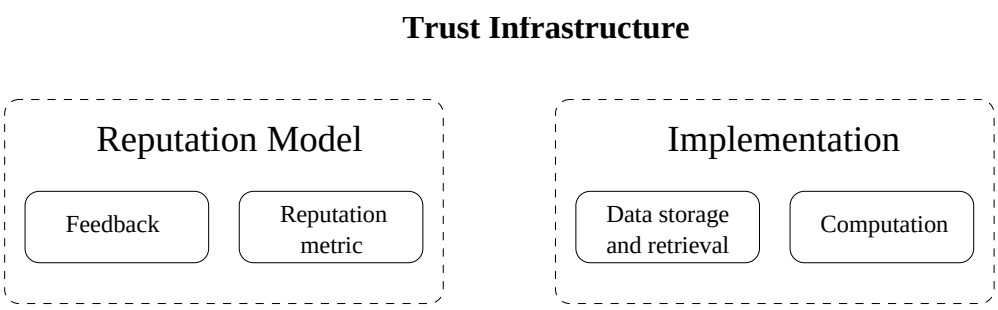



Figure 2.3.2: Trust system

### 2.3.3 Reliable Trust System for P2P

As discussed earlier in Section 2.2, reputation is a main indicator of trustworthiness in online settings, such as P2P. A trust system (or trust infrastructure) for P2P provides the knowledge necessary for peers to evaluate the trustworthiness of each other and then make trust decisions. The knowledge, in particular, is in the form of reputation values (or scores). The design of such an infrastructure must have the following properties:

- *Reliability.* Peers do not make incorrect trust decisions even if the system is given inaccurate information. This property can be translated into the requirement that the system generates *intuitive* reputation scores and is resilient to manipulation. In other words, well behaved peers are given high scores and badly behaved ones are given low scores. In addition, badly behaved peers are prevented from colluding with each other in order to boost their reputations.

- *Efficiency.* The system is efficient with respect to its response time and the resources (memory, bandwidth, etc.) it consumes.

Figure 2.3.2 depicts two components of a reputation-based trust system for P2P. First, the component marked as Reputation Model represents the computational model for reputation, which consists of a feedback model and a reputation metric. The feedback model defines the process (or the mechanism) by which one peer can give ratings to another. It also describes the nature and values of the ratings. The reputation metric specifies how to combine feedback into reputation scores. Peers are then sorted in descending

order of their reputation values. The positions in the ordered list are called the *reputation ranks*, i.e. peers with lower ranks have higher scores. The reliability property of the trust system is concerned with the reputation model. More specifically, it requires the feedback mechanism to elicit feedback which are then combined by the reputation metric to yield intuitive reputation scores which are also resilient against manipulation. Second, the details of how to manage feedback and compute reputation values are included in the component marked as Implementation. The efficiency property of the trust system is concerned with this component. It states that the implementation of the trust system is efficient with respect to its response time and its required resources.

#### 2.3.3.1 Reputation metric

Aberer *et al.* [3] proposed a reputation metric that is the inverse of the number of complaints (the feedback is in the form of complaints). In [103], the metric is the average function over feedback. These functions are vulnerable to manipulation. For instance, a malicious peer can alter its reputation ranks by giving fake feedback to others. Finding better reputation functions has been an active area of research. In the literature, the advanced metrics make use of the system's *trust graph* $\mathcal{G}(V, E, W)$ that is constructed from the set of feedback. Each peer is represented by a graph node in $V$. An edge $(i, j) \in E$ indicates that $i$ and $j$ have been involved in a transaction. The weight $w(i, j)$ of the edge $(i, j)$ represents the rating that $i$ gives to $j$ regarding the past transactions. Cheng *et al.* [20] divides the advanced metrics into two groups:

- *Symmetric:* these functions return global, objective reputation scores, i.e. the values that are agreed by all peers. They are dependent of the structure of the trust graph and are invariant under node-relabeling. The global reputation is useful to newly joined peers, ones that have not had interaction with others. Furthermore, for peers that have never interacted with each other and there is no path connecting them in the trust graph, the global reputation is the only information that could help them evaluate the trustworthiness of each other. The PageRank metric [73] is the most

well-known example of this group.

- *Asymmetric:* these functions produce reputation values that are relative to a set of *root nodes*, i.e. reputation is subjective with respect to the root nodes. When the set of root nodes includes all the peers in the network, these functions become symmetric. In Advogato [59], the reputation of a node $j$ relative to a root node $i$ is the maximum flow from $i$ to $j$ in the trust graph. Other reputation metrics in this group are based on personalized PageRank, k-step Markov chain and shortest path [102]. They are computationally more expensive than the symmetric ones. Additionally, they cannot evaluate the reputation value for a node that is unreachable from the root nodes.

This thesis chooses to investigate the PageRank metric. One reason for it is that in P2P environments, it is common for the trust graphs to be partitioned into regions that cannot reach each other, therefore a symmetric reputation function like PageRank is more suitable than an asymmetric one. Furthermore, computing PageRank is less expensive than the other asymmetric functions. In particular, the main operation in the computation of PageRank is matrix multiplication, whose complexity is $O(N^2)$ where $N$ is the size of the network. On the contrary, the cost of a network flow algorithm, for example, is $O(N^3)$.

All existing metrics are subject to manipulation attacks in which peers collude or use their Sybils to boost their reputations. In the collusion attack, peers would participate only if they could all have their reputations increased as the result. Zhang *et al.* [105] showed that in PageRank, a group of nodes could collude and change the trust graph structure in between them to boost their PageRank scores by up to $\frac{1}{\epsilon}$ (where $\epsilon$ is the *jumping factor*, whose typical value is 0.15).

The Sybil attack, in which a node attempts to boost its reputation at the expense of its Sybils, is one of the focuses of this thesis. Cheng *et al.* [20] showed that under certain conditions, the asymmetric functions are immune from Sybil manipulation. The PageRank function was shown to have a degree of resilience against such attack. The

next chapter extends this result and describes how the PageRank's resilience in P2P trust graphs can be improved.

#### 2.3.3.2 Feedback model

In building a reputation-based trust system, the role of feedback model is often understated. It has as an important role as that of reputation metrics, because it describes the elements necessary to construct trust graphs upon which the metrics operate. A number of problems faced by the feedback mechanisms have just recently been identified.

First, the lack of incentives for peers to leave feedback could hinder the trust system. Insufficient feedback results in an incomplete trust graph that is likely to consist of many unconnected regions. In addition, theory predicts that a minimum amount of feedback is required for the reputation effects to induce cooperation (Bakos *et al.* [8]). Dellarocas [25] explained that economic theory predicts voluntary feedback is under-provided, because feedback is considered as public goods, and once provided, every peer will costlessly benefit from it. Avery *et al.* [6] proposed a mechanism in which early evaluators were paid for their information. The authors concluded that out of the three desirable properties of their mechanism: voluntary participation, no price discrimination, and budget balance, only two can be achieved. In [63], Liu *et al.* designed a system that reinforces the relationship between the feedback incentive and the Beta reputation metric. The Beta metric is quite simple and therefore subject to manipulation. Nevertheless, the work is interesting because on the one hand, the incentive induces sufficient feedback so that the Beta function can yield meaningful and robust values, and on the other hand, a peer's *recommendation reputation* is used by another to determine whether to provide its feedback information to that peer. Therefore, the peer has an incentive to leave more feedback in order to maintain and increase its recommendation reputation.

Second, inducing honest feedback is also important, as most game-theoretic models of reputation rely on truthful information. Dellarocas [26] argued that dishonest feedback causes a noisy environment and drives the system to a less efficient equilibrium (with a sub-

optimal cooperation level). The incentive mechanism in [63] was shown, via simulations, to also encourage peers to be truth tellers.

Third, negative feedback is commonly used in the literature to represent *distrust*. This type of feedback plays the role of stabilizing the system, as it punishes peers that misbehave. Studies of eBay [53] showed that the fear of being retaliated discourages users from giving negative feedback. A possible solution suggested by Dellarocas *et al.* [28] involves keeping the feedback secret until after the period for leaving feedback expires. The number of reputation metrics in the literature supporting negative feedback, even when sufficiently elicited, is limited. Chapter 3 first argues for a clearer semantic of this type of feedback in P2P, and then proposes a new reputation function based on PageRank that supports negative feedback.

Fourth, existing works assume that a peer is always able to evaluate its transactions with others before giving feedback. It is trivial in a market application, for example, for a buyer to assess the quality of the goods and then leave the appropriate feedback. This thesis presents two scenarios in structured P2P environments where it is difficult to check if the outcome of a transaction is good or bad. In particular, the lookup protocol in a structured overlay returns a node claiming to be the root node of the search key, but it is problematic for the searching peer to know if what returned is the correct root node in the current configuration of the network. Chapter 4, 6 and 7 discuss the mechanisms that enable the peer to verify results of the routing protocol. These mechanisms equip the peer with the necessary knowledge so that it can leave feedback (regarding the quality of routing) to the nodes involved in the routing path.

#### 2.3.3.3 Implementation

Despite not being in the scope of this thesis, the efficiency requirements arising when implementing trust systems in P2P could influence the re-designing of the reputation model. The first challenge is to be able to store and retrieve feedback efficiently. Next, the computation of reputation metrics must be done in a reasonable time frame. In the

literature, progress has been made in addressing these obstacles:

1. **Storage and retrieval**. Aberer *et al.* [3] and Li *et al.* [103] used P-Grid, a structured P2P overlay, to store feedback which are in the form of complaints and transaction's ratings. The object key used to identify a peer's feedback is defined as the hash of the peer's ID.

2. **Computation**. Even in a centralized environment, the computation of advanced reputation metrics, which involves large matrix multiplications or other exponentially complex algorithms, is expensive. In P2P, it becomes even more problematic. First, the data is scattered across the network. Second, the computation needs to be split into independent sub-tasks which are carried out at different peers and later aggregated to yield the overall result. Parreira *et al.* [74] proposed a decentralized evaluation of the PageRank metric. Each peer is responsible for a small portion of the network, and it runs the PageRank algorithm locally. When two peers meet, the local PageRank results are merged to produce a closer approximation of the global result. Experiments via simulation showed that with a large number of meetings among peers, the decentralized PageRank metric closely approximates the centralized version.

# CHAPTER 3

# REPUTATION METRICS

This chapter extends an existing reputation metric, namely PageRank, by improving its resilience against Sybil manipulation, and enabling support for negative feedback. Its contents come from a joint work with Mark Ryan. It has been discussed in Chapter 2 that a Sybil-resilient reputation function with support for negative feedback is a building block for a reliable trust system. Section 3.1 presents the computational model of reputation in more detail. It is followed by the introduction of a new reputation metric based on PageRank, called Cluster-based PageRank (or CPR). CPR is designed to be more resilient to Sybil manipulation than the original PageRank, which is confirmed by experimental analysis. Finally, Section 3.3 describes another reputation metric based on PageRank, called PRN, that takes negative feedback into account. This new metric is shown to meet a set of desirable properties.

## 3.1 Formal Computational Model of Reputation

As highlighted in Section 2.2, a computational model of reputation consists of two main components: a feedback mechanism and a reputation metric. This section describes in more detail the model in a P2P environment.

### 3.1.1 System Model

The building blocks for evaluating reputation are user feedback in the form of *transaction ratings*. A transaction could be an upload/download operation in a content distribution applications, or a buying/selling interaction in an online market. The feedback mechanisms in P2P determine how peers giving ratings to each other. The result is a set of ratings used by the reputation model.

More formally, a P2P system is modeled as the tuple $(\mathcal{P}, \mathcal{T}, init, resp, Rt)$ where:

- $\mathcal{P} = \{0, 1, 2, .., (n-1)\}$ is the set of peers. It is the same as the set of peers first introduced in Section 2.1, except that the peers' IDs are in the range $[0, n)$.
- $\mathcal{T} = \{t_0, t_2, ..., t_{s-1}\}$ is the set of transactions.
- $init : \mathcal{T} \rightarrow \mathcal{P}$ and $resp : \mathcal{T} \rightarrow \mathcal{P}$ are functions returning the *initiator* and *responder* of a transaction respectively. In a content distribution application, for example, the initiator and responder of a transaction is the peer downloading and the peer uploading the file respectively. Suppose $p_i, p_r \in \mathcal{P}$ completed a transaction $t$, then $init(t) = p_i$ and $resp(t) = p_r$.
- $Rt : \mathcal{T} \rightarrow \mathbb{R}$ is a function returning the rating that one peer gives to another. More precisely, $Rt(t)$ is the rating that $init(t)$ gives to $resp(t)$.

Notice that this model does not stop a peer from leaving ratings to itself. In particular, there could exists $t$ such that $init(t) = resp(t)$. From this model, the set of feedback $\mathcal{F}$ mentioned in Section 2.2 can be defined as follows:

$$\mathcal{F} \quad = \quad \{(t, resp(t), Rt(t) \,|\, t \in \mathcal{T}\}$$

### 3.1.2 Trust Graph

A trust graph $\mathcal{G}(V, E, W)$ is constructed from the system model, in which:

- $V$ is the set of nodes, which is the same as $\mathcal{P}$.

- $E$ is the set of edges. For any $i, j \in V$, $(i, j) \in E$ if there exists $t \in \mathcal{T}$ such that $init(t) = i \wedge resp(t) = j$. This means an edge exists between two nodes if they have engaged in a transaction with each other. The direction of the edge is from the initiator to the responder.

- $W : E \to \mathbb{R}$ returns the weight of a given edge. $W((i, j))$, for example, represents the average rating that $i$ has given to $j$ over the past transactions with $j$. In particular, denote $\mathcal{T}_{ij} = \{t \in \mathcal{T} \mid init(t) = i \ \wedge \ resp(t) = j\}$, then:

$$W((i,j)) = \frac{\sum\limits_{(t \in \mathcal{T}_{ij})} Rt(t)}{|\mathcal{T}_{ij}|}$$

If the graph $\mathcal{G}(V, E, W)$ is *undirected* if it satisfies the condition:

$$\forall i, j \in V \bullet (i, j) \in E \ \Leftrightarrow \ (j, i) \in E$$

Otherwise, it is *directed*.

### 3.1.3 Reputation Metric

A reputation metric uses the trust graph to evaluate peers' reputations. Let $R_i$ be the reputation of peer $i$. The metric $\mu$ is applied to $\mathcal{G}$ and returns reputations for all the peers. In other words:

$$\mu(\mathcal{G}) = (R_1, R_2, ..., R_n)$$

The simple reputation function in [103], which returns the averages of the feedback as reputations, can be represented in this model as follows. Let $E_i^+, W_i^+$ be the set of edges to $i$ and ratings given to $i$ respectively. More precisely, $E_i^+ = \{(j, i) \mid (j, i) \in E\}$ and

$W_i^+ = \{W(e) \,|\, e \in E_i^+\}$. Then,

$$\mu(\mathcal{G}) = (\mu_1(\mathcal{G}), \mu_2(\mathcal{G}), ..., \mu_n(\mathcal{G}))$$

where

$$\mu_i(\mathcal{G}) = \frac{\sum_{x \in W_i^+} x}{|W_i^+|} \tag{3.1.1}$$

More advanced reputation functions are divided into two groups: symmetric and asymmetric. They differ mainly in the semantics of $R_i$: in the former, $R_i$ is agreed by all peers, whereas in the latter, $R_i$ is subjective to a set of root nodes. As explained in Section 2.3, this thesis chooses to investigate a symmetric function, namely PageRank.

## 3.2 Sybil-resilient Metric

A reputation metric is defined as being *Sybil-resilient* if it does not allow the adversary to arbitrarily increase its reputation by introducing a large number of Sybils. In other words, the reputation scores produced by the metric are not easily manipulated by the Sybil strategies executed by the adversary.

As an example, the reputation metric introduced in [103] and formalized in Section 3.1.3 is not Sybil-resilient. Consider an adversary that has been behaving badly and as the consequence been given bad ratings by other peers. Equation 3.1.1 indicates that the adversary's reputation is likely to be low. However, the adversary can cheat the system by introducing a large number of Sybils, then faking transactions between the Sybils and itself. The outcomes of those transactions are positive ratings being given to the adversary. As the result, the adversary reputation, as computed by Equation 3.1.1, increases proportionally with the number of Sybils. This means the reputation metric is subject to manipulation.

### 3.2.1 PageRank reputation metric

PageRank [73] is the most well-known symmetric reputation function. It is used by Google for ranking web pages. PageRank can be explained intuitively by the random surfer model. In particular, the random surfer starts at a random node $i$. With a probability $(1-\epsilon)$ where $0 < \epsilon < 1$, she moves to a neighbor node $j$ with the choice of $j$ determined by $W((i,j))$. With the probability of $\epsilon$, she jumps to another randomly chosen node. $\epsilon$ is called the *jumping factor*, whose typical value used in PageRank is 0.15. The portion of time the surfer would spend on a node if she continued the walk forever is interpreted as the node's reputation. This fits well with the intuition that a good web page is linked from other good web pages and therefore is visited more often.

Formally, the surfer's random walk is modeled as a Markov process. Let $\overrightarrow{R}$ be the vector containing $R_1, R_2, .., R_n$. $\overrightarrow{R}^T$, the transposed vector of $\overrightarrow{R}$, is the stationary vector of the Markov process (the eigen vector with the corresponding eigenvalue of 1). More specifically, let $E_i^-, W_i^-$ be the set of edges from $i$ and ratings given by $i$. In other words, $E_i^- = \{(i,j) \,|\, (i,j) \in E\}$ and $W_i^- = \{W(e) \,|\, e \in E_i^-\}$. $T$ is the $n \times n$ transition matrix derived from $\mathcal{G}$, in which:

$$T_{i,j} = \begin{cases} \frac{W((i,j))}{\sum W_i^-} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

PageRank assumes $\mathcal{G}$ contains no negative edge, so that $W((i,j)) \geqslant 0$ for all $(i,j) \in E$. $\overrightarrow{R}$ is the solution of the following equation:

$$\overrightarrow{R}^T = (1-\epsilon).\overrightarrow{R}^T.T + \epsilon.\overrightarrow{1}^T \tag{3.2.1}$$

in which $\overrightarrow{1}$ is the identity vector with $n$ elements. The reputation rank of peer $i$ is the position of $R_i$ in the list obtained from sorting the elements in $\overrightarrow{R}$ in descending order. Equation 3.2.1 always has a solution. In practice, the *iterative method* is used to compute $\overrightarrow{R}$ that is very close to the real solution. More specifically, $\overrightarrow{R}$ is refined over many

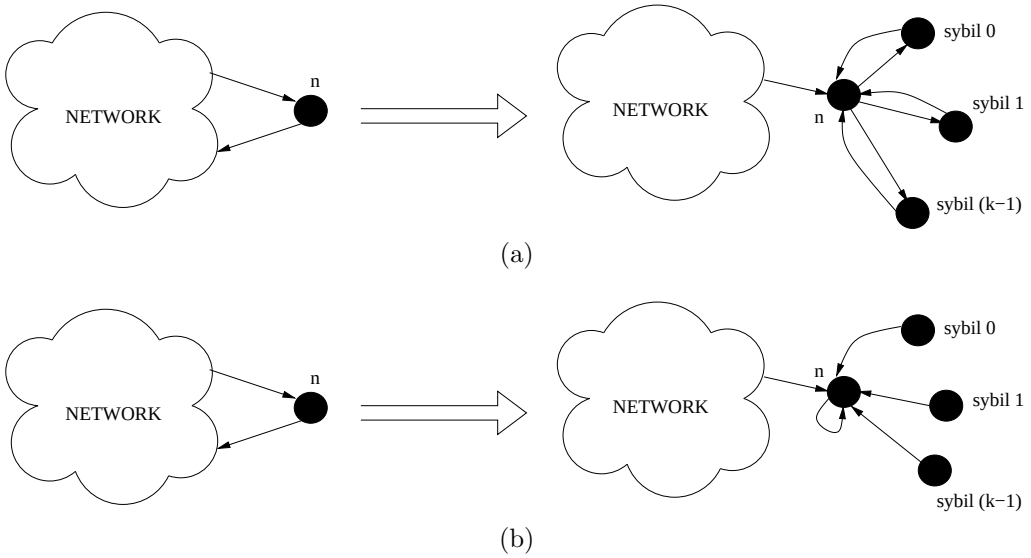



Figure 3.2.1: Two Sybil strategies for manipulating PageRank, in which the attacker $n$ introduces $k$ new Sybils and uses them in different ways. (a) is considered in [21]

iterations. Let $\overrightarrow{R}^{it}$ be the value of $\overrightarrow{R}$ at iteration $it$. One starts with the vector $\overrightarrow{R}^{0}$ whose elements have the same value of $\frac{1}{n}$. Next, for $it \geqslant 0$:

$$(\overrightarrow{R}^{T})^{it+1} = (1-\epsilon).(\overrightarrow{R}^{T})^{it}.T + \epsilon.\overrightarrow{1}^{T}$$

This process is repeated until it converges, i.e. the difference between $\overrightarrow{R}^{it+1}$ and $\overrightarrow{R}^{it}$ is insignificantly small for a value of $it$.

### 3.2.2 Sybil Manipulation in the Original PageRank

In the following, the resilience of PageRank against Sybil manipulation is discussed. Unless otherwise stated, the edges in $\mathcal{G}$ are assumed to have the same weight, i.e. $W((i,j)) = 1$ for all $(i,j) \in E$.

Cheng *et al.* [21] provided a formal analysis of the PageRank 's resilience against Sybil manipulation. The Sybil strategy considered in [21] is depicted in Figure 3.2.1a. Peer $n$ is an attacker. It introduces $k$ new Sybils into the network, each Sybil has a link to and from $n$. The attacker also removes its outgoing links to the other nodes in the network. Cheng

*et al.* concluded that $n$'s new reputation value, $R'_n$, satisfies the following inequality:

$$R_n + k.\frac{1-\epsilon}{2-\epsilon} \leqslant R'_n \leqslant \frac{R_n}{\epsilon.(2-\epsilon)} + k.\frac{1-\epsilon}{2-\epsilon} \tag{3.2.2}$$

This result shows that the attacker can boost its reputation considerably by increasing the number of Sybils. The improvement is more noticeable if the $R_n$ is small. In particular, with $\epsilon = 0.15$, $1 + 0.46\frac{k}{R_n} \leqslant \frac{R'_n}{R_n} \leqslant 3.6 + 0.46\frac{k}{R_n}$. If $R_n = 0.3$, for example, the adversary needs only 1 Sybil to double its reputation score.

[21] only considered one simple Sybil strategy. In the following, a new investigation on the vulnerability of PageRank against a different Sybil strategy is presented. In this strategy, as depicted in Figure 3.2.1b, $n$ adds a link to itself and removes its links to the other nodes, including the Sybils. Compared with the strategy in Figure 3.2.1a, this allows the attacker to keep the random walk at the node for longer, which according to PageRank's surfer model could result in higher reputation.

**Theorem 3.2.1.** *Let $R'_n$ be the reputation value of node $n$ after executing the Sybil strategy in Figure 3.2.1b. Then:*

$$(2-\epsilon).R_n + (1-\epsilon).k \leqslant R'_n \leqslant \frac{R_n}{\epsilon} + (1-\epsilon).k$$

*Proof.* The detailed proof can be found in Appendix A. Briefly, in addition to the trust graph $\mathcal{G}, \mathcal{G}'$ representing the network before and after the attack, another graph $\mathcal{G}''$ is considered. In the network represented by $\mathcal{G}''$, $n$ removes its outlinks and creates a link to itself without adding Sybils to the network. Let $R''_n$ be the reputation value of $n$ in $\mathcal{G}''$, then $R'_n = R''_n + (1-\epsilon).k$, and for all $i \leqslant n$, $R_n \geqslant R''_n$ □

Theorem 3.2.1 indicates that this Sybil strategy has a different impact on the original

PageRank. More specifically, because $k, R_n \geqslant 0$ and $0 < \epsilon < 1$, the following is true:

$$(2-\epsilon).R_n + (1-\epsilon).k \;>\; R_n + k.\frac{1-\epsilon}{2-\epsilon} \tag{3.2.3}$$

$$\frac{R_n}{\epsilon} + (1-\epsilon).k \;>\; \frac{R_n}{\epsilon.(2-\epsilon)} + k.\frac{1-\epsilon}{2-\epsilon} \tag{3.2.4}$$

These inequalities imply that compared with the Sybil strategy depicted in Figure 3.2.1a, the attacker using this strategy can achieve larger minimum and maximum gain in reputation. In other words, this Sybil strategy is more effective at boosting the attacker's reputation. If $R_n = 0.3$ and $k = 1$, for example, the adversary using the strategy depicted in Figure 3.2.1a and Figure 3.2.1b can boost its reputation by the factor of at least 2.5 and 4.25 respectively.

### 3.2.3 PageRank with Undirected Trust Graph

Most of the graphs used for studying PageRank are directed. A trust graph derived from a typical P2P system, however, is likely to be undirected. In a P2P environment, two node $i$ and $j$ are likely to have more than one transactions with each other. In some transactions, $i$ is the initiator and $j$ is the responder; in others, the roles reverse. In a P2P-based online market application, for example, $i$ might buy an item from $j$ in one transaction, and it might sell another item to $j$ in another transaction. In trackerless Bittorrent, $i$ might use the tracker service run at $j$ to download a file in one transaction, it might provide the tracker service for $j$ in another transaction regarding another file. In structured P2P routing, $i$ might forward a query to $j$ in one transaction, it might forward another query from $j$ in another transaction. Assume these scenarios and that the initiator always leaves a feedback to the responder for each transaction, it follows that $(i, j) \in E$ and $(j, i) \in E$ for all $i, j$. In other words, $\mathcal{G}$ is undirected. In practice, two peers do not always rate each other the same. This section assumes, however, for the sake of simplicity, that $W((i, j)) = W((j, i)) = 1$.

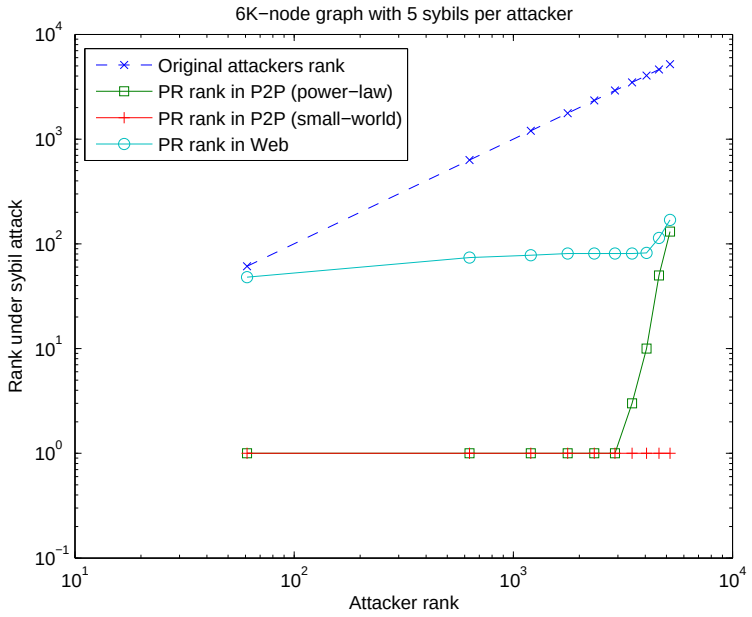



Figure 3.2.2: Sybil effect on Web vs P2P graph

For directed graph, simulation results in [21] showed that on average, the attacker can increase its reputation rank by more than an order of magnitude using as few as 5 Sybils. Figure 3.2.2 illustrates the effect of Sybils on undirected (or P2P) trust graphs, and compares it with the effect on directed (or Web) trust graphs. The graphs used for the experiments consists of over 6000 nodes. The Web graph is derived from actual Web links published in the academic Web link database [85]. In a simulated P2P graph, the node's degree follows a power-law distribution. The other generated P2P graph has the small-world property. The Sybil strategy consists of 5 Sybils. The attacker creates extra links to and from its Sybils. In the Web graph, the reputation gains agree with the result in [21]. In the P2P graphs, the improvement in reputations is significantly higher. In particular, in the small-world graph, the attacker seems to always obtain the highest rank after introducing the Sybils, regardless of its original rank. An observation that could help explain this is that the distribution of reputation values in the P2P graphs has a smaller variance than in the Web graph. In undirected graphs, a link from a low-ranked node to a high-ranked one requires another link going in the opposite direction, which is not the case in Web graphs where it is more difficult to be linked from a high-ranked

Web page. This could have been the reason behind the small variance in the distribution of the reputation values, which means a little increase in the reputation value leads to a noticeable boost in rank.

### 3.2.4 Cluster-Based PageRank (CPR)

Figure 3.2.2 shows that PageRank is very sensitive to Sybil manipulation in P2P graphs. This subsection presents a modification to PageRank that alleviates the problem.

The intuition behind the new reputation function is that the attacker creates a graph region consisting of itself and the Sybils, in which it creates extra links in order to trap the random walk process inside the region for as long as possible. As a result, the region would have many internal links and only a small number of links with the rest of the network. In other words, it forms a typical *graph cluster*. The denser the region, the more likely it is controlled by the attacker. Therefore, the clustering information of a graph can be helpful in detecting and dealing with the Sybil attacks.

In the new metric, called Cluster-based PageRank (or CPR), when the random walk arrives at a highly clustered region, it adjusts the jumping factor $\epsilon$ to quickly jump out of the region, and therefore avoid being trapped. This modification would not significantly affect the rank of an originally high-ranked node, because the node would have links from many parts of the graph, therefore the random walk process would still end up spending more time at the node.

Given the trust graph $\mathcal{G}(V, E, W)$ [1], CPR comprises the following:

- A clustering algorithm $CA$ that partitions $\mathcal{G}$ into a set of clusters $C = \{C_0, C_1, ..., C_{c-1}\}$. A function $CL : V \rightarrow C$ can be derived from $CA$ that maps a node to the cluster it belongs.

- $dens : C \rightarrow \mathbb{R}$ returns the density value of a given cluster in $C$. Cluster $C_i$ is consider denser than $C_j$ if $dens(C_i) > dens(C_j)$.

[1] $W((i, j)) = W((j, i)) = 1$ for any $(i, j) \in E$

- $dens2ep : \mathbb{R} \rightarrow (0,1)$ maps a density value to a value of the jumping factor. In addition, $dens2ep$ satisfies the following condition:

$$\forall d_1, d_2 \in \mathbb{R} \bullet \ dens2ep(d1) > dens2ep(d2) \ \Leftrightarrow d1 > d2$$

In CPR, each node has a personalized jumping factor, as opposed to every node having the same global value $\epsilon$. More precisely, the personalized jumping factor $\epsilon_i$ of node $i$ is defined as:

$$\epsilon_i = dens2ep(dens(CL(i)))$$

It can be seen that two nodes belonging to the same cluster have the same jumping factor. Moreover, if the cluster containing $i$ is denser than that containing $j$, then $\epsilon_i > \epsilon_j$, which means the random walk is less likely to follow the graph links from $i$ than from $j$. Let $M$ be a $n \times n$ matrix defined as follows:

$$M = \begin{bmatrix} (1-\epsilon_1).T_{1,1} & (1-\epsilon_2).T_{1,2} & \cdots & (1-\epsilon_n).T_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ (1-\epsilon_1).T_{n,1} & (1-\epsilon_2).T_{n,2} & \cdots & (1-\epsilon_n).T_{n,n} \end{bmatrix} + \frac{1}{n}. \begin{bmatrix} \epsilon_1 & \epsilon_2 & \cdots & \epsilon_n \\ \vdots & \vdots & \ddots & \vdots \\ \epsilon_1 & \epsilon_2 & \cdots & \epsilon_n \end{bmatrix}$$

where $T_{i,j}$ is the element of the transition matrix $T$ defined in Section 3.2.1. The reputation vector $\overrightarrow{R}$ is defined as the stationary vector of the Markov process having $M$ as its transition matrix. Because $M$ is *ergodic*, i.e. stochastic and irreducible, $\overrightarrow{R}$ always exists, and it is the solution of the equation:

$$\overrightarrow{R}^T = \overrightarrow{R}^T.M$$

Using the iterative method as in PageRank, $R_i$ can be approximated by iterating the following:

$$R_i^{it+1} = \sum_j (1-\epsilon_j).M_{j,i} \cdot R_j^{it} + \frac{1}{n}. \sum_j \epsilon_j.R_j^{it} \tag{3.2.5}$$

To implement CPR, one needs to implement $CA$ and specify the $dens$ and $dens2ep$

functions. Graph clustering is a NP-complete problem. There are different approaches in finding good clusters in graphs: multi-level, hyper-graph, etc. In this thesis, a Markov-based clustering algorithm, called MCL [99], is used. MCL has been shown to produce good clusters out of large graphs. In MCL, a random walk starting from a node in a dense region tends to stay within the region for a long time before moving on to another region. Given a transition matrix $M$, the algorithm consists of two main operations:

- The expansion operation, $EXT_e(M)$, extends the current random walk to $e$ extra hops, i.e. $EXT_e(M) = M^e$
- The inflation operation with a parameter $lr$, $\Gamma_{lr}$, promotes the probability of jumping to a node in the local region: $\Gamma_{lr}(M) = M'$ where $M'$ is a matrix in which $M'_{i,j} = \frac{(M_{i,j})^{lr}}{\sum_{j'=0}^{n} (M_{j',j})^{lr}}$

These operations are applied alternately on $M$ until reaching the matrix $M'$ that is *doubly idempotent*, i.e. $EXT_e(M') = M'$ and $\Gamma_{lr}(M') = M'$. The matrix $M'$ is extremely sparse, and its structure is interpreted as consisting of independent clusters.

### 3.2.5 Experimental Study of CPR

#### 3.2.5.1 Setup

Using the results in [89, 107], P2P trust graphs are generated that resemble real P2P systems. In particular, small-world graphs are created using the Watts-Strogatz model [101], and have the *clustering coefficient* value of 0.018. Power-law graphs are created using the Eppstein model [34]. The sizes of these graphs vary from 1,000 to 50,000 nodes.

The attacker's initial rank varies from the top 1% to 95%. A number of Sybils, $nSybils$, as large as 5% of the total number of nodes, are introduced into the network. The attacker has links to and form the Sybils. The Sybils do not have links among them.

An open-source implementation of MCL [98] is used to derive clusters from the graphs. Let $iL(C_i)$ and $eL(C_i)$ be the functions returning the number of external and internal links

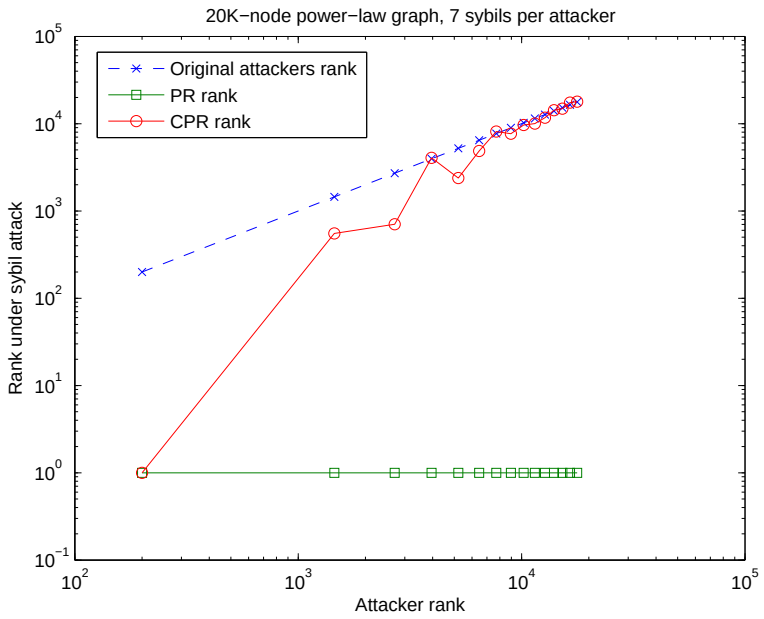



Figure 3.2.3: Sybil resilience under a power-law graph with 7 Sybils per attacker

of the given cluster, then $dens(C_i) = \frac{eL(C_i)}{2.iL(C_i)}$ and $dens2ep(x) = 0.15^x$. Finally, the value of $\epsilon$ is 0.15.

#### 3.2.5.2 Sybil resilience

Figure 3.2.3 and 3.2.4 illustrate the resilience of CPR against Sybil manipulation in comparison with the original PageRank. CPR shows significantly better resilience. In small-world graphs (Figure 3.2.4), the attacker's new ranks are always close to the initial ones. More interestingly, for most of the time, the attacker gets lower ranks than before, i.e. it gets punished. In power-law graphs (Figure 3.2.3), attacks caused by high-ranked nodes are hard to detect. But as the initial rank of the attacker declines, CPR becomes more and more effective in dealing with the attacks. More specifically, when the attacker's initial ranks are below the top 50%, its new ranks are close to the original ones, and sometimes event lower. Figure 3.2.5 shows the resilience of CPR when varying the number of Sybils. The attacker's initial rank is around $10,000$. Its new ranks are close to the initial one, even when it controls up to 1000 Sybils.

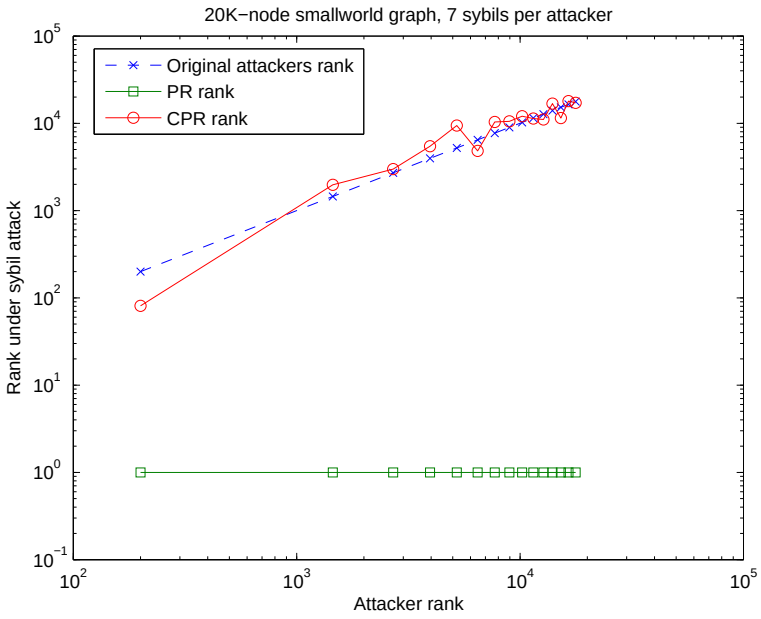



Figure 3.2.4: Sybil resilience under a small-world graph with 7 Sybils per attacker

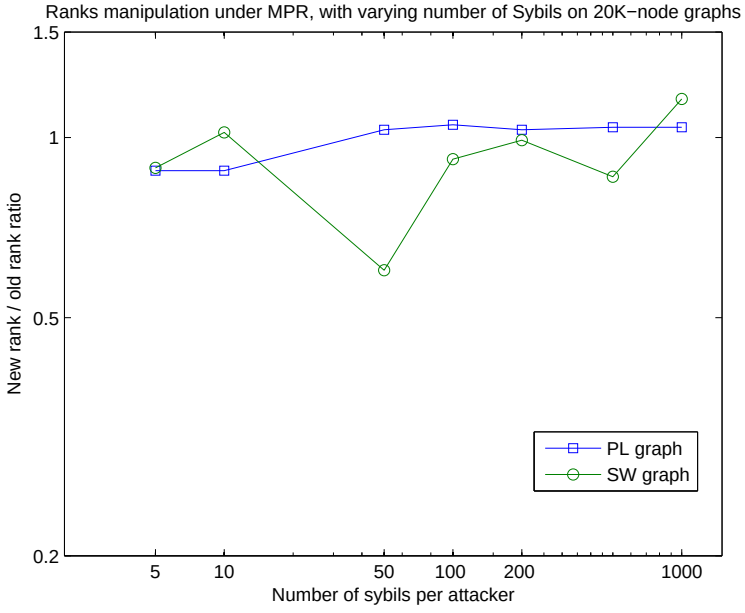



Figure 3.2.5: Resilience against Sybil attacks, with varying number of Sybils per attacker. The attacker's initial rank is around $10,000$.

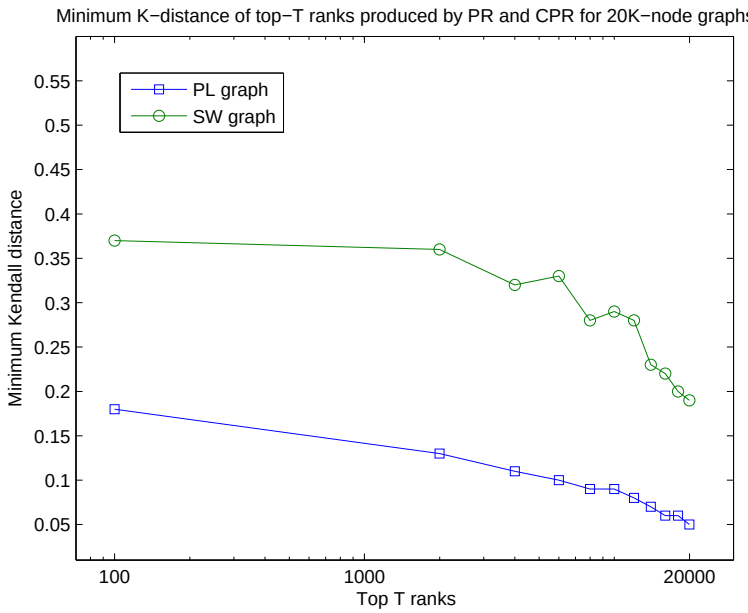



Figure 3.2.6: Minimum Kendall distance of top-T ranks produced by PageRank and by CPR

#### 3.2.5.3 Intuitiveness of CPR rankings

In the absence of Sybils, the intuitiveness of the reputations produced by CPR can be assessed by comparing them with what produced by PageRank. The similarity between the two rank lists is measured by the *normalized minimum Kendall distance* (or K-distance) [35] metric. Essentially, this metric quantifies the number of pairwise disagreement between two lists. The larger the distance, the more dissimilar the two lists are. As shown in Figure 3.2.6, the power-law and small-world graphs have different K-distances to the original PageRank. In both cases, however, the top-100 ranks produced by CPR are quite similar to those produced by PageRank. More specifically, the K-distance is always smaller than 0.4. The similarity increases as longer lists are compared. For the lists containing all the nodes, the K-distance is under 0.2 (and less than 0.05 in case of the small-world graph).

### 3.2.6 Related Work and Discussion

This section has discussed the impact of Sybils in the context of reputation metrics for P2P systems. It assumes the existence of Sybils in the system, then analyzes and proposes a method to mitigate the effect. Danezis *et al.* [24] and Castro *et al.* [16] studied the effect of Sybils in the context of P2P routing. [24] took advantage of the *bootstrap graph* during routing to avoid having a small set of nodes being predominantly present in all the query paths. In other words, it proposed to alleviate the impact of Sybils by diversifying the nodes used in routing. It assumed existing off-line relationships between nodes, and a node joins the network via another. During routing, the statistics of the nodes (in the bootstrap graph) that appear in the previous query paths is used to determine the next hop. [16] discussed two approaches that restrict the number of Sybils introduced into the network. One approach relies on a central authority (CA) to assign an unique ID to each participant. The other approach makes use of cryptographic puzzles during the assignment of node IDs, which only limits the rate at which Sybils can be introduced, but does not completely eliminate the attack.

This section has investigated PageRank's resilience against Sybil manipulation in undirected trust graphs. The result led to the design of a new reputation metric, namely CPR, that is based on graph clustering. Experimental analysis of the metric that uses MCL as the clustering algorithm suggests that CPR is intuitive and more resilient than PageRank in undirected graphs. Scalability is the weakness of MCL, for it involves many large-matrix multiplications. However, there exists a number of other clustering algorithms [50], since cluster analysis has been an active area of research. A comparative study of those algorithms in the context of CPR could be an interesting direction for the future work. In addition, the future work could investigate the effect of more complex *dens* and *dens*2*ep* functions on CPR.

Finally, the undirected graphs studied in this section have the common property that $W((i,j)) = W((j,i)) = 1$ for all $(i,j) \in E$. This is, to an extent, an oversimplification of the real P2P systems. In particular, the weights $W((i,j))$ and $W((j,i))$ may not be the

same, and either of them may be negative. When $W((i,j)) \neq W((j,i))$ but $W((i,j)) > 0$ and $W((j,i)) > 0$, it means $i$ and $j$ both rate each other positively, but they have different degrees of satisfaction with each other behavior. In this case, a possible hypothesis is that the result in Section 3.2.3 can be extrapolated, i.e. the Sybil effect would have a similar trend to what depicted in Figure 3.2.2. It would have been caused by the small variance in the distribution of the reputation values. More work is needed to confirm or invalidate this hypothesis. Finally, both PageRank and CPR do not consider the case when $W((i,j)) < 0$ or $W((j,i)) < 0$. The next section improves upon PageRank with added support for negative edges.

## 3.3 Support for Negative Feedback

Both PageRank and CPR are based on the Markov model. The transition matrices need to be strictly non-negative for the stationary vectors to exist. As the consequence, PageRank and CPR lack the support for negative feedback. This section first discusses the nature of feedback in reputation systems. It then argues for the importance of having negative feedback in the system, which then leads to the design of a new reputation metric based on PageRank, called PRN.

### 3.3.1 Negative Feedback is Different from Distrust

In the (computing) literature, the trust graph $\mathcal{G}(V, E, W)$ is commonly used in studying the notion of trust. In many works such as those from Maurer [65], Guha *et al.* [45] and Ziegler *et al.* [108], $W((i,j)) > 0$ indicates that $i$ trusts $j$ to the degree of $W((i,j))$. Interestingly, in [3], [39], [44] and [60], $W((i,j)) < 0$ is used to represent a low level of trust, which is then identified as *distrust*. These interpretations of the trust graph are misleading, for the following reasons.

First, to say we distrust you means that we believe you do *not* have the right intentions towards us or that you are not competent enough to do what we trust you to do. Distrust

includes both the absence of trust, which is different from having a low level of trust, and something more — suspicion [67].

Second, according to the definition of trust in Section 2.2, trust (and similarly distrust) is a boolean-valued relation. Therefore, it is inaccurate to represent this concept with real values.

Finally, assigning $W((i,j))$ with a value in $\{-1,1\}$ raises both philosophical and technical challenges:

- $W((i,j)) = 1$ or $W((i,j)) = -1$ indicates that $i$ either trust or distrust $j$, but not both. However, research in social sciences has shown that trust and distrust can co-exist. $i$ may trust $j$ in one task, but distrust it in another task. Even for the same task, $i$ may trust $j$ at one time, but distrust it at another time. It has also been shown that both trust and distrust are important to a functional society. Without distrust, one makes himself vulnerable to unnecessary risks when the trustee misbehaves. Without trust, one is deprived of the benefits that would have resulted from trusting. Luhmann [64] suggested that trust cannot exist without distrust, for trust and distrust simplify humans' process of making decisions. Trust reduces the complexity by compelling one to take actions that expose her to risks; distrust reduces the complexity by inducing her to take measures that reduce risks. McKnight *et al.* [67] argued that trust and distrust have distinct emotions. While trust is like a "docile zoo elephant munching on hay", distrust is like a "raging wild bull elephant protecting the herd from attack".

- The question concerning how distrust is propagated in a network is difficult to answer. Trust can be transitive, that is assuming $A$ trusts $B$ and $B$ trusts $C$, then it follows that $A$ trusts $C$. But the same cannot be said for distrust.

In this thesis, $W((i,j))$ is the feedback that $i$ gives to $j$, which represents the degree of trustworthiness of $j$ from the perspective of $i$. Being a rating, $W((i,j))$ can be given any value in $\mathbb{R}$. The decision to trust and distrust a peer can be made from inspecting

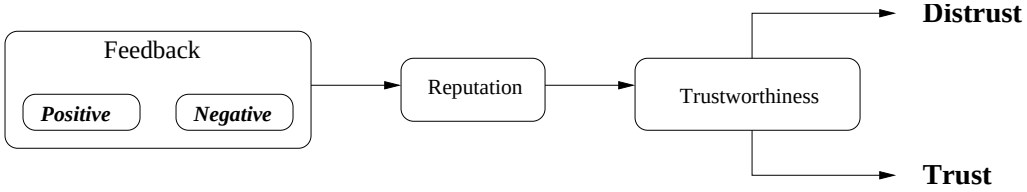



Figure 3.3.1: The relationship between feedback, reputation, trustworthiness, trust and distrust. Feedback is combined into reputation by using a reputation metric. Reputation can be used as a direct indicator of trustworthiness. The decision to trust or distrust is positively or negatively influenced by the degree of trustworthiness respectively.

the peer's trustworthiness. More specifically, $W((i,j)) > 0$ indicates a positive evaluation of $j$'s trustworthiness by $i$. When $i$ rates $j$ as untrustworthy, it assigns a negative value to $W((i,j))$. The feedback is combined (by a reputation function) to produce reputation values. As indicated in Section 2.2, a high reputation value implies a high level of trustworthiness. Benamati *et al.* [10] hypothesized that one's belief of another trustworthiness negatively influences her distrust decision. This hypothesis was later verified by the empirical data from the interactions between users and online banking systems. The relation between negative feedback and distrust is shown in Figure 3.3.1.

### 3.3.2 Why Negative Feedback?

This section argues that for reputation to be a reliable indicator of trustworthiness, at least two types of feedback: negative and positive are needed. While positive feedback is used for rewarding good behavior, negative feedback is a means of punishing bad behavior. The terms *negative* and *positive* are purely qualitative, they do not imply the numerical value of feedback. For example, in a binary feedback system like the one used by eBay, negative and positive feedback have the value $-1$ and 1 respectively. In a Amazon-like feedback system based on 1-5 star ratings, feedback with the value less than 3 may be considered as negative.

Consider a feedback system that supports only positive feedback. Translated into the trust graph $\mathcal{G}$, it means $(i,j) \in E$ and $W((i,j)) = 1$ if and only if $i$ had a satisfactory transaction with $j$. Transactions in which one party cheats are not represented in $\mathcal{G}$. The

reputation of node $i$ is therefore an indicator of $i$'s popularity among other peers, and is proportional to the number of good transactions that $i$ had with others. This value is clearly not an accurate reflection of $i$'s trustworthiness, because it does not take into account the bad behavior of $i$. For example, let $x$ be the number of positive feedback received by $i$ and $j$ from a same set of nodes. Assume that $x$ is reasonably large, and $i$ has cheated in $2.x$ transactions, while $j$ has never cheated. $i$ and $j$ would have the same reputation (at least when PageRank is used as the reputation metric), but $i$ is in fact less trustworthy than $j$.

Another implication of the hypothetical reputation system above is that it induces peers to cheat, because cheating goes unpunished and the peers are able to build up high reputation just by having many transactions. As a consequence, the level of distrust will increase, resulting in a low level of cooperation in the system. Negative feedback is the necessary extension, because it punishes peers that behaved badly. A reputation metric taking both positive and negative feedback into account is likely to produce a good indicator for trustworthiness, since peers having high reputations must not have been cheating frequently in the past. Furthermore, peers are discouraged from cheating in order to maintain their reputations. Dellarocas [27] showed analytically that a simple feedback mechanism with binary values (one for positive feedback, and another for negative feedback) can achieve maximum level of cooperation between traders in online environments. Interestingly, the effect of reputation on the price and level of cooperation cannot be improved with more complex feedback systems such as a 1-5 star rating system.

### 3.3.3 Reputation Metric with Negative Feedback

The previous section has explained the importance of (qualitatively) negative feedback. PageRank and CPR use numerically positive trust graphs, i.e. $0 < W((i,j)) \leqslant 1$ for all $(i,j) \in E$. The range $(0,1]$ can be split into two sets representing negative and positive feedback. For instance, let $0 < n_l, n_r, p_l, p_r \leqslant 1$ be the end points of the range $(n_l, n_r)$ and $(p_l, p_r)$. An edge $(i,j)$ is considered as representing a negative feedback

if $W((i,j)) \in (n_l, n_r)$. When $W((i,j)) \in (p_l, p_r)$, $(i,j)$ is considered as representing a positive feedback.

An advantage of this approach, splitting the range $(0,1]$ to represent both positive and negative feedback, is that PageRank can be used as it is. However, it is the interpretation of the results that poses problems. More specifically, PageRank has an interesting property that for any node $i$ and $j$, $E_i^+ \subseteq E_{i'}^+$ implies $R_i \leqslant R_{i'}$ [1]. Suppose $W((j,i')) \in (n_l, n_r)$ for all $(j,i') \in E_{i'}^+$, then the fact that $R_i \leqslant R_{i'}$ is not intuitive, because $i'$ has received more negative feedback and should be given lower reputation.

The new metric proposed in this section, called PRN (for PageRank with Negative feedback), uses numerically negative values to represent negative feedback. Specifically, the co-domain of $W$ is extended to the range $[-1,1]$. The main ideas behind PRN are as follows:

1. Similar to PageRank, $R_i$ is proportional to the reputations of the nodes that have links to $i$ and the values of the links. However, unlike in PageRank, only nodes with positive reputations can contribute to $i$'s reputation. The intuition is that a node with negative reputation should not be able to influence the reputations of others.

2. As in PageRank, node $i$ also gets a reputation value of $\frac{\epsilon}{n}$ from other nodes that do not have links to $i$.

Let $T$ and $\epsilon$ be the transition matrix and jumping factor defined in Section 3.2. Then, $R_i$ is a solution of the following equation:

$$R_i = (1-\epsilon).\sum_{R_j>0} T_{j,i}.R_j + \frac{\epsilon}{n} \tag{3.3.1}$$

If a solution for Equation 3.3.1 exists, I conjecture that it can be approximated using the iterative method similar to the one used in PageRank. More precisely, let $R_i^0$ be the

[1] This can be derived directly from Equation 3.2.1

initial value of $R_i$ which is chosen at random. Then for $it \geqslant 0$:

$$R_i^{it+1} = (1-\epsilon).\sum_{R_j^{it}>0} T_{j,i}.R_j^{it} + \frac{\epsilon}{n} \tag{3.3.2}$$

in which $R_i^{it}$ is the value of $R_i$ at the $it^{th}$ iteration. Let $t$ be the number of iterations after which the difference between $\overrightarrow{R}^t$ and $\overrightarrow{R}^{t+1}$ is insignificantly small, then $R_i^t$ can be used as $i$'s reputation.

### 3.3.4 Experimental Study

Experiments (via simulations) were carried out to evaluate five properties of PRN:

1. The iterative method for approximating $R_i$ converges. In particular, for a reasonably large value of $t$, the difference between $\overrightarrow{R}^t$ and $\overrightarrow{R}^{t+1}$ gets very small.
2. A node having links from others with high reputations will have high reputation.
3. Receiving negative feedback from others with positive reputations will reduce the node's reputation.
4. A node with low or negative reputation has small (or zero) impact on others' reputations.
5. Being resilient against Sybil manipulation.

#### 3.3.4.1 Experiments Setup

The trust graph $\mathcal{G}$ used in all the experiments contains $49,290$ nodes. It is derived from the Epinion data-set [94], which has almost $500,000$ users ratings. An user in Epinion is rated for her reviews about certain topics. The edges in $\mathcal{G}$ have values in $\{-1, 1\}$. The fraction of negative edges in $\mathcal{G}$, denoted as $rNEdges$, varies from 0.1 to 0.5.

The attacker is introduced to the network as a new node. The distributions of nodes having link from (and similarly to) the new node, called the *edge distributions*, have the

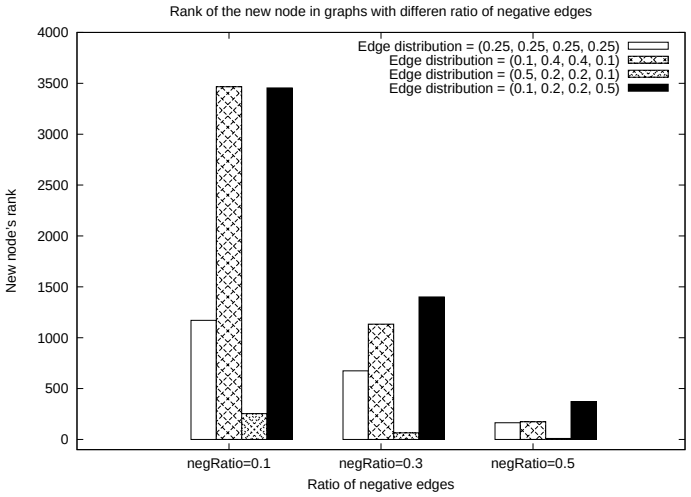



Figure 3.3.2: Ranks of the new node with 7 positive incoming edges

form of $(d_1, d_2, d_3, d_4)$. In particular, a fraction of $d_1$ nodes are from the top 25% nodes with highest ranks, $d_2$ from the next $26 - 50\%$, $d_3$ from the next $51 - 75\%$ and $d_4$ from the set of nodes ranked in the bottom 25%. The fraction of negative edges in the set of edges coming to and from the attacker is $rNEdges$.

A simple Sybil strategy is implemented, in which the number of Sybils varies from 3 to 10. The strategy is the same as the one depicted in Figure 3.2.1b, except the attacker does not remove its links to other nodes, nor does it create a link to itself.

Finally, as in PageRank and CPR, the value of $\epsilon$ is 0.15.

#### 3.3.4.2 Results and Analysis

First, the experiments suggest that the iterative method converges quickly (Property 1). Interestingly, the rate of convergence is slower than that of PageRank. However, as the number of iterations increases beyond 60, the differences between the subsequent runs become insignificantly small.

Figure 3.3.2 and 3.3.3 show the ranks of the new node having 7 and 15 incoming edges, while varying $nEdges$ and the edge distribution. The attacker's reputation is highest and lowest when the edge distribution is $(0.5, 0.2, 0.2, 0.1)$ and $(0.1, 0.2, 0.2, 0.5)$ respectively. In the former, most edges come from nodes with high reputations. In the latter, most edges

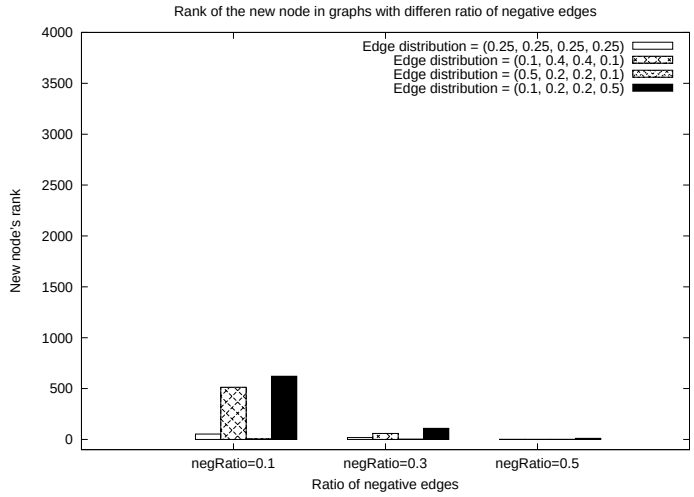



Figure 3.3.3: Ranks of the new node with 15 positive incoming edges

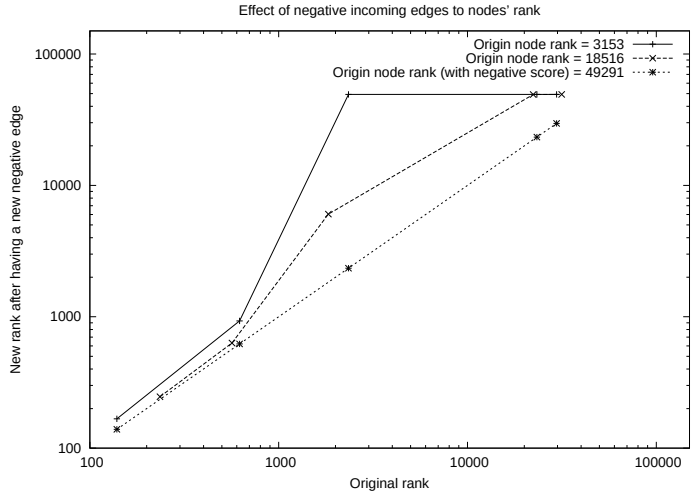



Figure 3.3.4: The effect of negative edges on reputations. $rNEdges = 0.1$

are from nodes with low reputations. These results suggest that a node receiving positive feedback from other nodes with high reputations also have high reputation (Property 2).

It is interesting to notice that the new node's rank increases with *rNEdges*, and with the number of incoming edges. The possible explanations are as follows. First, as *rNEdges* increases, more nodes are assigned negative reputations, and the variance in reputation values decreases. Second, adding more positive incoming links increases the reputation value, which subsequently improves the rank.

Figure 3.3.4 illustrates the effect of negative feedback on reputations. In the experiments, negative edges come from the attacker whose rank is 3153, 18516 or 49291. The

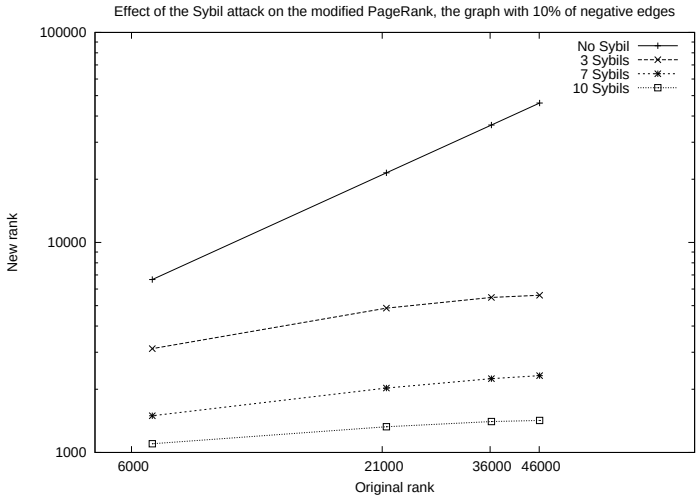



Figure 3.3.5: Effect of Sybils in the graph having negative edges

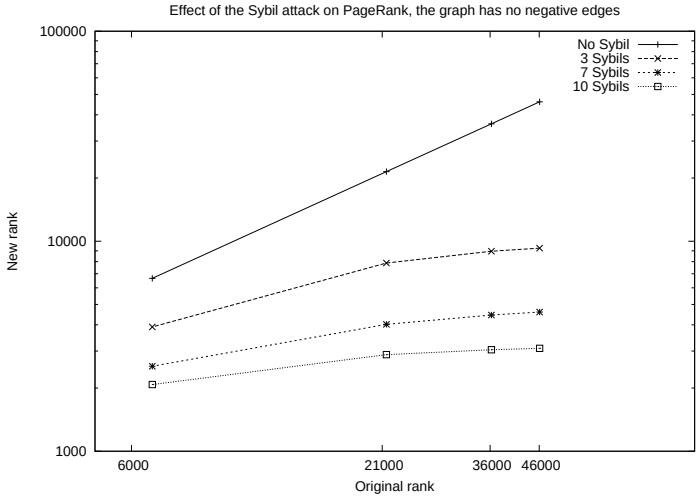



Figure 3.3.6: Effect of Sybils in the graph having no negative edges

nodes receiving the negative edges have their original ranks shown in the $x-$axis. Their new ranks, after the edges are added, are represented in the $y-$axis. It can be seen that the attacker with a higher rank can bring the other's reputations down more substantially (Property 3). The attacker negative reputation has no impact on the other's reputations (Property 4). For example, the attacker whose rank is 3153 brings the rank of another node from 2342 down to 29289. With the rank of 18516, it reduces the rank of another node from 1834 down to 6044. With negative reputation (ranked 49291), however, the attacker has no impact on the reputations of others.

Finally, Figure 3.3.5 illustrates the resilience of PRN against Sybil manipulation in the

graph having negative edges, in comparison to the resilience of PageRank in the positive graph depicted in Figure 3.3.6. The attacker in PRN gains higher rank than in PageRank, with the same number of Sybils, which can be explained by the smaller variance in the reputation values produced by PRN. The attacker's rank produced PRN differs to the one computed by PageRank by less than an order of magnitude. Therefore, the Sybil resilience of PRN can be considered as comparable with that of PageRank (Property 5).

### 3.3.5 Related Work and Discussion

In [45] and [108], the authors proposed methods to propagate trust and distrust in P2P-like environments. It has been argued earlier in Section 3.3 that it can be misleading to represent trust and distrust as real-valued edges in the trust graph. Instead, by considering the edges as representing negative and positive feedback, both [45] and [108] offered different approaches that integrate negative feedback into reputation metrics. In [45], negative feedback is converted to positive and a new graph is constructed in addition to the original graph. Two sets of reputation values are evaluated on the two graphs and then combined together. As discussed in [108], this approach may yield counter-intuitive results. In particular, it *super-imposes* the computation of negative reputations after the computation of positive reputations, therefore allows for a node with equally high numbers of positive and negative feedback to have disproportionately large impact on others. In [108], an Advogato-like function that takes into account both types of feedback is presented. This reputation function is asymmetric, and therefore is different from PRN which is symmetric.

As discussed in Section 3.3.2 and Section 3.3.3, the word *negative* does not impose that negative feedback should only be represented by negative numbers. Section 3.3.3 has argued that it is not straightforward to simply take a range $(0, x) \subseteq (0, 1]$ to represent negative feedback, and then to use the PageRank function as it is. It would be interesting to investigate on modifying PageRank or other advanced reputation function that use a range of positive numbers to represent negative feedback.

The results from Section 3.3.4 are encouraging, but only preliminary. More experiments are needed to further validate the properties of PRN. It would also be interesting to examine PRN's performance in undirected graphs, which could lead to exploring a combination of PRN and CPR. Finally, a more formal analysis is necessary to study mathematical properties of PRN, or at least to conclude whether the iterative method used to compute PRN reputations (Equation 3.3.2) does indeed converge.

CHAPTER 4

# DETECTION OF MISBEHAVIOR IN P2P USING TRUSTED PLATFORM MODULES

Feedback mechanisms rely on peers' abilities to evaluate outcomes of their transactions with each other. This implies the need for a peer to be able to securely detect if another has misbehaved in the transaction. This chapter discusses how peers can achieve such capabilities using security devices. It considers two case studies that demonstrate how nodes can misbehave in different ways. The chapter starts with an overview of the challenges in detecting misbehavior in structured P2P, and outlines the two case studies. Section 4.2 introduces the Trusted Computing paradigm and Trusted Platform Modules (TPMs). The following sections present in detail new protocols for each case studies that allow an honest peer to tell if another peer has misbehaved. These protocols are results of collaborative efforts involving Mark Ryan and Tom Chothia. Finally, Section 4.5 discusses the related works and open issues.

## 4.1 Overview

Recall that the reputation model, described in Section 3.1.1, consists of $\mathcal{T}$ — the set of transactions — and a partial function $Rt : \mathcal{T} \rightarrow \mathbb{R}$ returning the rating that a peer receives for a given transaction. For a transaction $t$, $Rt(t)$ represents the feedback given to $resp(t)$ of the transaction $t$, and $Rt(t) \in \{-1, 1\}$. So far, the model has implicitly assumed

that $Rt$ can be readily implemented in P2P settings. However, this section argues that in structured P2P, the implementation of $Rt$ is not always straightforward.

This thesis investigates the nature of transactions and the realization of $Rt$ at two layers of abstraction: routing layer and application layer. As shown in Figure 2.3.1, the routing layer implements the lookup protocol and returns the root node of a given search key. The application layer consists of protocols specific to the application. Most existing P2P applications based on structured overlays use Chord, Pastry or Kademlia. Given a key $k$, these overlays use very similar functions to determine the root node of $k$. In Chord, $root(k)$ is the node closest on the right of $k$ in the ID ring. In Pastry or Kademlia, the root node is the one with closest numerical or XOR distance to $k$. When the correctness of $root(k)$ is the main concern, any of these overlays can be chosen to study, because the results in one overlay can be translated to the results in another overlay. In fact, this and the following chapters (except for Chapter 7) assume that Chord is the underlying the overlay.

At the routing layer, a transaction involves a peer asking another to route its query for a key $k$. It is assumed in this thesis that the routing protocol always terminates, which means that it is always possible to define the transaction's outcome to be another node returned as the root node of $k$. The searching peer then gives ratings to other peers in the routing path. If the returned node is the correct destination node of $k$, positive feedback is given. Otherwise, the peers in the routing path are considered as having misbehaved, and are given negative feedback. However, detecting such misbehavior, or in other words verifying if the returned node is the correct root node, is difficult because peers do not have full knowledge of which nodes currently in the network. Furthermore, adversarial nodes might collude to impersonate the destination node.

At the application layer, a marketplace application based on structured P2P is considered. In such a system, a seller publishes its offer for an item $k$ to a listing node that is in fact the root node of $k$. A transaction involves a buyer finding the offers for a particular item at the listing node. The transaction's outcome is a list of offers for the items

that have been published by the sellers. It is difficult to verify if the list is complete, because the listing node has total control over the offers and can decide not to report them. If the list returned to the buyer is incomplete, the listing node is considered as having misbehaved, and negative feedback is given accordingly.

The main focus of this thesis is on protocols that make the misbehavior mentioned above detectable by honest peers. This chapter presents protocols based on Trusted Platform Modules (TPMs). At the routing layer, the TPMs are used for guaranteeing the freshness of the neighbor information. At the application layer, the TPMs are used for building undeniable histories of transactions that can be verified by the buyer. More efficient protocols using a new type of secure hardware are discussed in Chapter 5.

## 4.2 Trusted Computing and Trusted Platform Modules

### 4.2.1 Trusted Computing and TPMs

*Trusted Computing* is a collection of current and future initiatives to root security in hardware that have been under development since about 2003. It is set to transform the computing security landscape over the next decade. Currently, the most noticeable manifestations are the *Trusted Platform Module* (TPM), Intel's *Trusted eXecution Technology* (TXT) and *Virtualization Technology* (VT-d).

The TPM is a hardware chip currently shipped in high-end laptops, desktops and servers made by all the major manufacturers and destined to be in all devices within the next few years. It is specified by an industry consortium [95], and the specification is now an ISO standard [1]. There were 100 million TPMs in existence in 2008, and this figure is expected to reach 250 millions in 2010 [96, 97]. The TPM provides hardware-secured storage, secure platform integrity measurement and reporting, and platform authentication. Software that uses this functionality will be rolled out over the coming years. More

specifically, TPMs are designed as passive devices that enable the following:

1. **Secure storage.** TPM is the root of trust for storage. Encryption keys are generated from and protected by a Storage Root Key (SRK). The SRK is embedded inside the TPM.

2. **Platform authentication.** A TPM can create a public/private key pair $\langle KPriv, KPub \rangle$, called an *Attestation Identity Key* (AIK), and uses this to authenticate itself to another party. The authenticity of the AIK can be certified by a certificate authority (Privacy CA), or directly by using the Direct Anonymous Attestation (DAA) protocol [12]. The latter does not require an online, trusted party to be available when the authenticity of the AIK is being verified. In the protocols using TPMs that are proposed in this thesis, peers identities are linked in certain ways with the AIKs. Therefore, the authentication protocol is essential to establish peers' identities. Additionally, it limits the adversary's capability of introducing a large number of fake identities (or Sybils).

3. **Platform measurement and reporting.** TPM is the root of trust for measuring the platform, and for reporting the measurements. TPM *measures* an application before passing the control to it. The chain of trust up to the point when the application is running is:

$$\text{TPM} \rightarrow \text{BIOS} \rightarrow \text{Hardware} \rightarrow \text{Bootloader} \rightarrow \text{OS} \rightarrow \text{Application} \tag{4.2.1}$$

Here, $X \rightarrow Y$ means that $X$ measures $Y$, then extends the PCR, and finally passes control to $Y$. TPM has a set of Platform Configuration Registers (PCRs), to which integrity measurements of software (hashes of the binaries) are recorded. In particular, a new integrity measurement (IM) is recorded to a PCR by the following operation:

$$PCR \leftarrow \text{SHA-1}(\text{concat}(PCR, IM))$$

Here, the arrow means assignment, and *concat* is the bitwise concatenation function. The PCR values are used as a report of the platform's current configuration, as they contain descriptions of software that has run or is running. To prove the authenticity of the report, the TPM signs the PCR values with one of its AIK.

The implementation of TPM also supports other features, two of which are monotonic counters and transport sessions.

#### 4.2.1.1 Monotonic counters.

TPM has a set of monotonic counters, each is identified by a unique counter ID. The counters can only be updated by incrementing the current values. The TPM restricts the rate at which updates happen, so that the values do not wrap around, at least not until after a number of years. On counter *cid*, the following operations are possible:

1. `TPM_ReadCounter`(*cid*): returns the current value of the counter *cid*.
2. `TPM_IncrementCounter`(*cid*): increments and returns the new value of the counter *cid*.

#### 4.2.1.2 Transport sessions.

TPM commands can be grouped and executed together within a transport session. The session can be *exclusive*, meaning that no other commands can be executed outside of the session when it is active. Furthermore, the session's log, which includes inputs and outputs of the commands executed in the session, can be signed by the TPM with one of its AIKs.

1. `TPM_EstablishTransport`(*exc*): sets up a transport session. The flag *exc* indicates if the session is exclusive. A session handle, *sHandle*, is returned.
2. `TPM_ExecuteTransport`(*comm*, *sHandle*): executes *comm*, which contains a *wrapped* TPMs command, inside the session *sHandle*.

```
1. sHandle <- TPM_EstbalishTransport(true)
2. if (mode=read)
       wc <- wrap command TPM_ReadCounter(cid)
   else
       wc <- wrap command TPM_IncrementCounter(cid)
3. TPM_ExecuteTransport(wc, sHandle)
4. sig <- TPM_ReleaseTransportSigned(sHandle,n)
5. return sig
```

Figure 4.2.1: `getSignedCounterValues`$(mode, n, cid)$ is executed by the local TPM at B. Two modes: 'read' and 'inc' can be invoked that return different signatures. The TPM uses the private part of the AIK that can be authenticated to other parties as the signing key for the transport session's log.

3. `TPM_ReleaseTransportSigned`$(n, sHandle)$: closes the transport session and signs its log with an AIK, using $n$ as the non-replay nonce.

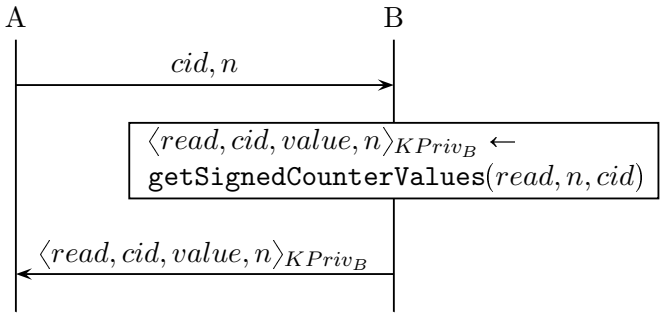
### 4.2.2 A Simple Protocol Involving Monotonic Counters and Transport Sessions

Figure 4.2.1 illustrates a procedure in which the current value of a given counter is read and signed by the TPM. Let *KPriv* be the signing key, which could be the private part of an AIK. If the mode is set to 'read', the returned signature contains the current value of counter *cid*, called *value*, and a non-replay nonce $n$. The signature is denoted as:

$$\langle read, cid, value, n \rangle_{KPriv}$$

If the mode is 'inc', the returned signature is $\langle inc, cid, value, n \rangle_{KPriv}$ where *value* is the latest value of the counter *cid* after it has been incremented.

Suppose that agent $A$ is interested in the latest value of the counter *cid* of agent $B$'s TPM. Protocol 4.2.1 illustrates how $A$ obtains the latest value of *cid*. First, $A$ sends the counter ID and a fresh nonce $n$ to $B$. $B$ then executes `getSignedCounterValues`$(read, n, cid)$ on its local TPM, which first establishes a transport session with the TPM, then executes `TPM_ReadCounter`$(cid)$ within that session. Finally, the session is closed and the TPM's




**Protocol** 4.2.1: $A$ queries $B$ for the latest value of its counter $cid$

signature on the session's log is returned. The TPM uses the private part of its AIK that can be authenticated to other parties as the signing key. $n$ is used as the non-replay nonce in the returned signature. It is not possible for $B$ to generate such a signature without having its TPM executing the `TPM_ReadCounter`($cid$) command inside a transport session.

### 4.2.3 Why Not Attestation?

Most TPMs' features are originally designed to enable platform remote attestation. The basic idea is that the local platform (the attestee) uses the TPM to measures and faithfully reports its states to a remote party. Having had the report, the attester then decides if the attestee's platform is trustworthy based on its own list of *approved* software.

Using an attestation-based approach, the problem of detecting misbehavior becomes that of attesting if remote peers are running trustworthy P2P software. The implication goes beyond reputation systems, as any problem in a P2P system can be solved by having all peers running the same trustworthy software. In practice, there are at least three approaches for software attestations, all having serious drawbacks:

1. *Binary attestation.* Suppose a trustworthy P2P application $A$ were written, whose binary hash is $H_A$. To check if a remote peer is not misbehaving, the verifying peer must make sure that: (1) all the components in the chain of trust (Equation 4.2.1) are trustworthy (2) the application $A$ is running (3) the communication channel

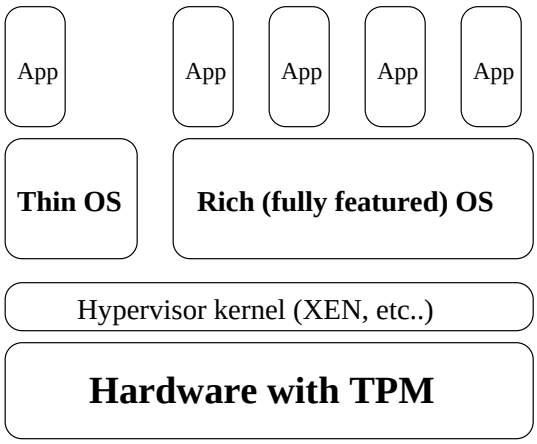



Figure 4.2.2: Attestation based on virtual machines. The hypervisor kernel provides multiple hardware interfaces, so that more than one operating systems could utilize the underlying hardware at the same time.

with $A$ is secure. In practice, this approach has serious weaknesses [19]:

- It is difficult to verify if the P2P software is trustworthy.
- Many different applications behave the same way but have different hash values. In addition, a particular software might have many different versions, each version hashes to different value. The list of trustworthy binary hashes, which the attester must maintain, could be infeasibly long.

2. *Virtual machine based attestation.* Notice that in the binary attestation approach, attesting an operating system is one of the biggest challenges, as the operating system may change regularly. In this approach, a hardware-based hypervisors like Xen is used so that applications can run on top of *thin* operating systems which are simple enough for them not to be changed frequently (Figure 4.2.2). Even though the list of trustworthy software is then smaller and thus easier to manage, the attester still needs to verify the integrity and especially the trustworthiness of the other components in the chain of trust. The latter is the most difficult.

3. *Property-based attestation.* In this approach, it is the properties of the software that get attested, not the binary hash [19]. As a consequence, two different applications are attested as being the same if they have the same set of properties. This could

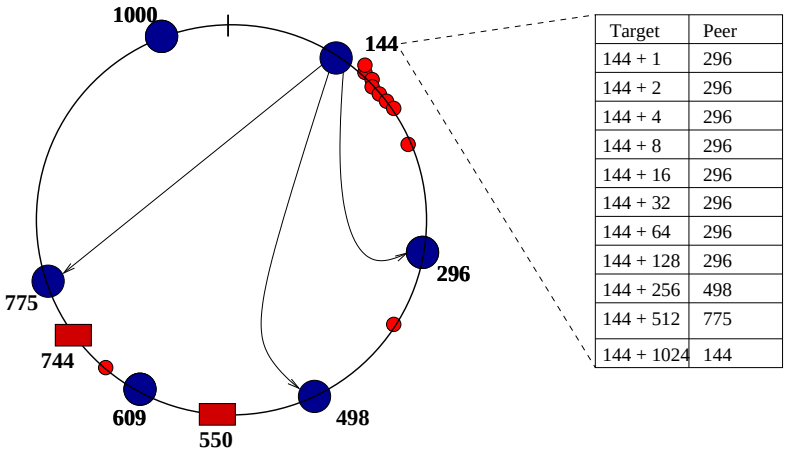
enormously reduce the work by the attester in keeping track of the variations and updates of the software. The biggest obstacle, however, is how to specify and verify properties of software. Code analysis, proof carrying code and relying on humans are the mentioned techniques. But current effort in these fields is not sufficient to correctly identify and analyze all the possible behavior of the software.

In summary, to use attestation with TPMs in practice, many challenges will needed to be addressed. Based on its current stage of development, software attestation is not used in this thesis. Instead, the protocols presented in this chapter make use of the low-level features of the TPM, particularly platform authentication, monotonic counters and transport sessions.

## 4.3 Detection of Misbehavior at the Routing Layer (DTR1)

This section presents the mechanism for DeTecting misbehavior at the Routing layer, called **DTR1** (as opposed to DTR2 which is presented in the next chapter). The protocols are devised in collaboration with Mark Ryan. It enables peers to assess the outcome of routing transactions. More specifically, given a node claiming to be the root node of a key $k$, an honest peer can verify if it is the correct root node of $k$, namely $root(k)$, in the current configuration of the network.

As briefly explained in Section 4.1, the verification is difficult, because of adversarial nodes colluding to impersonate the root nodes. In Figure 2.1.2 (shown again below), for example, the adversary controlling node 498 and 775 could convince node 144 that 775 is the root node of key 550 (the correct root node of 550 is in fact node 609). There is a practical explanation for this misbehavior. Since $k$ is stored at the $root(k)$, the adversary might misbehave to gain control or to censor a particular piece of data. In a P2P storage system, for example, the attacker having the data can modify or remove it from the system. In other applications such as P2P-based marketplaces, controlling more (or a




| Target | Peer |
|---|---|
| 144 + 1 | 296 |
| 144 + 2 | 296 |
| 144 + 4 | 296 |
| 144 + 8 | 296 |
| 144 + 16 | 296 |
| 144 + 32 | 296 |
| 144 + 64 | 296 |
| 144 + 128 | 296 |
| 144 + 256 | 498 |
| 144 + 512 | 775 |
| 144 + 1024 | 144 |

particular piece of) data could imply monetary gains. In a P2P-based communication system like VOIP or instant messaging, the data generally contains connection details of the communicating clients. Having gained control of such data, the attacker might be able to eavesdrop the communication, or even prevent it from happening.

A P2P system in which peers can correctly identify this misbehavior is said to have the *root authenticity* (or RA) property. A more formal definition of this property is given next, which is followed by descriptions of the protocols that aim to achieve this property.

### 4.3.1 Root Authenticity (RA) Property of a P2P System

For any $x, y \in \mathcal{I}$, denote $cd(x, y)$ as the function returning the clock-wise distance between $y$ and $x$. More precisely:

$$cd(x, y) = t \;\Leftrightarrow\; 0 \leqslant t < 2^m \;\wedge\; y \oplus t = x$$

where $\oplus$ is the addition operation in *modulo* $2^m$. Let $inBetween(z, x, y)$ be the predicate indicating if $z$ is in the clock-wise range between $x$ and $y$. In other words, going clockwise from $x$, one will reach $z$ before $y$. More precisely:

$$inBetween(z, x, y) = \big(\, cd(z, x) + cd(y, z) = cd(y, x) \,\big)$$

```
1. pl <- pv.getPredecessor(pd);
2. if (pv.neighborVerification(pl,pd))
      if (inBetween(k,pl,pd))
         return true;
3. return false;
```

Figure 4.3.1: Details of the $p_v$.`destVerification(k,`$p_d$`)` protocol, which is executed by $p_v$. It returns *true* if $p_d$ is correct root node of $k$, and returns *false* otherwise.

Let $p_v$ be an honest peer that searches for the root node of a key $k$ and verifies if the returned node is the correct root node. $p_v$ has the following set of operations:

1. $p_v.route(k)$ : the P2P routing protocol. *route* can be any function returning a peer.

2. $p_v.getPredecessor(p_d)$ : $p_v$ contacts $p_d$ and asks for its predecessor. $getPredecessor$ can be any function returning a peer.

3. $p_v.neighborVerification(p_l, p_r)$ : $p_v$ checks if $p_l$ is the predecessor of $p_r$ in the current network. This protocol is the main focus of Section 4.3.2.

4. $p_v.destVerification(k, p_d)$ : $p$ checks if $p_d$ is the root node of $k$ in the current network. The details of this protocol are shown in Fig.4.3.1. This protocol assumes a Chord overlay, therefore only returns true if $p_v$ thinks there is no node in between $k$ and $p_d$ in the ID ring.

**Definition 4.3.1** (**Root Authenticity (RA) Property**)**.** Let $\mathcal{P}^t$ be the set of current nodes in the P2P system, at a given time $t$. Assume that the system evolves from $t$ to $(t+1)$ as a new peer joins or an existing peer leaves the system. Let $p_v$ be the honest peer that performs lookup queries and verifies if the returned nodes are the correct root nodes of the search keys. Let $destVerf(k, p_d, t)$ be the operation $p_v.destVerification(k, p_d)$

executed by $p_v$ at time $t$. The RA property is defined as:

$$\forall p_d, k \in \mathcal{I}, t\,.\;\; destVerf(k, p_d, t)$$
$$\Rightarrow\; p_d \in \mathcal{P}^t \;\wedge\; \big(\,\forall p'_d \in \mathcal{P}^t \setminus \{p_d\}\,.\; cd(p'_d, k) > cd(p_d, k)\,\big)$$

Informally speaking, the RA property states that if $destVerification(k, p_d)$ returns true for any value of $k$ and $p_d$, then $p_d$ is the destination node of $k$ given the current configuration of the network.

**Definition 4.3.2** (**Neighbor Authenticity (NA) Property**)**.** Let $\mathcal{P}^t$ be the set of current nodes in the P2P system, at a given time $t$. Assume that the system evolves from $t$ to $(t+1)$ as a new peer joins or an existing peer leaves the system. Let $p_v$ be the honest peer that performs lookup queries and verifies if the returned nodes are the correct root nodes of the search keys. Let $neighVerf(p_l, p_d, t)$ be the operation $p_v.neighborVerification(p_l, p_d)$ executed by $p_v$ at time $t$. The NA property is defined as:

$$\forall p_l, p_d, t\,.\;\; neighVerf(p_l, p_d, t)$$
$$\Rightarrow\; \{p_l, p_d\} \subseteq \mathcal{P}^t \;\wedge\; \big(\,\forall p'_d \in \mathcal{P}^t \setminus \{p_l, p_d\}\,.\; cd(p'_d, p_l) > cd(p_d, p_l)\,\big)$$

This property states that if $neighborVerification(p_l, p_d)$ returns true for any value of $p_l$ and $p_d$, then $p_l$ and $p_d$ are currently in the network and $p_l$ is the immediate left neighbor of $p_d$.

**Theorem 4.3.1.** *Given the definition of NA and RA above, it follows that:*

$$NA \;\Rightarrow\; RA$$

*Proof.* The proof for this theorem can be found in Appendix B □

This theorem means that if the neighbor verification protocol is correct, then the system satisfies the RA property.

### 4.3.2 Proposed System

This section introduces protocols that promise to satisfy the NA property. The main idea is to rely on a trusted party to issue certificates to peers during churn events, and the certificates are linked to the latest counter values at the peers' TPMs in order to guarantee their freshness. The formal proof that the system does indeed satisfy the NA (and consequently RA) property is left until Chapter 6.

#### 4.3.2.1 Assumptions.

First, peers are equipped with running TPMs. Furthermore, counter *cid* at each TPM is used exclusively for the P2P application. This assumption could be used to establish a simple ID scheme for peers in which public part of the AIK is assigned as the peer ID. This scheme is simpler than the one proposed by Balfe *et al.* [9], but it still guarantees uniqueness even though multiple AIKs can be generated for the TPM. It is because *cid* is unique per TPM, therefore if multiple IDs are used then updating *cid* using one ID will invalidate the states of others.

Second, for the churn model, it is assumed that peers leave the network gracefully, meaning that they notify their neighbors (and other relevant entities) before exiting.

#### 4.3.2.2 Certificate Authority.

There exists a certificate authority (CA) that is trusted to issue *neighbor certificates* as peers join and leave the network. The CA does not need to run on trusted hardware. It is a single point of trust, but as discussed later, is unlikely to be a performance bottleneck.

The CA has an asymmetric key pair $\langle KPriv_{CA}, KPub_{CA} \rangle$. At the end of a joining process, for example, a new peer $p_n$ contacts CA to get a neighbor certificate, which is of the form:

$$\langle cid, v, p_n, p_l, p_r \rangle_{PrivK_{CA}}$$

where $v$ is the current value of the counter $cid$ of $p_n$'s TPM. $p_l, p_r$ are the immediate left and right neighbor of $p_n$, at the moment the certificate being issued. They also receive new neighbor certificates from the CA. It is important that the CA *knows* the correct immediate left and right neighbors of $p_n$ in order to issue such certificates. There are several ways for the CA to acquire this knowledge. For simplicity, it is assumed that the CA maintains a list of peers currently in the network. When $p_n$ joins, it checks that $p_n$ is not already in the list, then issues the relevant certificates and adds $p_n$ to the list. It performs the opposite when $p_n$ leaves the network.

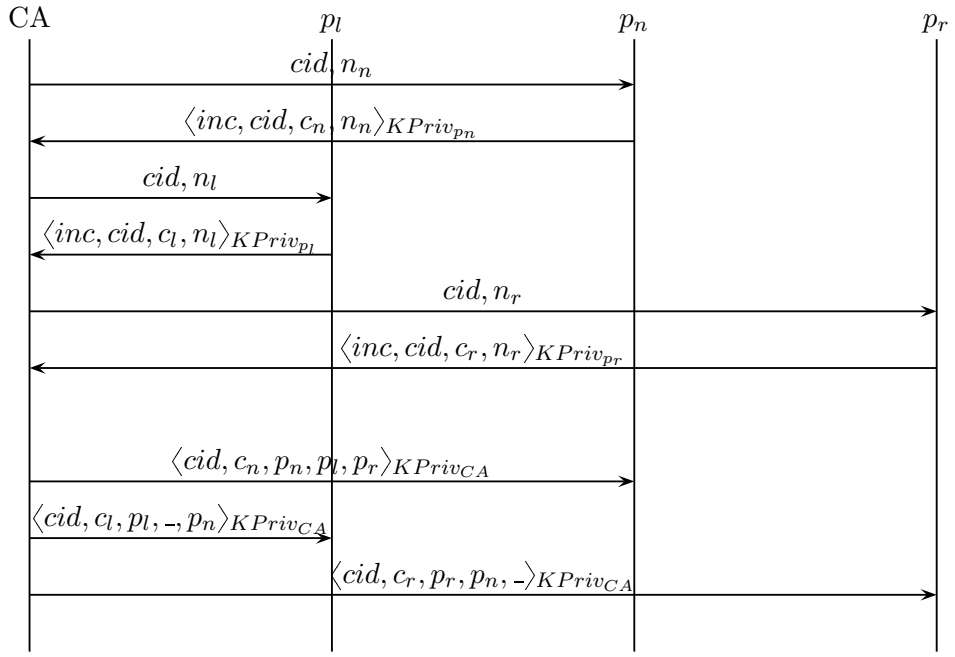
#### 4.3.2.3 Joining/Leaving Protocol.

Protocol 4.3.1 illustrates the protocol between the CA and other nodes when $p_n$ joins the network. The CA knows that $p_l$, $p_r$ are the immediate left and right neighbor of $p_n$ in the current network. First, it asks $p_n$, $p_l$ and $p_r$ to increment their counters. Once receiving the signatures on the new counter values, the CA adds $p_n$ to its list of existing peers, then issues new certificates for $p_n$, $p_l$ and $p_r$ containing information of their new neighbors.

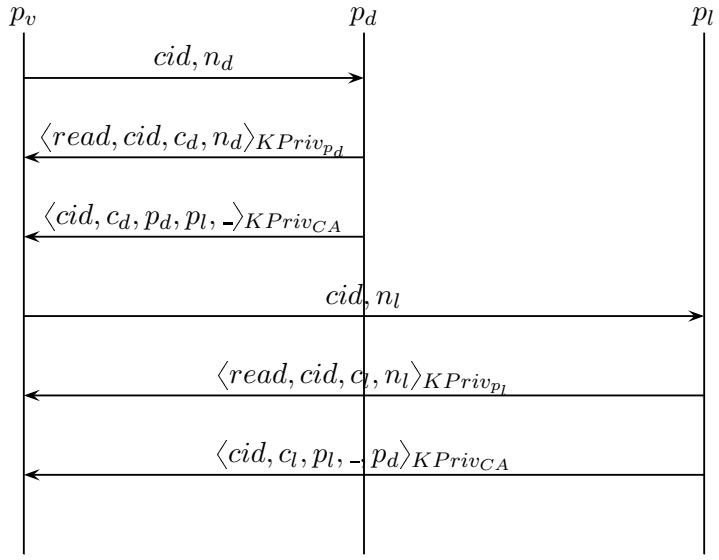
When a node leaves the network, the protocol is similar, except that the CA only issues certificates for the current neighbors of the leaving nodes.

#### 4.3.2.4 Routing Protocol.

Suppose that $p_v$ searches for the root node of a key $k$. The normal P2P routing protocol is executed first, which returns a peer $p_d$. As shown in Figure 4.3.1, before accepting $p_d$ as the destination of $k$, $p_v$ performs the verification protocol with $p_d$, which is depicted in Protocol 4.3.2. $p_v$ queries the latest value of $p_d$'s counter, namely $c_{p_d}$. It then asks $p_d$ for




**Protocol** 4.3.1: Peer $p_n$ joins in between $p_l$ and $p_r$ in the network. '-' indicates that the value of the field is not important




**Protocol** 4.3.2: Peer $p_v$ verifies if $p_l$ is the current left neighbor of peer $p_d$. '-' indicates that the value of the field is not important. This protocol essentially implements the $p_v.neighborVerification(p_l, p_d)$ operation.

the certificate of $Cert_{p_d}$ that contains $c_{p_d}$. $p_v$ can be confident that $Cert_{p_d}$ is the latest certificate issued by the CA to $p_d$.

$Cert_{p_d}$ contains information of $p_d$'s left neighbor, namely $p_l$. $p_v$ then asks $p_l$ for its latest certificate, namely $Cert_{p_l}$. The verification returns true if $Cert_{p_d}$ and $Cert_{p_l}$ match, i.e. in $Cert_{p_d}$, $p_d$ is the right neighbor of $p_l$ and in $Cert_{p_l}$, $p_l$ is the left neighbor of $p_d$.

Certificates from both $p_d$ and $p_l$ are required in order to avoid the following scenario. Assume that only the certificate from $p_d$ is asked for during verification, i.e. $p_v.neighborVerification(p_l, p_d)$ always returns true regardless of $p_l$. Suppose that $p_d$ is the adversary that executed the joining protocol properly and has already left the network (gracefully), but it is still online. The routing protocol returns $p_d$. Since it is still online, $p_d$ provides its out-of-date certificate during verification, which is accepted by the $p_v$. Consequently, $p_v.destVerification(k, p_d)$ could return true, violating the RA property that requires the destination node to be a node currently in the network.

## 4.4 Detection of Misbehavior at the Application Layer (DTA1)

This section presents the protocols for DeTecting misbehavior at the Application layer, called **DTA1** (as opposed to DTA2, which is presented in the next chapter). These protocols are the results from the joint work that was done with Tom Chothia. They targets a P2P-based marketplace application, and enables peers to assess the outcomes of transactions. Marketplace applications based on P2P offer a number of advantages over the centralized systems. First, they scale better and have no single points of failure. Second, there would be no limitation on the types of items being exchanged. Thus, the P2P infrastructure allows for a censorship-free environment. Finally, no centralized authority means a monopoly cannot arise.

In a P2P-based marketplace, the overlay consists of nodes and data representing the buyers, sellers and sale offers for items being exchanged. The sale offer for item $k$ is

listed at $root(k)$, called the *listing node*. Structured overlays are more suitable for this application, because:

1. The deterministic search helps locating the items more efficiently.

2. Structured overlays allow for the offers of a particular items from all the sellers to be found at one place. This reduces the time buyers spent on selecting the best offers.

A consequence of using structured overlays is that sellers need to trust the listing nodes to truthfully report their offers to potential buyers. There are economic incentives for a listing node to misbehave. For example, if it gets a percentage for each item sold, it will earn more profit by reporting items with the highest price. If it is also selling the same item, the rational choice is just to report its own offer. Such misbehavior is a real concern and needs to be addressed if P2P-based marketplaces are to be realized in practice. Notice that it does not help for the seller to constantly query the listing node to check that it reports the offer, since the listing node might check the identity of the querier before reporting, so that it could choose to report the offer only to the original seller, not to the potential buyers.

In the following system, each peer is assumed to have a running TPM. The main idea is to tie sale offers to counter values so that a peer cannot lie about the offers it stores without being detected. Essentially, TPM is used for constructing an undeniable history of offers being published at the peer. What returned by the publisher to the buyer is then checked against the history in order to determine if the publisher has truthfully reported the offers.

In the next section, the system model and problem's description are presented. Section 4.4.2 describes the protocols and stretches the proof that they allow for the detection of the misbehavior. More efficient protocols are discussed in Section 4.4.3.

### 4.4.1 System Model and Problem Description

#### 4.4.1.1 System Model

In the P2P-based marketplace, $\mathcal{P}$ represents the set of buyers and sellers, $\mathcal{D}$ represents the items being exchanged. Peers form a Chord overlay. It is assumed that the network is static, i.e. no churn. In addition:

- $\mathcal{S} : \mathcal{D} \to \mathbb{P}(\mathcal{P})$ is the function returning the set of sellers of a given item.
- $v : \mathcal{P} \times \mathcal{D} \to \mathbb{R}^+$ is the partial function returning the price that a peer offers for a given item. $v(p, d)$ is defined if $p \in \mathcal{S}(d)$.
- $\Delta : \mathcal{D} \to \mathbb{P}(\mathcal{P})$ is the function returning the set of listing nodes for a given item. Assume there exists a well-defined replication mechanism so that an item can be stored at more than one listing nodes (one of which is the root node, others are the replicas).
- $\mathcal{W}^p = \{(d, s, v(s, d)) \mid d \in \mathcal{D}, s \in \mathcal{S}(d), p \in \Delta(d)\}$ is the set of sale offers stored at peer $p$.
- $\mathcal{W}^p_d = \{(d', s, c) \in \mathcal{W}^p \mid d' = d\}$ is the set of sale offers for item $d$ stored at peer $p$. $\mathcal{W}^p_d \subseteq \mathcal{W}^p$
- $f \in \mathbb{R}^+, r \in (0, 1)$ are the flat rate and variable payment respectively. The payments are made by the seller to the listing nodes of its items.

Two main protocols in the system are:

1. $p.publish(d)$: a seller $p$ first selects a node $p_d \in \Delta(d)$, then executes the following:
   (a) $p$ sends $(d, p, v(d, p))$ to $p_d$.
   (b) If $p_d$ is honest, it updates the its states, i.e. $\mathcal{W}^{p_d} = \mathcal{W}^{p_d} \cup \{(d, p, v(d, p))\}$
   (c) $p$ sends the flat payment $f$ to $p_d$ for listing the sale offer.

2. $p_r.retrieve(d)$: a buyer $p_r$ interested in the item $d$ ($p_r \notin \mathcal{S}(d)$) first finds the set of listing nodes $\Delta(d)$. For every $p_d \in \Delta(d)$, $p_r$ requests the offers for $d$, for which a set $\rho_d \subseteq \mathcal{W}_d^{p_d}$ is returned. If there is an offer $(d, p_s, v) \in \rho_d$ satisfying $p_r$'s requirements, $p_r$ makes the payment of value $v$ to $p_s$. If $p_d$ takes a commission with the rate $r$ from selling $d$, then its total profit from the sale amounts to $r.v + f$.

#### 4.4.1.2 Problem Description

The misbehavior considered is that the listing node does not report all the offers for $d$. In other words,

$$\rho_d \neq \mathcal{W}_d^{p_d}$$

There is a number of reasons why this might arise. First, the variable profit that $p_d$ gets for the sale is $r.v$, therefore, if both $e = (d, p_s, v)$ and $e' = (d, p'_s, v')$ are in $\mathcal{W}_d^{p_d}$ and $v > v'$, $p_d$ might be likely to gain more profit by not including $e'$ in $\rho_d$. Second, if $v(p, d)$ is the same for all $p \in \mathcal{S}(d)$ and $p_d \in \mathcal{S}(d)$, meaning that $p_d$ also sells $d$, $p_d$ would gain the most profit by reporting only its own offer, i.e. $\rho_d = \{(d, p_d, v(p_d, d))\}$. Third, when $p_d \notin \mathcal{S}(d)$, $p_d$ might choose to only report offers from $p'_d$ which is colluding with $p_d$ or pays higher fees.

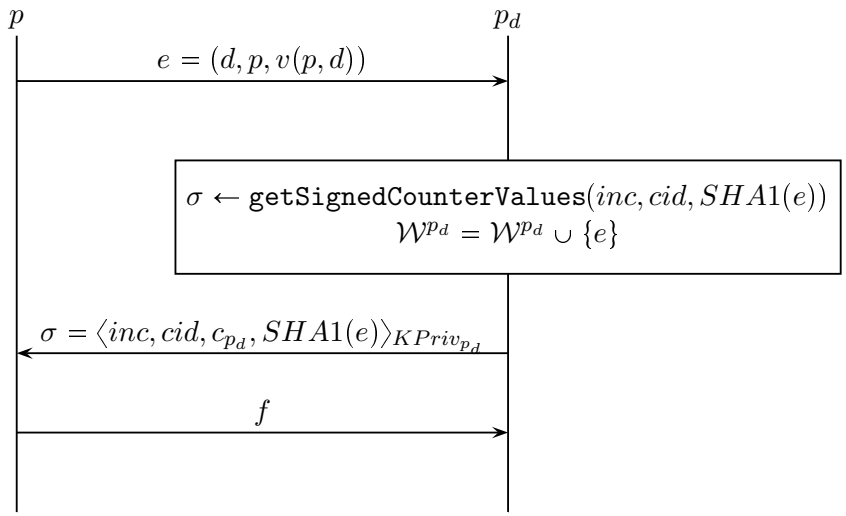
In the system described in Section 4.4.2, if the $retrieve(d)$ operation terminates successfully, it means $p_d$ is behaving properly and $p_r$ has received all the offers for $d$ stored at $p_d$, i.e. $\rho_d = \mathcal{W}_d^{p_d}$.

### 4.4.2 Proposed System

By utilizing the TPM at each node, the *publish* and *retrieve* operations are enhanced as follows:

#### 4.4.2.1 *publish(d)*:

It is illustrated in Protocol 4.4.1. First, $p$ sends its offer $e = (d, p, v(p, d))$ to $p_d$. $p_d$ asks




**Protocol** 4.4.1: Peer $p$ publishes its offer for item $d$ to $p_d$, where $p_d \in \Delta(d)$.

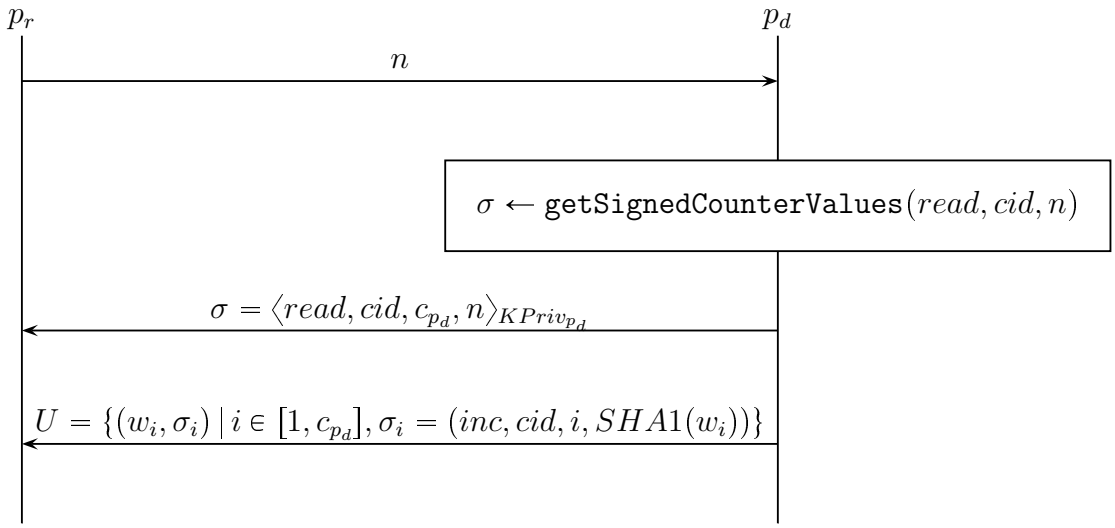
its TPM to increment the counter and sends back the signed receipt $\sigma$ of that operation. The hash of $e$ is used as the non-replay nonce in $\sigma$. $p_d$ also updates its states to include the new offer. $p$ verifies that $\sigma$ is correct. Finally $p$ sends the payment $f$ to $p_d$ and the operation terminates successfully. At the end of this operation, the offer $e$ is tied to the value $c_{p_d}$ of the counter $cid$.

#### 4.4.2.2 *retrieve(d)*:

It is depicted in Protocol 4.4.2. First, $p_r$ asks $p_d$ for its the latest value, to which $p_d$ returns the signature on a value $c_{p_d}$. Then, $p_d$ sends all the offers tied to the counter values from 1 to $c_{p_d}$. More specifically, it sends the set of tuples $U$ defined as:

$$U = \{(w_i, \sigma_i) \mid i \in [1, c_{p_d}], \sigma_i = (inc, cid, i, SHA1(w_i))\}$$

In other words, $U$ contains elements having the form $(w_i, \sigma_i)$ where $w_i$ is the offer tied to the counter value $i$, $\sigma_i$ is the receipt issued to the seller when the offer was published. The *retrieve* operation terminates successfully if $p_d$ has received all tuple $(w_i, \sigma_i)$ for $i \in [1, c_{p_d}]$ and all the signatures are correct. Finally, $p_r$ can extract the set of the offers for item $d$




**Protocol** 4.4.2: Peer $p$ publishes its offer for item $d$ to $p_d$.

from $U$ as follows:

$$\rho_d = \{(d', s', v') \mid ((d', s', v'), \sigma') \in U, d' = d\}$$

**Theorem 4.4.1.** *Suppose that the* publish *and* retrieve *operations terminate successfully, then $p_d$ is behaving properly, or:*

$$\rho_d = \mathcal{W}_d^{p_d}$$

*Sketch of proof.* This can be proved by showing that $U = \mathcal{W}^{p_d}$. First, $|W^{p_d}| = |U|$, because $U$ contains all the receipts of all the publishing operations with counter values from 1 to the latest value. Next, for all $w_i \in U$, it follows that $w_i \in W^{p_d}$. If this is not true, $p_d$ must be able find $w_i' \neq w_i$ such that SHA1$(w_i)$ = SHA1$(w_i')$ (so that the verification of $\sigma_i$ succeeds). This contradicts the non-collision property of the hash function. Thus, $w_i \in \mathcal{W}^{p_d}$. Therefore, $U = W^{p_d}$. □

### 4.4.3 More Efficient Solutions

The *retrieve* protocol requires $p_d$ to send back the set $\mathcal{W}^{p_d}$. This does not scale well, as $|\mathcal{W}^{p_d}|$ can be very large. This section presents two implementations that are more

scalable. It both cases, the number of offers being sent back to $p_r$ is at least $|\rho_d|$.

#### 4.4.3.1 Probabilistic Solution

$p_d$ builds a 2-3 Merkle trees [71] from $\mathcal{W}^{p_d}$. The hash value of an item $d$ is stored at a leaf of the tree. The leaves are ordered. On average, the insertion and update operations involve updating $O(h)$ tree nodes in the hash path. The latest root hash is included in the TPM's signature containing the latest counter value.

To check if the hash tree was constructed correctly by $p_d$, $p$ can use a *challenge - response* protocol. When publishing an offer, $p$ verifies if the new root hash is updated correctly by comparing it with its own calculation using the hash path given by $p_d$. Because the leaves are ordered, in *retrieve*($d$), $p_r$ can efficiently count the number of item $d$ stored in the leaf set. $p_r$ then ask $p_d$ for all the valid tuples of the form $(w_i, \sigma_i)$ where $d$ is included in $w_i$. There is a clear trade-off between the communication, computation overhead and the probability that $\rho_d = \mathcal{W}_d^{p_d}$.

#### 4.4.3.2 Extra Counters

So far, only one monotonic counter is used to tie offers to counter values. Having $c$ counters, $c > 1$, will reduce the communication and computation overhead to $O(\frac{|\mathcal{W}^{p_d}|}{c})$. More specifically, for an item $d$, both the *publish(d)* and *retrieve(d)* operations use the counter *cid* such that *cid* = SHA1($d$) *modulo* $c$. It can be seen that if $c$ is large enough, significant improvement can be achieved.

The current implementation of TPM supports a small number of counter. Sarmenta *et al.* [84] proposed a slight extension to the TPM that allows it to support up to $2^{160}$ monotonic counters. The basic idea is to build a Merkle tree whose leaves store the counter values. The TPM only stores the root of the tree, and it has operations to verify and update the leaves.

## 4.5 Related Work and Discussion

A closely related work on detecting the misbehavior in structured P2P routing is that by Wang *et al.* [100], which proposed a mechanism for Pastry allows an honest node to verify if another is the root node of the search key. It assumes the existence of a certificate authority (CA). When a node joins, the CA issues certificates to $l + 1$ neighbors, where $l$ is the leafset's size. This approach is probabilistic, and its effectiveness increases with $l$. Ganesh *et al.* [38] proposed another system that relies on peers regularly publishing their ID certificates. For verification, it relies on name-space density estimation, which is probabilistic. The protocols described in Section 4.3 and Section 4.4 allow the honest peer to tell categorically whether another node is the root node of the search key.

A major assumption in DTR1 and DTA1 is that peers are equipped with TPMs. For a large-scale P2P system, one might question if this assumption is reasonable. It is partly due to past controversy about the TPM [5] and the TPM being in early stages of development. However, more computers are being shipped with TPMs. Moreover, DTR1 and DTA1 are not necessarily bound to TPM. They can be implemented with any other infrastructure supporting platform authentication, monotonic counters and transport sessions. These alternatives could be in the form of smart-cards [61] or online services. If available in large scale, such devices or services could be better choices than TPMs because of their flexibility and wider ranges of trusted functionalities. In fact, Chapter 5 describes a new secure hardware that is more powerful than TPM and can improve the efficiency of the current systems.

The churn models assumed in DTR1 and DTA1 are quite strict. More specifically, DTR1 assumes peers leave the network gracefully, which can be made more realistic by taking into account *fail-stop* and *Byzantine* failure. To deal with these failures, a *time-out* mechanism is needed that indicates when a certificate expires. Peers would need to contact the CA regularly to have their certificates renewed, or else they would be considered as having left the network. This would imply more overhead for the CA, as it needs to issue more certificates and keep track of which peers have left the network.

Detailed investigation of the time-out mechanism is left for future work. In DTA1, only static networks are considered, which means more work is needed to study the system under dynamic conditions.

In DTR1, the CA is a relatively *off-line* entity, since it is only involved during churn events and is not consulted during routing. In a typical P2P system, the rate of query is considerably more frequent than the rate of churn. Therefore, it can be argued that the CA is unlikely to be a performance bottleneck. More specifically, churn events can be modeled by a Poisson distribution. Let $cr$ be the *churn rate*, so that the session times are exponentially distributed with the expected value of $\frac{1}{cr}$. It then follows that the expected number of churn events the CA has to deal with per time unit is $cr.|\mathcal{P}|$. For a large (but relatively stable) network, i.e. $\mathcal{P}$ is in order of millions and the average session time is in the order of days, $cr.|\mathcal{P}|$ is small enough so that the CA would not become a bottleneck.

In the current design, the CA maintains a list of peers currently in the network. For scalability, it will be better to relieve the CA from keeping such a list. One alternative would be for the CA to ask for the certificates of the joining or leaving peer as well as of its immediate neighbors during churn. If the certificates match, the CA then issues new certificates as usual. It can be conjectured that if all the certificates before a churn event were issued correctly, then so are the new ones after the event is completed.

The current design of DTA1 leaves some room for future work. The flat-rate payment $f$ is used as an incentive for peers to accept storing and reporting sale offers. In practice, more advanced incentive mechanisms might be needed to discourage peers from denying storing and reporting sale offers. As time passes, the *retrieve* operation will only send more (irrelevant) data. Even with the improvement proposed in Section 4, the amount of data grows without bound. One method to address this could be to use *sliding windows* to regulate the maximum number of offers stored at any given time. Finally, the system model in DTA1 can be made more realistic by letting $p_d$ remove an offer from $\mathcal{W}_d^{p_d}$ after the item has been paid for by a buyer. The system must ensure that $p_d$ cannot arbitrarily remove items without being detected.

CHAPTER 5

# NEW HARDWARE FOR DETECTION OF MISBEHAVIOR IN P2P

This chapter introduces a new type of security hardware, called TTM, that aims to be a general security device and be able to support more security applications than TPM. When used instead of TPM, this hardware offers improvements to the misbehavior detection mechanisms described in the previous chapter. Another use of TTM, which is presented in this chapter, is for access control systems. Section 5.1 discusses the motivations in designing TTM. The next three sections detail the main components, data structures and operations of the hardware. Section 5.4 shows how access control systems can benefit from TTM. Section 5.5 and Section 5.6 focus on using TTM to improve the misbehavior detection protocols (DTR1 and DTA1). Finally, Section 5.7 discusses the related works and open issues with TTM.

## 5.1 Motivation

In the previous chapter, DTR1 and DTA1 used Trusted Computing Modules (TPMs) as the underlying security mechanism. Due to the TPM's restricted set of functionalities, there are limitations when implementing DTR1 and DTA1 in practice. As discussed in Section 4.5, DTR1 has to rely on a single trusted party (the CA) to issue certificates as peers join and leave the system, which is not always desirable. DTA1 currently does

not scale very well, as TPM supports only a small number of monotonic counters. The improved version of DTA1, proposed at the end of Section 4.4, is probabilistic and incurs noticeable overhead.

Using only the TPMs, extending DTR1 to remove the role of the CA is difficult. Consider a hypothetical extension of DTR1, denoted as DTR1', that requires no CA. As in DTR1, a peer's left and right neighbor constitute the peer's state. The TPM's counter can be used to time-stamp the state, so that any peer can verify if a state is the latest one. This is similar to the way peers in DTA1 time-stamping the sale offers. As a peer joins the system, it creates a new local state and also causes the neighbors' states to change. In DTR1, the state of $p$ is certified by the CA. As a result, DTR1 satisfies the following property:

**Property 5.1.1** (Non-overlapping property)**.** For any peer $p$ and $p'$, let $(p_l, p_r)$ and $(p'_l, p'_r)$ be the states of $p$ and $p'$ respectively. Then:

$$\nexists x \bullet \ inBetween(x, p_l, p_r) \ \wedge \ inBetween(x, p'_l, p'_r)$$

DTR1', however, does not meet this property, mainly because a peer in DTR1 can generate any valid state by itself . Failure to meet this property gives rise to the following challenges when verifying the correctness of a (latest) state:

1. A history of states is needed so that a peer can verify if a state at time $t$ is correctly derived from the previous state at time $(t-1)$. More specifically, suppose $p_l$ and $p_r$ are the left and right neighbor of $p$ at time $(t-1)$, and $p_n$ joins between $p_l$ and $p$ at time $t$, then $p$'s state at $t$ is derived correctly from the state at $(t-1)$ if it contains $p_n$ and $p_r$ as its immediate neighbors. In addition, one has to verify that the state at $(t-1)$, $(t-2)$, etc. are also correctly constructed from their previous states. In other words, the history starting from the beginning (or at least from time $t'$ where

it can be trusted) need to be verified. Notice that such the history grows without bounds as nodes join and leave the system.

2. The correctness of a peer's state also depends on the correctness of its neighbors' states. For example, if $p$'s state indicates that $p_l$ and $p_r$ are its left and right neighbor, then $p_l$'s and $p_r$'s state must also indicate that $p$ is the right and left neighbor respectively. As a consequence, the current state of $p$ depends on a long chain of trust consisting of many nodes whose states need to be verified for correctness.

In a large system with frequent churn, these additional checks imply very expensive verification processes. For this reason, to design an efficient, yet completely decentralized system satisfying Property 5.1.1, TPM alone is not sufficient. In other words, a new security hardware is needed. Such the hardware must be able to maintain representations of peers' states. Furthermore, for Property 5.1.1 to hold, when one peer changes its state, the state of another peer must also be updated. The following sections describe the design of a new hardware module called Trusted Tokens Module (or TTM). TTM is designed with the aim of improving DTR1 and DTA1, and at the same time being general enough to be used in other security applications.

## 5.2 Design Overview of TTM

A TTM is a security hardware that is designed to provide trusted operations involving *tokens*. A token is simply a piece of data maintained by a TTM. As seen later, it is a tuple consisting of three other data fields. The TTM can perform the following operations:

1. Creation and removal of tokens.

2. Transferring tokens from one TTM to another. This operation is irreversible, meaning that the giving TTM is no longer in possession of the tokens until they are given back to it by the other TTM.

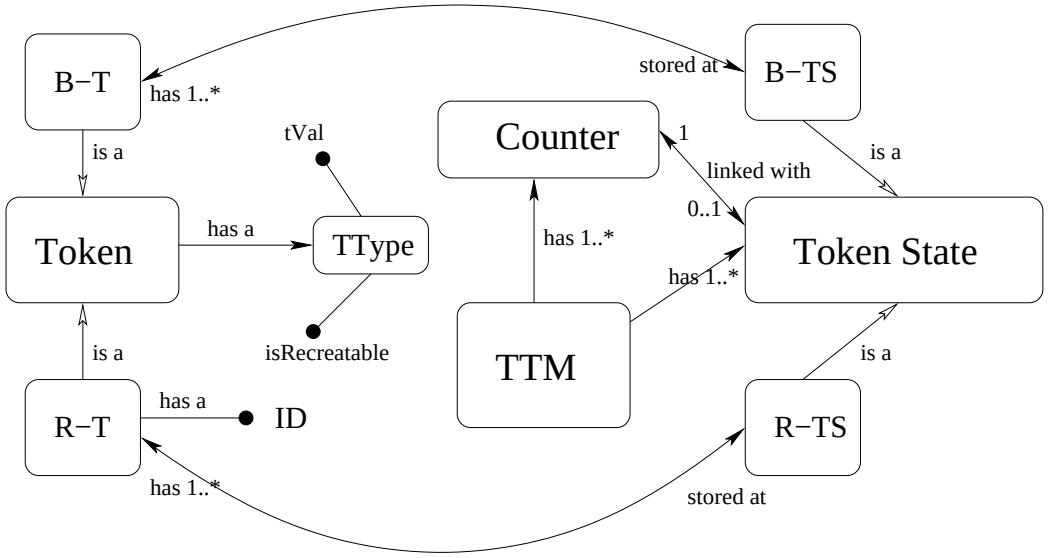



Figure 5.2.1: TTM logical design. A TTM contains a number of states and monotonic counters. A token state is linked with a unique counter. It stores either a number of R-T tokens having the same type, or a number of B-T tokens having the same type. The type of a token consists of a type ID and a flag indicating if tokens of this type could be created more than once. A token is either a B-T token, which is identified by its type, or a R-T token which is identified by its type and an identifier.

### 5.2.1 Logical Design

Figure 5.2.1 illustrates the components of a TTM and their relationships at the logical level. Basically, a TTM consists of a number of token states and monotonic counters. R-T tokens or B-T tokens of the same type are maintained at a token state.

#### 5.2.1.1 Token

Every token has a type belonging to $TType$. Every type has a value and a flag indicate if the type is unique. More specifically, a type *ttype* has:

- *tVal*: the type ID.
- *isRecreatable*: the flag indicating if this type could be created more than once.

A token is either a B-T token or a R-T token.

- A B-T token (or *bulk token*) is identified by its type *ttype*.

- A R-T token (or *range token*) is identified by its type *ttype* and an identifier belonging to $\mathcal{I}$.

For example, *ttype* represents a B-T token of type *ttype*, whereas $(ttype, id)$ represents a R-T token whose type is *ttype* and whose ID is *id*.

#### 5.2.1.2 Token State

A token state is where the TTM stores the tokens of the same type. More precisely, a token state, denoted as $ts(ttype)$ where $ttype \in TType$, belongs to one the following groups:

- B-T Token State: $ts(ttype)$ stores B-T tokens of type *ttype*. It basically stores the number of B-T tokens of type *ttype* that the TTM has.

- R-T Token State: $ts(ttype)$ stores R-T tokens of type *ttype*. It basically groups tokens with consecutive ID together into ranges of tokens, then stores the ID of the tokens at the end points of those ranges. More specifically,

  **Definition 5.2.1** (**Range of tokens**)**.** A range of R-T tokens of type *ttype*, written as $[a, b]_{ttype}$ where $a, b \in [0, 2^m)$ is a set of tokens of type *ttype* whose IDs are all the values in between and including $a$ and $b$. In other words:

  $$[a, b]_{ttype} = \{(ttype, i) \mid i = a \ \vee \ i = b \ \vee \ inBetween(i, a, b)\}$$

  The efficient datastructure for storing ranges of tokens in the token state is discussed later in Section 5.2.2.

Each token state has a pointer to a unique monotonic counter, which is incremented when the token state is updated (addition or removal of tokens). The counter is an useful time-stamping mechanism for the operations involving the token state.

#### 5.2.1.3 Counter

A monotonic counter in TTM is similar to that in TPM: it has an ID and contains a value field *cv* that can only be read or incremented. In addition, it has a flag *ced* indicating if the counter is pointed to by a token state.

### 5.2.2 Architectural Design

#### 5.2.2.1 Data Structures

A token type *ttype* is represented in TTM as a tuple $(tVal, isRecreatable)$ where $tVal$ is a bit string ($tVal \in \{0,1\}^*$) and *isRecreatable* is a boolean value indicating if this type can be created more than once.

A token state is represented in TTM as a tuple consisting of the following fields:

- $ttype \in TType$: the type of the tokens stored at the token state.
- $isRT$: a boolean value indicating if the tokens stored at this token state are R-T or B-T tokens.
- *cid*: the ID of the monotonic counter linked with this token state. It is assumed that there exists $c$ counters with the IDs $0, 1, .., (c-1)$.
- *count*: the number of tokens stored in this state.

When $isRT = T$, that is the tokens stored at the state are R-T tokens, they are grouped into ranges which are then stored in the leaves of an external $h$-level Merkle tree. A Merkle leaf has an ID $id \in [0, 2^h)$ and two other data fields: $offset \in [0, 2^m)$ and $size \in [0, 2^m)$. It represents the range $[a, b]_{ttype}$ where $a$ and $b$ are derived as follows:

$$a = (id \ll (m - h)) \oplus offset \tag{5.2.1}$$

$$b = a \oplus (size - 1) \tag{5.2.2}$$

| | |
|---|---|
| TType | $tVal \in \{0,1\}^*$<br>$isRecreatable \in \{T, F\}$ |
| Token State | $ttype \in TType$<br>$isRT \in \{T, F\}$<br>$cid \in [0, c)$<br>$count \in [0, 2^m)$<br>$h \in [0, m)$<br>$root \in \{0,1\}^*$ |
| Counter | $cid \in [0, c)$<br>$cv \in \mathbb{N}$<br>$ce \in \{T, F\}$ |
| Merkle Leaf | $id \in [0, 2^m)$<br>$offset \in [0, 2^m)$<br>$size \in [0, 2^m)$ |

Table 5.2.1: Summary of the data structures used in TTM. $c$ is the number of monotonic counters supported by the TTM. $m$ is the security parameter, whose typical value is 160.

($\ll$ is the shift-left operation). The height $h$ and the root value $root$ of the Merkle tree are included in the token state and stored inside the TTM.

It can be noticed that R-T tokens are never represented explicitly in TTM. Instead, they are stored externally in the leaves of a Merkle tree whose root node is maintained inside the TTM. The trade-off for such the compact representation of R-T tokens is the computational complexity required when checking if the TTM contains a specific R-T token.

A monotonic counter in TTM is represented by the following data fields:

- $cid$: its ID, which is in $[0, c)$.
- $cv$: its value, which is a natural number.
- $ce$: a boolean value indicating if the counter is linked with a token state.

Table 5.2.1 summarizes the data structures used in TTM.

#### 5.2.2.2 Hardware Components of a TTM.

Figure 5.2.2 sketches the main hardware components of TTM. A TTM is identified by a unique asymmetric key pair, $(KPriv, KPub)$, the private part of which is protected by the

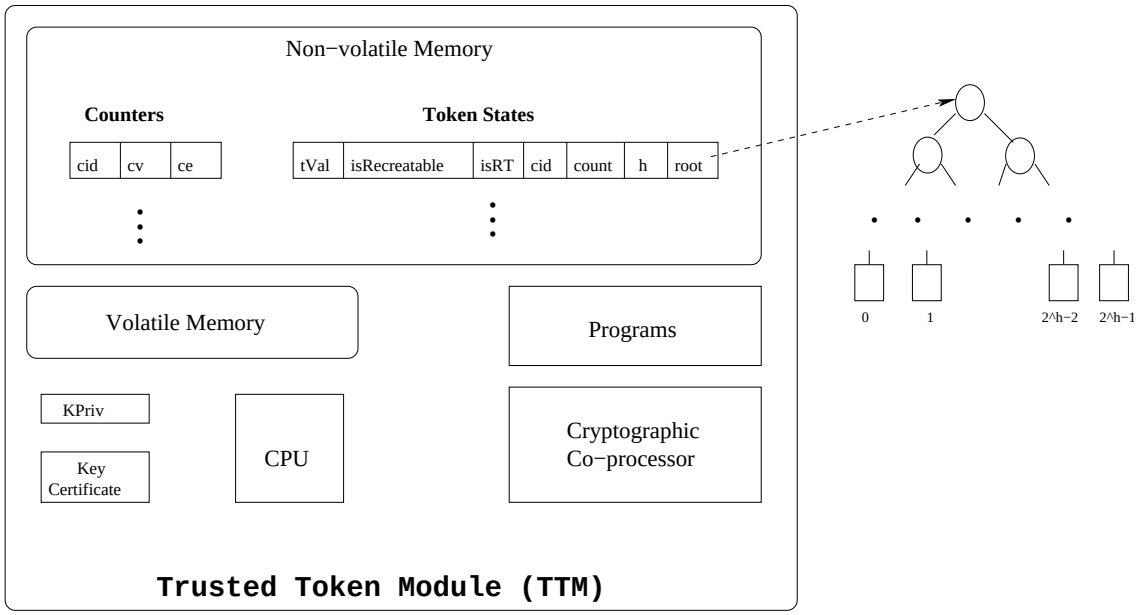



Figure 5.2.2: High-level hardware design of the Trusted Tokens Module (TTM). Note that the token state does not store R-T tokens explicitly. Instead, those tokens are stored in the leaves of an external Merkle tree whose root value is maintained inside the TTM.

device. The manufacturer supplies the key certificate embedded in the device that can be used as the authenticity proof for the TTM.

The TTM has a limited amount of non-volatile memory used by a number of token states and monotonic counters. The cryptographic co-processor is used for fast execution of basic cryptographic operations such as SHA-1, RSA signing, AES encryption. TTM's commands are provided in the Programs module. The volatile memory is used during the execution of the commands.

## 5.3 Operations

### 5.3.1 Creation and Removal of Tokens

A TTM can create a new set of tokens using `init`(n,cid,isRT,isRecreatable,h,even) command, as detailed in Figure 5.3.1. An existing token state *tks* can be removed using the `remove`(tks) command. If `init` returns successfully, $2^m$ tokens of type *ttype* are created, where *ttype* is initialized as follows:

**Command:** `init`
**Inputs:** *n* - random nonce
*cid* - counter ID
*isRT, isRecreatable*
*h* - height of the tree
*even* - flag used to initialize the tree
**Outputs:** If successful, returns the new state and a receipt
Else returns error
**Actions:**

1. Check the number of active token states, ABORT if not less than $c$
2. Check that $ct.ce = F$ where $ct.cid = cid$. ABORT if not true
3. If (*intOpt* is *True*) then
   Generate a random nonce $n'$ using the RNG engine
4. Create a new type $ttype$
   (a) Set $ttype.isRecreateable = isRecreatable$
   (b) If ($isRT = T$) then
   Set $ttype.tVal$ = SHA-1($KPub||n$)
   (c) Else
   Set $ttype.tVal$ = SHA-1($KPub||n||n'||c.cv$)
5. Create a new token state $tks$.
6. Set $tks.ttype = ttype$
7. Set $tks.isRT = isRT$, $tks.cid = cid$ and $tks.h = h$
8. Set $tks.count = 2^m$
9. Initialize $tks.root$ value
   (a) If ($isRT = F$) then
   Set $tks.root$ = *DEFAULT_ROOT*
   (b) else set $tks.root$ = *initialRoot*(*h*,*even*)
10. Increment the counter, *c.cv++*
11. Create the signature
    $\sigma = (tks; c.cv; \textit{INTERNAL\_FLAG})_{K_{priv}}$
12. Return $tks, \sigma$

NOTES: (1) the function *initialRoot*(*h*,*even*) is explained in Section 5.3.1
(2) *INTERNAL_FLAG* indicates the signature is generated using data within the device

**Command:** `remove`
**Inputs:** *tks* - token state to be removed
**Outputs:** error code if the removal fails
**Actions:**

1. Check that $tks$ is one of the active states
   ABORT if not true
2. Remove $tks$ from the set of active states
3. Set $c.ce = F$ where $c.cid = tks.cid$

Figure 5.3.1: `init`(n,cid,isRT,isRecreatable,h,even) and `remove`(tks) initializes new tokens and removes the given token state respectively.

- *ttype.tVal* is set to SHA1($Kpub||n$) or SHA1($Kpub||n||n'||c.cv$) depending on the value of the input *isRecreatable*. When *isRecreatable* $= T$, $tVal =$ SHA1($Kpub||n$), which means that tokens of this type can be created again by invoking the same command. When *isRecreatable* $= F$, $tVal =$ SHA1($Kpub||n||n'||c.cv$), which means it would be infeasible to recreate tokens of this type, because the type value uses a second nonce $n'$ generated internally by the TTM's Random Number Generator (RNG).

- *ttype.isRecreatable* is set to value of the input *isRecreatable*.

The newly created tokens are then stored in a new token state $tks$, which is initialized as follows:

- $tks.ttype$ is set to the newly created type $ttype$.

- $tks.cid$, $tks.isRT$, $tks.h$ are set to the values of the corresponding inputs.

- $tks.count$ is set to $2^m$.

- When $isRT = T$, meaning that the stored tokens are R-T tokens, the function $initialRoot(h, even)$ is used to compute the root of the initial Merkle tree whose leaves together represent all the tokens created. *initialRoot* can be implemented efficiently as follows:

  - When $even = T$, the Merkle tree consists of $2^h$ leaves, all of which are the same. More precisely, $L_i.offset = 0$ and $L_i.size = 2^{m-h}$ for all $i \in [0, 2^h)$. Because the tree is symmetric, computing the root takes only $O(h)$ steps
  - When $even = F$, the tree consists of $2^h$ leaves, in which $L_0.offset = 0$, $L_0.size = 2^m$, and $L_i.offset = L_i.size = 0$ for all $i \in [1, 2^h)$. This tree is mostly symmetric, and computing the root can also be done in $O(h)$ steps.

  It can be seen that there are at most $2.m$ possible initial roots, and they can all be kept in the device's ROM. As a result, implementing *initialRoot* simply requires reading values from the memory.

Once the initialization completes, the device *owns* $2^m$ new tokens of the newly created type. Using `init`, some types of tokens cannot be re-issued. More specifically, TTM satisfies the following property:

**Property 5.3.1** (Non-recreatability)**.** Let $tks$ be the state returned from a successful initialization in which $tks.isRecreatable = F$. It is infeasible to find another state $tks'$ such that $tks'$ is the result of another initialization, and that $tks.ttype = tks'.ttype$.

### 5.3.2 Transferring Tokens

Both R-T and B-T tokens can be transferred from one TTM to another by the protocol depicted in Figure 5.3.2. Let $p_1$, $p_2$ be the peer that requests for the tokens and gives out the tokens respectively. The protocol consists of three phases:

1. The TTM of $p_1$ prepares a request to be forwarded to $p_2$.

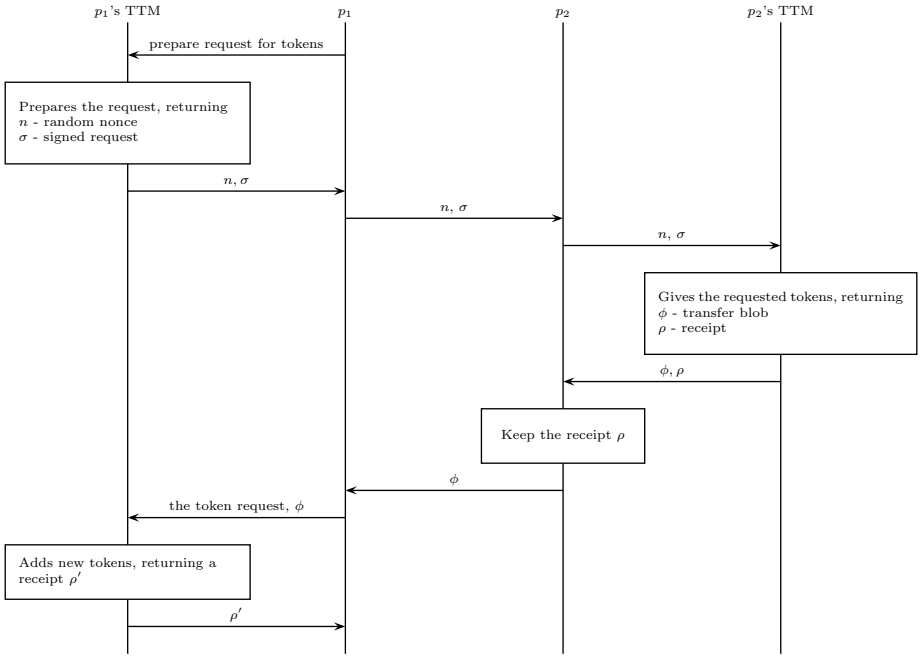



Figure 5.3.2: Information flow of the protocol for transferring tokens from $p_2$ to $p_1$

2. The TTM of $p_2$, once received the request, creates a transfer blob containing the requested tokens.

3. Finally, the TTM of $p_1$ updates its token state with the tokens included in the transfer blob.

#### 5.3.2.1 Preparing Request

In the first phase of the token transferring protocol $p_1$ asks its device to prepare a request for the tokens. It is important that the transfer requests come from the trusted devices.

`prepareBTokensTransferRequest` and `prepareRTokensTransferRequest` commands (detailed in Figure 5.3.3 and Figure 5.3.4) return the TTM-prepared requests for B-T and R-T tokens respectively. In these commands, the requested tokens' type, *ttype*, is passed as a parameter. Another parameter is the requested number of tokens in case of B-T tokens, or the requested range in case of R-T tokens. If the requesting TTM is not currently in possession of the requested token type, it creates a new token state (line 2). The new token state is different from the ones created when executing the `init` commands in two

**Command:** `prepareBTokensTransferRequest`

**Inputs:** $ttype$ - type of the tokens
$cid$ - the counter ID
quantity - number of the tokens requested

**Outputs:** If successful, return a signed request
Else return error

**Actions:**

1. Generate a random nonce $n$, using the RNG
2. If there is no active token state $tks$ such that $tks.ttype = ttype$ and $tks.isRT = F$, then: Initializes a new states
   (a) Check the number of active token states. ABORT if equal to $c$
   (b) ABORT if $c.ce = T$ where $c.cid = cid$
   (c) Create a new state $nTks$
   (d) Set $nTks.ttype = ttype$, $nTks.isRT = F$, $nTks.count = 0$
   (e) Set $nTKs.cid = cid$, $nTKs.h = 0$, $nTKs.root =$ *DEFAULT_ROOT*
3. Store $(nTks, quantity, n)$ in its memory
4. Create the signature
   $\sigma = (nTks, quantity, n, \textit{INTERNAL\_FLAG})_{K_{priv}}$
5. Return $n; \sigma$

NOTES: the *INTERNAL_FLAG* indicates that signature is generated on data within the device

**Command:** `giveBTokens`

**Inputs:** $ttype$ - type of the tokens
$cid$ - counter ID
quantity - number of the token requested
n - random nonce
$\sigma$ - signature on the request
KCert - key certificate

**Outputs:** If successful, return a transfer blob containing the given quantity of tokens, and a receipt
Else return error

**Actions:**

1. Check if it has the requested tokens
   (a) Check that there is an active token state $tks$ that $tks.ttype = ttype$, $tks.isRT = F$ and $tks.cid = cid$
   ABORT if not true
   (b) Check that $tks.count \geq quantity$
   ABORT if not true
2. Check the request
   (a) Check that $KCert$ is the valid certificate for a public key $K'_{pub}$. ABORT if not true
   (b) Check the signature
   $\sigma = (tks', quantity, n, \textit{INTERNAL\_FLAG})_{K'_{priv}}$
   ABORT if not valid, or $tks'.ttype \neq ttype$
3. Prepare the transfer blob
   (a) Create the signature
   $\phi = (tks, quantity, n, \textit{INTERNAL\_FLAG})_{K_{priv}}$
   (b) Remove *quantity* tokens from $tks$
   $tks.count = tks.count -$ *quantity*
4. Prepare a receipt
   (a) *ct.cv++* where $ct.cid = cid$
   (b) Create the signature
   $\rho = (tks, ct.cv, K'_{pub}, \textit{INTERNAL\_FLAG})_{K_{priv}}$
5. Return $\phi$, $\rho$

**Command:** `takeBTokens`

**Inputs:** $ttype$ - type of the tokens
$cid$ - counter ID
quantity - number of the token requested
n - random nonce
$\sigma$ - signature on the transfer blob
KCert - key certificate

**Outputs:** If successful, return a a receipt
Else return error

**Actions:**

1. Check that there is an active state $tks$ such that $tks.ttype = ttype$, $tks.isRT = F$ and $tks.cid = cid$
   ABORT if not true
2. Check that there is a tuple $(tks, quantity, n)$ in the memory
3. Check the validity of the transfer blob
   (a) Check that $KCert$ is the valid certificate for a public key $K'_{pub}$. ABORT if not true
   (b) Check the signature
   $\sigma = (tks', quantity, n, \textit{INTERNAL\_FLAG})_{K'_{priv}}$
   ABORT if not valid or $tks'.ttype \neq ttype$
4. Set $tks.count = tks.count + quantity$
5. Prepare a receipt
   (a) *ct.cv++* where $ct.cid = cid$
   (b) Create the signature
   $\rho = (tks, ct.cv, K'_{pub}, \textit{INTERNAL\_FLAG})_{K_{priv}}$
6. Remove the tuple $(tks, quantity, n)$ from the memory
7. Return $\rho$

Figure 5.3.3: Commands for transferring B-type tokens

**Command:** `prepareRTokensTransferRequest`

**Inputs:** $ttype$ - type of the tokens
$cid$ - counter ID
$h$ - height of the tree
a,b - representing the requested range $[a, b]_{ttype}$

**Outputs:** If successful, returns a signed request
Else returns error

**Actions:**

1. Generate a random nonce $n$, using the RNG
2. If there is no active state $tks$ such that $tks.ttype = ttype$, $tks.isRT = T$ and $tks.cid = cid$, then
   Initialize a new state
   (a) Check that the number of active states is less than *c*
   ABORT if not true.
   (b) ABORT if $c.ce = T$ where $c.cid = cid$
   (c) Create a new state $nTks$
   (d) Set $nTks.ttype = ttype$, $nTks.isRT = T$, $nTks.cid = cid$
   (e) Set $nTks.count = 0$, $nTks.h = h$ and *nTks.root=initialRootEmptyTree(auth.h)*
3. Store $(nTks, a, b, n)$ in its memory
4. Create a signature
   $\sigma = (nTks, a, b, n, INTERNAL\_FLAG)_{K_{priv}}$
5. Return $n, \sigma$

NOTES: the *initialRootEmptyTree(h)* function returns the root hash of a Merkle tree whose height is *h* and all of its leaves are empty (do not store any token)

Figure 5.3.4: Prepare a request for a range of tokens

ways:

- The field *count* is set to 0, as opposed to $2^m$ in `init`.
- For the state storing R-T tokens, the field *root* is set to *initialRootEmptyTree(h)* that returns the root of the Merkle tree whose leaves represent empty ranges. It is as opposed to using *initialRoot(h,even)* which computes the root of a different Merkle tree in which at least one of its leaves represents a non-empty range.

Finally, the TTM signs the request using a non-replay nonce generated by its internal Random Number Generator. The nonce and the signed request are returned at the end of `prepareBTokensTransferRequest` and `prepareRTokensTransferRequest`.

#### 5.3.2.2 Giving Tokens

$p_1$ forwards the signed request it received from the TTM to $p_2$ which then asks its own TTM to remove the requested tokens from the corresponding state and include them

in a transfer blob. `giveBTokens` and `giveRTokens` command (depicted in Figure 5.3.3 and Figure 5.3.5) ask the TTM to prepare the transfer blobs. Both commands take as parameters a signed request and an attestation proving its validity. `giveRTokens` requires, in addition, the Merkle's leaf containing the requested range and the verification path for the Merkle tree. The TTM then executes the following steps:

1. First, it uses the given attestation to verify that the signed request comes from another TTM.

2. Next, it checks if the requested tokens are stored in an active token state $tks$. For B-T tokens, it checks if $tks.count$ is greater than the requested quantity. For R-T tokens, the check requires two more steps:

   (a) It uses the given leaf and verification path to compute the root of the Merkle tree. The result must match with $tks.root$.

   (b) It checks that the range represented by the given leaf can be split into two smaller ranges, one of them is the requested range.

3. The requested tokens are then removed from $tks$ and as the result, the fields $tks.count$ and $tks.root$ (if applicable) are updated.

4. The transfer blob, constructed and signed by the TTM, contains the same information as in the original request.

5. Finally, the TTM increments the counter linked with $tks$, and then produces a signed receipt containing the latest $tks$.

#### 5.3.2.3 Taking Tokens

Having received the transfer blob and receipt from the TTM, $p_2$ keeps the receipt and forwards the transfer blob to $p_1$. The $p_1$'s TTM then uses the blob to add new tokens to its token state.

**Command:** `giveRTokens`

**Inputs:** $ttype$ - type of the tokens
$cid$ - counter ID
$h$ - height of the tree a,b - representing the requested range $[a, b]_{ttype}$
n - random nonce
$\sigma$ - signature on the request
KCert - key certificate
$L$ - a leaf of a Merkle tree
*path* - verification path, the leaf $L$ is *path*[0]

**Outputs:** If successful, return a transfer blob containing given range of tokens, and a receipt
Else return error

**Actions:**

1. Check the request
   (a) Check that $KCert$ is the valid attestation for a public key $K'_{pub}$. ABORT if not true
   (b) Check the signature $\sigma = (tks, a, b, n, \textit{INTERNAL\_FLAG})_{K'_{priv}}$ is valid, and $tks.ttype \neq ttype$ and $tks.isRT = F$ and $tks.h = h$. ABORT if not true
2. Check if it has the requested range
   (a) Check that there is an active token state $tks'$ such that $tks'.ttype = ttype$, $tks'.isRT = T$, $tks'.cid = cid$ and $tks'.h = h$ ABORT if not true
   (b) Check that $tks'.count > cd(b, a)$. ABORT if not true
   (c) Derive a range $[x, y]_{ttype}$ from $L$
   (d) Check that ($y$=$b$ OR $x$=$a$) AND $[a, b]_{ttype} \subseteq [x, y]_{ttype}$ ABORT if not true
   (e) Compute the root hash $rHash$, using *path* and $L$
   (f) Check that $rHash = tks'.root$. ABORT if not true
3. Create transfer blob $\phi = (tks', a, b, n, \textit{INTERNAL\_FLAG})_{K_{priv}}$
4. Update the state $tks'$
   (a) Set $L.size = L.size - |[a, b]_{ttype}|$
   (b) If ($x$=$a$), then set $L.offset = L.offset + cd(b, a)$
   (c) Compute the new root hash $newRHash$, using *path* and the updated leaf.
   (d) Set $tks'.count = tks'.count - |[a, b]_{ttype}|$
   (e) Set $tks'.root = newRHash$
5. Prepare a receipt
   (a) $ct.v$++ where $ct.cid = cid$
   (b) Create the signature $\rho = (tks', ct.cv, K'_{pub}, \textit{INTERNAL\_FLAG})_{K_{priv}}$
6. Return $\phi$, $\rho$

**Command:** `takeRTokens`

**Inputs:** $ttype$ - type of the tokens
$cid$ - counter ID
$h$ - height of the Merkle tree
a,b - representing the requested range $[a, b]_{ttype}$
n - random nonce
$\sigma$ - signature on the transfer blob
KCert - attestation
$L$ - a leaf of a Merkle tree
*path* - verification path, the leaf $L$ is *path*[0]

**Outputs:** If successful, return a a receipt
Else return error

**Actions:**

1. Check that there is an active token state $tks$ such that $tks.ttype = ttype$, $tks.isRT = T$, $tks.cid = cid$ and $tks.h = h$. ABORT if not true
2. Check that there is the tuple $(tks, a, b, n)$ in the memory ABORT if not true
3. Check the validity of the transfer blob
   (a) Check that $KCert$ is the valid key certificate for a public key $K'_{pub}$. ABORT if not true
   (b) Check the signature $\sigma = (tks', a, b, n, \textit{INTERNAL\_FLAG})_{K'_{priv}}$ is valid and $tks'.ttype = ttype$, $tks'.isRT = T$, $tks'.cid = cid$ and $tks'.h = h$. ABORT if not valid
4. Compute the root hash $rHash$ using *path* and $L$
5. Check that $rHash = tks.root$. ABORT if not true
6. If ($L.size = 0$):
   (a) Set $L.offset = cd((L.id \ll (m - h)), a)$
   (b) Set $L.size = |[a, b]_{ttype}|$
7. Else
   (a) Derive the range $[x, y]_{ttype}$ from $L$ and *leafID*
   (b) Check that $[x, y]_{ttype}$ and $[a, b]_{ttype}$ can be merged into one range. ABORT if not true
   (c) Set $L.size = L.size + |[a, b]_{ttype}|$
   (d) If $[a, y]$ is the merged range, then Set $L.offset = L.offset - |[a, b]_{ttype}|$
8. Update $tks$
   (a) Compute the new root hash $newRHash$, using *path* and the updated leaf $L$.
   (b) Set $tks.root = newRHash$
   (c) Set $tks.count = tks.count + |[a, b]_{ttype}|$
9. Prepare a receipt
   (a) $ct.v$++ where $ct.cid = cid$.
   (b) Create the signature $\rho = (tks, ct.v, K'_{pub}, \textit{INTERNAL\_FLAG})_{K_{priv}}$
10. Remove the tuple $(tks, a, b, n)$ from the memory
11. Return $\rho$

Figure 5.3.5: Commands for transferring a range of tokens

`takeBTokens` and `takeRTokens` command, depicted in Figure 5.3.3 and Figure 5.3.5, update the TTM state with the tokens included in the given transfer blob. A key certificate is passed as a parameter as a proof of the blob's authenticity. For R-T tokens, `takeRTokens` additionally requires the Merkle leaf to which the new tokens can be added, and the verification path for the Merkle tree. Given the inputs, the TTM performs the following steps:

1. First, it checks that it has previously issued a request for the tokens, and there exists an active token state $tks$ that stores the tokens. It then verifies the validity and authenticity of the transfer blob.

2. For R-T tokens:

   (a) The leaf and verification path are used to compute the root of the Merkle tree, the result of which must match with $tks.root$.

   (b) TTM checks if the range represented by the given leaf and the range included in the transfer blob can be merged together into a bigger range (line 7, Figure 5.3.5).

3. New tokens are added to $tks$, and as the result, $tks.count$ and $tks.root$ (if applicable) are updated accordingly.

4. Finally, TTM increments the counter linked with $tks$, and then produces a receipt containing the latest $tks$.

#### 5.3.2.4 Property.

Given the the above operations and Property 5.3.1, it can be seen that for R-T, non-recreatable tokens, it is not feasible to find two TTMs that have overlapping ranges of the tokens. More specifically:

**Property 5.3.2.** Consider a type *ttype* of R-T tokens where $ttype.isRecreatable = F$.

Let $[a,b]_{ttype}$ and $[x,y]_{ttype}$ be two ranges of tokens stored in some TTMs. Then:

$$\nexists i \bullet \ inBetween(i,a,b) \ \wedge \ inBetween(i,x,y)$$

### 5.3.3 Cryptographic Operations

Asymmetric keys can be generated and protected by TTM. They are wrapped in key *blobs* and stored outside of the device. Similar to wrapped keys in TPM, one can specify conditions on which the key blobs can be loaded and used. Such unwrapping conditions in TPM are usually based on PCR values. In TTM, these conditions are related to the tokens stored in the device. Figure 5.3.6 show the details of the commands for creating, loading and using wrapped keys.

#### 5.3.3.1 Creating Wrapped Keys.

TTM creates asymmetric keys and wraps them in key blobs that can be used only when certain conditions are met. `TTM_CreateWrappedKey` command takes the unwrapping condition *cond* as a parameter and returns a signed key blob. The blob contains the encrypted private part of the newly generated key and the unwrapping condition. The public part of the new key pair is returned together with the blob.

The possible values of *cond* are:

1. *NULL*: the key can be unwrapped without any condition.

2. $(ttype, cid, h, count, range, disc)$: the parameters represent the type of tokens, the counter ID, the height of a Merkle tree, the number of tokens, a specific range of tokens, and the discount value respectively. The *range* element is used only for R-T tokens. The value of *disc*, where $0 \leqslant disc < 2^{m-1}$, is directly related to the number of time this blob can be unwrapped. More specifically, let *ct* be the current number

**Command:** `TTM_CreateWrappedKey`

**Inputs:** *cond* - condition to unwrapped the key.
*NULL* if key can be unwrapped unconditionally

**Outputs:** If successful, return a key blob
Else return error

**Actions:**

1. Generate an asymmetric key pair ($Kpub'$, $Kpriv'$)
2. Set *blob* = (**enc**($Kpriv'$, $Kpub$), *cond*)
3. Create the signature
   $\sigma = (blob, INTERNAL\_FLAG)_{K_{priv}}$
4. Return $Kpub'$, *blob*, $\sigma$

**Command:** `TTM_LoadBKey`

**Inputs:** *blob* - the key blob
$\sigma$ - signature on the blob

**Outputs:** If successful, return the key handle
Else return error

**Actions:**

1. Check that $\sigma$ is the correct signature of *blob*
   ABORT if not true
2. Check that blob is of the form (**enc**($Kpriv'$, $Kpub$), *cond*)
   ABORT if not true
3. If (*cond* is not *NULL*) then
   (a) Check that *cond* is of the form $(ttype, cid, h, count, range, disc)$. ABORT if not true
   (b) Check that there is an active token state $tks$ such that $tks.ttype = ttype$, $tks.cid = cid$ and $tks.isRT = F$. ABORT if not true
   (c) Check that $tks.count \geq count + disc$
   ABORT if not true
   (d) Set $tks.count = tks.count - disc$
4. Decrypt **enc**($Kpriv'$) and set *handle* to $Kpriv'$
5. Return *handle*

**Command:** `TTM_LoadRKey`

**Inputs:** *blob* - the key blob
$\sigma$ - signature on the blob
$L$ - a Merkle tree's leaf
*path* - verification path, in which $L$ is *path*[0]

**Outputs:** If successful, return the key handle
Else return error

**Actions:**

1. Check that $\sigma$ is the correct signature of *blob*
   ABORT if not true
2. Check that blob is of the form (**enc**($Kpriv'$, $Kpub$), *cond*)
   ABORT if not true
3. If (*cond* is not *NULL*) then
   (a) Check that *cond* is of the form $(ttype, cid, h, count, range, disc)$. ABORT if not true
   (b) Check that there is an active token state $tks$ such that $tks.ttype = ttype$, $tks.cid = cid$, $tks.h = h$ and $tks.isRT = T$. ABORT if not true
   (c) Check that *range* represents $[x, y]_{ttype}$. ABORT if not true
   (d) Check that $|[x, y]_{ttype}| = count$, and $tks.count \geq count + dis$. ABORT if not true
   (e) Derive $[a, b]_{ttype}$ from the leaf $L$
   (f) Check that $[x, y]_{ttype} \subseteq [a, b]_{ttype}$, and $disc \leq cd(y, b)$
   ABORT if not true
   (g) Compute the root hash $rHash$, using *path* and $L$
   (h) Check that $rHash = tks.root$. ABORT if not true
   (i) Set $L.size = L.size - disc$
   (j) Compute new root hash $newHash$ using *path* and the updated leaf
   (k) Set $tks.root = newHash$
   (l) Set $tks.count = tks.count - disc$
4. Decrypt **enc**($Kpriv'$) and set *handle* to $Kpriv'$
5. Return *handle*

**Command:** `TTM_Decrypt`/`TTM_Sign`

**Inputs:** *keyHandle* - key handle, *DEFAULT_HANDLE* if the identity key is used
*data* - data to be decrypt/sign

**Outputs:** If successful, return the encrypted data or signature
Else return error

**Actions:**

1. If *keyHandle* is not *DEFAULT_HANDLE*, then
   set *key* to the key pointed to by the handle, namely $KPriv$.
   ABORT if such key does not exist
2. If this is a sign operation, then
   set *res* = $(data, EXTERNAL\_FLAG)_{key}$
3. If this is a decrypt operation, then
   set *res* = **dec**$(data, key)$
4. If *keyHandle* is not *DEFAULT_HANDLE*, then
   remove *key* from the memory
5. Return *res*

Figure 5.3.6: Commands for generating and loading asymmetric keys. **enc**(KPriv', KPub) encrypts KPriv' using KPub as the encryption key

of tokens with type *ttype* that belong to the TTM. The blob can be unwrapped only if the following conditions are met:

(a)

$$ct \geqslant (count + disc) \tag{5.3.1}$$

(b) For R-T tokens, let $[x, y]_{ttype}$ be the range of tokens represented by *range*, and let $[a, b]_{ttype}$ be the range belonging to the TTM, then:

$$cd(y, x) = count \ \wedge \ [x, y]_{ttype} \subset [a, b]_{ttype} \ \wedge \ cd(y, b) > disc \tag{5.3.2}$$

These conditions basically imply that the TTM must have at least $count + disc$ tokens. For R-T tokens, it must additionally have a range bigger than the one specified by *range*.

Once unwrapped successfully, a number of *disc* tokens are removed from the device. As a result, the blob can be unwrapped at most $\lfloor \frac{ct - count}{disc} \rfloor$ times.

#### 5.3.3.2 Loading Wrapped Keys.

Key blobs are stored outside of TTM, and they need to be loaded into the device, using either `TTM_LoadBKey` or `TTM_LoadRKey` command before being used. `TTM_LoadBKey` is called when unwrapping requires B-T tokens. `TTM_LoadRKey` is called when R-T tokens are needed to unwrap the blob. `TTM_LoadRKey` requires as input the details of the Merkle's leaf containing the specified range. TTM then checks if the conditions in Equation 5.3.1 and Equation 5.3.2 are met. If true, the key is unwrapped (or decrypted) and stored in the device's memory where it is accessible via a handle.

#### 5.3.3.3 Decrypting and Signing.

Loaded keys can be used for decrypting (using `TTM_Decrypt` command) or signing (using `TTM_Sign` command). Before returning the decrypted or signed data, TTM removes the

loaded key from the memory. As the consequence, the wrapped key is used at most once after loaded, meaning that the blob needs to be re-loaded for the key to be used again.

### 5.3.4 Other Operations.

#### 5.3.4.1 Transport session.

As in TPM, a number commands can be grouped and executed within a transport session.

1. `TTM_EstablishTransportSession`($exc, isLog$): opens a transport session and returns the session handle *handle*. *exc* and *isLog* flag indicate if the session is exclusive and logged. When the session is exclusive, no command can be executed outside of the session when it is still active.

2. `TTM_ExecuteTransport`($handle, comm$): executes command *comm* in the transport session. If the session is logged, the inputs and outputs of *comm* are written to a log.

3. `TTM_ReleaseTransport`($handle, nonce$): closes the session. If the session is logged, TTM also returns its signature on the log, using *nonce* as the non-replay nonce.

#### 5.3.4.2 Counters.

The monotonic counters in TTM can either be read or incremented:

1. `TTM_ReadCounter`($cid$): returns the current value of the counter *cid*.

2. `TTM_IncrementCounter`($cid$): increments the counter *cid* and returns the new value.

#### 5.3.4.3 States.

An active token state in the TTM can be read by the following command:

`TTM_ReadState`($ttype$): returns the token state storing tokens with type *ttype*.

| Condition / Usage | n times | unlimited |
|---|---|---|
| R-type tokens | Type 1 | Type 2 |
| B-type tokens | Type 3 | Type 4 |

Table 5.4.1: Access scenarios for systems and objects

## 5.4 Example With Access Control Systems

This section describes how access control systems can benefit from TTM. Tokens stored in TTM can be used as access tokens for accessing systems or objects. Gaining access to a system may require one to have particular tokens, whereas access to objects may require the ability to use wrapped keys. Table 5.4.1 summarizes different types of access policies that can be implemented with TTM. In a system supporting Type 3, for instance, access is granted up to $n$ times for agents having certain B-T tokens. In the following, examples of systems supporting these access policies are discussed.

### 5.4.1 Online Access Control.

Consider a system consisting of a number users, an admin $AD$ and an access control server $AS$. An user $U$ needs to prove his *credentials* to $AS$ in order to gain access to the system's resources. Suppose that users are equipped with TTMs. $AD$ initializes new tokens whose types are *ttype*, and distributes them to existing users. Assume for simplicity that B-T tokens are used. The admin can implement two interesting policies detailed as below.

1. $AS$ grants access only to users that have the tokens, and for unlimited number of times (Type 4 access). The protocol is as follows:

   (a) $AS$ sends a nonce to $U$.

   (b) $U$ opens a transport session with its TTM, in which `TTM_ReadState`(*ttype*) is executed. When the session is released, its log containing a token state $tks$ is signed by the TTM using the nonce given by $AS$.

   (c) $AS$ receives the log containing $tks$, and the signature on the log by $U$'s TTM.

It verifies the signature and checks that $tks.ttype = ttype$ and $tks.isRT = F$. It grants $U$ access to the resources only if $tks.count > 0$.

2. $AD$ grants limited access, namely $n$ times per user, to users who have the tokens (Type 3 in Table 5.4.1). In this scenario, $AD$ gives $n$ tokens to every user. The protocol is similar to that of Policy 1, except in the final step, before granting the access, $AS$ asks $U$ to transfer one token to it. $U$ will not be able to gain access to the system's resources once it has run out of the tokens. Users could lend their tokens to each other, but the maximum number of access granted to all users is $n \times u$ ($u$ is the number of users).

In these policies, the users can send their tokens back to $AD$ at anytime and consequently give up their abilities to access the system's resources.

### 5.4.2 Offline Access Control.

Consider a party $DP$ that wishes to implement an access policy to its objects. The objects considered in this case are signing or decryption keys. The delegated key can be used by an user $U$ only if $U$ has certain R-T tokens. This scenario resembles a Digital Right Management (DRM) system consisting of a party $DP$ that distributes encrypted objects to users. $U$ needs to have specific tokens (given by to it by a separate payment system, for example) in order to decrypt the objects. A typical use-case scenario for such the system is as follows:

- The payment system gives $U$ a range of tokens.
- $DP$ ask $U$ to create a wrapped key whose unwrapping condition are linked with the tokens that $U$ just received.
- $DP$ encrypts the data with a symmetric key $sK$ which is in turned encrypted by the public part of the wrapped key given by $U$. The encrypted data and key are sent back to $U$.

- $U$ is only able to decrypt the data if it can load and use the wrapped key to decrypt the encryption key.

$DP$ can implements the following two policies.

1. For unlimited usage (Type 2):

   (a) $DP$ asks $U$ to prove its possession of a range $[x, y]_{ttype}$. The proof can be generated by $U$ returning its TTM's signature on the token state $tks$ where $tks.ttype = ttype$ and $tks.isRT = T$. $U$ also returns the Merkle leaf containing $[x, y]_{ttype}$ and the verification path. $DP$ then computes the root of the Merkle tree and checks that it is the same as $tks.root$.

   (b) $DP$ asks $U$ to create a wrapped key inside a transport session. The wrapped key $wK$ is returned together with the proof (the session log and signature on the log from $U$'s TTM) that the key was created properly. In particular, the unwrapping condition is set to be:

$$(ttype, cid, h, |[x, y]_{ttype}|, x, y, 0)$$

   (c) After verifying that $wK$ was generated correctly, the public part of $wK$ can be certified (when $wK$ is used as a signing key) or used to encrypt data (when $wK$ is used as a decryption key) by $DP$

   To use $wK$ for signing or decrypting, $U$ must load it into the device using `TTM_LoadRKey` command. Because the unwrapping condition specifies $disc = 0$, $U$ can use $wK$ the key for as long as it has the range $[x, y]_{ttype}$.

2. For n-time usage (Type 1):

   (a) $DP$ asks $U$ to prove its possession of a range consisting of at least $(n + 2)$ tokens whose types are $ttype$. $U$ returns the proof for a range $[x, y]_{ttype}$ where $|[x, y]_{ttype}| = n + 2$. The protocol for this is similar to the one for the unlimited usage case.

(b) $U$ transfers the range $[y, y]_{ttype}$ to $DP$.

(c) Similar to the unlimited usage case, $U$ creates a wrapped key $wK$ inside a transport session and sends it back to $DP$. The main data fields of the unwrapping condition are set as follows:

i. $cond.ttype = ttype$

ii. $cond.count = 1$

iii. $cond.range = (x, x)$

iv. $cond.disc = 1$

To sign or decrypt with $wK$, $U$ needs to load it into TTM using `TTM_LoadRKey` command. *disc* tokens are removed from the range each time the key is used. Notice that step $b$ is necessary, for it ensures that $U$ cannot merge ranges together in order to use $wK$ for more than $n$ times.

The policies above could be implemented with B-T tokens (instead of R-T). However, $U$ may need to maintain different types of tokens for different wrapped keys. Because the number of states in TTM is limited, such an implementation would not scale well. Using R-T tokens, up to $2^{160}$ ranges can be maintained in one state. Therefore R-T tokens allow many wrapped keys to be linked with different non-overlapping ranges maintained by the same state.

## 5.5 Detecting Misbehavior at the Routing Layer (DTR2)

As described in the previous chapter, DTR1 consists of a set of protocols that allow a peer to verify if another is the correct root node of a given search key. DTR1's main limitation is that it relies on a centralized CA to issue neighbor certificates as peers join and leave the network. As noted earlier in this chapter, the CA plays an important role in ensuring the non-overlapping property. This section shows how the TTM's non-recreatability prop-

erty (Property 5.3.1) can be leveraged to construct a new set of protocols for DeTecting misbehavior at the Routing layer, called **DTR2**, that also satisfies the non-overlapping property without requiring a CA. In other words, DTR2 is a decentralized mechanism for detecting misbehavior at the routing layer.

Let $\mathcal{B}$ be the set of bootstrapping nodes for the P2P systems. $\mathcal{B}$ comprises a number of peers that start and never leave the network. Assume that peers are equipped with TTMs, and they leave the network gracefully.

**Initialization.** First, a node $p_{b_0} \in \mathcal{B}$ initializes a new type of R-T tokens, namely *ttype*. It distributes the new tokens to the other members of $\mathcal{B}$, in such a way that $p_{b_i}$ gets the range $[p_{b_j} \oplus 1, p_{b_i}]_{ttype}$ where $p_{b_j} \in \mathcal{B}$ is the immediate left neighbor of $p_{b_i}$ in the ID ring consisting of only nodes in $\mathcal{B}$.

**Routing.** Suppose $p_v$ is searching for the root node of a key $k$. First, the normal P2P routing protocol (Chord) is executed, which returns a node $p_d$. Next, $p_v$ asks $p_d$ for its immediate left neighbor, for which $p_l$ is returned. $p_v$ verifies that $p_l$ is indeed the correct left neighbor of $p_d$ in the current network by asking $p_d$ to prove that it has the range $[p_l \oplus 1, p_d]_{ttype}$. Let $tks$ be the token state in $p_d$'s TTM such that $tks.ttype = ttype$ and $tks.isRT = T$. The proof can be constructed from the signature of $p_d$'s TTM on the latest value of $tks$, together with the Merkle leaf representing the range, and with the verification path for the Merkle tree. $p_v$ computes the Merkle root from the leaf's details, and then verifies that the result is the same as $tks.root$.

**Joining.** A new node $p_n$ joins the network through a bootstrapping peer $p_b \in \mathcal{B}$ that it knows. The joining protocol is as follows:

1. $p_n$ asks $p_b$ to perform lookup for the root node of the key that has the same ID as $p_n$'s. Using the routing protocol above, a node $p_d$ is returned as the root node. Suppose that $p_d$ returns $p_l$ as its left neighbor, and that $p_d$ is accepted by $p_n$ after verification to be the correct root node.

2. $p_n$ asks $p_d$ to transfer the range $[p_l \oplus 1, p_n]_{ttype}$ to it.

3. Only after successfully receiving the requested range does $p_n$ become part of the network and the joining process terminate successfully.

**Leaving.** Suppose $p_n$ is leaving the network. Let $p_r$ be its immediate right neighbor. When leaving gracefully, $p_n$ simply transfers its current range of tokens to $p_r$. Only after $p_r$ has received the range is $p_n$ considered as having left the network successfully. Otherwise, it is assumed to still be in the network but fail temporarily.

## 5.6 Detecting Misbehavior at the Application Layer (DTA2)

As described in Section 4.4, DTA1 consists of protocols that enable an honest peer to detect misbehavior of another peer in a P2P-based marketplace. It has been noted early that scalability is one of the main obstacles that must be addressed before DTA1 can be realized in practice. This section describes how TTM helps improve the scalability of DTA1. The new set of protocols for DeTecting misbehavior at the Application layer, called **DTA2**, reduce the cost of verifying if a publishing peer has truthfully reported the sale offers. DTA2 also allows the publishing peer to securely remove an offer from its list, but only when agreed by the seller.

Assume that peers are equipped with TTMs. In addition, at each TTM, one monotonic counter, namely *cid*, is used exclusively for the marketplace application. For simplicity, assume further that the initial values of *cid* at all peers are 0. Let $2^{si}$ be the maximum number of types of items that is accepted by a publishing peer, for $0 \leqslant si \leqslant m$. The main idea behind DTA2 is that each peer creates $2^m$ R-T tokens, from which $2^{m-si}$ tokens are used for sale offers of each item. When a seller publishes its sale offer for an item, it takes away one token from the range of tokens reserved for the item. Once the the sale offer is accepted and the item is sold, the seller gives the token back to the publishing node.

During verification, the publishing node returns the Merkle's root, list of missing tokens and the list of sale offers for the item.

**Initialization.** A peer $p_i$ initializes a set of new tokens as follows:

1. Establishes a transport session with the TTM.
2. Executes `TTM_IncrementCounter`($cid$) inside the session.
3. Executes `init`(n,cid,isRT,isRecreatable,h,even) where:
   - $isRecreatable = F$
   - $isRT = T$
   - $h = m$
4. Closes the session. The session's log $sl_i$ is returned, together with a TTM's signature $\sigma_i$ on the log.

At the end of the initialization, $2^m$ tokens whose types are $ttype_i$ are created. The external Merkle tree storing the tokens have $2^m$ leaves, and $L_i.offset = 0$, $L_i.size = 1$ for all $i \in [0, 2^m)$. For item $d$, the sale offers stored at $p_d$ are linked with tokens in the range $[d_{si}.2^{m-si}, (d_{si}+1).2^{m-si}-1]_{ttype_i}$ where $d_{si} = \text{SHA1}(d)$ $modulo$ $2^{si}$. Notice that different peers initialize different types of tokens; but all the tokens are R-T, and non-recreatable.

**publish(d).** The seller $p$ first asks the publisher $p_d$ to prove that it has tokens of the appropriate types, then publishes its offer for $d$. More specifically:

1. $p$ asks $p_d$ to execute `TTM_ReadCounter`($cid$) and `TTM_ReadState`($ttype_{p_d}$) command inside a transport session. The results of these commands and the proof that they are executed inside a transport session are sent back to $p$. $p_d$ also sends the proof $\sigma_d$ that shows tokens whose types are $ttype_{p_d}$ were initialized correctly at $p_d$, and details of a Merkle's leaf representing one token.

2. $p$ verifies the proof, and checks that the leaf is part of the current Merkle tree. In addition, the leaf represents a token in the range $[d_{si}.2^{m-si}, (d_{si}+1).2^{m-si}-1]_{ttype_{p_d}}$ where $d_{si} = \text{SHA1}(d)\ modulo\ 2^{si}$. If the verifications are correct, $p$ asks $p_d$ to transfer to it the token represented by the returned leaf.

3. Finally, $p$ sends the payment $f$ to $p_d$ and the protocol terminates successfully.

**retrieve(d).** To retrieve the list of offers for item $d$ stored at $p_d$, the buyer $p_r$ first asks $p_d$ to prove its latest token state $tks$ storing the tokens of type $ttype_{p_d}$. This is similar to the first step in publish(d) protocol.

Next, $p_d$ returns a list of tokens missing from the range $[d_{si}.2^{m-si}, (d_{si}+1).2^{m-si}-1]_{ttype_{p_d}}$. In particular, the list consists of details of the Merkle's leaves representing the tokens in the range $[d_{si}.2^{m-si}, (d_{si}+1).2^{m-si}-1]_{ttype_{p_d}}$ that have been taken away. $p$ verifies the list by computing the Merkle's root using the leaves' details and checking that the result is the same as $tks.root$.

Finally, $p_d$ returns a list of sale offers for item $d$, together with the receipts generated when the offers were published. $p$ can verify the completeness of such the list, because the offers are tied to the missing tokens. The list contains the same number of sale offers for $d$, and therefore is smaller than the what returned in DTR1, namely $W^{p_d}$, which also contains offers of other items. In other words, TTM helps reduce the cost of verification.

**Post-transaction.** Once the buyer accepts an offer for the item $d$, it pays the listed price to the seller $p$. But before the transaction is considered as completed, $p$ returns the token it acquired when publishing the sale offer to $p_d$. The publishing peer then adds the returned token back to its state so that the token can be reused later. This mechanism increases the publishing peer's capacity as more sale offers can be stored over time.

## 5.7 Discussion

This chapter introduces a general-purpose secure hardware, called TTM, that can initialize, store and transfer tokens. TTMs can be used in P2P systems to improve the mechanisms for detecting misbehavior. Levin *et al.* [61] proposed another secure hardware abstraction, called *TrInc*, that proves to be useful for large distributed system. TrInc is basically a stripped down version of TPM, and its main operations are related to trusted monotonic counters. It is designed to combat equivocation problems in distributed systems. Such problems can be reduced to ones that ensure freshness of the information used in the systems. However, in the context of misbehavior detection mechanisms discussed in this thesis, TrInc does not offer any new feature beside what can already be found in TPM.

In this chapter, TTMs have been used to improve upon DTR1 and DTA1. In particular, DTR2 requires no centralized party, and DTA2 is more efficient than DTA1. What have not been improved are the churn models. More specifically, DTR2 still assumes nodes leave the network gracefully, and DTA2 presumes that the network is static. It would be interesting to investigate the extensions to DTR2 and DTA2 that allow for more realistic churn models. In case of DTR2, for example, when a node leaves permanently without giving its tokens to another node, those tokens will disappear forever. Therefore, a mechanism is needed to re-issue such tokens, which could be done either by a trusted party or by the network switching to a new type of tokens.

In TTM, computing the root of a Merkle tree is a frequent operation involving a number of SHA1 operations, and therefore it could be a potential bottleneck. Particularly, TTM may be asked to perform up to 160 SHA1 operations in one command. Anecdotal evidence suggests the latency of a SHA1 operation in smartcards is around $10 - 15ms$. Thus, it would take $2 - 2.4s$ for computing a Merkle root in the worst case, which is a reasonable delay. Speedup could be achieved using caching techniques.

For computing the Merkle's root, `giveRTokens`, `takeRTokens` and `TTM_LoadRKey` command take as a parameter an array consisting of up to 160 elements of 20 bytes each. This

requires TTM to reserve at least 3.2KB in input memory for each of these commands. In TPM, the size of the input buffer is up to 4KB, which is large enough to accommodate these commands. In a smartcard implementation of TTM, the input buffer is usually much smaller. One could overcome this by implementing a *computeRoot* operation that computes the Merkle's root by taking one element of the input array at a time. Consequently, the commands needs to be modified to include multiple calls to the *computeRoot* operation.

In the current design, TTM supports only a limited number of monotonic counters and states. A direct extension would be to add support for more counters and states without requiring linear extension of memory. Such an extension could be based on the work by Sarmenta *et al.* [84] in which the authors' proposal allows TPM to support up to $2^{160}$ monotonic counters. The basic idea is to maintain *virtual* states and counters in the leaves of a Merkle tree. The tree is stored outside of the device, but its root is maintained inside the TTM. When a state or a counter is updated, the corresponding leaf and verification path are provided to the TTM which then uses them to compute the root and check the result against the current root stored in the memory. If the two roots are the same, the update is accepted and a new root is evaluated and replaces the one in the TTM's memory. Having almost unlimited number of counters help lift the assumption made in Section 5.6 that the reserved counters at all peers must have the same initial values. More precisely, different applications, equipped with $2^{160}$ counters, can now be linked with unique counters that are very likely to never be used again. This allows for a node to participate in many different applications at the same time.

A number of properties of TTM have been put forward in this chapter. They look intuitive, but more formal analysis is still needed to ascertain that they are indeed true statements. For now, TTM is still at the design stage and has not yet been implemented. The next step is to implement it on smartcards. Only by doing so can its real performance be assessed and improvements to the current design be identified. Finally, since TTM aims to be a general-purposed device, it would be interesting to find other security applications

where it can be useful.

# CHAPTER 6

# FORMAL ANALYSIS

The previous chapters have presented two mechanisms for detecting misbehavior at the P2P routing layer: DTR1 and DTR2. Both DTR1 and DTR2 are designed to meet the root authenticity (or RA) property, which allows an honest peer to detect if others are misbehaving with regards to routing operations. This chapter uses formal methods based on Communicating Sequential Processes (CSP) to model these mechanisms and verify that the RA property is indeed met. Section 6.1 discusses the verification of RA property in general, and it explains the choice of CSP and the methodology adopted in verifying the property. Section 6.2 introduces CSP and the model abstraction technique called data independence which proves to be a useful tool for verification. Section 6.4 and 6.5 detail the CSP models and formal analysis showing that DTR1 and DTR2 satisfy the RA property. The analysis of DTR1 is a joint work with Mark Ryan. Finally, the related work and discussions are presented in Section 6.6.

## 6.1 Overview

### 6.1.1 Formal Verification of RA Property

In Section 4.3.1, the RA property is defined together with another property called the Neighbor Authenticity (or NA) property that concerns with the correctness of the verifica-

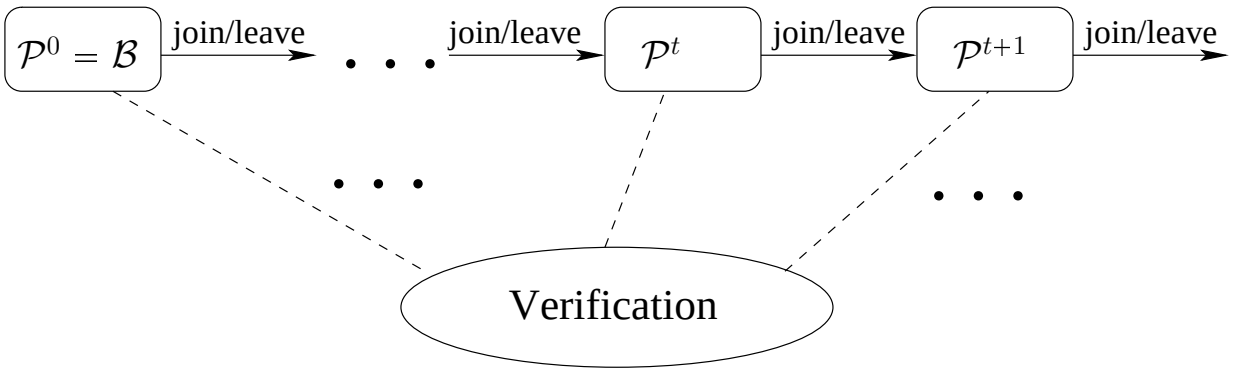



Figure 6.1.1: The P2P system evolves from time $t$ to time $t+1$ (for $t \geqslant 0$) as peers join and leave the network

tion that one node is the immediate neighbor of another (Definition 4.3.2). The NA property specifically states that for any peer $p_l$ and $p_r$, if the verification *neighborVerification*$(p_l, p_r)$ protocol returns true then $p_l$ is the immediate left neighbor of $p_r$ in the current network. Theorem 4.3.1 indicates that to prove that a system meets the RA property, it is sufficient to show that the system meets the NA property.

In the definition of RA and NA property, the system's state at time $t$ (for $t \geqslant 0$) is represented by $\mathcal{P}^t$ (the set of nodes currently in the network). The system evolves to a new state at time $(t+1)$ (represented as $\mathcal{P}^{t+1}$) when a peer joins or leaves the network. Figure 6.1.1 illustrates the evolution of the system, starting with the set of bootstrapping nodes $\mathcal{B}$. The system satisfies the NA property only if the verification outcome is correct given any state reached by system. The joining and leaving protocols are considered as atomic operations that move the system from one state to another. Verification is not performed when joining or leaving is in progress, and vice versa.

For any state $\mathcal{P}^t$, the NA property imposes that if the verification terminates successfully, what returned is correct with regards to nodes being immediate neighbors of each other. More specifically, if *neighborVerification*$(p_l, p_d)$ returns true for some values of $p_l$ and $p_r$, then $p_l$ and $p_d$ are members of $\mathcal{P}^t$ and within this set $p_d$ is the immediate right neighbor of $p_l$. More precisely:

$$\forall p'_d \in \mathcal{P}^t, p'_d \neq p_l \,.\, cd(p_d, p_l) > cd(p'_d, p_l)$$

### 6.1.2 Why CSP?

Process calculi are formal languages often used in studying properties of security systems. They offer the rigor of mathematical approaches and some degree of automation. A system is modeled as consisting of processes interacting with each other in a common environment. Difference calculi have different ways of specifying the processes and reasoning about their interactions and equivalence relations. Many calculi are constructed based on the Calculus for Communicating System (CCS) by Milner [68]. This thesis uses the Communicating Sequential Process (CSP) process algebra by Hoare [47]. The decision to select CSP over CCS as the formal language for modeling and reasoning about P2P protocols, beside the more numerous resources and references available for CSP, is based on the following arguments by Hoare [48].

While CCS is designed with the goal of providing a minimal set of basic agents and operators capable of modeling and reasoning about concurrent systems, CSP provides a broader range of useful operators so that one has more flexibility in studying the system. The primary semantics supported by CSP is the denotational semantics, whereas CCS mainly supports the operational semantics. This implies that definitions of the operators in CSP are more complex than those in CCS. Nevertheless, the complexity in the definitions is compensated by the improved algebraic properties of the operators in CSP.

Properties of a system modeled in a calculus are often expressed by *specification* processes using notations of the calculus. The system model, called the *implementation* process, is then checked against the specification for an equivalence relation. In CCS, the main relation is bisimulation equivalence which requires that the specification and the implementation are indistinguishable within the calculus. This relation is very strict, but a proof for it can be automated quite efficiently. CSP supports a more general relation known as *refinement*. The implementation is said to refine the specification if the set of observable behavior of the former is a subset of the behavior permitted by the latter. In CSP, one can define a very abstract specification at the beginning. Such the specification can be refined by different implementations whose details can be left until the later stages

of the system design. The refinement relation is less efficient to check automatically, but the Failure Divergence Refinement (FDR) model checker for CSP [37] overcomes this by reducing the specification to normal forms before checking for the correctness of an implementation. Expressing properties in CSP is not constrained by using specification processes, which means any mathematically sound description is possible, thus allowing for flexible and elegant definitions and proofs of the system's properties.

A well-known denotational semantics of CSP is based on traces. The trace set of a process consists of all the possible sequences of events produced by the process. This is strongly related to the *safety* properties — ones concerning with the absence of certain undesirable events or sequences of events. The definition of NA property (Definition 4.3.2) suggests that the property can be readily translated into a trace property and therefore fits naturally into CSP.

Finally, proofs in CSP involve concepts and operators mainly from discrete mathematics such as sets, sequences and functions. In contrast, proofs in CCS are usually more complex and innovative because they mainly involve more advanced techniques such as recursion and induction. With regards to this criterion, CSP is chosen, as a matter of personal preference, for the seemingly more straightforward approach in deriving the proofs.

### 6.1.3 Methodology

The CSP model for DTR1 (and similarly for DTR2) consists of a set of processes representing the peers, a process representing the CA, an adversary process whose aims is to break the NA property, and a process representing the honest peer that implements the *neighborVerification* protocol. These processes are combined using parallel operators. Verifying that the model meets the NA property is done by checking that it refines a specification process that trivially satisfies the NA property.

As it turns out, the CSP model for DTR1 is large and could lead to a lengthy proof. The first step in forming the proof is to derive models with smaller state spaces, called

abstractions, that the original model refines. Since refinement is a transitive relation, if the abstractions refine the specification process, then so does the original model. In this thesis, two techniques are used for deriving the abstractions.

1. Weakening the adversary. The adversary in the original model is very powerful, but it is shown that the set observable behavior of the systems does not become smaller when the adversary is weakened in particular ways.

2. Data independence technique. The original model contains data types of unlimited sizes. The data independence technique helps reduce the sizes of the data types while preserving the system's set of observable behavior.

Next, an instance of an abstraction of the original model, which consists of three peers, is implemented in FDR and checked against the specification. FDR confirmation that the instance of the model does indeed meet the NA property serves as a concrete evidence that the DTR1 (and DTR2) model of at least 3 peers satisfies the NA property. This result is generalized for any number of peers by a hand-written formal proof.

## 6.2 Introduction to CSP

### 6.2.1 Syntax

The building blocks of CSP are *events* that are communicated by *processes*.

**Events.**

| | |
|---|---|
| $a, b : DT$ | events belonging to the data type $DT$ which are communicated by CSP processes |
| $\Sigma$ | universal set of events |
| $\alpha_P$ | set of all events communicated by the process $P$ |
| $ch.v$ | communication of event $v$ on channel $ch$ |
| $\|ch\|$ | set of all events communicated on channel $ch$ |

**Processes.**

| | |
|---|---|
| $P, Q$ | CSP process |
| $STOP$ | do nothing |
| $a \rightarrow P$ | event prefix: communicate event $a$ then behave as $P$ |
| $ch?v : T \rightarrow P$ | input prefix choice: communicate any event $ch.v$ where $v \in T$, then behave as $P$ |
| $P \Box Q$, $\Box_{P_i \in Ps} P_i$ | external choice |
| *if* $b$ *then* $P$ *else* $Q$ | conditional |
| $P \underset{X}{\parallel} Q$ | $P$ and $Q$ executed concurrently, but synchronize on events in $X$ |
| $P \mid\mid\mid Q$, $\mid\mid\mid_{P_i \in Ps} P_i$ | interleaving |
| $P[\![Ren]\!]$ | renaming operator by which the events are renamed using $Ren$ |
| $P \setminus X$ | hiding operator, by which events in $X$ are hidden |

### 6.2.2 Trace semantics

The behavior of a process can be defined by it traces, each of which is a sequence of events communicated by the process. The trace semantics of some basic processes and operators are given below as examples. The trace semantics of the other operators can be found in Appendix C.

- $traces(STOP) = \{\langle\rangle\}$. This process does nothing.
- $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces(P)\}$. This process either does nothing or communicate $a$ and then follows a trace of $P$.
- $traces(\Box_{P_i \in Ps} P_i) = \bigcup\{traces(P_i) \mid P_i \in Ps\}$. This process can perform traces of any process in $Ps$.
- $traces(P[\![Ren]\!]) = \{tr \mid \exists tr' \in traces(P) \,.\, tr\ Ren^*\ tr'\}$ where $tr\ Ren^*\ tr'$ is the relation satisfied when events in $tr$ is mapped to events in $tr'$ by $Ren$.

- For a recursive process $P$ defined by $P = F(P)$,

$$traces(P) = \bigcup\{traces(F^n(STOP)) \mid n \in \mathbb{N}\}$$

  In trace semantics, $traces(P)$ is in fact the least fixed point of function $F$. When $P$ is guarded (occurrence of $P$ in $F(P)$ appears within the scope of an event prefix), $traces(P)$ is the unique fixed point of $F$.

**Refinement.** A process $Q$ is said to refine process another process $P$, written as $P \sqsubseteq Q$, if:

$$traces(Q) \subseteq traces(P)$$

In practice, $P$ represents the system's specification that defines the set of acceptable behavior, and $Q$ represents a particular implementation of the system satisfying the specifications. The refinement relation has two useful properties. First, it is transitive:

$$P \sqsubseteq Q, Q \sqsubseteq S \models P \sqsubseteq S$$

Second, it is *monotone*, meaning that for any process context $C[.]$:

$$P \sqsubseteq Q \models C[P] \sqsubseteq C[Q]$$

When designing a complex system consisting of many components, the second property allows one to use the specification of a component directly at the beginning and postpone the detailed implementation until later. This technique, in software engineering, is called *compositional development*.

**Trace properties.** The notation $P$ `sat` *Prop* is often used to denote that all the traces of $P$ satisfies the property *Prop*. More precisely:

$$\forall tr \in traces(P)\,.\,Prop(tr)$$

**Fixed point induction.** When proving trace properties for a recursive process $P$ (defined by $P = F(P)$), the following rule can be used:

$$\frac{\forall P'\,.\,P' \texttt{ sat } Prop \;\Rightarrow\; F(P') \texttt{ sat } Prop}{P \texttt{ sat } Prop} \quad [Prop(\langle\rangle)] \tag{6.2.1}$$

## 6.3 Data Independence Technique

The data independence techniques are first developed by Lazic [56] and followed by Roscoe and Broadfoot [13]. A process $P$ is data-independent of type $DT$ ($DT \subseteq \Sigma$) if it places no constraints on what $DT$ is [80, 13]. More specifically, $P$ can perform equality checks, polymorphic operations (tupling, listing) on members of $DT$, but it cannot depend on the size of $DT$ or have operations that involve values of its members, such as $>, <, +/-$.

Lazic [56] introduced a condition necessary to reduce a large (potentially infinite) data independent type $DT$ to a finite one.

**Definition 6.3.1 (PosConjEqDT).** A process $P(DT)$, which is independent of the data type $DT$, satisfies PosConjEqDT condition precisely when the failure of any equality check involving $DT$ values results in $STOP$.

The subsequent work from Roscoe and Broadfoot [80] applied transformation in the middle of the execution of a data independent process satisfying PosConjEqDT condition. Given a process $P$ satisfying PosConjEqDT, any state $P'$ reached in the operational semantic of $P$ will also meet PosConjEqDT. Applying a function $\phi$ to the data independent

type in $P'$ yields:

$$\{\phi(tr) \mid tr \in traces(P')\} \subseteq traces(\phi(P')) \tag{6.3.1}$$

where $\phi$ is applied to the values of the data type when they appear in traces or in parameters included in the definition of a process. This result can be useful when $\phi$ is a *collapsing* function that reduces size of the type. The inequality may be strict because $P'$ may carry distinct data values that get mapped to the same value by $\phi$, resulting in equality checks returning *true* in $\phi(P')$ where it returns *false* in $P'$. Examples can be found in [13] and [30].

**Extending PosConjEqDT Condition.** The equality in Equation 6.3.1 is restored when another (stricter) is met.

**Definition 6.3.2 (PosConjEqDTStrict(ET)).** A process $P(DT)$, which is independent of the data type $DT$, satisfies PosConjEqDTStrict(ET) condition, where $ET$ is a set of events containing values in $DT$, precisely when:

1. $P(DT)$ satisfies PosConjEqDT condition.
2. There is no constant of type $DT$ in $P(DT)$
3. Let $sq$ be a sequence of events. Let $e \in ET$ and let $P'(sq, e, DT)$ be a state reached by the process $P(DT)$ after the sequence $sq^\frown\langle e\rangle$. Let $\pi_{sq,DT}$ be the set of $DT$ values appeared in $sq$. Then, $P'(sq, e, DT)$ does not use values in $\pi_{sq,DT}$.

Essentially, the third condition in the definition requires that no previously used $DT$ value will appear in the future events. This allows one to safely reuse the previous values, subsequently reducing the size of $DT$. Consider the following processes:

$$P1(DT) = ch1?x : DT \rightarrow ch2?x' : DT \rightarrow ch3.x \rightarrow STOP$$
$$P2(DT) = ch1?x : DT \rightarrow ch2?x' : DT \rightarrow STOP$$

in which $DT = \{1, 2\}$. $P2(DT)$ satisfies PosConjEqDTStrict($\{|ch2|\}$) condition. Let $P_1'(\langle ch1.1\rangle, ch2.2, DT)$ be the process reached by $P1(DT)$ after the sequence $\langle ch1.1, ch2.2\rangle$. The event $ch3.1$ in $P_1'(\langle ch1.1\rangle, ch2.2, DT)$ uses the value 1, which appears in $\langle ch1.1, ch2.2\rangle$. Therefore it violates the third condition in the definition of PosConjEqDTStrict($\{|ch2|\}$). In other words, $P1(DT)$ does not satisfy PosConjEqDTStrict($\{|ch2|\}$) condition.

For any $ET$ and a process $P(DT)$ satisfying PosConjEqDTStrict(ET), let $P'(sq, e, DT)$ where $e \in ET$ be a state reached by $P(DT)$ after the sequence $sq^\frown\langle e\rangle$. Let $DT'$ be the set of all $DT$ values used in $P'(s, e, DT)$. For any set $R_{e,DT}$ satisfying the following:

1. $R_{e,DT} \subseteq DT'$

2. $|R_{e,DT}| = |\pi_{sq,DT}|$

3. $R_{e,DT} \cap \pi_{sq,DT} = \varnothing$

a transformation $\phi$ is defined as follows:

- $\phi$ maps each value in $R_{e,DT}$ to a distinct one in $\pi_{sq,DT}$.

- $\forall v \in DT' \backslash R_{e,DT}\,.\ \ \phi(v) = v$. In other words, $DT$ values that are not in $R_{e,DT}$ are mapped to themselves.

$\phi$ essentially reuses the old $DT$ values in the future events. Because the third condition in the definition of PosConjEqDTStrict(ET) states that values in $\pi_{sq,DT}$ are not checked for equality in $\phi(P'(sq, e, DT))$, the transformation $\phi$ ensures that there does not exists equality checks in $\phi(P'(sq, e, DT))$ returning true where they return false in $P'(sq, e, DT)$. I conjecture that the following equation is true:

$$traces(\phi(P'(sq, e, DT))) = \{\phi(tr) \mid tr \in traces(P'(sq, e, DT))\} \tag{6.3.2}$$

## 6.4 Verifying RA Property for DTR1

### 6.4.1 The DTR1 Model in CSP

#### 6.4.1.1 Data types.

The following data types are used:

- $Nonces = \{Nonce.id \mid id \in \mathcal{I}\}$ represents nonces.
- $\mathcal{P}^\infty$ represents all peers that could take part in the system.
- $Counts$ represents the counter values. For simplicity, assume $Counts = \mathbb{N}$.
- $Agents = \mathcal{P}^\infty \cup \{NM, CA, VF\}$ represents all agents in the network except for the adversary. This includes peers, the agent that manages nonces, the CA and the honest peer that carries out the *neighborVerification* protocol.

#### 6.4.1.2 Events.

The events used in the model come from the following sets:

1. $ChurnEvents = \{Churn.x \mid x \in \{join, leave\} \times \mathcal{P}^\infty\}$. This represents churn events.
2. $SigMessages = \{SqR.x, SqI.x \mid x \in Nonces \times \mathcal{P}^\infty \times Counts\}$. This represents TPM-signed messages containing counter values after being read or incremented.
3. $NonceMessages = \{SqN.\langle n \rangle \mid n \in Nonces\}$.
4. $CertMessages = \{Cert.x \mid x \in \mathcal{P}^\infty \times \mathcal{P}^\infty \times \mathcal{P}^\infty \times Counts\}$. This represents neighbor certificates issued by the CA to peers during churn events. The event $Cert.\langle p, l, r, c \rangle$, for example, is the neighbor certificate issued for $p$, and it asserts that $p$'s immediate neighbors when its counter value is $c$ is $l$ and $r$.
5. $Messages = ChurnMessages \cup SigMessages \cup NonceMessages \cup CertMessages$

#### 6.4.1.3 Channels.

The following channels are used in the model:

- $send, receive, take, fake : Agents.Agents.Messages$
- $learn, say : Messages$
- $output : \mathcal{P}^{\infty}.\mathcal{P}^{\infty}$
- $completeChurn : ChurnMessages$
- $unlock : Agents.\mathcal{P}^{\infty}$

#### 6.4.1.4 Nonce Manager process.

This process supplies fresh and unique nonces to other agents via the *send* channel. The nonces are used during churn and verification.

$$NonceSender(n) = \Box_{j \in Agents} send.NM.j.SqN.\langle n \rangle \rightarrow STOP$$

$$NonceManager = |||_{n \in Nonces} NonceSender(n)$$

Compared with other CSP implementation of the nonce manager, the process *NonceManager* is more efficient to compile and run in FDR (see [30] for more details).

#### 6.4.1.5 Peer processes.

Each peer is modeled by a process representing a TPM.

$$TPMs = |||_{i \in \mathcal{P}^{\infty}} TPM(i,0)$$

$$TPM(i,c) =$$

$$\Box_{\substack{n \in Nonces \\ j \in Agents}} \left( \begin{array}{l} receive.j.i.SqN.\langle n \rangle \\ \rightarrow \left( \begin{array}{l} \Box_{d \geqslant c} send.i.j.SqR.\langle n,i,d \rangle \rightarrow unlock.VF.i \rightarrow TPM(i,d) \\ \Box \Box_{d > c} send.i.j.SqI.\langle n,i,d \rangle \rightarrow unlock.CA.i \rightarrow TPM(i,d) \end{array} \right) \end{array} \right)$$

Each TPM process receives nonces on the *receive* channel, then sends back *SigMessages* events on the *send* channel. The TPM is locked when verification or churn is in progress,

which is to ensure the atomicity of these protocols (discussed early in Section 6.1). The events on the *unlock* channel signal that the churn or verification protocol has finished.

#### 6.4.1.6 CA Process.

This process models the CA that issues neighbor certificates when a peer joins or leaves the network. The high-level protocol is described in Section 4.3.2. In $CAProcess(ps, pn)$, $ps$ is the set of nodes that are currently in the network, $pn$ is the set comprising nodes that have left or are about to join the network.

First, the process receives a churn request from a peer, either for joining or leaving the network. The request is in the form of the event $receive.i.CA.Churn.\langle join, i\rangle$ or $receive.i.CA.Churn.\langle leave, i\rangle$ for some $i$. It checks that $i \in ps$ if $i$ is joining the network, and that $i \in pn$ if $i$ is leaving the network. Next, it gets fresh nonces from the Nonce Manager process, in the form of the event $receive.NM.CA.SqN.\langle n\rangle$ for some value of $n$. The nonces are forwarded to the relevant nodes (the joining/leaving node and its immediate neighbors) in the form of the event $send.CA.i.SqN.\langle n\rangle$ for some values of $i$ and $n$. The processes representing these nodes are then locked, meaning that they cannot communicate events that are not related to the churn process, until the churn process completes. Once received the signed messages from those nodes containing their newly incremented counter values, which are in the form of the event $receive.i.CA.SqI.\langle n, i, c\rangle$ for some values of $n$, $i$ and $c$, the CA issues new neighbor certificates for these nodes, then sends them to the nodes in the form of the event $send.CA.i.Cert.\langle i, l, r, c\rangle$ for some values of $i$, $l$, $r$ and $c$. Next, it outputs on channel *completeChurn*, for example $completeChurn.Churn.\langle join, i\rangle$, to signal that the churn event for some node $i$ has completed, or in other words, that the network has moved to a new state (Figure 6.1.1). Finally, the nodes involved during the churn process are unlocked by the events of the form $unlock.CA.i$, so that they can start communicating other events that are not churn-related.

$$
\begin{aligned}
CAProcess(ps,pn) \quad = \quad & |ps| == 0 \ \&\ Join0(ps,pn) \\
\Box \ & |ps| == 1 \ \&\ Join1(ps,pn) \\
\Box \ & |ps| > 1 \ \&\ JoinAndLeaveN(ps,pn)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{JoinAndLeaveN(ps,pn)} \quad = \quad \Box_{i \in pn}\ \mathit{receive.i.CA.Churn.}\langle \mathit{join}, i\rangle \rightarrow \mathit{JoinN(i,ps,pn)} \\
&\qquad\qquad \Box \Box_{i \in ps} \left( \begin{array}{l} \mathit{receive.i.CA.Churn.}\langle \mathit{leave}, i\rangle \\ \rightarrow\ \mathit{if}\ |ps| > 2\ \mathit{then}\ \mathit{LeaveN(i,ps,pn)} \\ \quad \mathit{else}\ \mathit{Leave2(i,ps,pn)} \end{array} \right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{JoinN(i,ps,pn)} = \\
&\Box_{n1,n2,n3 \in Nonces} \left( \begin{array}{l} \mathit{receive.NM.CA.SqN.}\langle n1\rangle \rightarrow \mathit{send.CA.i.SqN.}\langle n1\rangle \\ \rightarrow \Box_{c1,c2,c3 \in Counts} \left( \begin{array}{l} \mathit{receive.i.CA.SqI.}\langle n1, i, c1\rangle \rightarrow \\ \quad \mathit{let}\ S = ps \cup \{i\} \\ \qquad (l, r) = \mathit{neighbor}(i, S) \\ \qquad (l1, r1) = \mathit{neighbor}(l, S) \\ \qquad (l2, r2) = \mathit{neighbor}(r, S)\ \mathit{within} \\ \qquad\quad \mathit{send.CA.i.Cert.}\langle i, l, r, c1\rangle \\ \qquad \rightarrow \mathit{receive.NM.CA.SqN.}\langle n2\rangle \\ \qquad \rightarrow \mathit{send.CA.l.SqN.}\langle n2\rangle \\ \qquad \rightarrow \mathit{receive.l.CA.SqI.}\langle n2, l, c2\rangle \\ \qquad \rightarrow \mathit{send.CA.l.Cert.}\langle l, l1, i, c2\rangle \\ \qquad \rightarrow \mathit{receive.NM.CA.SqN.}\langle n3\rangle \\ \qquad \rightarrow \mathit{send.CA.r.SqN.}\langle n3\rangle \\ \qquad \rightarrow \mathit{receive.r.CA.SqI.}\langle n3, r, c3\rangle \\ \qquad \rightarrow \mathit{send.CA.r.Cert.}\langle r, i, r2, c3\rangle \\ \qquad \rightarrow \mathit{completeChurn.Churn.}\langle \mathrm{join}, i\rangle \\ \qquad \rightarrow \mathit{unlock.CA.i} \rightarrow \mathit{unlock.CA.l} \\ \qquad \rightarrow \mathit{unlock.CA.r} \rightarrow \mathit{CAProcess(S,pn\backslash\{i\})} \end{array} \right) \end{array} \right)
\end{aligned}
$$

The function $\mathit{neighbor}(p, ps)$ returns the left and right neighbor of $p$ in the set $ps$. More precisely,

$$\mathit{neighbor}(p, ps) = (\mathit{left}(p, ps), \mathit{right}(p, ps))$$

$$\mathit{left}(p, ps) = l \quad \text{if} \ \ l \in ps \ \wedge\ \forall p' \in ps \backslash \{l\}\,.\, cd(p, l) \leqslant cd(p', l)$$

$$\mathit{right}(p, ps) = r \quad \text{if} \ \ r \in ps \ \wedge\ \forall p' \in ps \backslash \{r\}\,.\, cd(r, p) \leqslant cd(r, p')$$

The details of other sub-processes, namely *Join0(ps,pn), Join1(ps,pn), Leave2(ps,pn)* and *LeaveN(ps,pn)*, can be found in Appendix D.

#### 6.4.1.7 Adversary Process.

This process models the adversary whose aim is to break the NA property. In particular, it attempts to make the Verifier accept incorrectly that two nodes are immediate neighbors of each other in the current network. The adversary is modeled as having control of all peers, which means that it could ask TPMs to increment their counters and generate signed messages on their values. However, it cannot fake the signatures coming from TPMs, which reflects the property that TPM operations are trusted.

$$
\begin{aligned}
\mathit{MemoryNonce(n)} &= \mathit{learn.SqN}.\langle n\rangle \rightarrow \mathit{ReplayNonce(n)} \\
\mathit{ReplayNonce(n)} &= \mathit{say.SqN}.\langle n\rangle \rightarrow \mathit{ReplayNonce(n)} \\
\\
\mathit{MemorySigR(n,i,c)} &= \mathit{learn.SqR}.\langle n,i,c\rangle \rightarrow \mathit{ReplaySigR(n,i,c)} \\
\mathit{MemorySigI(n,i,c)} &= \mathit{learn.SqI}.\langle n,i,c\rangle \rightarrow \mathit{ReplaySigI(n,i,c)} \\
\mathit{ReplaySigR(n,i,c)} &= \mathit{say.SqR}.\langle n,i,c\rangle \rightarrow \mathit{ReplaySigR(n,i,c)} \\
\mathit{ReplaySigI(n,i,c)} &= \mathit{say.SqI}.\langle n,i,c\rangle \rightarrow \mathit{ReplaySigI(n,i,c)} \\
\\
\mathit{MemoryCert(i,l,r,c)} &= \mathit{learn.Cert}.\langle i,l,r,c\rangle \rightarrow \mathit{ReplayCert(i,l,r,c)} \\
\mathit{ReplayCert(i,l,r,c)} &= \mathit{say.Cert}.\langle i,l,r,c\rangle \rightarrow \mathit{ReplayCert(i,l,r,c)} \\
\\
\mathit{Memory} &= \Vert\!\Vert_{n\in Nonces} \mathit{MemoryNonce(n)} \\
&\quad \Vert\!\Vert_{n\in Nonces, i\in\mathcal{P}, c\in Counts} \left( \begin{array}{c} \mathit{MemorySigR(n,i,c)} \\ \Vert\!\Vert\ \mathit{MemorySigI(n,i,c)} \end{array} \right) \\
&\quad \Vert\!\Vert_{i,l,r\in\mathcal{P}, c\in Counts} \mathit{MemoryCert(i,l,r,c)} \\
\\
\mathit{ChurnInitiator} &= \Box_{i\in\mathcal{P}} \left( \begin{array}{c} \mathit{say.Churn}.\langle \text{join}, i\rangle \rightarrow \mathit{ChurnInitiator} \\ \Box\ \mathit{say.Churn}.\langle \text{leave}, i\rangle \rightarrow \mathit{ChurnInitiator} \end{array} \right) \\
\mathit{Adversary} &= \mathit{Memory} \Vert\!\Vert \mathit{ChurnInitiator}
\end{aligned}
$$

The adversary can start a churn event for any node in the network, modeled by the *ChurnInitiator* sub-process. It can eavesdrop (on the *learn* channel), remember, and replay (on the *say* channel) all messages sent between peers and the other agents in the network. Its infinite memory for remembering messages is modeled by the *Memory* sub-process. In the literature, modeling such the powerful adversary has been used when analyzing security protocols in CSP [83].

#### 6.4.1.8 Verifier Process.

This process models the *neighborVerification* protocol described in Section 4.3.2, i.e. it models the honest peer that verifies if two other nodes are immediate neighbors of each other. The process sends and receives messages using the *send* and *receive* channel respectively. The $output.l.r$ event, for example, indicates that the verifying peer accepts that $l$ is the immediate left neighbor of $r$ in the current state of the system.

*VerifierProcess* =

$$\Box_{n \in Nonces} \left( \begin{array}{l} receive.NM.VF.SqN.\langle n \rangle \\ \rightarrow \Box_{i,l,r \in \mathcal{P}^\infty} \left( \begin{array}{l} send.VF.i.SqN.\langle n \rangle \\ \rightarrow \Box_{c \in Counts} \left( \begin{array}{l} receive.i.VF.SqR.\langle n, i, c \rangle \\ \rightarrow receive.i.VF.Cert.\langle i, l, r, c \rangle \\ \rightarrow \textit{if } l = r \textit{ and } l = i \textit{ then} \\ \quad output.i.i \rightarrow unlock.VF.i \\ \quad \rightarrow STOP \\ \textit{else } VerifierProcessN(l,i) \end{array} \right) \end{array} \right) \end{array} \right)$$

*VerifierProcessN(l,i)* =

$$\Box_{n \in Nonces} \left( \begin{array}{l} receive.NM.VF.SqN.\langle n \rangle \rightarrow send.VF.r.SqN.\langle n \rangle \\ \rightarrow \Box_{cl \in Counts} \left( \begin{array}{l} receive.l.VF.SqR.\langle n, l, cl \rangle \\ \rightarrow \Box_{ll \in \mathcal{P}^\infty} \left( \begin{array}{l} receive.l.VF.Cert.\langle l, ll, i, cl \rangle \\ \rightarrow output.l.i \rightarrow unlock.VF.l \\ \rightarrow unlock.VF.i \rightarrow STOP \end{array} \right) \end{array} \right) \end{array} \right)$$

Notice that *VerifierProcess* models the verification that happens only once, unlike *CAProcess* which allows for the many churn events to happen. There are two reasons for it:

- *VerifierProcess* is combined with the other processes by a parallel operator, which means that the verification can occur at any state of the system. If on all the traces of the system model, verification returns the correct result, then it can be concluded that the *neighborVerification* protocol returns the correct result for any state of the system (or the $NA$ property is satisfied).

- Form the adversary's perspective, no more benefit is gained by having the veri-

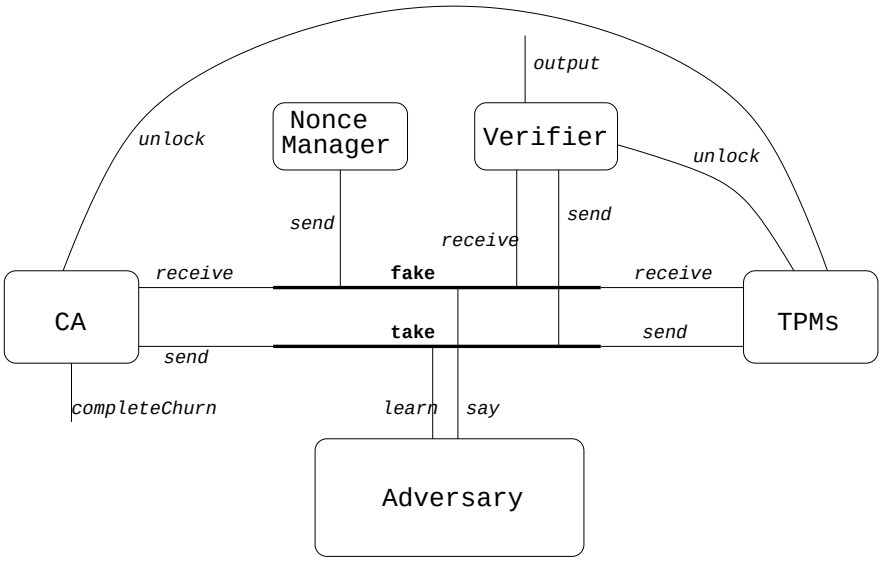



Figure 6.4.1: Channels and processes in the model of DTR1

fication executed more than once. If *VerifierProcess* were defined recursively, so that the verification protocol would occurs more than once in a trace, the results of these verification protocols would be completely independent of each other. In other words, the adversary cannot rely on running more verifications to help it succeed in making another verification return the wrong result.

#### 6.4.1.9 Putting it together.

Figure 6.4.1 illustrates how the processes are joined together to create the complete model for DTR1. To model the adversary eavesdropping on the communication between peers and the other agents, the *send* channel, which is used by the CA, TPM and Verifier to send messages, are mapped to the *take* channel to which the adversary process listens (by renaming its *learn* channel to *take*). To model the adversary replaying messages, its *say* channel is mapped to the *fake* channel on which messages are received by the CA, TPM and Verifier process. Notice that the *send* channel used by the Nonce Manager process is mapped to the *fake* channel, so that the communication between Nonce Manager and the other agents is not accessible by the adversary.

Table 6.4.1 details the renaming relations used by the processes. $RA_1$ and $RA_2$, used

| Name | Details | Applied to |
|---|---|---|
| $RAd_1$ | $learn \leftarrow take.i.j \mid i,j \in Agents, \{i,j\} \cup \mathcal{P} \neq \emptyset$ | $Adversary$ |
| $RAd_2$ | $say \leftarrow fake.i.j \mid i,j \in Agents, \{i,j\} \cup \mathcal{P} \neq \emptyset$ | $Adversary$ |
| $RCom_1$ | $send \leftarrow take$ | $TPMs$, $CAProcess$ and $VerifierProcess$ |
| $RCom_2$ | $receive \leftarrow fake$ | $TPMs$, $CAProcess$ and $VerifierProcess$ |
| $RNonce_1$ | $send.NM.i \leftarrow take.NM.i \mid i \in \mathcal{P}$ | $NonceManager$ |
| $RNonce_2$ | $send.NM.j \leftarrow fake.NM.j \mid j \notin \mathcal{P}$ | $NonceManager$ |
| $\chi_i$ | $\{\|take.i.a, fake.a.i \mid i \in \mathcal{P}, a \in Agents\|\}$ | |
| $\chi_e$ | $\{\|take.a.i, fake.i.a \mid i \in \mathcal{P}, a \in Agents\|\}$ | |

Table 6.4.1: Renaming relations and synchronization sets

by the adversary process to map *learn* and *say* channels to *take* and *fake*, are many-to-one mappings. They introduce nondeterminism, which increases the adversary's power. In particular, the mapping allows the adversary to send a message originally intended for one process to another different process.

$$
\begin{aligned}
Network &= \Big( Adversary \underset{\chi_i}{\parallel} TPMs \Big) \setminus \chi_i \\
CAandVFProcess &= CAProcess(\{\}, \mathcal{P}) \;|||\; VerifierProcess \\
OtherAgents &= \Big( NonceManager \underset{\{|fake.NM|\}}{\parallel} CAandVFProcess \Big) \\
Impl &= \Big( OtherAgents \underset{\chi_e}{\parallel} Network \Big) \setminus \{|take, fake, unlock|\}
\end{aligned}
$$

### 6.4.2 Specification

The NA property is formalized by a specification process, against which the DTR1 model is checked for trace refinement. Such the process specifies the system's evolution and the correct outcomes of the *neighborVerification*.

$$
\begin{aligned}
Spec(ps, pn) = & \Box_{i \in pn} completeChurn.Churn.\langle join, i\rangle \rightarrow Spec(ps \cup \{i\}, pn \setminus \{i\}) \\
& \Box \Box_{i \in ps} completeChurn.Churn.\langle leave, i\rangle \rightarrow Spec(ps \setminus \{i\}, pn \cup \{i\}) \\
& \Box \Box_{i \in ps} output.i.right(i,ps) \rightarrow Spec(ps, pn)
\end{aligned}
$$

### 6.4.3 Verification

To check that the DTR1 model satisfies the NA (and subsequently RA) property, one can show the following:

$$Spec(\{\}, \mathcal{P}^{\infty}) \sqsubseteq Impl \tag{6.4.1}$$

It can be seen that *Impl* is large and complex. Even for a system with a small number of peers, *Impl* has too many states and transitions for it to be checked automatically by FDR. In this thesis, the approach for verifying Equation 6.4.1 involves first finding an abstraction for *Impl* that has a smaller state space. Next, a small instance of the abstraction is implemented in FDR, which confirms that the refinement relation is true. The final step is to derive a general proof for the abstraction model of any size. In the following, the main steps in arriving at the final the proof are presented informally in order. The general proof is shown in Appendix F. More details can be found in [30].

#### 6.4.3.1 Weakening the adversary.

The adversary modeled in *Impl* is given infinite memory which allows it to remember and replaying messages. Consider the weakened adversary that has no memory and therefore is restricted to only relay messages. More specifically, the weakened adversary is constructed by replacing $MemoryNonce(n)$, $MemorySigI(n,i,c)$, $MemorySigR(n,i,c)$ and $MemoryCert(i,l,r,c)$ with the following processes:

$$\begin{aligned}
RelayNonce(n) &= learn.SqN.\langle n\rangle \rightarrow say.SqN.\langle n\rangle \rightarrow STOP \\
RelaySigR(n,i,c) &= learn.SqR.\langle n,i,c\rangle \rightarrow say.SqR.\langle n,i,c\rangle \rightarrow STOP \\
RelaySigI(n,i,c) &= learn.SqI.\langle n,i,c\rangle \rightarrow say.SqI.\langle n,i,c\rangle \rightarrow STOP \\
RelayCert(i,l,r,c) &= learn.Cert.\langle i,l,r,c\rangle \rightarrow say.Cert.\langle i,l,r,c\rangle \rightarrow STOP
\end{aligned}$$

Let *Impl1* be the new model constructed by replacing the old adversary with the weakened one, then it can be shown that:

$$traces(Impl) = traces(Impl1) \tag{6.4.2}$$

#### 6.4.3.2 Reducing *Nonces* using the data independence technique.

In *Impl1*, the Nonce Manager supplies fresh nonces to the other agents from the potentially infinite set *Nonces*. The model is therefore dependent of type *Nonces*, because the adversary and Nonce Manager process are constructed using parallel operators indexed over *Nonces*. To apply the data independence technique for *Nonces*, another abstraction to *Impl1* that is independent of *Nonces* must be found. First, notice that *Nonces* can be divided into three distinct sets: $Nonces_{ad}$, $Nonces_{ca}$ and $Nonces_{vf}$ that contain nonces supplied to the adversary, CA and Verifier process respectively. Let *Impl2* be the model derived from *Impl1* as follows:

- $Nonces_{ad}$ is removed, meaning that the data type for nonces used in *Impl2* is $Nonces_{vf} \cup Nonces_{ca}$. It is shown in [30] that removing these nonces does not change the trace set of *Impl1*.

- The Nonce Manager and adversary process are rewritten in recursive form so that they no longer consist of parallel sub-processes indexed over the set $Nonces_{vf}$ and $Nonces_{ca}$. For example, the adversary sub-process that relay TPM signatures can be rewritten as:

$$\begin{aligned} \mathit{RelaySigR(i)} &= \mathit{learn?SqR.}\langle n,i,c\rangle \\ &\quad \rightarrow \left( \begin{array}{l} \quad \mathit{RelaySigR(i)} \\ \Box\ \mathit{say.SqR.}\langle n,i,c\rangle \rightarrow \mathit{RelaySigR(i)} \end{array} \right) \\ \mathit{RelaySigI(i)} &= \mathit{learn?SqI.}\langle n,i,c\rangle \\ &\quad \rightarrow \left( \begin{array}{l} \quad \mathit{RelaySigI(i)} \\ \Box\ \mathit{say.SqI.}\langle n,i,c\rangle \rightarrow \mathit{RelaySigI(i)} \end{array} \right) \end{aligned}$$

It is shown in the technical report [30] that:

$$\mathit{traces}(\mathit{Impl1}) \subseteq \mathit{traces}(\mathit{Impl2}) \tag{6.4.3}$$

All the sub-processes of *Impl2*, except for the Nonce Manager process, are independent of $Nonces_{vf}$ and $Nonces_{ca}$. They satisfy the PosConjEqDTStrict($ET_{vf}$) and

PosConjEqDTStrict($ET_{ca}$) condition where:

$$ET_{vf} = \{fake.NM.VF.SqN.\langle n \rangle \mid n \in Nonces_{vf}\}$$

$$ET_{ca} = \{fake.NM.CA.SqN.\langle n \rangle \mid n \in Nonces_{ca}\}$$

Let *Impl3* be derived from *Impl2* by removing the Nonce Manager process and replacing $Nonces_{vf}$ and $Nonces_{ca}$ with $\{n_{vf}\}$ and $\{n_{ca}\}$ respectively. Using the collapsing functions that map $Nonces_{vf}$ and $Nonce_{ca}$ to $\{n_{vf}\}$ and $\{n_{ca}\}$ respectively, Equation 6.3.2 and the fact that events containing nonces are hidden in the model, it is shown in [30] that:

$$traces(Impl2) = traces(Impl3) \tag{6.4.4}$$

#### 6.4.3.3 Reducing *Counts* using the data independence technique.

*Impl3* is dependent of type *Counts*, because the operator '>' and parallel operator indexed over *Counts* are used in the TPMs and adversary process. Before reducing *Counts* to a smaller size, a new abstraction to *Impl3* that is independent of *Counts* needs to be found. Let *Impl4* be the model derived from *Impl3* as follows:

- The TPMs process is rewritten so that new counter values are selected from the set of values that have not been used, instead of restricting them to be greater than the current value. This replacement is sound, because in the original model, the '>' operator is used only to guarantee the freshness of the new value.

- The adversary process is rewritten in recursive form so that it no longer comprises parallel sub-processes indexed over *Counts*.

The following holds:

$$traces(Impl3) \subseteq traces(Impl4) \tag{6.4.5}$$

*Impl4* is independent of type *Counts* and in fact satisfies the PosConjEqDT condition. Let *Abstraction* be the model derived from *Impl4* in which the only counter value used is

$c_d$. By applying a collapsing function that maps *Counts* to $\{c_d\}$, it is shown in [30] that:

$$traces(Impl4) \subseteq traces(Abstraction) \tag{6.4.6}$$

This abstraction does not allow replay attacks (even though only two nonces and one counter value are ever used in *Abstraction*), because the adversary's memory has been removed.

#### 6.4.3.4 Automated verification.

The state space of *Abstraction* is smaller than that of *Impl*. Because $Abstraction \sqsubseteq Impl$, to prove that Equation 6.4.1 holds, it is sufficient to show:

$$Spec(\{\}, \mathcal{P}^\infty) \sqsubseteq Abstraction \tag{6.4.7}$$

To provide preliminary evidence that Equation 6.4.7 holds, the refinement is checked for a small model in which $|\mathcal{P}^\infty| = 3$. $Spec(\{\}, \mathcal{P}^\infty)$ and *Abstraction* are implemented in FDR. The detailed implementation can be found in Appendix E. The refinement check returns **true** after evaluating **13,501,797** states and **73,831,002** transitions. This automated proof confirms that Equation 6.4.7 is true for the DTR1 model of at least 3 peers.

#### 6.4.3.5 Generalizing the automated proof.

To prove that Equation 6.4.7 holds for $\mathcal{P}^\infty$ of any size, it is necessary to show that $traces(Abstraction) \subseteq traces(Spec(\{\}, \mathcal{P}^\infty))$. The proof is constructed by induction as follows (for more details, see Appendix F):

1. (Base case). Let $tr$ be a trace of *Abstraction* such that $tr \upharpoonright \{|completeChurn|\} = \langle\rangle$ ($\upharpoonright$ is the restriction operator, for example $sq \upharpoonright X$ removes non-$X$ elements from $sq$). Then $tr \in traces(Spec(\{\}, \mathcal{P}^\infty))$.

2. (Inductive case). For any $\theta \neq \langle\rangle$, let $tr$ be a trace of *Abstraction* such that:

$$tr \upharpoonright \{|completeChurn|\} = \theta \ \wedge \ tr \in traces(Spec(\{\}, \mathcal{P}^{\infty}))$$

Let $tr'$ be another trace of *Abstraction*, then:

$$\forall e \,.\, tr' \upharpoonright \{|completeChurn|\} = \theta \hat{} \langle e \rangle \ \Rightarrow \ tr' \in traces(Spec(\{\}), \mathcal{P}^{\infty})$$

## 6.5 Verifying RA Property for DTR2

This section presents a CSP model for DTR2, and verifies that the NA property is met by the model. Unlike the initial DTR1 model described in Section 6.4, the DTR2 model uses only 2 nonces, and its adversary is restricted to only relaying messages. The model is not as realistic as the one in which the infinite number of nonces are allowed and the adversary has infinite memory for remembering and replaying messages. However, the model abstraction techniques (weakening the adversary and data independence) used in Section 6.4 suggest that such the complex model would be likely to refine the simpler model presented in this section.

### 6.5.1 The Model in CSP

All the notations, data types, channels and renaming relations used in the DTR2 model are similar to those used in the DTR1 model. The signed messages generated by TTMs are represented by events in the following set:

1. $RangeMessages = \{Ran.\langle x \rangle \,|\, x \in \mathcal{P}^{\infty} \times \mathcal{P}^{\infty} \times \mathcal{P}^{\infty}\}$. This represents the transfer blobs exchanged among peers. For example, $Ran.\langle i, l, r \rangle$ models the transfer blob created by $i$ and containing the range $[l, r]_{ttype}$ where *ttype* is the default token type known to all peers.

2. $CertMessages = \{Cert.\langle x \rangle \,|\, x \in Nonces \cup (Nonces \times \mathcal{P}^\infty \times \mathcal{P}^\infty)\}$. This represents the signed messages from TTM certifying the ranges that it has. For example, $Cert.\langle i, l, r \rangle$ models the signed message from $i$ asserting that it has the range $[l, r]_{ttype}$. This type of message is used for verification, whereas *RangeMessages* are used during churn events when peers transfer their ranges to each other.

An event on channel $join, leave : \mathcal{P}^\infty.\mathcal{P}^\infty$ signals that a churn event has completed and the system has moved to a new state. This is similar to the *completeChurn* channel in the DTR1 model.

#### 6.5.1.1 Peer Processes.

Each peer is modeled by a process representing a TTM. Peer $i$ is considered as having joined the network (an event on the $join.i$ channel is performed) once it has received a range $[l, i]_{ttype}$. The peer can transfer tokens belonging to its range (or a sub-range) to other peers. It is considered as having left the network once its entire range has been transferred to its neighbor, and the neighbor has added the range to its own range. At the beginning, there exists a bootstrapping node $b$ that has the entire range of tokens.

$$
\begin{aligned}
TTMN(i) \quad = \quad & receive.VF.i.SqN.\langle n_{vf} \rangle \rightarrow send.i.VF.Cert.\langle n_{vf} \rangle \\
& \rightarrow unlock.VF.i \rightarrow TTMN(i) \\
& \Box \Box_{l,r} \; receive.r.i.Ran.\langle r, l, i \rangle \rightarrow join.i.r \rightarrow TTM(i,l) \\
\\
TTM(i,l) \quad = \quad & receive.VF.i.SqN.\langle n_{vf} \rangle \rightarrow send.i.VF.Cert.\langle n_{vf}, l, i \rangle \\
& \rightarrow unlock.VF.i \rightarrow TTM(i,l) \\
& \Box \Box_{j,ll} \; receive.j.i.Ran.\langle l, ll, l \rangle \rightarrow leave.l.i \rightarrow TTM(i,ll) \\
& \Box \Box_{j} \; \textit{if } mid(j,l,i) \textit{ then } send.i.j.Ran.\langle i, l, j \rangle \rightarrow TTM(i,j) \\
& \qquad \textit{else } send.i.j.Ran.\langle i, l, i \rangle \rightarrow TTMN(i) \\
\\
TTMs \quad = \quad & \Box_{b \in \mathcal{P}^\infty} \left( TTM(b,b) \;|||\; |||_{i \neq b} TTMN(i) \right)
\end{aligned}
$$

#### 6.5.1.2 Verifier Process.

$$\begin{aligned} \mathit{VerifierProcess} \quad = \quad & \Box_{r} \; \mathit{send.VF.r.SqN.}\langle n_{vf}\rangle \rightarrow \mathit{receive.r?VF.Cert.}\langle n_{vf}, l, r\rangle \\ & \rightarrow \mathit{if}\ l = r\ \mathit{then}\ \mathit{output.l.r} \rightarrow \mathit{unlock.VF.r} \rightarrow \mathit{STOP} \\ & \quad \mathit{else}\ \mathit{send.VF.l.SqN.}\langle n_{vf}\rangle \rightarrow \mathit{receive.l.VF?Cert.}\langle n_{vf}, ll, l\rangle \\ & \quad \rightarrow \mathit{output.l.r} \rightarrow \mathit{unlock.VF.l} \rightarrow \mathit{unlock.VF.r} \rightarrow \mathit{STOP} \end{aligned}$$

#### 6.5.1.3 Adversary Process.

Notice that the verification process checks only two certificates, therefore the adversary could do no better than relaying only two messages of type *CertMessages*.

$$\begin{aligned} \mathit{RelayNonce} &= \mathit{learn.SqN.}\langle n_{vf}\rangle \rightarrow \mathit{say.SqN.}\langle n_{vf}\rangle \rightarrow \mathit{RelayNonce} \\ \mathit{RelayCert} &= \mathit{learn?Cert.}\langle n_{vf}, l, r\rangle \rightarrow \mathit{say.Cert.}\langle n_{vf}, l, r\rangle \rightarrow \mathit{STOP} \\ \mathit{RelayRange(i,l,r)} &= \mathit{learn.Ran.}\langle i, l, r\rangle \rightarrow \mathit{say.Ran.}\langle i, l, r\rangle \rightarrow \mathit{RelayRange(i,l,r)} \\ \mathit{RelayRanges} &= |||_{i,l,r}\ \mathit{RelayRange(i,l,r)} \\ \mathit{Adversary} &= \mathit{RelayNonce} \;|||\; \mathit{RelayCert} \;|||\; \mathit{RelayCert} \;|||\; \mathit{RelayRanges} \end{aligned}$$

#### 6.5.1.4 Putting it together.

The DTR2 model is constructed from the processes above by first renaming their channels in the same way as in the DTR1 model, and then joining them together with parallel operators. More specifically,

$$\mathit{Impl} \quad = \quad \mathit{Verifier} \quad \underset{\Omega_1}{\parallel} \quad \Big( \mathit{Adversary} \quad \underset{\Omega_2}{\parallel} \quad \mathit{TTMs} \Big)$$

where $\Omega_2 = \alpha_{Adversary} \cap \alpha_{TTMs}$ and $\Omega_1 = \alpha_{Verifier} \cap \Omega_2$.

### 6.5.2 Specification

The specification process models an ideal system that starts with a bootstrapping node and satisfies the NA property.

$$SpecProcess(ps,pn) = \Box_{i,j \in \mathcal{P}^\infty} \left( \begin{array}{c} join.i.j \rightarrow SpecProcess(ps \cup \{i\},\ pn \backslash \{i\}) \\ \Box\ \ leave.i.j \rightarrow SpecProcess(ps \backslash \{i\},\ pn \cup \{i\}) \end{array} \right) \Box_{i \in ps} output.i.right(i,ps) \rightarrow SpecProcess(ps,pn)$$

$$Spec = \Box_{b \in \mathcal{P}^\infty} SpecProcess(\{b\}, \mathcal{P}^\infty \backslash \{b\})$$

### 6.5.3 Verification

To prove that the DTR2 model satisfies the NA property, it is sufficient to show that:

$$Spec \sqsubseteq Impl \backslash \{|take, fake, unlock|\} \tag{6.5.1}$$

#### 6.5.3.1 Automated Verification

An instance of *Impl* and *Spec* where $|\mathcal{P}^\infty| = 3$ is implemented in FDR, the detailed implementation can be found in Appendix G. FDR returns **true** for the refinement check in Equation 6.5.1 after evaluating **85,477** states and **245,394** transitions. This automated proof confirms that the NA property is met by the DTR2 model of at least 3 peers.

#### 6.5.3.2 Generalizing the Result

To prove that Equation 6.5.1 holds for $\mathcal{P}^\infty$ of any size, it is necessary to show that $traces(Impl \backslash \{|take, fake, unlock|\}) \subseteq traces(Spec)$. The proof is derived from the following theorem [31]:

**Theorem 6.5.1.** *Let $X$, $sq$ be a set and a sequence of events respectively. Let $\gamma(X, sq)$ be the function defined as follows:*

$$\gamma(X, sq) = \begin{cases} X & \text{if } sq = \langle\rangle \\ \gamma(X \cup \{i\}, t) & \text{if } sq = \langle join.i.j \rangle^\frown t \quad \text{for some } i, j, t \\ \gamma(X \backslash \{i\}, t) & \text{if } sq = \langle leave.i.j \rangle^\frown t \quad \text{for some } i, j, t \\ \gamma(X, t) & \text{if } sq = \langle x \rangle^\frown t \quad \text{for some } x \notin \{|join, leave|\} \end{cases}$$

*Then for any trace tr of Impl\{|take, fake, unlock|}, the following holds:*

$$\forall s, t, l, r \,.\, tr = s^\langle output.l.r \rangle^ t \;\Rightarrow\; \{l, r\} \subseteq \gamma(\{\}, s) \;\wedge\; l = left(r, \gamma(\{\}, s))$$

The function $\gamma$ returns the set of peers in the network after a given trace is executed. In other words, $\gamma$ returns the state of the system after the evolution represented by the given trace. The theorem states that if $output.l.r$ occurs at a given state, then $l$ is in fact the immediate left neighbor of $r$ in that state. The detailed proof of this theorem is included in Appendix H.

## 6.6 Related Work and Discussion

This chapter formalizes DTR1 and DTR2 in CSP, and verifies that they satisfy the RA property. The main implication for trust systems implementing DTR1 and DTR2 is that an honest peer can check if other peers have misbehaved in routing transactions, and can accordingly leave feedback for those peers. The work on formalizing and verifying properties of P2P systems in the literature is limited in number. Borgstrom *et al.* [11] modeled a structured overlay called Distributed K-ary Search (DKS) in CCS. They showed that the routing protocol in DKS, when there is no churn, is correct. Bakhshi *et al.* [7] modeled Chord in $\pi$-calculus and verified that the stabilization protocol is correct.

So far, the definitions of NA and RA property assume that verification and churn are atomic operations occurring one after another. The DTR1 model, for example, represents this by locking the TPM during verification and churn so that one cannot start while another is in progress. It would be interesting to investigate the implication of lifting this assumption. In particular, the verification protocols would be allowed to start while the churn protocols are being executed, which might present new opportunities for the adversary to break the NA property.

In the DTR2 model, only 2 nonces are used and the adversary is restricted to only

relaying messages. It was suggested that the model could be refined by the more complicated model that uses infinite number of nonces and whose adversary has infinite memory for remembering and replaying messages. This hypothesis needs to be verified in future work. One may start with the more complex model and use model abstraction techniques such as weakening the adversary and data independence to demonstrate that the model does indeed refine the one presented in this thesis.

Finally, NA and RA are considered as *safety* properties, because they concern with the attacker not being able to fool the honest peer. While safety properties require that undesirable behavior will not happen, *liveness* properties require that good behavior will eventually happen. In case of DTR1 and DTR2, the liveness property means that peers will eventually be able to engage in successful and correct routing transactions. In CSP, liveness properties are supported by the stable failure semantics model, as opposed to the trace semantics model that supports safety properties. Examining the liveness property of DTR1, DTR2 in CSP would be an interesting avenue for future work.

# CHAPTER 7

# EXPERIMENTAL ANALYSIS

This chapter provides further assessments of the proposed mechanisms for detecting misbehavior at the routing layer (DTR1 and DTR2) presented in Chapter 4 and Chapter 6. More specifically, the high-level performance of DTR1 and DTR2 are evaluated by simulation. A distributed simulation platform, which is called dPeerSim, is used for the large-scale simulation of DTR1 and DTR2 in dynamic network conditions. dPeerSim is a collaborative work involving Michael Lees, Georgios Theodoropoulos and Rob Minson. The simulation results suggest that DTR1 and DTR2 are comparable with respect to their performance, and that the latter is more scalable but is less robust under frequent churn. In both systems, under churn a high number of queries are found to be forwarded to the wrong destination nodes. Section 7.1 explains why simulation, particularly distributed large-scale simulation, is a useful method for evaluating P2P systems. Section 7.2 follows with the design and analysis of the scalability of dPeerSim. Section 7.3 describes the simulation of DTR1 and DTR2 using dPeerSim, and discusses the simulation results. Finally, the related works and discussions are presented in Section 7.4.

## 7.1 Why Large-Scale Distributed Simulation of P2P?

### 7.1.1 Why Simulation of P2P?

Current research on P2P can be categorized into two groups: studying and improving properties of current systems, and designing new applications on top of the existing overlays. There exists a number of techniques available for studying P2P systems, including mathematical and analytical modeling, real-time monitoring and simulation.

The traditional techniques - mathematical and analytical modeling - involve devising and analyzing formal models of P2P systems, which is similar to the approach adopted in Chapter 6. In the P2P context, formal models have been used in studying the search mechanisms, the impact of churn on the system's robustness, and other properties like the RA property presented in Chapter 6. But those models are complex, in which many system variables are often ignored in order to reduce the models to manageable sizes. Therefore, they do not approximate real systems very closely, and they are not ideal for studying the real performance of the systems. The real-time monitoring technique is mainly used in evaluating the peers behavior and traffic patterns in file-sharing applications. It makes use of data coming from a real P2P system. But it is expensive, because in order to monitor real-time traffic, it requires the control of a large number of peers, or access to top-tier Internet Service Providers (ISPs) or to testbeds such as PlanetLab [2].

Simulation is an appealing alternative when it comes to the verification and testing of both new and existing P2P systems. It is simpler than mathematical modeling and less expensive than real-time monitoring. It has proved to be an effective tool for studying large and complex systems. In particular, a survey covering 76 research papers on P2P was carried out to map the current landscape of P2P evaluation methods [1]. The results, summarized in Figure 7.1.1 and 7.1.2, illustrate the popularity of simulation, especially in structured P2P research.

[1]The list of surveyed papers can be found at `http://www.cs.bham.ac.uk/~ttd/survey.xls`

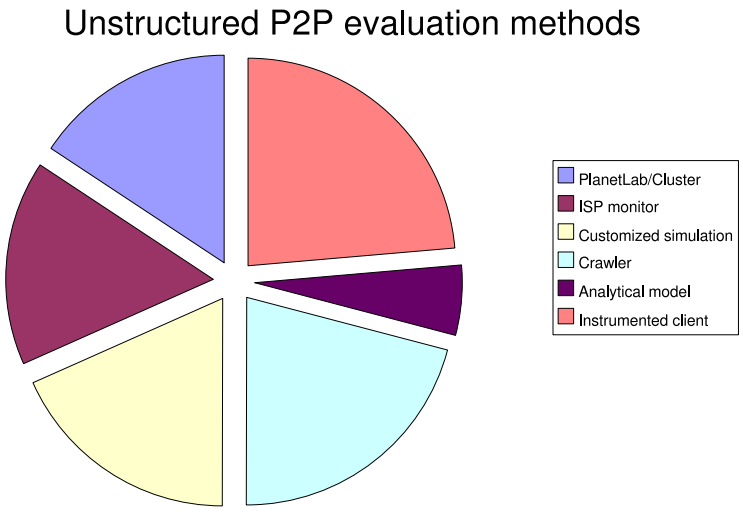



Figure 7.1.1: Evaluation methods used in unstructured P2Pstudies

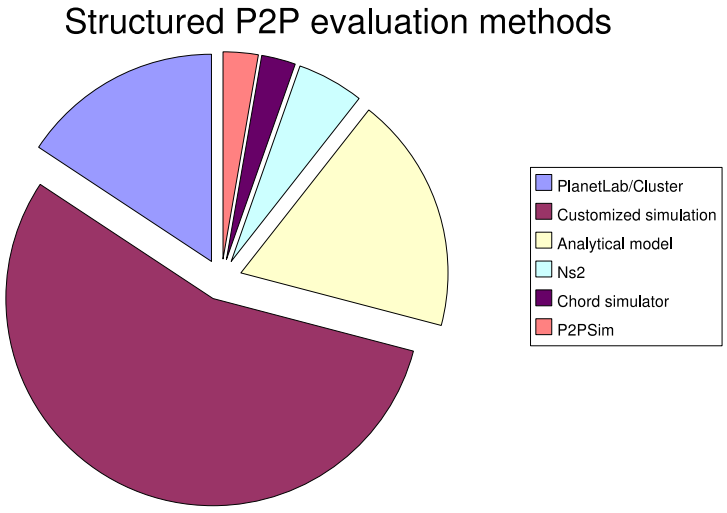



Figure 7.1.2: Evaluation methods used in structured P2Pstudies

### 7.1.2 Why Large-Scale Distributed Simulation?

Existing simulators fall short as most are only able to simulate relatively small numbers of nodes. Since the size of a typical real-world P2Psystem is in the order of millions of nodes and growing, the demand for a large-scale simulator becomes even more apparent. Such the simulator would enable testing systems of real world size or even larger, and in a reasonable time scale. It could help reveal interesting properties that are not observable when studying the systems of smaller sizes. For instance, the property that connectivity and sharing patterns in P2P file-sharing networks follow a heavy tailed distribution only unfolded when studying the networks in large scale.

There are two main challenges needed to be addressed when implementing a large-scale simulation:

1. Memory constraint: the simulation model is too big to fit into the memory of a single machine.

2. CPU constraint: the computational complexity of the simulation model is too high for it to be executed in a reasonable time scale.

These challenges can be overcome by either using super computers or distributing the model over many commodity machines. The latter, i.e. distributed simulation, is adopted in this thesis because it is less expensive and more scalable.

## 7.2 Distributed Simulation Platform (dPeerSim) for P2P Systems

This section presents the new distributed simulation platform called dPeerSim. It first introduces the design of dPeerSim, followed by the results from experimenting with dPeerSim. The results indicate that dPeerSim is capable of simulating large P2P systems in reasonable time scales.

### 7.2.1 Overview

#### 7.2.1.1 Level of abstraction.

Simulation of P2P systems is typically performed at two levels of abstraction:

1. Application level: the system is modeled as consisting of a set of nodes and a set of edges. This level is suitable for evaluating high-level properties of the network such as the hop counts or the number of exchanged messages.

2. Packet level: the underlying communication network (the Internet, for example) is also modeled. This level is suitable for studying the system's efficiency (its latency, for instance) and its impact on the underlying network.

As suggested in the survey described in the previous section, the split in the demand for application-level and for packet-level simulation is more or less even. The distributed simulation platform described in this chapter supports application-level simulation.

#### 7.2.1.2 Level of parallelism.

In distributing the sequential simulation model of the system over multiple machines, there are more than one levels of parallelism that can be exploited within the simulation [36].

1. Application level: uses the same simulation with different input parameters, requires no coordination. Limited when using large simulations as each node requires enough memory to run whole simulation. This is the most basic form of distribution and is only applicable if more than one run of a simulation is needed.

2. Subroutine level: distributes iterations of a loop, the number of processors that can be employed is limited and so is speedup.

3. Component level: is based on the natural parallelism inherent in the physical system. Each processor corresponds to a component of the physical system.

4. Event level (centralized event list): is based on a centralized master processor distributing small numbers of events onto slave processors. This is appropriate for shared memory multiprocessors.

5. Event level (distributed event list): is the most effective way of performing a simulation in parallel. Schemes exploiting this type of parallelism require local synchronization mechanisms.

Recall that the main objective of distributing a simulation is to make it run faster (CPU distribution), and to run larger simulations (memory distribution). All the levels above support CPU distribution, since the distributed models are smaller than the original model. Level 5 achieves very high level of CPU distribution, because the inter-dependency among the distributed models is very small. For memory distribution, it is necessary to distribute the simulation states across multiple machines without a centralized coordinator. This implies that either level 3 or level 5 is needed. The distributed simulation platform described in this chapter implements the level 5 of parallelism.

### 7.2.2 Design of dPeerSim

#### 7.2.2.1 Main components of dPeerSim.

The distributed simulation platform, called dPeerSim [29], is based on PeerSim [78], an open source P2P simulator written in Java. The source code and documentation can be found at: `http://www.cs.bham.ac.uk/~ttd/dPeerSim.tar.gz`.

PeerSim is an application-level simulation tool for P2P systems. It provides two modes of operation: cycle driven and and event driven. dPeerSim extends the event-driven simulation engine of PeerSim. The distribution of the simulation model is implemented using the Parallel Discrete Event Simulation (or PDES) paradigm. In particular, the system being modeled is decomposed into a number of Logical Processes (LPs), each of which is executed on a different machine. An LP is responsible for modeling a portion of the simulation states. The state changes and other information are communicated between

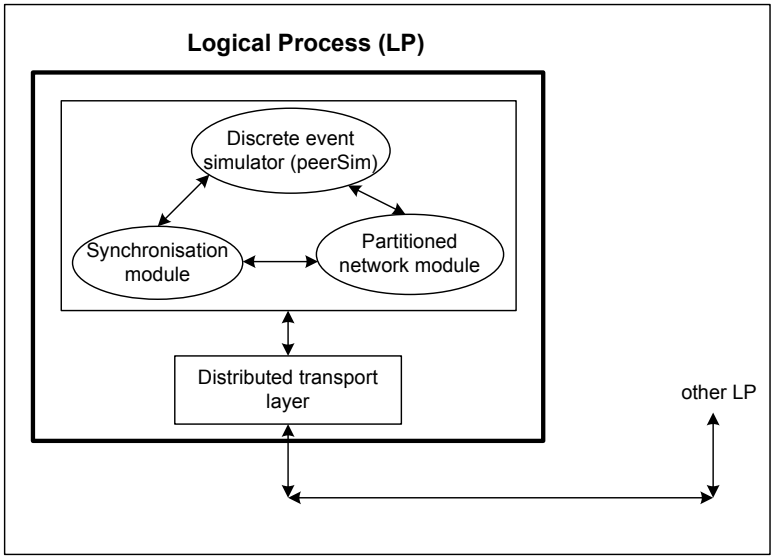



Figure 7.2.1: Components of a Logical Process in dPeerSim

LPs by exchanging messages. Every event and message in the simulation is associated with a timestamp. Each LP contains a local clock and an event list. The simulation proceeds with the LP processing the event with the lowest timestamp and increasing its local clock.

In PDES, a synchronization mechanism between LPs is needed to ensure that events are processed in the correct global order, so that results of the distributed simulation are the same as those of the equivalent sequential simulation. Synchronization mechanisms are broadly classified into two groups: conservative [18] and optimistic [51]. The conservative synchronization restricts the execution of the simulation so that an event is processed only if there guarantees to be no other events having lower timestamps. The optimistic synchronization does not restrict the execution of the simulation, instead it comes with techniques to undo (or to rollback) processed events that violate the causality constraint.

Figure 7.2.1 shows the main components of dPeerSim that together represent a LP. The discrete event simulator maintains an event queue and executes events in order, which is largely the same as the event-driven simulation engine of PeerSim. The partitioned network module manages a portion of the network assigned to the LP. The distributed transport layer provides a medium with which the LP can communicate with the other

LPs. The synchronization module ensures that simulation events are processed in correct global order.

dPeerSim implements a conservative synchronization mechanism, called Null-Message Algorithm (NMA) [18], a simple algorithm which is often used as a benchmark. NMA provides methods of describing safe and unsafe events through *lookahead* and special events called Null messages. Lookahead is defined as the minimum time increment between the current event and any future generated event. The details of the synchronization mechanism is as follows:

```
While(simulation running) {
 wait until all input queues contain at least one message
 remove the earliest time stamped event from any input queue
 execute that event (timestamp t)
 send null messages to all other LPs with time stamp (t+lookahead),
       indicating lower bound on future messages sent from this LP
}
```

To reduce the overall number of Null messages, an optimization was added to the standard algorithm. In particular, for each incoming communication channel of an LP, only one Null message exists at any given time [70]. Therefore, when a Null message is received on an incoming channel, all the existing Null messages on the same channel are removed.

#### 7.2.2.2 Distributing the P2P model into multiple LPs.

Once a network of LPs has been implemented, the P2P model is distributed into multiple LPs, as illustrated in Figure 7.2.2. The upper part of Figure 7.2.2 shows the connectivity of the network, in which the arrows represent links from one node to another. When distributed, each LP has a portion of the network (not necessarily with an equal number of nodes). Some links would stay in the local LP, others would point to other nodes in the remote LPs. The nodes and their links are maintained in the partitioned network

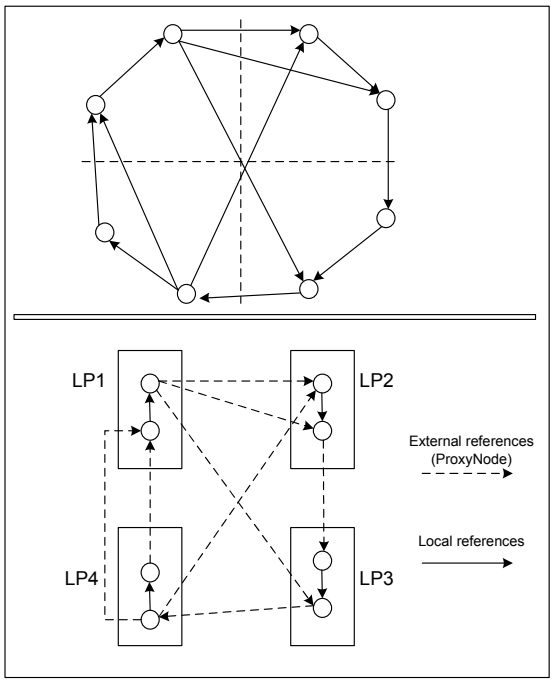



Figure 7.2.2: Distribution of P2P nodes with multiple LPs

module. A node uses the distributed transport layer module to exchange messages with its remote neighbors. An event can both be generated locally or received from a remote LP. The LP executes the synchronization algorithm (implemented in the synchronization module) to choose a safe event to process.

### 7.2.3 Scalability of dPeerSim

To evaluate the scalability of dPeerSim, experiments with different P2P systems, under static network condition and under frequent churn, are carried out. More specifically, the following protocols are implemented in dPeerSim:

- Routing protocols of Chord and Pastry. Every node initiates a lookup query according to a Poisson process.

- Churn and a simple maintenance protocol for Pastry. The node's session time follows an exponential distribution. Every node periodically sends keep-alive messages to its neighbors to check if they are still active. In addition, the nodes send their

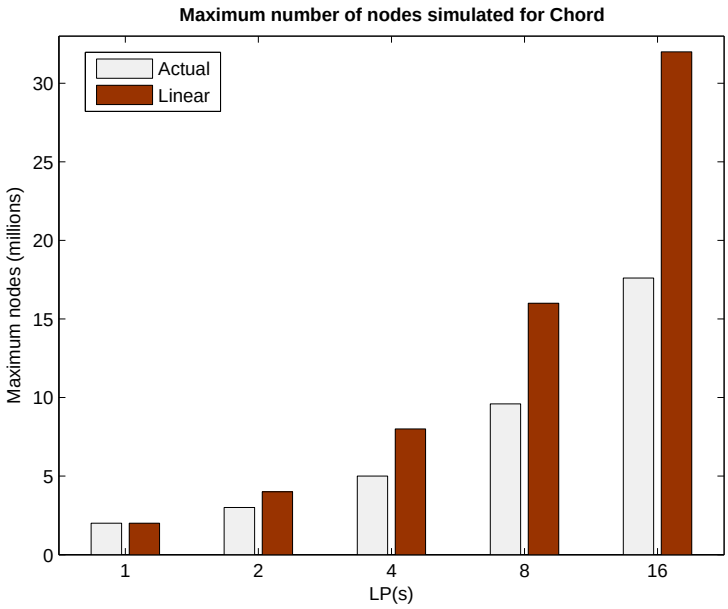



Figure 7.2.3: Maximum number of nodes that can be simulated for Chord. In a single LP, up to 2 millions nodes can be simulated. If the simulation scaled linearly, one would be able to simulate 32 millions nodes using 16 LPs (linear scalability). In practice, however, the overhead introducing by distributing the simulation model restricts the scalability: using 16 LPs, around 18 millions nodes can be simulated (actual scalability).

routing tables to each other in order to be informed of the system's latest states.

In the following, a brief analysis of dPeerSim's scalability is presented. For more details, see [32].

#### 7.2.3.1 Memory scalability.

Figure 7.2.3 and 7.2.4 show the memory scalability of the simulation for Chord and Pastry under static network condition. They demonstrate that the topology of the simulated P2P system has direct influence on the scalability of the simulation.

For the non-distributed simulation, it is possible to simulate approximately 2 million nodes for Chord and 1 million nodes for Pastry. For more than one LPs, Pastry achieves linear scaling. For Chord, the distribution overhead in terms of memory, which is proportional to the amount of information that must be replicated in different LPs, becomes apparent as the number of LPs increases. On the one hand, Pastry nodes keep far more

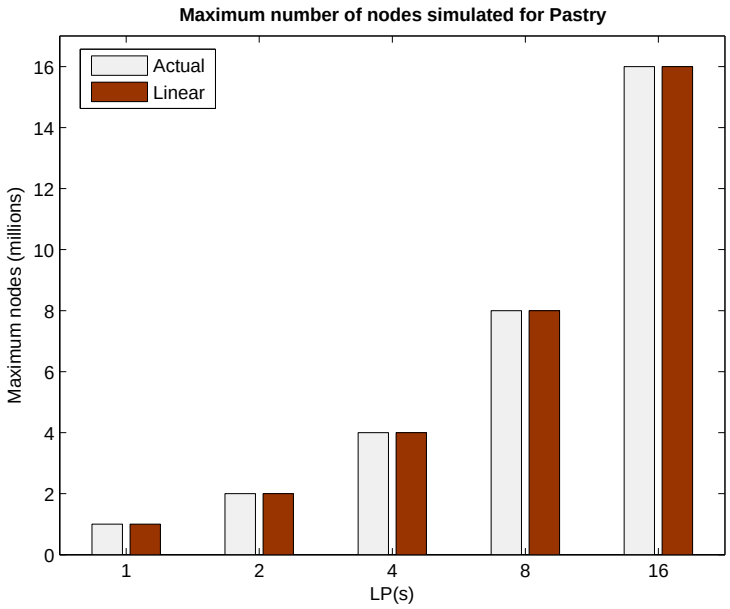



Figure 7.2.4: Maximum number of nodes that can be simulated for Pastry. In a single LP, around 1 millions nodes can be simulated. However, the simulation scales linearly, meaning that one can simulate 16 millions nodes using 16 LPs.

routing information than Chord nodes do, which explains why double numbers of Chord nodes can be simulated in a single LP. On the other hand, because of the bit-wise routing mechanism in Pastry, most neighbors of a Pastry nodes are in the local LP. This implies smaller numbers of references to remote LPs are needed. Therefore, less information needs to be replicated when distributing a Pastry model than when distributing a Chord model.

#### 7.2.3.2 Execution speedup.

Figure 7.2.5 shows the execution time when simulating Pastry protocols under dynamic network condition. The speedup is clear in the graph, as shown by the rapid decreases in execution time when more LPs are added. By distributing the simulation model to multiple machines, super linear speedup in execution time can be achieved. For example, with $16K$ nodes and $kal = 100$ (simulation time-steps), the simulation is executed over 100 times quicker with 32 LPs than with 1 LP.

It can be seen from Figure 7.2.5 that when the number of LPs goes beyond a certain point, the simulation starts to slow down. This phenomenon occurs at smaller numbers

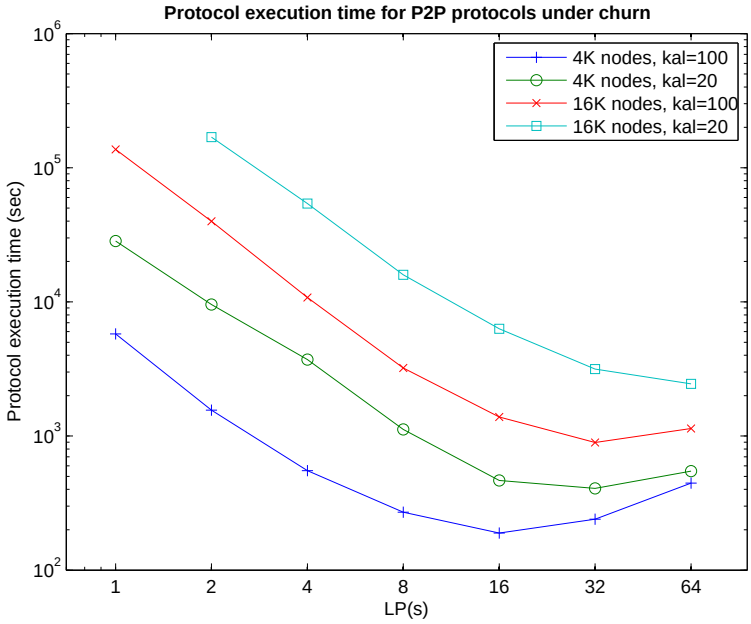



Figure 7.2.5: Execution time of Pastry under churn. The variable *kal* represents the interval after which a Pastry node sends keep-alive messages to its neighbors.

of LPs for smaller simulations. It can be attributed to the growth of communication overhead as the simulation is distributed over more LPs. To understand the effect of the communication overhead, the average numbers of Null and P2P messages handled by each LP are recorded and the results are shown in Figure 7.2.6. The number of Null messages, generated by the conservative synchronization mechanism, grows almost linearly with the number of LPs. On the contrary, the number of P2P messages, including routing, churn and maintenance messages, diminishes rapidly. This decrease in the numbers of P2P messages reflects the distribution of computational load onto LPs, and it explains the speedup in execution time. However, the gap between the numbers of Null and P2P messages becomes wider with more LPs added. Clearly, the simulation reaches a point after which the increase in the number of Null messages imposes more overhead than the speedup gained by the reduction of P2P messages. Such a threshold always exists, because the number of P2P messages is a constant for a given simulation model, whereas the number of Null messages always increases with more LPs added.

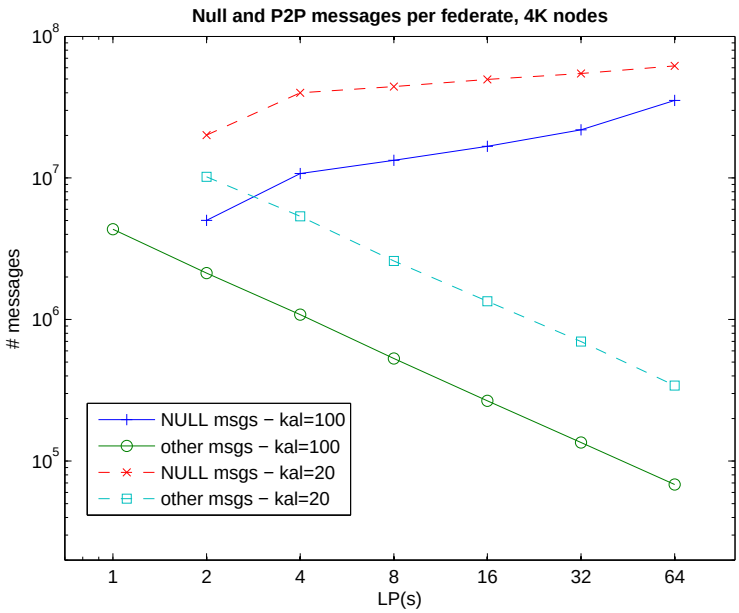



Figure 7.2.6: Average number of messages handled at each LP during simulation of Pastry under churn

## 7.3 Simulation-Based Analysis of DTR1 and DTR2

The main objective of simulating DTR1 and DTR2 using dPeerSim is to evaluate their high-level performance, namely their scalability and their robustness under churn. In particular, DTR1 and DTR2 are added on top of an existing implementation of Pastry in dPeerSim. As seen in Figure 7.2.3 and 7.2.4, dPeerSim allows scalable simulations for both Pastry and Chord under static network condition. But Pastry is chosen to be extended with DTR1 and DTR2, mainly because its implementation in dPeerSim supports churn and maintenance protocols.

For simplicity, adversarial behavior (that aims to break the system's RA property), is not taken into account when simulating DTR1 and DTR2.

### 7.3.1 Methodology

#### 7.3.1.1 Experiment setup

The simulation models contain a number of variables.

1. *nNodes* — number of simulated nodes, varying from 4096 to over 1 million.

2. $\mathcal{I}$ — the identifier space from which a peer's ID is chosen. In the simulation, $\mathcal{I} = 2^{30}$.

3. $ET$ — the number of simulation time-steps. In the simulation, $ET = 1000$.

4. $nLPs$ — the number of LPs over which the simulated nodes are distributed. In a simulation of DTR1, an extra LP is needed for simulating the CA. $nLPs$ varies from 1 to 64.

5. *qRate* — the query rate. Every node initiates queries according to a Poisson process, with the average of *qRate* queries per simulation time-step. In the simulation, $qRate = 0.02$, meaning that on average, each node starts 2 queries every 100 time-steps.

6. *cRate* — the churn rate. When a node joins the system, it is given a session time which follows the exponential distribution with the average of $\frac{1}{cRate}$ time-steps. In the simulation, *cRate* varies from 0.005 to 0.01, meaning that the average session time varies from 20 to 100 time-steps.

7. *kal* — the keep-alive interval. Each node sends a keep-alive message to its neighbors every *kal* time-steps. If a neighbor does not response, it is marked as having left the system. In the simulation, $kal = 20$ (time-steps).

8. *mPeriod* — the maintenance period. Every node sends the maintenance message containing its latest routing state to its neighbors every *mPeriod* time-steps. Once received such the message, the neighbors update their routing states accordingly. In the simulation, *mPeriod* varies from 50 to 100 time-steps.

In Pastry, the joining of a node $p_n$ is considered successful once it receives the leafset from a node $p_d$ that claims to be the closest neighbor of $p_n$ in the current network. The implementation of DTR1 and DTR2 extend the joining protocol as follows:

- In DTR1, after receiving the leafset, $p_n$ contacts the CA (in the remote LP) asking for a neighbor certificate. The joining protocol is considered successful when $p_n$ has received such the certificate.

- In DTR2, after receiving the leafset, $p_n$ asks $p_d$ for a range of tokens. $p_n$ checks that it gets the correct tokens (as described in Section 5.5). If it is true, the joining protocol is considered successful.

A leaving operation is considered successful after the leaving node has contacted the relevant party. In DTR1, the CA is contacted. The CA then issues updated certificates to the neighbors of the leaving node. In DTR2, the leaving node contacts its immediate neighbor and transfers its tokens to the neighbor. In both DTR1 and DTR2, the leaving protocols always complete successfully.

In the original Pastry, a lookup query initiated by a node $p_i$ is considered successful when it arrives at a node $p_d$ from which it cannot be forwarded any further. In DTR1 and DTR2, the query is considered as completed. Additionally, $p_i$ performs verification that $p_d$ is the correct root node by checking its neighbor certificate (in DTR1) and tokens (in DTR2). Only if the verification returns true is the query considered successful.

All the experiments were run on the University of Birmingham's BlueBear cluster [1]. Each cluster node has two 64bit AMD Opteron dual core processors and 8GB of RAM. Each LP is given one core and 1.5GB of RAM. Experiments were allowed to run for up to 10 days. The results were averaged over multiple runs.

#### 7.3.1.2 Evaluation metrics.

Various measurements are taken from the experiments to evaluate DTR1's and DTR2's scalability and their robustness churn. More specifically:

- For scalability:

[1] http://www.bear.bham.ac.uk

1. The workload at the CA, measured by the number of certificates it issues during the simulation.
2. Simulation execution time.
3. Average hop count of the routing operation.

- For robustness:

1. Average ratio of successful joins.
2. Average rate of query failure per completed query.

### 7.3.2 Results and Analysis

#### 7.3.2.1 Scalability.

Section 4.5 has argued that in DTR1, the CA is unlikely to be a performance bottleneck in a relatively stable P2P system (with low churn rate). The CA maintains a list of peers that are currently in the network, which would not scale well. However, such the list could be removed by using a more complex mechanism for issuing certificates. The main role of the CA is to issue neighbor certificates to peers during churn events. Figure 7.3.1 shows the number of certificates produced by the CA in the simulation. As expected, such the workload grows with the number of nodes and the churn rate.

To see if the workload at the CA could be a performance bottleneck, the simulation execution time of DTR1 and DTR2 are recorded and the results are shown in Figure 7.3.2. It can be seen that the simulation of DTR1 takes longer to complete than that of DTR2, but the differences are small, even with the high the workload incurred at the CA. Therefore, it can be concluded that the negative impact of the CA on the system's performance is visible, but compared to the non-CA system (DTR2) it is not substantial enough for the CA to be adjudged a bottleneck.

Figure 7.3.3 and 7.3.4 show the average hop count in DTR1 and DTR2. The hop count is close to $log_{2^b}(N)$ - the hop count of Pastry under static network condition. DTR1

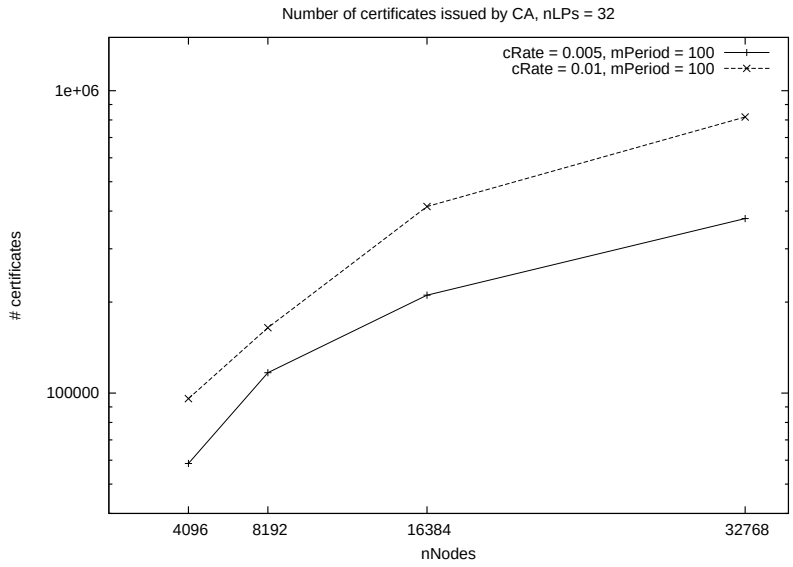



Figure 7.3.1: The workload at the CA during the simulation. $nLPs = 32$, $nNodes$ varies from 4096 to 32768, $cRate$ varies from 0.005 to 0.01, and $mPeriod = 32$

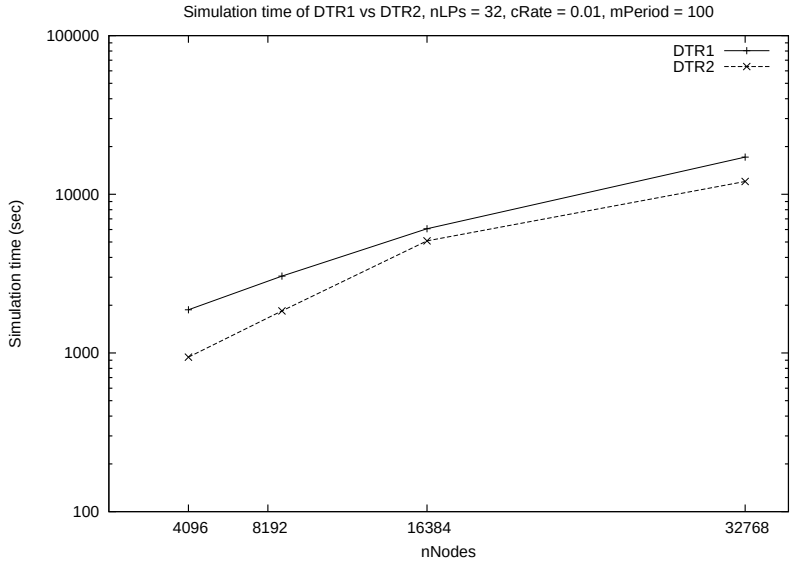



Figure 7.3.2: Simulation execution time of DTR1 vs DTR2. $nLPs = 32$, $cRate = 0.01$, $mPeriod = 100$, $nNodes$ varies from 4096 to 32768

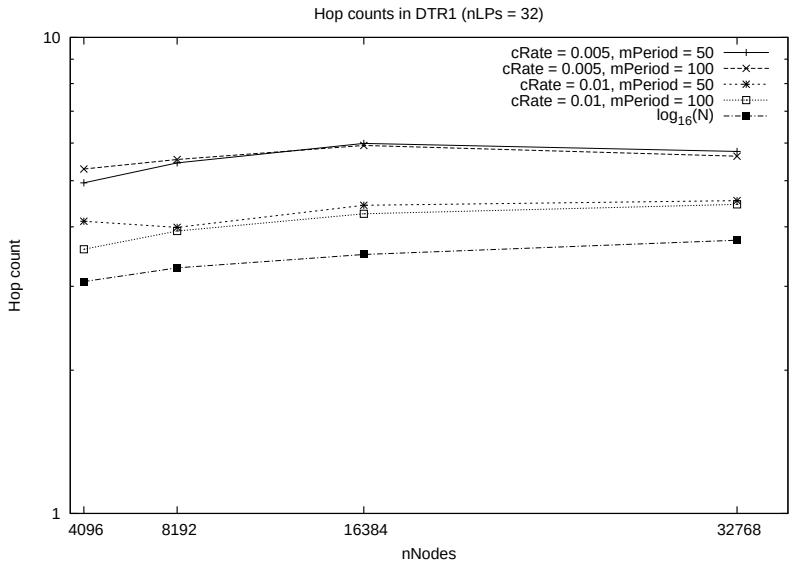



Figure 7.3.3: The average hop-count in DTR1, $nLPs = 32$ and $nNodes$ varies from 4096 yo 32768

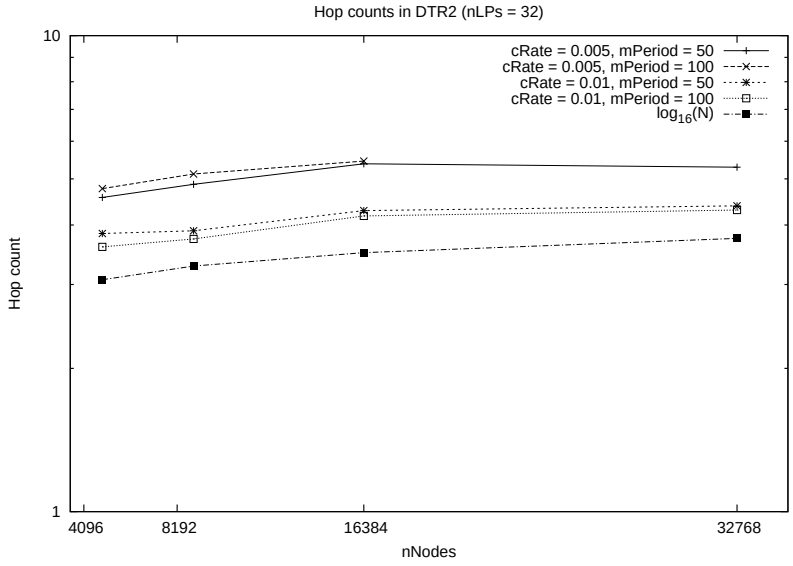



Figure 7.3.4: The average hop-count in DTR2, $nLPs = 32$ and $nNodes$ varies from 4096 yo 32768

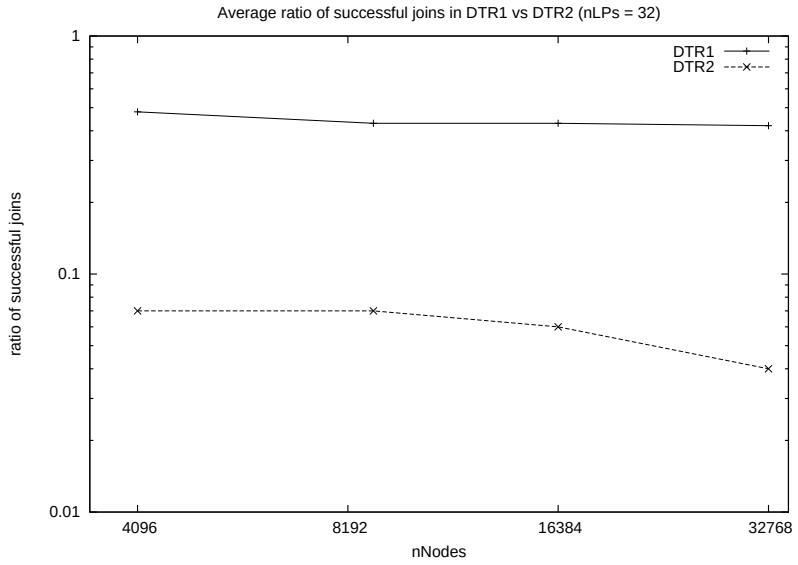



Figure 7.3.5: Average rate of successful joins in DTR1 vs DTR2, $nLPs = 32$ and $nNodes$ varies from 4096 to 32768

and DTR2 are similar regarding this metric, because the verifications occur only at the very last steps of the routing. In fact, these figures demonstrate that the hop-count is mainly influenced by the churn rate. In particular, higher churn rates result in higher hop count.

#### 7.3.2.2 Robustness.

The rates of successful joins for varying values of $cRate$ and $mPeriod$ are averaged and the results are then shown in Figure 7.3.5. There are two noticeable observations from the figure. First, unsuccessful join events always exist, as the rates of successful joins are always below 1. One explanation is that the churn condition causes the routing towards the closest neighbors of the joining nodes to fail occasionally. Second, the rate of successful joins in DTR1 is always larger than that in DTR2, which can be explained as follows. In DTR1, join events are always considered successful once the joining nodes receive a leafset from other nodes. In DTR2, a join event can fail when the joining node fails to get the correct tokens from its neighbor, which could happen under churn because the routing of the joining query ended up at the wrong neighbor. Therefore, the number of

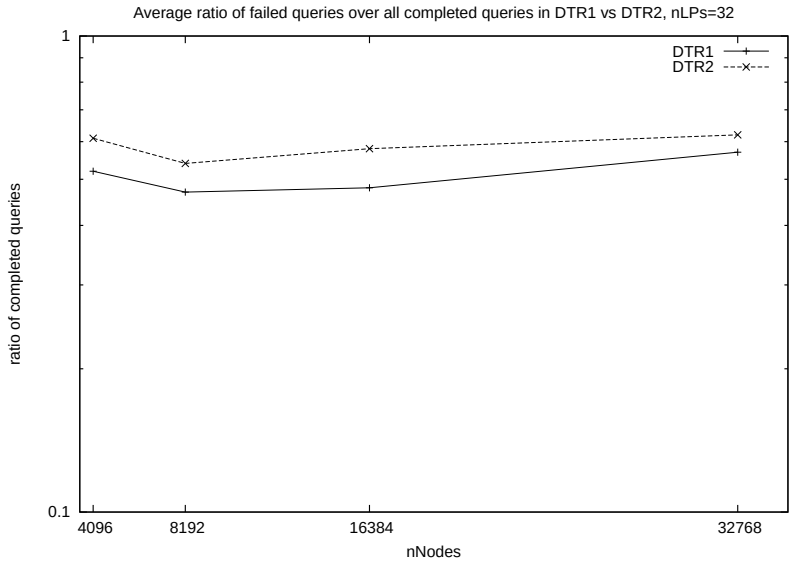



Figure 7.3.6: Average rate of query failure per completed query in DTR1 vs DTR2. $nLPs = 32$, $nNodes$ varies from 4096 to 32768

successful joins in DTR2 is smaller than that in DTR1.

Figure 7.3.6 illustrates the rate of query failure per completed query in DTR1 and DTR2. The results are averaged over those with varying values of $cRate$ and $mPeriod$. As stated earlier, a completed query arrives the node that cannot forward the query further. A successful query requires, in addition, the node to show evidence that it is indeed the root node of the search key. In other words, a query is deemed unsuccessful or failed in two cases:

1. The query is dropped because it is forwarded to a node which is no longer in the network.

2. The query arrives at the node that cannot present the correct certificate (in DTR1) or tokens (in DTR2).

The first case depends on the churn rate, while the second is attributed to the implementation of DTR1 and DTR2. Figure 7.3.6 demonstrates that DTR1 and DTR2 incur high, but comparable rates of query failure per completed query. In both systems, of all the queries arrived at some destination nodes, less than 50% of those nodes are the correct root nodes of the search keys. This provides more evidence for the negative effects of

churn in DTR1 and DTR2. Interestingly, the failure rate in DTR1 is always lower than in DTR2. This can be related to the higher level of successful queries in DTR1, as explained by the following example. Consider a node $p_n$ joining the system, and the joining query arrives at a node $p_d$ which is not the closest neighbor of $p_n$ in the current network. $p_n$ always gets its certificate in DTR1, but fails to do so in DTR2. In both cases, $p_n$ stays in the network and participates in the routing protocol. When query for a key $k$ whose root node is $p_n$ arrives at $p_n$, the query will succeed in DTR1 but fail in DTR2. Therefore, the chance of a given query being unsuccessful in DTR2 is higher than that in DTR1.

#### 7.3.2.3 Summary of the results.

The discussion of the simulation results above can be summarized as follows:

1. The CA introduces overheads to DTR1. For a typical P2P system, such overheads are not substantial enough for the CA to be considered as a performance bottleneck. However, it is clear that for systems under highly frequent churn, the CA can cause scalability issues.

2. The negative effects of churn in DTR1 and DTR2 are evident. The rates of successful joins and the rates of query failure per completed query indicate that the performance of DTR1 and DTR2 are comparable, but DTR1 seems more robust under churn.

## 7.4 Related Work and Discussion

This chapter presents experimental analysis of DTR1 and DTR2 using a large-scale simulation tool called dPeerSim. Other simulation tools for studying P2P systems exist. Generic network simulators such as NS-2 [40] and p2pSim [41] are used occasionally, while bespoke simulators are found in many other P2P studies. The detailed implementation and source codes of bespoke simulators are often unavailable, which makes it difficult

to reproduce the results or to carry out comparative evaluations. The aim of PeerSim is to fulfill the need for a generic, easily extendable open-source P2P simulator.

Most existing simulators are limited by the total number of nodes they can simulate. Bespoke simulators achieve $100,000$ nodes [17] at most, while NS-2 claims the scalability of $260,000$ nodes. This limit is largely due to resource constraints at the simulation machine. PDNS is a distributed simulation platform based on NS. It supports packet level simulation, as opposed to dPeerSim which supports application-level simulation.

There are a number of limitations with the current simulation models of DTR1 and DTR2 implemented in dPeerSim. First, the detailed protocols involving the security devices (TPMs in DTR1 and TTMs in DTR2) are not implemented. In the current churn model in which nodes fail gracefully, and with the assumption that the issuing of certificate is an atomic operation, the actual number of messages handled by the devices could be extrapolated from the number of successful joining and leaving events. It would be interesting to implement these protocols into dPeerSim and to observe the workload at the devices when more complex churn models (Byzantine failure, for example) are considered. Second, the maintenance protocol is very simple, in which peers send keep alive messages and routing tables to each other after fixed intervals. dPeerSim has been used to evaluate more complex, adaptive maintenance protocols [76]. This current protocol might be accountable for the high rates of query failure per completed query observed in DTR1 and DTR2. Therefore, it would be interesting to investigate how other maintenance schemes help improve the performance of DTR1 and DTR2 under churn. Third, the network adversaries are not included in the simulation models. In practice, the adversary could perform Denial of Service (DoS) attacks to undermine the system's performance. Simulating the network adversary and analyzing its impact on the system is an interesting and challenging domain of future work.

As discussed in Section 4.5 and 5.7, the current designs of DTR1 and DTR2 present a number of avenues for future work. For example, the churn model can be made more realistic by taking fail-stop or Byzantine failures into consideration. In addition, a more

complex mechanism for issuing certificate in DTR1 could remove the need for the CA to maintain the list of peers. Having redesigned DTR1 and DTR2, a necessary step is to evaluate their performance by experimenting with the new protocols using dPeerSim.

Regarding dPeerSim, an extension that adds support for packet-level simulation would make the tool more useful for studying P2P. From a distributed simulation perspective, it would be interesting to study the effect of optimistic synchronizations on the scalability of the simulation. Finally, implementing a load balancing technique into dPeerSim could prove useful when simulating P2P systems under frequent churn, because the more balanced workload among LPs would result in the shorter simulation execution time.

# CHAPTER 8

# CONCLUSION AND FUTURE WORK

This concluding chapter summarizes the results presented in the previous chapters, and discusses the extent to which the goal of the thesis, which has been to investigate the reliability of trust systems for structured P2P, has been met. Section 8.1 discusses this thesis's contributions to knowledge and highlights its limitations. Section 8.2 presents a number of research directions that may be taken in the future.

## 8.1 Contributions and Evaluation

Peer-to-Peer (P2P) infrastructure has received a great amount of research attention. Thanks to its scalability, structured P2P in particular has been used for designing many large-scale distributed systems. Security is one of many challenges that must be addressed before the potential of structured P2P can be fully utilized. Having a trust system for P2P that allows one node to assess the trustworthiness of another before interacting with it can help mitigate security as well as some other problems in P2P. A trust system comprises a reputation metric and a feedback mechanism, and it should be both *reliable* and *efficient*. This thesis set out to investigate on making the current states of trust systems for P2P more reliable. Recall that the goal of this thesis is to seek the answers to following questions:

1. To what extent are existing reputation metrics such as PageRank vulnerable to

Sybil manipulations? Can they be improved? Are the feedback models used by those metrics strong enough? Can they be made more realistic?

2. Regarding the existing feedback mechanisms for structured P2P applications, is it always possible for peers to leave feedback to each other after their transactions? If not, can we design mechanisms that overcome such the limitation?

Overall, this thesis has answered both questions. To a large extent, it has demonstrated that the existing reputation metrics and feedback mechanisms can be improved to make the trust system more reliable. It has also made other contributions regarding the methods for evaluating P2P systems.

### 8.1.1 Summary of Contributions

1. Chapter 3 has presented the limitations of the reputation metric called PageRank. It studied a new Sybil strategy and compared that against the strategy studied in [21]. It was shown that the adversary executing the new strategy can increase its reputation more than when it uses the strategy in [21]. Furthermore, when the underlying trust graphs are undirected PageRank was demonstrated to be more sensitive to Sybil manipulations. Another limitation of PageRank and of some existing reputation metrics is the failure to take negative feedback into consideration. The importance of negative feedback in trust systems has been explained in Chapter 3.

   These results have led to two new reputation metrics being proposed. One is called Cluster-based PageRank (or CPR), and the other is called PRN (PageRank with Negative feedback). Both metrics are based on the original PageRank. CPR has proved more resilient to Sybil manipulations under undirected graphs, and also capable of generating intuitive reputation values. PRN takes negative feedback into account when computing the reputation scores. Experimenting with PRN has shown that it generates intuitive reputation values and is resilient to Sybil manipulations.

2. Chapter 4 has explained a limitation of existing feedback mechanisms, that is the difficulty for an honest node in structured P2P environments to securely detect if another node has misbehaved. Such the problem arises because of the node's limited knowledge of the other parts of the network, and the presence of network adversaries. Two case studies have been presented in Chapter 4 to demonstrate the problem at two different layers of P2P abstraction: the routing layer and the application layer.

   Using Trusted Computing, particularly the Trusted Platform Modules (TPMs), two sets of protocols called DTR1 (for detecting misbehavior at the routing layer) and DTA1 (for detecting misbehavior in the P2P-based marketplace application) have been proposed.

   To improve upon DTR1 and DTA1, a new general-purpose security hardware called Trusted Token Module (or TTM) has been proposed in Chapter 5. Using TTMs, the centralized component in DTR1 can be removed, resulting in a new mechanism called DTR2. In addition, TTMs help improve the efficiency of DTA1 and increase the range of operations supported by DTA1. The improved mechanism for detecting misbehavior in the P2P-based marketplace application is called DTA2.

3. Chapter 6 has presented the details of using Communicating Sequential Process (CSP) as the formal method for modeling DTR1 and DTR2. It has also presented the verification showing that DTR1 and DTR2 indeed allow for the secure detection of misbehavior in structured P2P routing. It has contributed to the limited number of works that use formal methods for modeling and checking properties of P2P systems. In the process of devising the proofs, model abstraction techniques such as weakening the adversary and data independence have been used to reduce the state spaces of the original models. Chapter 6 has included the automated proofs, generated by the model checker FDR, for the DTR1 and DTR2 models of limited number of nodes.

A new simulation platform has been presented in Chapter 7. Simulation has proved to be an effective evaluation method when studying P2P systems. The new simulation tool, called dPeerSim, has been shown to be capable of simulating large P2P systems in reasonable time scales. Such the scalable simulator is useful for studying systems under dynamic conditions such as churn. In fact, dPeerSim has been used to evaluate the high-level performance of DTR1 and DTR2.

### 8.1.2 Implications of This Thesis

First, this thesis has merged a number of research fields together in the journey of finding the answers to the two research questions described in early in Section 8.1.

1. P2P and particularly structured P2P infrastructure. The thesis is motivated by the problems faced by existing systems based on structured P2P. Security is one of the hurdles that need to be overcome before structured P2P can be widely employed.

2. Trust and reputation. These concepts emerge from human interaction and have attracted great deals of research attention in social sciences. In the context of structure P2P, it has been argued that having a trust system based on reputation can help mitigate a number of problems including security.

3. Hardware-based security. The protocols proposed in this thesis: DTR1, DTR2, DTA1 and DTA2 all assume the presence of a trusted device at each peer. These protocols aim to facilitate the secure detection of misbehavior in structured P2P applications. Such detection mechanisms are important parts of the trust systems. DTR1 and DTA1 leverage TPMs, while DTR2 and DTA2 use the new devices called TTMs. In both cases, the devices' trusted operations that cannot be compromised are utilized.

4. Methods for analysis. There exists at least two types of methods for analyzing a system: analytical or formal methods and simulation-based methods. In this thesis,

methods belonging to both types have been used to analyze DTR1 and DTR2. CSP has been chosen as the formal method for modeling and proving that DTR1 and DTR2 satisfy the Root Authenticity property. dPeerSim — a scalable distributed simulation platform — has been used to evaluate the performance of DTR1 and DTR2.

Second, the answers provided by this thesis to the two research questions suggest that the existing trust systems for P2P can be made more reliable.

- *Answers to Question 1: reputation metrics with improved resilience against Sybil manipulations, and with support for more realistic feedback model.* It has been shown that PageRank has limited resilience against Sybil manipulations, especially when the underlying trust graphs are undirected. Furthermore, most metrics lack the support for negative feedback. The new metrics, called CPR and PRN, have been shown to improve upon the original PageRank. For example, CPR has been demonstrated to be more resilient to Sybil manipulation under undirected graphs.

- *Answers to Question 2: mechanisms for enabling the detection of misbehavior in structured P2P applications.* Current feedback mechanisms are not concerned with the difficulty in detecting misbehavior in structured P2P environments. New protocols have been proposed to overcome the difficulty in routing and in the P2P-based marketplace application. They rely on trusted hardware such as TPMs and TTMs. Formal analysis has shown that the new protocols for routing does enable peers to securely detect the misbehavior of each other.

Third, this thesis has demonstrated that hardware-based security can be useful in enabling trust for P2P systems. TPMs, which come in the forms of hardware chips being shipped in high-ends laptops, desktops and servers, have been utilized to improve upon existing feedback mechanisms. The newly proposed trusted devices, called TTMs, have the potential to be very useful in future applications, as they are more powerful and flexible than TPMs.

Fourth, to the best of my knowledge, this thesis has presented the first work in which CSP is used for modeling and checking properties of P2P systems. The refinement relation supported by CSP allows for more flexibility and elegance in the definitions of the models and of their properties. The thesis has contributed another case study that proves the usefulness of the data independence technique as a method for reducing the complexity of the formal models.

Last but not least, future research on P2P can benefit hugely from dPeerSim. The simulation platform has proved to be a powerful tool for studying large and complex P2P systems.

## 8.2 Limitations and Future Work

This thesis should only be seen as another step towards building a reliable, efficient trust system for P2P. The future work can seek to address the assumptions made and the limitations contained in this thesis. It can follow a number of directions, as detailed below.

1. This thesis did not consider the efficiency property of the trust systems. In other words, the implementation details of the reputation metrics and feedback mechanisms are outside the scope of this thesis. As noted in Chapter 2, such details must be considered before the trust systems can be used in practice. In particular, two research questions exist, which are: how to store and retrieve feedback, and how to compute the reputation values efficiently. These questions are challenging, especially when no centralized, trusted party is available. In such an environment, it could prove extremely hard to implement a symmetric reputation metric like PageRank, which hints at a simpler option such as the Beta metric [63]. However, the trade-off of using such simple metrics is that they are more vulnerable to Sybil manipulations.

2. This thesis focuses on symmetric reputation metrics, especially ones based on PageRank. Other metrics, for example the asymmetric ones that are based on network-

flow, are worthy of investigation in the context of reliable trust systems. More specifically, it would be interesting to examine their resilience against Sybil manipulations under undirected graphs, and how they can be modified to incorporate negative feedback.

Regarding feedback mechanisms, there exists other problems beside the detection of misbehavior. For example, the question of how to incentivize nodes to leave feedback to each other, and the question of how to eliminate dishonest feedback are difficult to answer. These questions have been investigated in a number of works, but there is still much room left for future work. The work by Liu *et al.* [63], for example, has shown positive results in giving nodes the incentives to leave honest feedback. However, the results hold only for a simple reputation metric which are vulnerable to Sybil manipulations.

3. The protocols proposed in Chapter 4 and Chapter 5 leverage the trusted operations provided by the security hardware. If the assumption that each peer must be equipped with a security device were to be removed, the protocols could still work if an online service providing those trusted operations is available. The detailed architecture of such the trusted service can be explored in future work.

4. In CPR and PRN, the underlying trust graphs contain edges whose weights are given values in $\{-1, 1\}$. For the undirected graphs used in CPR, the edges between any two nodes have the same value of 1, or $W((i,j)) = W((j,i)) = 1$ for any $(i,j) \in E$. It could be an interesting to investigate the effect of assigning the weights with real values in $[-1, 1]$, and of having $W((i,j)) \neq W((j,i))$.

   PRN has been evaluated under directed trust graphs. A possible direction of future work is to study PRN under undirected graphs, which could subsequently lead to the investigation of how to combine CPR and PRN together.

5. The current churn models considered in DTR1, DTA1, DTR2 and DTR2 are quite strict. More specifically, in DTR1 and DTR2, nodes are assumed to leave the system

gracefully, meaning that they inform the relevant parties before leaving. This can be made more realistic by incorporating fail-stop and Byzantine failure to the churn models. However, such change would then necessitate complex mechanisms that deal with expiring certificates (in DTR1) or the re-issuing of tokens (in DTR2). Designing and evaluating of these mechanisms are interesting avenues for future work.

DTA1 and DTA2 currently assume that the network is static. Therefore, more work is needed to allow them to be useful even when dynamic network conditions are considered.

6. There exists a number of possible improvements that can be made to the current design of TTM before it can be implemented efficiently on a smart-card. For instance, caching techniques could help speed up the computation of Merkle roots. Furthermore, due to the memory constraint in smart-cards, the function that computes the Merkle roots may need to be changed so that it takes a smaller number of inputs at a time. A TTM can also be extended to support more monotonic counters by following the technique proposed by Sarmenta *et al.* [84].

   Since TTM aims to be a general-purposed device, it would be interesting to find other security applications that can benefit from using TTMs. For now, TTM is still at the design stage, therefore the most needed work might be to implement the design in smart-cards, and to evaluate the performance when executing real commands.

7. In the formal analysis of DTR1 and DTR2, the verification and churn operations are assumed to be atomic. In other words, a churn event cannot start while the verification is in progress, and similarly the verification cannot start while joining or leaving protocols are taking place. Future work may examine the implications of removing this assumption. Allowing the verification to start while churn is still in progress, for example, might present the adversary with the opportunities to succeed

in his attacks.

In this thesis, the RA property has been translated into a safety property in CSP. A very interesting direction of future work may lie in defining and checking the liveness property for DTR1 and DTR2. The liveness property states that the honest peer will eventually succeed in routing its queries to the correct root nodes.

8. The experimental analysis of DTR1 and DTR2 suggests that their performance under churn leaves much to be desired. In particular, the rates of successful joins are low, and the rates of query failure per completed query are high. These results may be attributed to the maintenance protocol being very simple. In future work, one could explore and compare the effects of different, more complex maintenance schemes on the performance of DTR1 and DTR2.

   The current simulation models of DTR1 and DTR2 do not include the adversary. Simulating the adversarial behavior and assessing the system's performance when the adversary is active can be another direction of future work.

   Regarding the distributed simulation platform, future work could focus on adding the support for packet-level simulation, which would make the tool inevitably more useful for studying P2P systems. Future work could also investigate if using optimistic synchronization instead of the current conservative synchronization mechanism could improve the scalability of dPeerSim. Finally, future work may follow the direction of implementing a load balancing for dPeerSim, which could improve the scalability of dPeerSim even further, since the more balanced workload among LPs would result in the shorter simulation execution time.

9. This thesis has only analyzed the correctness and performance of DTR1 and DTR2. Therefore, future work could extend this by performing formal analysis and simulation-based analysis for DTA1 and DTA2.

# APPENDIX A

# PAGERANK'S RESILIENCE AGAINST SYBIL MANIPULATIONS

**Theorem 1.** *Let $R'_n$ be the reputation value of node $n$ after executing the Sybil strategy in Figure 3.2.1b. Then:*

$$(2-\epsilon).R_n + (1-\epsilon).k \leqslant R'_n \leqslant \frac{R_n}{\epsilon} + (1-\epsilon).k$$

*Proof.* In addition to the original graph $\mathcal{G}$ and the one under attack $\mathcal{G}'$, we consider an intermediary graph $\mathcal{G}''$, in which the adversary removes its links to other nodes and then creates a link to itself. Let $T$, $T'$ and $T''$ be the transition matrices derived from $\mathcal{G}$, $\mathcal{G}'$ and $\mathcal{G}''$ respectively.

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ T_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix}$$

$$T'' = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} T_{n-1} \\ O_1 \end{bmatrix}$$

$$T' = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1N} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} T_{n-1} & O_2 \\ O_1 & O_4 \\ O_5 & O_6 \end{bmatrix}$$

Replacing $T'$ and $T''$ into Equation 3.2.1, we have:

$$(1-\epsilon).\left[R''_{1..n-1}.T_{n-1} + R''_n.O_1\right] + \overrightarrow{\epsilon}^T = \left[R''_{1..n-1}, R''_n\right] \tag{A.0.1}$$

$$(1-\epsilon).\left[R'_{1..n-1}.T_{n-1} + R'_n.O_1 + \left[0..\sum_{n+1}^{n+k} R'_i\right], [0..0]\right] + \overrightarrow{\epsilon}^T = [R'_{1..n-1}, R'_n, R'_{n+1..n+k}] \tag{A.0.2}$$

Solving equation A.0.1 and A.0.2, we have $R'_{1..n-1} = R''_{1..n-1}$ and

$$R'_n = R''_n + (1-\epsilon).k \tag{A.0.3}$$

**Lemma 1.** *For all $i \leqslant n$, $R_i \geqslant R''_i$*

*Proof.* This lemma is proved by induction. The power method used to compute the stationary vector $\overrightarrow{R}$ proceeds as follows:

$$\begin{cases} R_i^{it} = & (1-\epsilon).\sum \frac{T_{ji}}{|W_j^-|}.R_j^{it-1} + \epsilon \\ R_i^0 = & 1 \end{cases}$$

where $|W_j^-| = \sum_{x \in W_j^-} x$. It can be seen that $R_i^1 \geqslant R_i''^1$ for all $i \leqslant n$, because the adversary in $\mathcal{G}$ does have links that point back to other nodes, unlike in $\mathcal{G}''$ where those links are

removed. Assuming $R_i^{it} \geqslant R_i''^{it}$ for $i \leqslant n$ and $it \geqslant 1$. We have:

$$\begin{aligned} R_i^{it+1} &= (1-\epsilon).\sum \frac{T_{ji}}{|W_j^-|}.R_j^{it} + \epsilon \\ &\geqslant (1-\epsilon).\sum \frac{T_{ji}}{|W_j^-|}.R_j''^{it} + \epsilon \\ &\geqslant R_i''^{it+1} \end{aligned}$$

By induction, we have $R_i^{it} \geqslant R_i''^{it}$ for all value of $it$. In other words, $R_i \geqslant R_i''$ for $i \leqslant n$. □

Using the Lemma above, it can be seen that $\sum_{i<n} R_i - (1-\epsilon).R_n \geqslant \sum R_i''$. Or,

$$R_n'' \geqslant (2-\epsilon).R_n \tag{A.0.4}$$

Also, applying Equation 3.2.1 for the adversary we have:

$$R_n'' \leqslant \frac{R_n}{\epsilon} \tag{A.0.5}$$

Combining equation A.0.3, A.0.4 and A.0.5 together, we derive the same formula as stated in the theorem. □

# APPENDIX B

# ROOT AUTHENTICITY AND NEIGHBOR AUTHENTICITY PROPERTY

**Theorem 2.** $NA \Rightarrow RA$

*Proof.* The proof can be constructed as follows. First, assume that $NA$ holds. Next assume that $destVerf(p_v, k, p_d, t)$ holds for any $k$, $p_d$ and $t$. Then we need to show the following:

$$p_d \in \mathcal{P}^t \tag{B.0.1}$$

$$\forall p'_d \in \mathcal{P}^t \backslash \{p_d\}.\ cd(p'_d, k) > cd(p_d, k) \tag{B.0.2}$$

Because $destVerf(p_v, k, p_d, t)$ and $NA$ hold, there exists $p_l$ such that:

$$\{p_l, p_d\} \in \mathcal{P}^t \tag{B.0.3}$$

$$cd(k, p_l) + cd(p_d, k) = cd(p_d, p_l) \ \wedge \ \forall p''_d \in \mathcal{P}^t \backslash \{p_l, p_d\}.\ cd(p''_d, p_l) > cd(p_d, p_l) \tag{B.0.4}$$

It can be seen that (B.0.1) can be derived directly from (B.0.3).

(B.0.4) implies (B.0.2) because assuming (B.0.4), for any $p'_d \in \mathcal{P}^t \backslash \{p_d\}$:

1. If $p'_d = p_l$, using the observation that $cd(x, y) = 2^m - cd(y, x)$ and $cd(k, p_l) + cd(p_d, k) = cd(p_d, p_l)$, we can derive that $cd(p'_d, k) > cd(p_d, k)$

2. If $p'_d \neq p_l$. (B.0.2) implies:

$$
\begin{aligned}
& cd(p'_d, p_l) > cd(p_d, p_l) \\
\Rightarrow\ & cd(p'_d, p_l) = cd(p'_d, p_d) + cd(p_d, p_l) \\
\Rightarrow\ & cd(p'_d, p_l) = cd(p'_d, k) + cd(k, p_l) \\
\Rightarrow\ & cd(p'_d, k) + cd(k, p_l) > cd(p_d, p_l) > cd(k, p_l) + cd(p_d, k) \\
\Rightarrow\ & cd(p'_d, k) > cd(p_d, k)
\end{aligned}
$$

□

# APPENDIX C

# CSP TRACE SEMANTICS

$$
\begin{array}{rcl}
traces(STOP) & = & \{\langle\rangle\} \\
traces(a \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces(P)\} \\
traces(P \,\Box\, Q) & = & traces(P) \cup traces(Q) \\
traces(P[\![R]\!]) & = & \{\, tr \mid \exists s \in traces(P) \bullet s\ R^*\ tr \,\} \\
traces(P \,|||\, Q) & = & \bigcup\{s \,|||\, t \mid s \in traces(P), t \in traces(Q)\} \\
traces(P \underset{X}{\parallel} Q) & = & \bigcup\{s \underset{X}{\parallel} t\} \mid s \in traces(P), t \in traces(Q)
\end{array}
$$

where $R^*$, $|||$ and $\underset{X}{\parallel}$ are defined as follows:

$$
\langle a_1, .., a_n\rangle\ R^*\ \langle b_1, .., b_m\rangle \quad \leftrightarrow \quad n = m \ \wedge\ \forall i \leqslant n \bullet a_i\ R\ b_i
$$

$$
\begin{array}{rcl}
\langle\rangle \,|||\, s & = & \{s\} \\
s \,|||\, t & = & t \,|||\, s \\
\langle a\rangle^\frown s \,|||\, \langle b\rangle^\frown t & = & \{\langle a\rangle^\frown u \parallel u \in s \,|||\, \langle b\rangle^\frown t\} \\
 & & \cup\ \{\langle b\rangle^\frown u \parallel u \in t \,|||\, \langle a\rangle^\frown s\}
\end{array}
$$

$$
\begin{array}{rcll}
s \underset{X}{\parallel} t & = & t \parallel s & \\
\langle\rangle \underset{X}{\parallel} \langle\rangle & = & \{\langle\rangle\} & \\
\langle\rangle \underset{X}{\parallel} \langle x\rangle & = & \{\langle\rangle\} & (x \in X) \\
\langle\rangle \underset{X}{\parallel} \langle y\rangle & = & \{\langle y\rangle\} & (y \notin X) \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle y\rangle^\frown t & = & \{\langle y\rangle^\frown u \mid u \in \langle x\rangle^\frown s \underset{X}{\parallel} t\} & \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x\rangle^\frown t & = & \{\langle x\rangle^\frown u \mid u \in s \underset{X}{\parallel} t\} & \\
\langle x\rangle^\frown s \underset{X}{\parallel} \langle x'\rangle^\frown t & = & \{\langle\rangle\} & (x, x' \in X \ \wedge\ x \neq x') \\
\langle y\rangle^\frown s \underset{X}{\parallel} \langle y'\rangle^\frown t & = & \{\langle y\rangle^\frown u \mid u \in s \underset{X}{\parallel} \langle y'\rangle^\frown t\} & \\
 & & \cup\{\langle y'\rangle^\frown u \mid u \in t \underset{X}{\parallel} \langle y\rangle^\frown u\} &
\end{array}
$$

# APPENDIX D

# CSP MODEL FOR DTR1

### D.0.0.1 Nonce Manager process.

$$NonceSender(n) = \Box_{j \in Agents} send.NM.j.SqN.\langle n \rangle \rightarrow STOP$$

$$NonceManager = |||_{n \in Nonces} NonceSender(n)$$

### D.0.0.2 Peer processes.

$$TPMs = |||_{i \in \mathcal{P}^{\infty}} TPM(i,0)$$

$$TPM(i,c) = \Box_{\substack{n \in Nonces \\ j \in Agents}} \left( \begin{array}{l} receive.j.i.SqN.\langle n \rangle \\ \rightarrow \left( \begin{array}{l} \Box_{d \geqslant c} send.i.j.SqR.\langle n,i,d \rangle \rightarrow unlock.VF.i \rightarrow TPM(i,d) \\ \Box \Box_{d > c} send.i.j.SqI.\langle n,i,d \rangle \rightarrow unlock.CA.i \rightarrow TPM(i,d) \end{array} \right) \end{array} \right)$$

### D.0.0.3 CA Process.

$$\begin{aligned} CAProcess(ps,pn) = \quad & |ps| == 0 \ \& \ Join0(ps,pn) \\ & \Box \ |ps| == 1 \ \& \ Join1(ps,pn) \\ & \Box \ |ps| > 1 \ \& \ JoinAndLeaveN(ps,pn) \end{aligned}$$

$$
\begin{array}{l}
\mathit{Join0(ps,pn)} = \\
\quad \Box_{i \in pn} \left( \begin{array}{l} \mathit{receive.i.CA.Churn.}\langle \mathit{join}, i \rangle \\ \quad \rightarrow \Box_{n \in Nonces} \left( \begin{array}{l} \mathit{receive.NM.CA.SqN.}\langle n \rangle \rightarrow \mathit{send.CA.i.SqN.}\langle n \rangle \\ \quad \rightarrow \Box_{c \in Counts} \left( \begin{array}{l} \mathit{receive.i.CA.SqI.}\langle n, i, c \rangle \\ \quad \rightarrow \mathit{send.CA.i.Cert.}\langle i, i, i, c \rangle \\ \quad \rightarrow \mathit{completeChurn.Churn.}\langle \mathit{join}, i \rangle \\ \quad \rightarrow \mathit{unlock.CA.i} \rightarrow \mathit{CAProcess}(\{i\}, pn \backslash \{i\}) \end{array} \right) \end{array} \right) \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Join1(i,pn)} = \\
\quad \Box_{j \in pn} \left( \begin{array}{l} \mathit{receive.j.CA.Churn.}\langle \mathit{join}, j \rangle \\ \quad \rightarrow \Box_{n1, n2 \in Nonces} \left( \begin{array}{l} \mathit{receive.NM.CA.SqN.}\langle n1 \rangle \rightarrow \mathit{send.CA.i.SqN.}\langle n1 \rangle \\ \quad \rightarrow \Box_{c1, c2 \in Counts} \left( \begin{array}{l} \mathit{receive.i.CA.SqI.}\langle n1, i, c1 \rangle \\ \quad \rightarrow \mathit{send.CA.i.Cert.}\langle i, j, j, c1 \rangle \\ \quad \rightarrow \mathit{receive.NM.CA.SqN.}\langle n2 \rangle \\ \quad \rightarrow \mathit{send.CA.j.SqN.}\langle n2 \rangle \\ \quad \rightarrow \mathit{receive.j.CA.SqI.}\langle n2, j, c2 \rangle \\ \quad \rightarrow \mathit{send.CA.j.Cert.}\langle j, i, i, c2 \rangle \\ \quad \rightarrow \mathit{completeChurn.Churn.}\langle \mathit{join}, j \rangle \\ \quad \rightarrow \mathit{unlock.CA.i} \rightarrow \mathit{unlock.CA.j} \\ \quad \rightarrow \mathit{CAProcess}(\{i, j\}, pn \backslash \{j\}) \end{array} \right) \end{array} \right) \end{array} \right)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{JoinAndLeaveN(ps,pn)} & = & \Box_{i \in pn} \; \mathit{receive.i.CA.Churn.}\langle \mathit{join}, i \rangle \rightarrow \mathit{JoinN(i,ps,pn)} \\
& & \Box \Box_{i \in ps} \left( \begin{array}{l} \mathit{receive.i.CA.Churn.}\langle \mathit{leave}, i \rangle \\ \rightarrow \; \mathit{if} \; |ps| > 2 \; \mathit{then} \; \mathit{LeaveN(i,ps,pn)} \\ \quad \mathit{else} \; \mathit{Leave2(i,ps,pn)} \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Leave2(i,\{i,j\},pn)} = \\
\quad \mathit{receive.i.CA.Churn.}\langle \mathit{leave}, i \rangle \\
\quad \rightarrow \Box_{n \in Nonces} \left( \begin{array}{l} \mathit{receive.NM.CA.SqN.}\langle n \rangle \rightarrow \mathit{send.CA.j.SqN.}\langle n \rangle \\ \quad \rightarrow \Box_{c \in Counts} \left( \begin{array}{l} \rightarrow \mathit{receive.j.CA.SqI.}\langle n, j, c \rangle \\ \rightarrow \mathit{send.CA.j.Cert.}\langle j, j, j, c \rangle \\ \rightarrow \mathit{completeChurn.Churn.}\langle \mathit{leave}, i \rangle \\ \rightarrow \mathit{unlock.CA.j} \rightarrow \mathit{CAProcess}(\{j\}, pn \cup \{i\}) \end{array} \right) \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{JoinN(i,ps,pn)} = \\
\qquad \Box_{n1,n2,n3 \in Nonces} \Bigg( \mathit{receive.NM.CA.SqN.}\langle n1\rangle \rightarrow \mathit{send.CA.i.SqN.}\langle n1\rangle \\
\qquad\qquad \rightarrow \Box_{c1,c2,c3 \in Counts} \Bigg( \mathit{receive.i.CA.SqI.}\langle n1, i, c1\rangle \rightarrow \\
\qquad\qquad\qquad \mathit{let}\ S = ps \cup \{i\} \\
\qquad\qquad\qquad\qquad (l, r) = \mathit{neighbor}(i, S) \\
\qquad\qquad\qquad\qquad (l1, r1) = \mathit{neighbor}(l, S) \\
\qquad\qquad\qquad\qquad (l2, r2) = \mathit{neighbor}(r, S)\ \mathit{within} \\
\qquad\qquad\qquad\qquad\quad \mathit{send.CA.i.Cert.}\langle i, l, r, c1\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{receive.NM.CA.SqN.}\langle n2\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{send.CA.l.SqN.}\langle n2\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{receive.l.CA.SqI.}\langle n2, l, c2\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{send.CA.l.Cert.}\langle l, l1, i, c2\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{receive.NM.CA.SqN.}\langle n3\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{send.CA.r.SqN.}\langle n3\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{receive.r.CA.SqI.}\langle n3, r, c3\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{send.CA.r.Cert.}\langle r, i, r2, c3\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{completeChurn.Churn.}\langle \mathrm{join}, i\rangle \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{unlock.CA.i} \rightarrow \mathit{unlock.CA.l} \\
\qquad\qquad\qquad\qquad \rightarrow \mathit{unlock.CA.r} \rightarrow \mathit{CAProcess(S,pn\backslash\{i\})} \Bigg) \Bigg)
\end{array}
$$

$$
\begin{array}{l}
\mathit{LeaveN(i,ps,pn)} = \\
\quad \mathit{let}\ (l, r) = \mathit{neighbor}(i, ps) \\
\qquad (l1, r1) = \mathit{neighbor}(l, ps\backslash\{i\}) \\
\qquad (l2, r2) = \mathit{neighbor}(r, ps\backslash\{i\})\ \mathit{within} \\
\qquad\quad \Box_{n1,n2 \in Nonces} \Bigg( \mathit{receive.NM.CA.SqN.}\langle n1\rangle \rightarrow \mathit{send.CA.l.SqN.}\langle n1\rangle \\
\qquad\qquad \rightarrow \Box_{c1,c2 \in Counts} \Bigg( \mathit{receive.l.CA.SqI.}\langle n1, l, c1\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{send.CA.l.Cert.}\langle l, l1, r, c1\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{send.CA.r.SqN.}\langle n2\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{receive.r.CA.SqI.}\langle n2, r, c2\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{send.CA.r.Cert.}\langle r, l, r2, c2\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{completeChurn.Churn.}\langle \mathit{leave}, i\rangle \\
\qquad\qquad\qquad \rightarrow \mathit{unlock.CA.l} \rightarrow \mathit{unlock.CA.r} \\
\qquad\qquad\qquad \rightarrow \mathit{CAProcess}(ps\backslash\{i\}, pn \cup \{i\}) \Bigg) \Bigg)
\end{array}
$$

The function $\mathit{neighbor}(p, ps)$ basically returns the left and right neighbor of $p$ in $ps$.

More precisely,

$$neighbor(p, ps) = (left(p, ps), right(p, ps))$$

$$left(p, ps) = l \quad \text{if} \;\; l \in ps \;\wedge\; \forall p' \in ps \backslash \{l\} \,.\, |p \ominus l| \leqslant |p' \ominus l|$$

$$right(p, ps) = r \quad \text{if} \;\; r \in ps \;\wedge\; \forall p' \in ps \backslash \{r\} \,.\, |r \ominus p| \leqslant |r \ominus p'|$$

#### D.0.0.4 Verifier Process.

$$\begin{array}{l} VerifierProcess = \\ \quad \Box_{n \in Nonces} \left( \begin{array}{l} receive.NM.VF.SqN.\langle n \rangle \\ \quad \rightarrow \Box_{i,l,r \in \mathcal{P}^\infty} \left( \begin{array}{l} send.VF.i.SqN.\langle n \rangle \\ \quad \rightarrow \Box_{c \in Counts} \left( \begin{array}{l} receive.i.VF.SqR.\langle n, i, c \rangle \\ \quad \rightarrow receive.i.VF.Cert.\langle i, l, r, c \rangle \\ \quad \rightarrow \textit{if } l = r \textit{ and } l = i \textit{ then} \\ \qquad\qquad output.i.i \rightarrow unlock.VF.i \\ \qquad\qquad \rightarrow STOP \\ \quad \textit{else } VerifierProcessN(l,i) \end{array} \right) \end{array} \right) \end{array} \right) \end{array}$$

$$\begin{array}{l} VerifierProcessN(l,i) = \\ \quad \Box_{n \in Nonces} \left( \begin{array}{l} receive.NM.VF.SqN.\langle n \rangle \rightarrow send.VF.r.SqN.\langle n \rangle \\ \quad \rightarrow \Box_{cl \in Counts} \left( \begin{array}{l} receive.l.VF.SqR.\langle n, l, cl \rangle \\ \quad \rightarrow \Box_{ll \in \mathcal{P}^\infty} \left( \begin{array}{l} receive.l.VF.Cert.\langle l, ll, i, cl \rangle \\ \quad \rightarrow output.l.i \rightarrow unlock.VF.l \\ \quad \rightarrow unlock.VF.i \rightarrow STOP \end{array} \right) \end{array} \right) \end{array} \right) \end{array}$$

#### D.0.0.5 Adversary Process.

$$
\begin{aligned}
\mathit{MemoryNonce(n)} &= \mathit{learn.SqN.}\langle n\rangle \rightarrow \mathit{ReplayNonce(n)}\\
\mathit{ReplayNonce(n)} &= \mathit{say.SqN.}\langle n\rangle \rightarrow \mathit{ReplayNonce(n)}\\
\\
\mathit{MemorySigR(n,i,c)} &= \mathit{learn.SqR.}\langle n,i,c\rangle \rightarrow \mathit{ReplaySigR(n,i,c)}\\
\mathit{MemorySigI(n,i,c)} &= \mathit{learn.SqI.}\langle n,i,c\rangle \rightarrow \mathit{ReplaySigI(n,i,c)}\\
\mathit{ReplaySigR(n,i,c)} &= \mathit{say.SqR.}\langle n,i,c\rangle \rightarrow \mathit{ReplaySigR(n,i,c)}\\
\mathit{ReplaySigI(n,i,c)} &= \mathit{say.SqI.}\langle n,i,c\rangle \rightarrow \mathit{ReplaySigI(n,i,c)}\\
\\
\mathit{MemoryCert(i,l,r,c)} &= \mathit{learn.Cert.}\langle i,l,r,c\rangle \rightarrow \mathit{ReplayCert(i,l,r,c)}\\
\mathit{ReplayCert(i,l,r,c)} &= \mathit{say.Cert.}\langle i,l,r,c\rangle \rightarrow \mathit{ReplayCert(i,l,r,c)}\\
\\
\mathit{Memory} &= \;\;\underset{n\in Nonces}{|||}\; \mathit{MemoryNonce(n)}\\
&\quad ||| \underset{n\in Nonces, i\in\mathcal{P}, c\in Counts}{|||} \left(\begin{array}{c} \mathit{MemorySigR(n,i,c)} \\ ||| \;\; \mathit{MemorySigI(n,i,c)} \end{array}\right)\\
&\quad ||| \underset{i,l,r\in\mathcal{P}, c\in Counts}{|||} \mathit{MemoryCert(i,l,r,c)}\\
\\
\mathit{ChurnInitiator} &= \underset{i\in\mathcal{P}}{\Box} \left(\begin{array}{c} \mathit{say.Churn.}\langle \text{join}, i\rangle \rightarrow \mathit{ChurnInitiator} \\ \Box\; \mathit{say.Churn.}\langle \text{leave}, i\rangle \rightarrow \mathit{ChurnInitiator} \end{array}\right)\\
\mathit{Adversary} &= \mathit{Memory} \;|||\; \mathit{ChurnInitiator}
\end{aligned}
$$

#### D.0.0.6 Implementation process (DTR1 model).

$$
\begin{aligned}
\mathit{Network} &= \left(\; \mathit{Adversary} \underset{\chi_i}{\|} \mathit{TPMs} \;\right) \backslash \chi_i\\
\mathit{CAandVFProcess} &= \mathit{CAProcess}(\{\},\mathcal{P}) \;\; ||| \;\; \mathit{VerifierProcess}\\
\mathit{OtherAgents} &= \left(\; \mathit{NonceManager} \underset{\{|fake.NM|\}}{\|} \mathit{CAandVFProcess} \;\right)\\
\mathit{Impl} &= \left(\; \mathit{OtherAgents} \underset{\chi_e}{\|} \mathit{Network} \;\right) \backslash \{|take, fake, unlock|\}
\end{aligned}
$$

#### D.0.0.7 Specification process.

$$
\begin{aligned}
Spec(ps, pn) \quad = \quad & \Box_{i \in pn} \; completeChurn.Churn.{<}\, join, i \,{>} \rightarrow Spec(ps \cup \{i\}, pn \backslash \{i\}) \\
& \Box \Box_{i \in ps} \; completeChurn.Churn.{<}\, leave, i \,{>} \rightarrow Spec(ps \backslash \{i\}, pn \cup \{i\}) \\
& \Box \Box_{i \in ps} \; output.i.right(i,ps) \rightarrow Spec(ps, pn)
\end{aligned}
$$

# APPENDIX E

# FDR IMPLEMENTATION FOR DTR1 CSP MODEL

```
PeerIDs = {0..2}

datatype AgentType = VF | CA | Peer.PeerIDs

datatype Data =  nonceCA | nonceVF | join | leave | countVal | SqN.Seq(Data)
                | SqR.Seq(Data) | SqI.Seq(Data) | Cert.Seq(Data)  | Churn.Seq(Data)

NonceSet =   {nonceCA, nonceVF}
CounterSet = {countVal}
PeerSet =    {Peer.id  | id  <- PeerIDs}
AgentSet =   union( PeerSet, {VF, CA} )
OtherAgentSet = {VF, CA}

NonceMessages = {SqN.<n> | n <- NonceSet}
SigMessagesR =  {SqR.<n,i,c> | n <- NonceSet, i <- PeerSet, c <- CounterSet}
SigMessagesI =  {SqI.<n,i,c> | n <- NonceSet, i <- PeerSet, c <- CounterSet}
SigMessages =  union( SigMessagesR, SigMessagesI )
CertMessages =  {Cert.<i,l,r,c> | i <- PeerSet, l <- PeerSet, r <- PeerSet, c <- CounterSet}
ChurnMessages = {Churn.<c,i> | c <- {join,leave},  i <- PeerSet}

MESSAGES = Union({NonceMessages, SigMessages, CertMessages, ChurnMessages})


channel  learn,say:  MESSAGES
channel  take,fake,send,receive:  AgentSet.AgentSet.MESSAGES
channel  output:  PeerSet.PeerSet
channel  completeChurn: ChurnMessages
channel  unlock: AgentSet.PeerSet
channel  nextVal: PeerSet.CounterSet

--------------------------- TPM AND ADVERSARY PROCESSES --------------------------------
TPM(i)= []j:OtherAgentSet, n:NonceSet @
                    receive.j.i.SqN.<n> -> (  ( send.i.j.SqR.<n,i,countVal> -> unlock.VF.i ->  TPM(i) )
                                           [](  nextVal.i.countVal -> send.i.j.SqI.<n,i,countVal>
                                               -> unlock.CA.i -> TPM(i))
                                          )

TPMsO = |||i:PeerSet @ TPM(i)

TPMs = TPMsO [[ send.i.j  <-  take.i.j  | i <- PeerSet,  j <- OtherAgentSet ]]
```

```
            [[ receive.j.i <- fake.j.i  | j <- OtherAgentSet, i <- PeerSet ]]
alpha_TPM = union ( {fake.j.i.nonceMes, take.i.j.sigMes | i <- PeerSet, j <- OtherAgentSet,
                                                 nonceMes <- NonceMessages, sigMes <- SigMessages},
                  {nextVal.i.c | i <- PeerSet, c <- CounterSet} )

RelayNonce0 = ( []n:NonceSet @
                               learn.SqN.<n> -> (  ( say.SqN.<n> -> RelayNonce0 )
                                                  []RelayNonce0
                                                 )
              )

RelayNonce = RelayNonce0  [[ learn <- take.j.i | i <- PeerSet, j <- OtherAgentSet ]]
                          [[ say <- fake.j.i | i <- PeerSet,  j <- OtherAgentSet ]]
alpha_RelayNonce = {take.j.i.mesN, fake.j.i.mesN | i <- PeerSet,  j <- OtherAgentSet,
                                                   mesN <- NonceMessages}

RelaySigR(i) = []n:NonceSet @
                               learn.SqR.<n,i,countVal> -> (  ( say.SqR.<n,i,countVal> -> RelaySigR(i) )
                                                               []RelaySigR(i)
                                                              )

RelaySigI(i) = []n:NonceSet @
                               learn.SqI.<n,i,countVal> -> (  ( say.SqI.<n,i,countVal> -> RelaySigI(i) )
                                                               []RelaySigI(i)
                                                              )

RelaySigs0 = |||i:PeerSet @ ( RelaySigR(i) ||| RelaySigI(i) )

RelaySigs = RelaySigs0  [[ learn <- take.i.j | i <- PeerSet,  j <- OtherAgentSet ]]
                        [[ say <- fake.i.j | i <- PeerSet, j <- OtherAgentSet]]
alpha_RelaySigs = {take.i.j.mes, fake.i.j.mes | i <- PeerSet, j <- OtherAgentSet,
                                                mes <- SigMessages}


RelayNonceSigs =  RelayNonce ||| RelaySigs
alpha_RelayNonceSigs = union ( alpha_RelayNonce, alpha_RelaySigs)


RelayCert(i) = []l:PeerSet, r:PeerSet @
                                    nextVal.i.countVal ->(  ( learn.Cert.<i,l,r,countVal> ->
                                                                (  ( say.Cert.<i,l,r,countVal>
                                                                     -> RelayCert(i) )
                                                                 []RelayCert(i)
                                                                )
                                                              )
                                                             []RelayCert(i)
                                                            )

RelayCerts = |||i:PeerSet @ RelayCert(i) ) [[ learn <- take.i.j | i <- OtherAgentSet,
                                                                  j <- PeerSet ]]
                                           [[ say <- fake.i.j | i <- PeerSet,  j <- OtherAgentSet ]]
alpha_RelayCerts = union( {take.i.j.mes, fake.j.i.mes | i <- OtherAgentSet,
                                                        j <- PeerSet,  mes <- CertMessages},
                        {nextVal.i.c | i <- PeerSet, c <- CounterSet} )

ChurnInitiator0 =  []i:PeerSet @
                                 (   say.Churn.<join,i> -> ChurnInitiator0
                                  [] say.Churn.<leave,i>  -> ChurnInitiator0
```

```
                                )

ChurnInitiator = ChurnInitiator0 [[ say <- fake.i.j | i <- PeerSet,  j <- OtherAgentSet ]]
alpha_ChurnInitiator = {fake.i.j.mes | i <- PeerSet, j <- OtherAgentSet, mes <- ChurnMessages}

Adversary = RelayNonceSigs  ||| RelayCerts ||| ChurnInitiator
alpha_Adversary = Union ( {alpha_RelayNonceSigs, alpha_RelayCerts, alpha_ChurnInitiator} )

Network = ( Adversary [| alpha_TPM |] TPMs ) \ alpha_TPM
alpha_Network = diff (alpha_Adversary, alpha_TPM)

--------------------------------------------------------------------------------



--------------------------- OTHER PROCESSES ------------------------------------------
CAProcess(ps,pn) =    ( card(ps)==0 & Join0(pn) )
                  [] ( card(ps) == 1 & ( Join1(ps,pn)   [] Leave1(ps,pn) ) )
                  [] ( card(ps) == 2 & ( JoinN(ps,pn) [] Leave2(ps,pn) ) )
                  [] ( card(ps) > 2 & ( JoinN(ps,pn) [] LeaveN(ps,pn) ) )

Join0(pn) = []i:pn @
                       receive.i.CA.Churn.<join,i>  -> send.CA.i.SqN.<nonceCA>
                       -> receive.i.CA.SqI.<nonceCA, i, countVal> -> send.CA.i.Cert.<i,i,i,countVal>
                       -> completeChurn.Churn.<join,i> -> unlock.CA.i -> CAProcess({i},  diff(pn, {i}))

Join1(ps,pn) = []i:pn @
                       receive.i.CA.Churn.<join,i>  -> send.CA.i.SqN.<nonceCA>
                       -> receive.i.CA.SqI.<nonceCA,i,countVal>
                       -> ( let (l,r) = neighbor(i, union(ps,{i})) within
                                (
                                       send.CA.l.SqN.<nonceCA>
                                       -> receive.l.CA.SqI.<nonceCA, l, countVal>
                                       ->  send.CA.i.Cert.<i,l,l,countVal>
                                       -> send.CA.l.Cert.<l,i,i,countVal>
                                       -> completeChurn.Churn.<join,i>  -> unlock.CA.i -> unlock.CA.l
                                       -> CAProcess( union({i}, ps), diff(pn, {i}) )
                                )
                          )

Leave1(ps,pn) = []i:ps @
                        receive.i.CA.Churn.<leave,i>  -> send.CA.i.SqN.<nonceCA>
                        -> receive.i.CA.SqI.<nonceCA,i,countVal> -> completeChurn.Churn.<leave,i>
                        -> unlock.CA.i -> CAProcess ({}, union (pn, {i}))

JoinN(ps,pn) = []i:pn @
                       receive.i.CA.Churn.<join,i> -> send.CA.i.SqN.<nonceCA>
                       -> receive.i.CA.SqI.<nonceCA,i,countVal>
                       -> ( let (l,r) = neighbor(i, union(ps, {i}))
                                    (ll, lr) = neighbor(l, union(ps, {i}))
                                    (rl, rr) = neighbor(r, union(ps, {i}))
                              within
                              (
                                       send.CA.l.SqN.<nonceCA> -> receive.l.CA.SqI.<nonceCA,l,countVal>
                                       -> send.CA.r.SqN.<nonceCA>
                                       -> receive.r.CA.SqI.<nonceCA,r,countVal>
                                       -> send.CA.i.Cert.<i,l,r,countVal>
                                       -> send.CA.l.Cert.<l,ll,i,countVal>
                                       -> send.CA.r.Cert.<r,i,rr,countVal>
```

```
                                    -> completeChurn.Churn.<join,i>
                                    -> unlock.CA.i -> unlock.CA.l -> unlock.CA.r
                                    -> CAProcess( union({i},ps), diff(pn,{i}) )
                                )
                            )

Leave2(ps,pn) = []i:ps @
                        receive.i.CA.Churn.<leave,i>
                        -> ( let (l,r) = neighbor(i, ps) within
                                (
                                  send.CA.l.SqN.<nonceCA> -> receive.l.CA.SqI.<nonceCA,l,countVal>
                                  -> send.CA.l.Cert.<l, l, l, countVal>
                                  -> completeChurn.Churn.<leave,i> ->  unlock.CA.l
                                  -> CAProcess({l}, union(pn,{i}))
                                )
                           )

LeaveN(ps,pn) = []i:ps @
                        receive.i.CA.Churn.<leave,i>
                        -> ( let (l,r) = neighbor(i, ps)
                                 (ll,lr) = neighbor(l, ps)
                                 (rl,rr) = neighbor(r,ps)
                                  within
                                   (
                                     send.CA.l.SqN.<nonceCA> -> receive.l.CA.SqI.<nonceCA,l,countVal>
                                     -> send.CA.r.SqN.<nonceCA> -> receive.r.CA.SqI.<nonceCA,r,countVal>
                                     -> send.CA.l.Cert.<l, ll,r, countVal>
                                     -> send.CA.r.Cert.<r,l,rr,countVal> -> completeChurn.Churn.<leave,i>
                                     -> unlock.CA.l -> unlock.CA.r
                                     -> CAProcess(diff(ps,{i}), union( pn, {i}))
                                   )
                           )

VerifierProcess = []i:PeerSet @
                              send.VF.i.SqN.<nonceVF> -> receive.i.VF.SqR.<nonceVF, i, countVal>
                              -> []l:PeerSet, r:PeerSet @
                                             (
                                               receive.i.VF.Cert.<i,l,r,countVal>
                                               -> (  ( (i==l and i==r) & output.i.i
                                                                       -> unlock.VF.i -> STOP )
                                                     []( (l==r and i!=l) & Verifier1(i,r) )
                                                  )
                                             )

Verifier1(i,r) =   send.VF.r.SqN.<nonceVF> -> receive.r.VF.SqR.<nonceVF, r, countVal>
                   -> []rr:PeerSet @ ( receive.r.VF.Cert.<r,i,rr,countVal> -> output.i.r
                                       -> unlock.VF.i -> unlock.VF.r -> STOP )

OtherAgents  = ( CAProcess({}, PeerSet) ||| VerifierProcess ) [[ send.i.j <- take.i.j | i <- OtherAgentSet,
                                                                                        j <- PeerSet ]]
                                                              [[ receive.i.j <- fake.i.j | i <- PeerSet,
                                                                                           j <- OtherAgentSet
                                                              ]]
alpha_OtherAgents = union ( {take.i.j.mes, fake.j.i.mes | i <- OtherAgentSet, j <- PeerSet, mes <- MESSAGES},
                            {unlock.j.i  |  j <- OtherAgentSet, i <- PeerSet} )

--------------------------------------------------------------------------------
```

```
-------------------------------- IMPLEMENTATION AND SPEC ------------------------------
alpha_Hidden = {|take,fake,unlock|}
alpha_Sync = union ( diff(alpha_Adversary, alpha_TPM),
                    {unlock.j.i | j <- OtherAgentSet, i <- PeerSet} )

Impl = (OtherAgents [|alpha_Sync|] Network) \ alpha_Hidden

Spec(ps,pn) =  ( []i:pn @ completeChurn.Churn.<join,i> -> Spec(union(ps,{i}),diff(pn,{i})) )
             [] ( []i:ps @ completeChurn.Churn.<leave,i> -> Spec(diff(ps,{i}),union(pn,{i})) )
             [] ( card(ps)>0 & []i:ps @ ( let r=right(i,ps) within
                                                          ( output.i.r -> Spec(ps,pn) )
                                      )
                )

assert Spec({}, PeerSet) [T= Impl
---------------------------------------------------------------------------------------

---------------------------------- AUXILILARY  ----------------------------------------
neighbor(Peer.0,S) = if (member(Peer.0,S) and member(Peer.1,S) and card(S)==2) then (Peer.1,Peer.1)
                     else if (member(Peer.0,S) and member(Peer.2,S) and card(S)==2) then (Peer.2,Peer.2)
                          else if (card(S)==3) then (Peer.2,Peer.1)
                               else if (member(Peer.0,S)) then (Peer.0,Peer.0)
                                    else if (member(Peer.1,S)) then (Peer.1,Peer.1)
                                         else (Peer.2,Peer.2)

neighbor(Peer.1,S) = if (member(Peer.0,S) and member(Peer.1,S) and card(S)==2) then (Peer.0,Peer.0)
                     else if (member(Peer.1,S) and member(Peer.2,S) and card(S)==2) then (Peer.2,Peer.2)
                          else if (card(S)==3) then (Peer.0,Peer.2)
                               else let {x}=S within (x,x)

neighbor(Peer.2,S) = if (member(Peer.2,S) and member(Peer.1,S) and card(S)==2) then (Peer.1,Peer.1)
                     else if (member(Peer.0,S) and member(Peer.2,S) and card(S)==2) then (Peer.0,Peer.0)
                          else if (card(S)==3) then (Peer.1,Peer.0)
                               else let {x}=S within (x,x)

left(i,P) = let (x,y)=neighbor(i,P) within x
right(i,P) = let (x,y)=neighbor(i,P) within y

---------------------------------------------------------------------------------------
```

# APPENDIX F

# THE PROOF FOR DTR1

## Preliminaries

$$precedes(X,Y) = \lambda\, tr \,.\, \forall y \in Y, s_1, s_2 \bullet \big(tr = s_1^\frown\langle y\rangle^\frown s_2 \Rightarrow \exists x \in X, t_1, t_2 \bullet\ s_1 = t_1^\frown\langle x\rangle^\frown t_2\big)$$

$$strictPrecedes(X,Y) = \lambda\, tr \,.\, \forall y \in Y, s_1, s_2 \bullet \big(tr = s_1^\frown\langle y\rangle^\frown s_2 \Rightarrow \exists x \in X, t_1, t_2 \bullet\ s_1 = t_1^\frown\langle x\rangle^\frown t_2 \ \wedge\ t_2 \upharpoonright X \cup Y = \langle\rangle\big)$$

$$strictFollowed(X,Y) = \lambda\, tr \,.\, tr \upharpoonright (X \cup Y) \in \{\langle\rangle, \langle a\rangle, \langle a,b\rangle \mid a \in X, b \in Y\}^*$$

$$\frac{s \upharpoonright \alpha \in (S \setminus X)^* \quad s \upharpoonright \beta \in (S \setminus Y)^*}{s \upharpoonright (\alpha \cup \beta) \in (S \setminus (X \cap Y))^*} \tag{F.0.1}$$

**Lemma 2.** *Let Abstraction' be the model consisting of the same processes as Abstraction without hidding the events in $\{take, fake, unlock\}$. Let $tr$ be any trace of Abstraction'. For any $l, r$, let $o = output.l.r$, $C(l,r) = \{fake.r.VF.Cert.\langle r, l, rr, c_d\rangle \mid rr \in \mathcal{P}^\infty\}$ and $ct \in C(l,r)$.*

*For any $s_1, s_2, s_3$ such that $tr = s_1^\frown\langle ct\rangle^\frown s_2^\frown\langle o\rangle^\frown s_3$, let $X1 = \gamma(\{\}, s_1)$ and $X2 = \gamma(\{\}, s_1^\frown s_2)$. Then:*

$$\{l,r\} \subseteq X1 \ \wedge\ r = right(l, X1) \ \Rightarrow\ \{l,r\} \subseteq X2 \ \wedge\ r = right(l, X2)$$

*Proof.* The proof is included in the technical report [30] □

**Lemma 3.** *Let Abstraction' be the model consisting of the same processes as Abstraction without hidding the events in $\{take, fake, unlock\}$. Let $tr \in traces(Abstraction')$ and $X = \gamma(\{\}, tr)$. For any $i$, denote $e_j = completeChurn.Churn.\langle join, i\rangle$, $e_l = completeChurn.Churn.\langle leave, i\rangle$. If there exists $e \in \{e_j, e_l\}$,*

*$s, t$ such that:*

$$tr = s^\frown\langle e\rangle^\frown t \ \wedge\ t \upharpoonright \{|completeChurn|\} = \langle\rangle$$
$$\wedge\ t \upharpoonright \{|fake.p.VF|\} = \langle fake.p.VF.Cert.\langle p, p_l, p_r, c_d\rangle\rangle$$

*then*

1. *If $e = e_j$, let $l = left(i, X)$, $r = right(i, X)$:*

$$p \in \{i, l, r\} \Rightarrow \{p, p_l, p_r\} \subseteq X \ \wedge\ p = right(p_l, X) = left(p_r, X) \tag{F.0.2}$$

2. *If $e = e_l$, let $X' = X \cup \{i\}$, $l = left(i, X')$, $r = right(i, X')$:*

$$p \in \{l, r\} \Rightarrow \{p, p_l, p_r\} \subseteq X \ \wedge\ p = right(p_l, X) = left(p_r, X)$$

*Proof.* The proof is included in the technical report [30] □

**Theorem 3.** $traces(Abstraction) \subseteq traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}}))$

*Proof.* The proof is constructed via induction as follows:

1. (Base case). Let $tr$ be a traces of *Abstraction* such that $tr \upharpoonright \{|completeChurn|\} = \langle\rangle$ where $\upharpoonright$ is the restriction operator (for example $sq \upharpoonright X$ removes non-$X$ elements from $sq$). Then $tr \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}}))$.

2. (Inductive case). For any $\theta \neq \langle\rangle$, let $tr$ be a trace of *Abstraction* such that:

$$tr \upharpoonright \{|completeChurn|\} = \theta \ \wedge\ tr \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}}))$$

   Let $tr'$ be another trace of *Abstraction*, then:

$$\forall e \,.\, tr' \upharpoonright \{|completeChurn|\} = \theta^\frown\langle e\rangle \ \Rightarrow\ tr' \in traces(Spec(\{\}), \mathcal{P}^{\mathcal{O}})$$

The base case is true because

$$tr \upharpoonright \{|completeChurn|\} = \langle\rangle \ \Rightarrow\ tr \upharpoonright \{|output|\} = \langle\rangle$$

since *Abstraction* `sat` $precedes(\{|completeChurn|\}, \{|output|\})$.

Let $Abstraction'$ be the model consisting of the same processes as $Abstraction$ does, but without hiding the events in $\Omega = \{|take, fake, unlock|\}$.

Let $\epsilon = \{|completeChurn|\}$, the proof for the inductive case is as follows. For any $\theta \neq \langle\rangle$, $e \in \epsilon$, consider $tr' \in traces(Abstraction')$ such that $tr' \upharpoonright \epsilon = \theta \frown \langle e \rangle$. On the assumption:

$$\forall tr \in traces(Abstraction) \bullet \ tr \upharpoonright \epsilon = \theta \Rightarrow tr \setminus \Omega \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}})) \tag{F.0.3}$$

we need to show the following:

$$tr' \setminus \Omega \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}})) \tag{F.0.4}$$

For $tr'$ described above, there exists $s, t$ such that:

$$tr' = s \frown \langle e \rangle \frown t \ \ \wedge \ \ s \upharpoonright \epsilon \neq \langle\rangle \ \ \wedge \ \ t \upharpoonright \epsilon = \langle\rangle$$

Consider $e = completeChurn.Churn.\langle join, i \rangle$ for any $i \in \mathcal{P}^{\mathcal{O}}$ (the proof is similar for the other values of $e \in \{completeChurn.Churn.\langle leave, i \rangle \mid i \in \mathcal{P}^{\mathcal{O}}\}$).

First, it is true that

$$\begin{aligned} s \frown \langle e \rangle \setminus \Omega &= s \frown \langle e \rangle \upharpoonright \alpha_{Spec} \\ &\in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}})) \end{aligned} \tag{F.0.5}$$

because:

1. Let $\delta = \{|output|\}$. Because $s \upharpoonright \epsilon = \theta$, it follows from the assumption in Eq.F.0.3 that

$$s \frown \langle e \rangle \upharpoonright \delta = s \upharpoonright \delta \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}}) \setminus (\alpha_{Spec} \setminus \delta))$$

2.

$$\begin{aligned} s \frown \langle e \rangle \upharpoonright \epsilon = s \frown \langle e \rangle \setminus \Omega &= \theta \frown \langle e \rangle \\ &= s \frown \langle e \rangle \upharpoonright (\alpha_{CAProcess \cap \epsilon}) \\ &\in traces(CAProcess \setminus (\alpha_{CAProces} \setminus \epsilon)) \\ &\in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}})) \setminus (\alpha_{Spec} \setminus \epsilon) \end{aligned}$$

From these results and Eq.F.0.1, we have

$$s^\frown\langle e\rangle \upharpoonright (\delta \cup \epsilon) = s^\frown\langle e\rangle \upharpoonright \alpha_{Spec}$$
$$\in traces(CAProcess)$$

**If** $t \upharpoonright \delta = \langle\rangle$ . This means:

$$tr' \setminus \Omega = s^\frown\langle e\rangle \setminus \Omega$$
$$\in traces(Spec(\{\}, \mathcal{P}^\infty)) \quad \text{(because of Eq.F.0.5)}$$

**If** $tr' \upharpoonright \delta = \langle output.l.r\rangle$ **for any** $l, r \in \mathcal{P}^\infty$**.** This means there exists $t_1, t_2$ such that:

$$tr' = s^\frown\langle e\rangle^\frown t_1^\frown\langle output.l.r\rangle^\frown t_2$$

Let $P_s = \gamma(\{\}, s^\frown\langle e\rangle)$, Eq.F.0.5 implies that we need to show that:

$$\{l, r\} \subseteq P_s \ \wedge \ r = right(l, P_s) \tag{F.0.6}$$

Let $O(l,r) = \{output.l.r\}$, $C1(l,r) = \{fake.l.VF.Cert.\langle l, ll, r, c_d\rangle \mid ll \in \mathcal{P}^\infty\}$ and $C2(l,r) = \{fake.r.VF.Cert.\langle r, l, rr, c_d\rangle \mid rr \in \mathcal{P}^\infty\}$. From the structure of the *VerifierProcess* process, we have the following:

1. *strictFollowed(C1(l,r), O(l,r))*$(tr')$
2. *strictFollowed(C2(l,r), O(l,r))*$(tr')$
3. $l \neq r \Rightarrow$ *strictFollowed(C1(l,r), C2(l,r))*

Let $ct_r = fake.r.VF.\langle r, l, rr, c_d\rangle \in C2(l,r)$, we therefore need to consider two cases:

**Case 1.** If there exists $s_1, s_2$ such that:

$$s = s_1^\frown\langle ct_r\rangle^\frown s_2 \ \wedge \ s_2^\frown t_1 \upharpoonright C2(l,r) = \langle\rangle$$

Notice that events in $O(l,r)$ only occurs in the trace of *VerifierProcess*. Let $s' = s_1^\frown\langle ct_r, output.l.r\rangle^\frown s_2$. We can deduce that:

$$s'^\frown\langle e\rangle^\frown t \ \in \ traces(Abstraction)$$

Because of Eq.F.0.3, $s' \setminus \Omega \in traces(Spec(\{\}, \mathcal{P}^\infty))$. Therefore, $\{l, r\} \subseteq \gamma(\{\}, s_1)$ and $r = right(l, \gamma(\{\}, s_1))$.

Lemma 2 implies that:

$$\{l, r\} \subseteq \gamma(\{\}, s^\frown\langle e\rangle) \ \wedge \ r = right(l, \gamma(\{\}, s^\frown\langle e\rangle))$$

**Case 2.** If there exists $r_1, r_2$ such that:

$$t_1 = r_1^\frown\langle ct_r\rangle^\frown r_2$$

Let $i_l = left(i, P_s)$, $i_r = right(i, P_s)$. Consider the following two sub-cases:

1. If $r \in \{i, i_l, i_r\}$. Lemma 3 implies that:

   $$\{r, l\} \subseteq P_s \ \wedge \ r = right(l, P_s)$$

   (same as Eq.F.0.6)

2. If $r \notin \{i, i_l, i_r\} \ \wedge \ l \in \{i, i_l, i_r\}$. Let $ct_l = fake.l.CA.Cert.\langle l, ll, r, c_d\rangle$ for some $l$. Because of *strictPrecedes*$(\{ct_l,\ ct_r\})(tr')$ being true and the structure of the CA process and *RelayCerts*, there exists $u_1, u_2$ such that:

   $$r_1 = u_1^\frown\langle ct_l\rangle^\frown u_2$$

   Similar to the previous case, Lemma 3 implies Eq.F.0.6

3. If $\{l, r\} \cap \{i, i_l, i_r\} = \varnothing$. Let $\alpha_v = \alpha_{VerifierProcess}$. It can be shown that:

   $$s^\frown(t \restriction \alpha_v)^\frown\langle e\rangle^\frown(t \setminus \alpha_v) \ \in \ traces(Abstraction)$$

   Let $s' = s^\frown(t \restriction \alpha_v)$, then according to Eq.F.0.3, we have $s' \setminus \Omega \in traces(Spec(\{\}, \mathcal{P}^{\mathcal{O}}))$. As $t \restriction \alpha_v$ contains $output.l.r$, it follows that $\{l, r\} \subseteq \gamma(\{\}, s')$ and $r = right(l, \gamma(\{\}, s'))$. Because of $\{l, r\} \cap \{i, i_l, i_r\}$ and $t \restriction \epsilon = \langle\rangle$, we can deduce that:

   $$\{l, r\} \subseteq P_s \ \wedge \ r = right(l, P_s)$$

□

# APPENDIX G

# FDR IMPLEMENTATION FOR DTR2 CSP MODEL

```
PeerIDs = {0..2}

datatype AgentType = VF

datatype Data = nonceVF |  Peer.PeerIDs | SqN.Seq(Data)
                | Ran.Seq(Data) | Cert.Seq(Data)

NonceSet = {nonceVF}
PeerSet =  {Peer.id  | id  <- PeerIDs}
AgentSet = union( PeerSet, {VF} )
OtherAgentSet = {VF}

NonceMessages = {SqN.<n> | n <- NonceSet}
RangesMessages = {Ran.<i,l,r> | i <- PeerSet, l <- PeerSet, r <- PeerSet}
CertMessages = {Cert.<n>, Cert.<n,l,r> | n <- NonceSet, l <- PeerSet, r <- PeerSet}

MESSAGES = Union({NonceMessages, RangesMessages, CertMessages})

channel  learn,say:  MESSAGES
channel  take,fake,send,receive:  AgentSet.AgentSet.MESSAGES
channel  output:  PeerSet.PeerSet
channel  join,leave: PeerSet.PeerSet
channel  unlock: AgentSet.PeerSet
channel  testEvent:PeerSet.PeerSet

-------------------------- DEVICE PROCESS ------------------------------------------

DeviceN(i) =   ( receive.VF.i.SqN.<nonceVF> -> send.i.VF.Cert.<nonceVF>
                 -> unlock.VF.i -> DeviceN(i)
               )
            [] ( []j:PeerSet @ ( []l:PeerSet, r:PeerSet @ (
                                                    receive.j.i.Ran.<r,l,i>
                                                    -> join.i.r -> Device(i,l)
                                                  )
                               )
               )

Device(i,l) =  ( receive.VF.i.SqN.<nonceVF> -> send.i.VF.Cert.<nonceVF,l,i>
                 -> unlock.VF.i -> Device(i,l)
               )
```

```
            []( []ll:PeerSet @ receive.l.i.Ran.<l,ll,l>  -> leave.l.i
                                -> Device(i,ll)
              )
            []( []j:diff(PeerSet,{i}) @ ( if ( mid(j,l,i) ) then ( send.i.j.Ran.<i,l,j>
                                                                   -> Device(i,j) )
                                        else (  send.i.j.Ran.<i,l,i> -> DeviceN(i) )
                                      )
              )

newPeerSet = diff(PeerSet, {Peer.0})
Devices0 = []i:PeerSet @ ( Device(i,i) ||| (|||j:diff(PeerSet,{i}) @ DeviceN(j) ) )
Devices = Devices0 [[ send.i.j  <- take.i.j | i <- PeerSet, j<- AgentSet ]]
                   [[ receive.j.i <- fake.j.i | j <- AgentSet, i <- PeerSet]]
joinAndLeaveSet = {join.i.j, leave.i.j | i <- PeerSet, j <- PeerSet}
alpha_Devices = Union ( { {fake.VF.i.nonMesg | i <- PeerSet, nonMesg <- NonceMessages},
                          {take.i.VF.certMesg | i <- PeerSet, certMesg <- CertMessages},
                          {fake.j.i.ranMesg | i <- PeerSet, j <- PeerSet,
                                              ranMesg <- RangesMessages},
                          {take.i.j.ranMesg | i <- PeerSet, j <- PeerSet,
                                              ranMesg <- RangesMessages},
                          joinAndLeaveSet }
                      )
-------------------------------------------------------------------------------------

--------------------------- ADVERSARY PROCESS ----------------------------------------

RelayNonce0 = learn.SqN.<nonceVF> -> say.SqN.<nonceVF> -> RelayNonce0
RelayNonce = RelayNonce0 [[learn <- take.VF.i | i <- PeerSet]]
                         [[say <- fake.VF.i | i <- PeerSet]]

RelayCert0 = []l:PeerSet, r:PeerSet @ (
                                       learn.Cert.<nonceVF, l, r>
                                       -> say.Cert.<nonceVF,l,r> -> STOP
                                     )
RelayCert = RelayCert0 [[learn <- take.i.VF | i <- PeerSet]]
                       [[say <- fake.i.VF | i <- PeerSet]]

RelayRange(i,l,r) = learn.Ran.<i,l,r> -> say.Ran.<i,l,r> -> RelayRange(i,l,r)
RelayRanges0 = |||i:PeerSet, l:PeerSet, r:PeerSet @ RelayRange(i,l,r)
RelayRanges = RelayRanges0 [[learn <- take.i.j | i <- PeerSet, j <- PeerSet]]
                           [[say <- fake.i.j | i <- PeerSet, j <- PeerSet]]

Adversary = RelayRanges ||| RelayNonce ||| RelayCert ||| RelayCert
alpha_Adversary = Union( {  {take.VF.i.nonMesg, fake.VF.i.nonMesg | i <- PeerSet,
                                                                    nonMesg <- NonceMessages},
                            {take.i.VF.certMesg, fake.i.VF.certMesg | i <- PeerSet,
                                                                      certMesg <- CertMessages},
                            {take.i.j.ranMesg, fake.i.j.ranMesg | i <- PeerSet, j <- PeerSet,
                                                                  ranMesg <- RangesMessages} }
                       )
-------------------------------------------------------------------------------------

---------------------------- VERIFIER PROCESS  ------------------------------------------
Verifier0 = []r:PeerSet @ ( send.VF.r.SqN.<nonceVF>
                            -> []l:PeerSet @ ( receive.r.VF.Cert.<nonceVF,l,r>
                                               -> if (l==r) then ( output.l.r -> unlock.VF.r -> STOP )
                                                  else ( send.VF.l.SqN.<nonceVF>
                                                         -> []ll:PeerSet @ (
```

```
                                                                    receive.l.VF.Cert.<nonceVF,ll,l>
                                                                    -> output.l.r -> unlock.VF.l
                                                                    -> unlock.VF.r -> STOP
                                                                    )
                                                    )
                                            )
                        )

Verifier = Verifier0 [[send.VF.i <- take.VF.i | i <- PeerSet]]
                     [[receive.i.VF <- fake.i.VF | i <- PeerSet]]
alpha_Verifier = Union( { {take.VF.i.nonMesg | i <- PeerSet, nonMesg <- NonceMessages},
                          {fake.i.VF.certMesg | i <- PeerSet, certMesg <- CertMessages},
                          {output.i.j | i <- PeerSet, j <- PeerSet} }
                      )
-------------------------------------------------------------------------------------------

----------------------------  IMPLEMENTAITON AND SPEC PROCESS  ----------------------------
networkSyncSet = inter(alpha_Adversary, alpha_Devices)
Network = Devices [|networkSyncSet|] Adversary

implSyncSet = inter(alpha_Adversary, alpha_Verifier)
Impl0 = Verifier [|union(implSyncSet, {|unlock|})|] Network
Impl = Impl0 \ {|take,fake,unlock|}

Spec(ps,pn) =  ( []i:union(ps,pn),j:union(ps,pn) @ join.i.j -> Spec(union(ps,{i}),diff(pn,{i})) )
            []( []i:union(ps,pn), j:union(ps,pn) @ leave.i.j -> Spec(diff(ps,{i}),union(pn,{i})) )
            []( card(ps)>0 & []i:ps @ ( let r = right(i,ps) within
                                                              output.i.r -> Spec(ps,pn) ) )

Specs = []i:PeerSet @ Spec({i}, diff(PeerSet,{i}))
assert Specs [T= Impl
-------------------------------------------------------------------------------------------

---------------------------------- AUXILILARY -----------------------------------------------
neighbor(Peer.0,S) = if (member(Peer.0,S) and member(Peer.1,S) and card(S)==2) then (Peer.1,Peer.1)
                     else if (member(Peer.0,S) and member(Peer.2,S) and card(S)==2)
                              then (Peer.2,Peer.2)
                          else if (card(S)==3) then (Peer.2,Peer.1)
                               else let {x}=S within (x,x)

neighbor(Peer.1,S) = if (member(Peer.0,S) and member(Peer.1,S) and card(S)==2) then (Peer.0,Peer.0)
                     else if (member(Peer.1,S) and member(Peer.2,S) and card(S)==2)
                             then (Peer.2,Peer.2)
                          else if (card(S)==3) then (Peer.0,Peer.2)
                               else let {x}=S within (x,x)

neighbor(Peer.2,S) = if (member(Peer.2,S) and member(Peer.1,S) and card(S)==2) then (Peer.1,Peer.1)
                     else if (member(Peer.0,S) and member(Peer.2,S) and card(S)==2)
                             then (Peer.0,Peer.0)
                          else if (card(S)==3) then (Peer.1,Peer.0)
                               else let {x}=S within (x,x)

left(i,P) = let (x,y)=neighbor(i,P) within x

right(i,P) = let(x,y)=neighbor(i,P) within y

mid(i,l,r) = left(i,{i,l,r}) == l and right(i,{i,l,r}) == r
```

# APPENDIX H

# THE PROOF FOR DTR2

**Notations** For any $i, l, r, k$, we use the following notations:

$$
\begin{array}{rcl}
\epsilon & = & \{|join, leave|\} \\
\Omega(i,k) & = & \{fake.x.i.Ran.\langle x,u,v\rangle \mid x \in \mathcal{P}^{\infty}, k \in (u,v]\} \\
\Phi(i,k) & = & \{take.i.x.Ran.\langle i,u,v\rangle \mid x \in \mathcal{P}^{\infty}, k \in (u,v]\} \\
\Omega(k) & = & \bigcup_{i \in \mathcal{P}^{\infty}} \Omega(i,k) \\
\Phi(k) & = & \bigcup_{i \in \mathcal{P}^{\infty}} \Phi(i,k) \\
Rt(i,j,l,r) & = & take.i.j.Ran.\langle i,l,r\rangle \\
Rf(i,j,l,r) & = & fake.i.j.Ran.\langle i,l,r\rangle \\
\Delta(i,l,r) & = & \{Rt(i,x,l,r) \mid x \in \mathcal{P}^{\infty}\} \\
(x,y] & = & \{i \mid inBetween(i,x,y)\} \cup \{y\}
\end{array}
$$

**Lemma 4.** *For any $i, l, r, k$, let $e \in \Omega(i,k)$, $e' \in \Phi(i,k)$ and $\delta \in \Delta(i,l,r)$. Let $tr \in traces(Impl)$, we then have:*

$$
\begin{array}{l}
\forall s, t \bullet tr = s^{\frown}\langle Rf(i,j,l,r)\rangle^{\frown}t \\
\qquad \Rightarrow \exists s_1, s_2 \bullet s = s_1{}^{\frown}\langle\delta\rangle^{\frown}s_2 \ \wedge \ s_2 \upharpoonright \Delta(i,l,r) \cup \{Rf(i,j,l,r)\} = \langle\rangle
\end{array}
\tag{H.0.1}
$$

$$
\begin{array}{l}
\forall s, t \bullet tr = s^{\frown}\langle e'\rangle^{\frown}t \\
\qquad \Rightarrow \ \exists s_1, s_2 \bullet s = s_1{}^{\frown}\langle e\rangle^{\frown}s_2 \ \wedge \ s_2 \upharpoonright (\,\Omega(i,k) \cup \Phi(i,k)\,) = \langle\rangle
\end{array}
\tag{H.0.2}
$$

*Proof.* The proof for this Lemma can be easily derived from the structure of the process $RelayRange(i,l,r)$ and $TTMN(i)$. □

**Lemma 5.** *For any $i,l$, let $tr \in traces(TTM(i,l))$. For any $k$, let $e,e' \in \Phi(i,k)$. Then the following holds:*

$$tr = s^\langle e\rangle^t^\langle e'\rangle^z \quad \Rightarrow \quad t \restriction \Omega(i,k) \neq \langle\rangle$$

*Proof.* Again, the proof can be derived directly from the structure of the process $TTM(i,l)$ for any $i,l$. □

**Lemma 6.** *For any $k,l,r$ such that $k \in (l,r]$ and any $i,j$, let $tr \in traces(Impl)$.*

1. *For any $s,t$ such that:*

$$tr = s^\langle Rt(i,j,l,r)\rangle^t \ \wedge \ t \restriction \{Rf(i,x,l,r) \mid x \in Peers\} = \langle\rangle$$

*Then we have:*

$$t \restriction \Phi(k) = \langle\rangle$$

2. *For any $s,t$ such that:*

$$tr = s^\langle Rf(j,i,l,r)\rangle^t \ \wedge \ t \restriction \Phi(i,k) = \langle\rangle$$

*Then we have:*

$$t \restriction \Phi(k) = \langle\rangle$$

*Proof.* We show the proof for the first result (the proof for the second result is similar). Without loss of generality, for any $x_n, y_n, l_n, r_n, i_n, j_n, u_n, v_n \in Peers$ such that $k \in (u,v] \cap (l,r]$, $Rt(i_m,j_m,u_m,v_m) \in \Phi(k)$ and $i_m \neq x_n$, consider a trace $tr \in traces(Impl)$. For any $s, t_1, t_2$ satisfying:

$$tr = s^\langle Rt(x_n,y_n,l_n,r_n)\rangle^t_1^\langle Rt(i_m,j_m,u_m,v_m)\rangle^t_2$$
$$\wedge \ t_1^t_2 \restriction \{Rf(x_n,y,l_n,r_n) \mid y \in Peers\} = \langle\rangle$$

We will show that such a trace does not exists.

Lemma 4 implies that there exists $s_0, s_1, .., s_{2n}$ such that:

$$s = s_0{}^\frown\langle Rt(b, y_0, l_0, r_0)\rangle{}^\frown s_1{}^\frown \ldots{}^\frown s_{2n-2}{}^\frown\langle Rt(x_{n-1}, y_{n-1}, l_{n-1}, r_{n-1})\rangle$$
$$^\frown s_{2n-1}{}^\frown\langle Rf(y_{n-1}, x_n, l_{n-1}, r_{n-1})\rangle{}^\frown s_{2n}$$
$$\wedge\ s_0 \restriction \Omega(b,k) = \langle\rangle \ \wedge\ k \in \bigcap_i (l_i, r_i]$$

Similarly, there exists $z_0, z_1, .., z_{2m}$ such that:

$$s{}^\frown\langle Rt(x_n, y_n, l_n, r_n)\rangle{}^\frown t_1$$
$$= z_0{}^\frown\langle Rt(b, j_0, u_0, v_0)\rangle{}^\frown z_1{}^\frown \ldots{}^\frown z_{2m-2}{}^\frown\langle Rt(i_{m-1}, j_{m-1}, u_{m-1}, v_{m-1})\rangle$$
$$^\frown z_{2m-1}{}^\frown\langle Rf(j_{m-1}, i_m, u_{m-1}, v_{m-1})\rangle{}^\frown z_{2m}$$
$$\wedge\ z_0 \restriction \Omega(b,k) = \langle\rangle \ \wedge\ k \in \bigcap_i (u_i, v_i]$$

Consider the following cases:

1. *If* $z_0 \neq s_0$. Lemma 5 and $s_0 \restriction \Omega(b,k)$, $z_0 \restriction \Omega(b,k)$ result in a contradiction.

2. *For any* $p < min(n,m)$, *assume that* $z_q = s_q$ *for all* $q < p$ *and* $z_p \neq s_p$. Denote $tr_q$ as the sub-sequence of $tr$ that ends with $s_q$. Consider:

   (a) $tr_{p-1}{}^\frown\langle take.x.y.\langle x,l,r\rangle\rangle{}^\frown s_p{}^\frown\langle fake.x.y'.\langle x,l,r\rangle\rangle$ and $tr_{p-1}{}^\frown\langle take.x.y.\langle x,l,r\rangle\rangle{}^\frown z_p{}^\frown\langle fake.x.y''.\langle x,l,r\rangle\rangle$. These two cannot both happen due to Eq.H.0.1.

   (b) $tr_{p-1}{}^\frown\langle fake.x.y.\langle x,l,r\rangle\rangle{}^\frown s_p{}^\frown\langle take.y.x'.\langle y,l',r'\rangle\rangle$ and $tr_{p-1}{}^\frown\langle fake.x.y.\langle x,l,r\rangle\rangle{}^\frown z_p{}^\frown\langle take.y.x''.\langle y,u,v\rangle\rangle$. First, as $k \in (l,r] \cap (u,v]$, Eq.H.0.2 implies $s_p \restriction \Omega(y,k) = z_p \restriction \Omega(y,k) = \langle\rangle$. The contradiction then arises because of Lemma 5 and the fact that $z_p \neq s_p$.

□

**Lemma 7.** *Let* $tr \in traces(Impl)$, *for any* $i, l, s, t$ *such that:*

$$tr = s{}^\frown\langle take.i.VF.Cert.\langle n_{vf}, l, i\rangle\rangle{}^\frown t$$

*Let* $P_s = \gamma(\{\}, s)$, *we have the following:*

*1.*

$$l = left(i, P_s \cup \{l\}) \tag{H.0.3}$$

2. *If* $t \upharpoonright \{unlock.VF.i\} = \langle\rangle$, *then we have*

$$\forall x \in (l,i) \bullet \ t \upharpoonright \{join.x.r \mid r \in Peers\} = \langle\rangle \tag{H.0.4}$$

*Proof.* In the following, we show the proof for Eq.H.0.3 (the proof for Eq.H.0.4 is similar).

When $tr \upharpoonright \epsilon = \langle\rangle$, the structure of the process $TTM(b,b)$ implies that Eq.H.0.3 holds true.

Consider $tr \upharpoonright \epsilon \neq \langle\rangle$. Assume that there exists $x \in (l,i), x_r, s_1, s_2$ such that:

$$s = s_1 \hat{} \langle join.x.x_r \rangle \hat{} s_2 \ \wedge \ s_2 \upharpoonright \{leave.x\} = \langle\rangle$$

To prove that Eq.H.0.3 is true, it is sufficient to show that such a trace above does not exists (because of the structure of $TTMN(i)$ and the fact that $i \in P_s$).

We show contradictions as follows. First, from the structure of $TTMN(i)$ and $RelayRanges$, it can be seen that there exists $y, i', u, v, z_0, z_1, z_2$ such that:

$$s = z_0 \hat{} \langle Rt(y,i',u,v) \rangle \hat{} z_1 \hat{} \langle Rf(y,i,u,v) \rangle \hat{} z_2 \ \wedge \ x \in (u,v] \ \wedge \ z_2 \upharpoonright \Phi(i,x) = \langle\rangle \tag{H.0.5}$$

Second, the structure of $TTMN(x)$ and $RelayRanges$ imply that there exists $a, d, x', w_0, w_1, w_2$ satisfying:

$$s_1 = w_0 \hat{} \langle Rt(a,x',d,x) \rangle \hat{} w_1 \hat{} \langle Rf(a,x,d,x) \rangle \hat{} w_2 \ \wedge \ w_2 \upharpoonright \Phi(x,x) = \langle\rangle \tag{H.0.6}$$

Since $i \neq x$, we consider the following cases:

1. *If* $s_1 \preceq z_0$ *($s_1$ is a prefix of $z_0$)*

   (a) *If* $s_2 \upharpoonright \Phi(x,x) = \langle\rangle$. Lemma 6-2 implies that $s_2 \upharpoonright \Phi(x) = \langle\rangle$, which is not true due to Eq.H.0.5.

   (b) *If* $s_2 \upharpoonright \Phi(x,x) \neq \langle\rangle$. Let $Rt(x,p,p_l,p_r) \in \Phi(x)$ and $s_2 = w_3 \hat{} \langle Rt(x,p,p_l,p_r) \rangle \hat{} w_4$ for any $p, p_l, p_r, w_3, w_4$.

      i. If $Rt(x,p,p_l,p_r)$ is in $z_1$. This is not true because of Lemma 6-1.

      ii. If $Rt(x,p,p_l,p_r)$ is in $z_2$. This is also not true, because of Lemma 6-2 implying that $z_2 \upharpoonright \Phi(x) = \langle\rangle$

      iii. If $Rt(x,p,p_l,p_r)$ is in $z_0$. It means there exists $w_{30}, w_{31}$ such that:

$$z_0 = s_1 \hat{} \langle join.x \rangle \hat{} w_{30} \hat{} \langle Rt(x,p,p_l,p_r) \rangle \hat{} w_{31}$$

Because of Lemma 6-1 and Eq.H.0.5, there exists $w_{310}, w_{311}, p'$ satisfying:

$$w_{31} = w_{310}{}^\wedge\langle Rf(x, p', p_l, p_r)\rangle{}^\wedge w_{311}$$

Lemma 6-2 then implies that $w_{311} \restriction \Phi(x, x) \neq \langle\rangle$. But it then follows from the structure of the process $TTMN(x)$ that $w_{311} \restriction \{leave.x\} \neq \langle\rangle$, which is a contradiction.

iv. If $Rt(x, p, p_l, p_r) = take.y.i'.\langle y, u, v\rangle$. It means $y = x$. Similar to the previous case, it follows that $z_2 \restriction \{leave.x\} \neq \langle\rangle$, which is a contradiction

2. *If $z_0 \prec s_1$ ($z_0$ is a strict prefix of $s_1$)*

   (a) First, $Rt(y, i', u, v) \neq Rt(a, x', d, x)$ as the opposite implies $i = x$, which is not true.

   (b) If $Rt(y, i', u, v)$ is in $w_0$. Lemma 6-1 implies that

   $$z_0{}^\wedge\langle Rt(y, i', u, v)\rangle{}^\wedge z_1 \ {}^\wedge\langle Rf(y, i, u, v)\rangle \prec w_0$$

   The contradiction arises because of $z_2 \restriction \Phi(x) = \langle\rangle$ and Eq.H.0.6

   (c) If $Rt(y, i', u, v)$ is in $w_1$ or $w_2$. This is not true due to Lemma 6-1 and $w_2 \restriction \Phi(x) = \langle\rangle$.

   (d) If $Rf(y, i, u, v)$ is in $w_0$. This is not true due to $z_2 \restriction \Phi(x) = \langle\rangle$.

   (e) If $Rf(y, i, u, v)$ is in $w_1$ or $w_2$. This case is the same as considering $Rt(y, i', u, v)$ being in $w_0, w_1$ or $w_2$ above. The result is a contradiction.

□

**Theorem 4.** *Then for any trace $tr$ of $Impl$, the following holds:*

$$\forall s, t, l, r \,.\, tr = s{}^\wedge\langle output.l.r\rangle{}^\wedge t \ \Rightarrow\ \{l, r\} \subseteq \gamma(\{\}, s) \ \wedge\ l = left(r, \gamma(\{\}, s))$$

*Proof.* Assume

$$tr = s{}^\wedge\langle output.l.r\rangle{}^\wedge t$$

for some $s$, $t$, $l$, $r$. Let $P_s = \gamma(\{\}, s)$. We need to show that:

$$\{l, r\} \subseteq P_s \ \wedge\ l = left(r, P_s) \tag{H.0.7}$$

1. If $l = r$. The structure of *VerifierProcess* implies that there exists $s_1, s_2$ such that:

$$s = s_1{}^\frown\langle take.l.VF.Cert.\langle n_{vf}, l, l\rangle\rangle{}^\frown s_2$$

Let $P_{s_1} = \gamma(\{\}, s_1)$, Lemma 7 implies that $l = left(l, P_{s_1} \cup \{l\})$ and $s_2 \upharpoonright \{join.x.x_r \mid x \in (l, l), x_r \in Peers\} = \langle\rangle$. In addition, $l \in P_{s_1}$. Therefore, Eq.H.0.7 holds.

2. If $l \neq r$. There exists $s_1, s_2, s_3, ll$ such that:

$$s = s_1{}^\frown\langle take.r.VF.Cert.\langle n_{vf}, l, r\rangle\rangle{}^\frown s_2{}^\frown\langle take.l.VF.Cert.\langle n_{vf}, ll, l\rangle\rangle{}^\frown s_3$$

Let $P_s = \gamma(\{\}, s)$, $P_{s_1} = \gamma(\{\}, s_1)$ and $P_{s_2} = \gamma(\{\}, s_1{}^\frown s_2)$. Lemma 7 implies:

(a) $r \in P_{s_1} \ \wedge \ l = left(r, P_{s_1} \cup \{l\}) \ \wedge \ s_2{}^\frown s_3 \upharpoonright \{join.x.x_r \mid x \in (l, r), x_r \in Peers\} = \langle\rangle$

(b) $l \in P_{s_2} \ \wedge \ ll = left(l, P_{s_s} \cup \{ll\}) \ \wedge \ s_3 \upharpoonright \{join.x.x_r \mid x \in (ll, l), x_r \in Peers\} = \langle\rangle$

In addition, $s_2{}^\frown s_3 \upharpoonright \{leave.l.l', leave.r.r' \mid l', r' \in Peers\} = \langle\rangle$ due to the structure of the process $TTMN(l)$ and $TTMN(r)$. Therefore, $\{l, r\} \subseteq P_s$ and $l = left(r, P_s)$, or Eq.H.0.7 holds.

□

# LIST OF REFERENCES


[1] ISO/IEC PAS DIS 11889: Information technology – Security techniques – Trusted platform module.

[2] Planetlab, an open platform for developing, deploying, and accessing planetary-scale services. `http://www.planet-lab.org/`.

[3] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In *10th international conference on Information and knowledge management*, pages 310–317, 2001.

[4] Karl Aberer and Zoran Despotovic. On reputation in game theory application on online settings. Working paper, 2004.

[5] Ross Anderson. Trusted computing faq tc/tcg/largrande/ngscb/longhorn/palladium. `http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html`.

[6] Christopher Avery, Paul Resnick, and Richard Zeckhauser. The market for evaluations. *American economic review*, 89(3):564–584, 1999.

[7] Rana Bakhshi and Dilian Gurov. Verification of peer-to-peer algorithms: A case study. *Electronic Notes in Theoretical Computer Science*, 181:35–47, 2007.

[8] Yannis Bakos and Chrysanthos Dellarocas. Cooperation without enforcement? a comparative analysis of litigation and online reputation as quality assurance mechanism. In *International Conference on Information System*, pages 127–42, 2002.

[9] Shane Balfe, Amit D. Lakhani, and Kenneth G. Paterson. Trusted computing: Providing security for peer-to-peer networks. In *International Conference on Peer-to-Peer Computing*, pages 117–124. IEEE Computer Society, 2005.

[10] John Benamati, Mark A. Serva, and mark A. Fuller. Are trust and distrust distinct constructs? and empirical study of the effects of trust and distrust among online banking users. In *39th Hawaii international conference on system sciences*, page 121b, 2006.

[11] Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a structured peer-to-peer overlay network: The static case. In *Global Computing*, pages 250–265, 2004.

[12] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anynymous attestation. In *11th ACM Conference on Computer and Communications Security*, pages 132–145, 2004.

[13] P. J. Broadfoot. *Data Independence in the Model Checking of Security Protocols*. PhD thesis, Oxford University, 2001.

[14] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings. Sosimple: A serverless, standards-based, p2p sip communication system. In *International Workshop on Advanced Archtectures and Algorithms for Internet Delivery and Applications*, pages 42–49, 2005.

[15] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *Internation workshop on Peer-to-Peer systesm (IPTPS)*, pages 80–87, 2003.

[16] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

[17] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE journal on selected areas in communications*, 20(8):100–110, 2002.

[18] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5:440–452, 1979.

[19] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-based attestation without a trusted third party. In *11th international conference on Information Security*, pages 31–46. Springer-Verlag, 2008.

[20] Alice Cheng and Eric Friedman. Sybilproof reputation mechanisms. In *ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 128–132, 2005.

[21] Alice Cheng and Eric Friedman. Manipulability of pagerank under sybil strategies. In *1st workshop of network systems*, pages 75–82, 2006.

[22] Bram Cohen. Bittorrent protocol specification v1.0. World Wide Web `http://wiki.theory.org/BitTorrentSpecification`, July 2007.

[23] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 33–44, 2003.

[24] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashhoek, and Ross Anderson. Sybil-resistant dht routing. In *10th European Symposium on Research in Computer Security (ESORICS)*, pages 305–318, 2005.

[25] Chrysanthos Dellarocas. The digitization of word of mouth: Promise and challenges of online feedback mechanisms. *Management Science*, 49(10):1407–1424, 2003.

[26] Chrysanthos Dellarocas. Efficiency and robustness of binary feedback mechanism in trading environments with moral hazard. Working paper, Janurary 2003.

[27] Chrysanthos Dellarocas. Reputation mechanism design in online trading environments with pure moral hazard. *Information systems research*, 16(2):209–230, 2005.

[28] Chrysanthos Dellarocas, Federico Dini, and Giancarlo Spagnolo. *Handbook of procurement*. Cambridge University Press, 2006.

[29] Tien Tuan Anh Dinh, Michael Lees, Georgios Theodoropoulos, and Rob Minson. Large scale distributed simulation of p2p networks. In *16th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2008)*, pages 499–507, Toulouse - France, Februray 2008. IEEE Computer Society.

[30] Tien Tuan Anh Dinh and Mark Ryan. Checking security property of p2p systems in csp. Technical Report CSR-10-07, School of Computer Science, University of Birmingham, 2010.

[31] Tien Tuan Anh Dinh and Mark Ryan. Secure hardware abstraction for distributed systems. Technical Report CSR-10-08, School of Computer Science, University of Birmingham, 2010.

[32] Tien Tuan Anh Dinh, Georgios Theodoropoulos, and Rob Minson. Evaluating large scale distributed simulation of P2P network. In *12th IEEE/ACM International Symposium on Distributed Simulation and Real-time application (DS-RT'08)*, pages 51–58, 2008.

[33] John R. Douceur. The sybil attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, 2002.

[34] David Eppstein and Joseph Yannkae Wang. A steady state model for graph power laws. In *2nd International Workshop on web dynamics*, May 2002.

[35] Ronald Fagin, Ravi Kumar, and D. Sivakuma. Comparing top k lists. In *14th annual ACM-SIAM symposium on discrete algorithms*, pages 28–36, 2003.

[36] Alois Ferscha and Satish K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical Report UMIACS-TR-94-100, University of Maryland at College Park, 1994.

[37] Formal System (Europe) Ltd. Fdr2 model checker tool. World Wide Web `http://www.fsel.com/software.html`.

[38] Lakshmi Ganesh and Ben Y. Zhao. Identity theft protection in structured overlays. *IEEE Workshop on Secure Network Protocols*, 0:49–54, 2005.

[39] G Gans, M Jarke, S Kethers, and G Lakemeyer. Modeling the impact of trust and distrust in agent networks. In *3rd International Bi-conference workshop on agent-oriented information systems*, 2001.

[40] Georgia Tech. The network simulator - ns2. `http://www.isi.edu/nsnam/ns/`.

[41] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim - a simulator for peer-to-peer protocols. `http://pdos.csail.mit.edu/p2psim/`.

[42] Gnutella Project. Gnutella specification. World Wide Web `http://rfc-gnutella.sourceforge.net/developer/testing/`, July 2007.

[43] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured p2p systems. In *International conference on computer communications (INFOCOM)*, 2004.

[44] Elizabeth Gray, Jean-Marc Seigneur, Yong Chen, and Christian Jensen. Trust propagation in small worlds. In *1st international conference on trust management*, pages 239–254, 2003.

[45] R. Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *13th international conference on World Wide Web*, pages 403–412, 2004.

[46] Russell Hardin. *Trust - Key Concepts*. Polity Press, 2006.

[47] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[48] Tony Hoare. Why ever csp? *Electronic notes in Theoretical Computer Science*, 162:209–215, 2006.

[49] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9, 2008.

[50] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Computing surveys*, 31(3):264–323, 1999.

[51] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[52] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International workshop on Peer-to-Peer systems (IPTPS)*, pages 98–107, 2003.

[53] Tobias J. Klein, Christian Lambertz, Giancarlo Spagnolo, and Konrad O. Stahl. Last minute feedback. manuscript, University of Mannheim and Stockholm school of economics, 2005.

[54] David M Kreps and Robert Wilson. Reputation and imperfect information. *Journal of Economic Theory*, 27(2):253–279, August 1982.

[55] Yoram Kulbak and Danny Bickson. The emule protocol specification. Technical report, Hebrew University of Jerusalem, 2005.

[56] Ranko S. Lazic. *A Semantic Study of Data-Independence with Applications to the Mechanical Verification of Concurrent Systems*. PhD thesis, Oxford University, 1997.

[57] Arnaud Legout, Guillaume Urvoy, and Pietro Michiardi. Rarest first and choke algorithms are enough. In *6th ACM SIGCOMM on Internet measurement*, pages 203–16, 2006.

[58] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the kazaa network. In *Workshop on Internet Application*, pages 112–20, 2003.

[59] Raph Levien. Attack resistant trust metric. Draft PhD thesis, `www.levien.com/thesis/compact.pdf`, 2004.

[60] Raph Levien and Alexander Aiken. Attack-resistant trust metrics for public key certification. In *7th USENIX Security symposium*, pages 229–242, 1998.

[61] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: small trusted hardware for large distributed systems. In *6th USENIX symposium on Networked systems design and implementation (NSDI'09)*, pages 1–14. USENIX Association, 2009.

[62] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *21st annual symposium on Principles of distributed computing (PODC)*, pages 233–242, 2002.

[63] Jinshan Liu and Valerie Issarny. An incentive compatible reputation mechanism for ubiquitous computing environments. *International Journal in Information Security*, 6(5):297–311, 2007.

[64] Niklas Luhmann. *Trust and Power*. John Wiley and Sons Inc, 1982.

[65] Ueli Maurer. Modelling a pulbic-key infrastructure. In *European Symposium on Research in Computer Security (ESORICS)*, pages 325–50, 1996.

[66] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *First International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[67] D.Harrison McKnight and Norman Chervany. While trust is cool and collected, distrust is fiery and frenzied: a model of distrust concept. In *Americas conference on information systems*, pages 35–59, 2001.

[68] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[69] Mininova. Mininova website. `http://www.mininova.org`.

[70] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing surveys*, 18(1):39–66, 1986.

[71] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *7th USENIX Security Symposium*, pages 217–28, 1998.

[72] Elinor Ostrom and James Walker. *Trust and Reciprocity, interdisciplinary lessons from experimental research*. Russell Sage Foundation, 2003.

[73] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[74] Josiane Xavier Parreira, Debora Donato, Sebastian Michel, and Gerhard Weikum. Efficient and decentralized pagerank approximation in a peer-to-peer web search network. In *32nd internation conference on very large databases*, pages 415–426, 2006.

[75] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, 1997.

[76] Richard Price, Tien Tuan Anh Dinh, and Georgios Theodoropoulos. Analysis of a self-organizing maintenence protocol under constant churn. In *International Symposium on Applications and the Internet*, pages 209–12, Turku, Finland, July 2008.

[77] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *2001 Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMMS'01)*, pages 161–172, 2001.

[78] Mark Relasity, Alberto Montersor, Gian Paolo Jesi, and Spyros Voulgaris. Peersim: A peer-to-peer simulator. `http://peersim.sourceforge.net/`.

[79] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. Technical Report CSD-03-1299, UC Berkeley, 2003.

[80] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999.

[81] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Operating Systems Review*, 35(5):188–201, 2001.

[82] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, 2001.

[83] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and anlysis of security protocols*. Addison Wesley, 2000.

[84] Luis F.G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limied objects using a tpm without a trusted os. In *1st ACM Workshop on Scalable Trusted Computing*, pages 27–42, 2006.

[85] Statistical Cybermetrics Research Group. Academic web link database project. `http://cybermetrics.wlv.ac.uk/database/`.

[86] Joe Stewart. Storm worm ddos attack. `http://www.secureworks.com/research/threats/storm-worm/`, February 2007.

[87] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, 2004.

[88] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *2001 ACM SIGCOMM Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[89] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *5th ACM SIGCOMM conference on Internet Measurement*, pages 49–62, 2005.

[90] Daniel Stutzbach, Shanyu Zhao, and Reza Rejaie. Characterizing files in the mordern gnutella network. *Multimedia Systems*, pages 35–50, March 2007.

[91] Paul Syverson. Onion routing for resistance to traffic analysis. *DARPA Information Survivability Conference and Exposition*, 2:108–110, 2003.

[92] Piotr Sztompka. *Trust: a sociological theory*. Cambridge University Press, 1999.

[93] The Pirate Bay. The pirate bay website. `http://thepiratebay.org`.

[94] Trust Let. Extended epinions dataset. `http://www.trustlet.org/wiki/Extended_Epinions_dataset`.

[95] Trusted Computing Group. TPM Specification version 1.2. Parts 1–3. `www.trustedcomputinggroup.org/specs/TPM/`, 2007.

[96] Trusted Computing Group. Press release. `www.trustedcomputinggroup.org/news/press/member_releases/WAVETCGPROMOTI%ONMW5_31_FINAL_.pdf`, 2008.

[97] Trusted Computing Group. TCG timeline. `www.trustedcomputinggroup.org/about/corporate_documents/`, 2008.

[98] Stijn van Dongen. Mcl - a cluster algorithm for graphs. `http://www.micans.org/mcl/`.

[99] Stijn Marinus van Dongen. *Graph clustering by flow simulation*. PhD thesis, University of Utrecht, 2000.

[100] Peng Wang, Nicholas Hopper, Ivan Osipkov, and Yongdae Kim. Myrmic: Secure and robust dht routing. Technical report, University of Minnesota, 2006.

[101] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440–42, 1998.

[102] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 266–275, 2003.

[103] Li Xiong and Ling Liu. Building trust in decentralized peer-to-peer electronic communities. In *International conference on electronic commerce research (ICECR-5)*, 2002.

[104] R. Yahalom, B. Klein, and Th. Beth. Trust relationships in secure systems-a distributed authentication perspective. In *IEEE Symposium on Security and Privacy*, pages 150–164, 1993.

[105] Hui Zhang, Ashish Goel, Ramesh Govindan, Kahn Mason, and Benjamin Van Roy. Making eigenvector-based reputation systems robust to collusion. In *Workshop on Algorithms and models for the Web graph*, pages 92–104, 2004.

[106] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, UC Berkeley, 2001.

[107] Runfang Zhou and Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions in Parallel and Distributed Systems*, 18(4):460–473, 2007.

[108] Cai Nicolas Ziegler and Georg Lausen. Propagation models for trust and distrust in social networks. *Information system frontiers*, pages 337–58, 2005.

# SUMMARY OF NOTATIONS

- $\mathcal{P}, \mathcal{P}^t$ - set of peers that currently in the network and at time $t$
- $\mathcal{B}$ - set of bootstrapping nodes which start the network
- $\mathcal{D}$ - set of data objects
- $\mathcal{I} = [0, 2^m)$ - the identifier space. $m$ is the security parameter whose typical value is 160.
- $root : \mathcal{D} \rightarrow \mathcal{P}$ - the function defining the root node of a key, given the set of nodes currently in the network
- $\oplus, \ominus$ - addition and substraction in *modulo* $2^m$
- $b$ - $2^b$ is the base on which IDs in Pastry are represented
- $l$ - size of the leafset
- $d$ - the number of dimension of the CAN space
- $\mathcal{F}$ - set of feeback
- $R_i$ - reputation of peer $i$
- $\mu_i$ - reputation metric or reputation function of $i$
- $\mathcal{T}, \mathcal{T}_{ab}$ - set of all transactions and of ones initiated by $a$
- $init, resp$ - returning initiator and responder of a transaction
- $Rt$ - returns the rating that one peer gives to another, with respect to a transaction
- $\mathcal{G}(V, E, W)$ - the trust graph
- $\mathcal{T}_{ij} = \{t \in \mathcal{T} \,|\, init(t) = i \ \wedge \ resp(t) = j\}$ - the set of transactions between i and j
- $E_i^+ = \{(j, i) \,|\, (j, i) \in E\}$ - set of edges to $i$
- $E_i^- = \{(i, j) \,|\, (i, j) \in E\}$ - set of edges from $i$
- $W_i^+ = \{W(e) \,|\, e \in E_i^+\}$ - set of ratings given to i

- $W_i^- = \{W(e) \,|\, e \in E_i^-\}$ - set of ratings given by i
- $T$ - transition matrix derived from $\mathcal{G}$
- $\epsilon$ - jumping factor, typical value is 0.15
- $CA$ - clustering algorithm returning a set of clusters $C$
- $CL : V \to C$ - maps a node to a cluster
- $dens : C \to \mathbb{R}$ - density function
- $dens2ep : \mathbb{R} \to (0, 1)$ - maps density to value of the jumping factor
- $rNEdges$ - ratio of negative edges, used in PNR
- concat - string concatenation operation
- $cid$ - counter ID
- $cd(x, y)$ - the clockwise distance from $y$ to $x$ in the ID ring
- $inBetween(z, x, y)$ - predicate returning true if going clockwise from $x$, one gets to $z$ before $y$.
- $route(k)$ - function returning the node that is the result of the routing protocols for the search key $k$. The function is assumed to return a random node.
- $getPredecessor(p_d)$ - returning a node representing the predecessor of $p_d$.
  The function is assumed to return a random node.
- $neighborVerification(p_l, p_r)$ - returns the result from checking whether $p_l$ is the immediate left neighbor of $p_r$.
- $destVerification(k, p_d)$ - returns the result from checking whether $p_d$ is the root node for $k$.
- $RootAuthenticity(RA)$ - property
- $NeighborAuthenticity(NA)$ - property
- $c_{p_d}$ - latest counter value at $p_d$
- $Cert_{p_d}$ - neighbor certificate of node $p_d$
- $\mathcal{S} : \mathcal{D} \to \mathbb{P}(\mathcal{P})$ - returning the set of sellers
- $v : \mathcal{P} \times \mathcal{D} \to \mathbb{R}^+$ - price function
- $\Delta : \mathcal{D} \to \mathbb{P}(\mathcal{P})$ - returning the set of listing nodes
- $\mathcal{W}^p$ - set of sale offers at peer $p$

- $\mathcal{W}_d^p$ - set of sale offers for $d$ stored at $p$
- $f, r$ - flat and variable rate of payment
- $publish(d), retrieve(d)$ - publish and retrieve offers for item $d$
- $tk \in TType \times \mathcal{I}$ - token in TTM
- $B-T$ and $R-T$ - bulk and range tokens respectively
- $TK$ - set of tokens of the same type.
- $S, s$ - set of TTM's states and a specific state.
- $[a, b]_{ttype}$ - range of tokens containing tokens whose IDs are from $a$ to $b$
- $nNodes$ - number of simulated nodes
- $ET$ - number of simulation time-steps
- $nLPs$ - number of LPs used in the simulation
- $qRate$ - the average query rate (query events follows a Poisson distribution)
- $cRate$ - churn rate (values of session time are expotentially distributed)
- $kal, mPeriod$ - time intervals after which keep-alive and maintenance messages are sent.

# INDEX