Tamper-resistant Peer-to-Peer Storage for File Integrity Checking

Alexander Zangerl
Dipl.-Ing. (TU Wien, Austria)

A thesis submitted for the degree of
Doctor of Philosophy at
Bond University
School of Information Technology

August 2006

“... oba es gibt kan Kompromiß,
zwischen ehrlich sein und link,
a wann’s no so afoch ausschaut,
und wann’s noch so üblich is ...”

— Wolfgang Ambros, 1975

Abstract

One of the activities of most successful intruders of a computer system is to modify data on the victim, either to hide his/her presence and to destroy the evidence of the break-in, or to subvert the system completely and make it accessible for further abuse without triggering alarms.

File integrity checking is one common method to mitigate the effects of successful intrusions by detecting the changes an intruder makes to files on a computer system. Historically file integrity checking has been implemented using tools that operate locally on a single system, which imposes quite some restrictions regarding maintenance and scalability. Recent improvements for large scale environments have introduced trusted central servers which provide secure fingerprint storage and logging facilities, but such centralism presents some new shortcomings.

This thesis describes an alternative, decentralised approach where peer-to-peer mechanisms are used to provide fingerprint storage for file integrity checking with more flexibility and scalability than offered by currently available systems. A research implementation has been developed to verify the approach as viable and practical, and experimental results obtained with that prototype are discussed.

Contents

1 Introduction **8**

1.1 Motivation 8

1.2 Current Practices 9

1.2.1 Introspective Operation 9

1.2.2 Fingerprint Database Storage 10

1.3 Problem Scope 10

1.4 Proposed Solution Outline 11

2 Related Work **13**

2.1 Intrusion Detection and Countermeasures 13

2.2 Distributed Consensus 15

2.3 Variants of the Consensus Problem 17

2.4 Distributed Services 18

2.5 Group Communication Systems 19

2.6 Practical Byzantine-Fault-Tolerant Services 21

2.7 Replicating Data and Quorum systems 22

2.8 Epidemics and Diffusion 23

2.9 Eventual Consistency 25

2.10 Distributed Storage 28

2.11 Secret Sharing 29

2.12 Trust Issues 31

2.13 Summary 33

3 Design Philosophy **34**

3.1 Introduction 34

3.2 Design Goals 34

3.3 Autonomy 35

3.4 Mistrust 35

3.5 Simplicity 37

3.6 Relaxing Consensus . . . 38
3.6.1 Versioning . . . 39
3.7 Minimal Cooperation and The “Good Citizen” Criteria . . . 40
3.7.1 Protocol and Format Conformance . . . 40
3.7.2 Communication Redundancy . . . 41
3.7.3 Verifiability . . . 41
3.8 Communication Fabric . . . 43
3.9 Document Scope . . . 43
3.10 Summary . . . 44

4 Design Details **45**
4.1 Environment and Principals . . . 45
4.2 Documents . . . 46
4.2.1 Namespace . . . 46
4.2.2 Versioning . . . 47
4.2.3 Document Lifecycle . . . 47
4.3 The Peer List . . . 49
4.4 The Policy Language . . . 50
4.5 Trust Levels and Signatures . . . 51
4.6 The Signature Tree . . . 52
4.6.1 Signing Once Only . . . 54
4.6.2 Multiple Signatures over Leaves . . . 56
4.6.3 Multiple Onion-shell Signatures . . . 56
4.6.4 Hierarchical Signature Tree . . . 57
4.7 Communications . . . 58
4.7.1 Low-level Protocol . . . 59
4.7.2 High-level Protocol . . . 59
4.7.3 Protocol Message Types . . . 60
4.8 Document Retrieval . . . 61
4.8.1 Handling Requests for Non-existent Documents . . . 62
4.9 Document Replication . . . 62
4.9.1 Initial Injection Phase . . . 63
4.9.2 Consolidation Phase . . . 64
4.10 Group Management . . . 65
4.10.1 Membership Control . . . 65
4.10.2 Administrator Capabilities . . . 66
4.10.3 Document Certification . . . 66
4.10.4 Adding and Removing Peers . . . 67
4.10.5 Alerts . . . 68
4.11 Summary . . . 68

5 Implementation **70**

5.1 Technological Considerations . . . 70

5.1.1 Event-driven Operation . . . 71

5.1.2 Code Reuse . . . 71

5.1.3 Modular Design . . . 72

5.2 Functional Hierarchy . . . 72

5.3 User Interface . . . 74

5.4 Practical Experiences . . . 75

6 Analysis **77**

6.1 Design Influences . . . 77

6.2 Threats and Countermeasures . . . 79

6.2.1 External Threats . . . 79

6.2.2 Malicious Administrators . . . 80

6.2.3 Local Threats . . . 81

6.2.4 Malicious Insertion of Documents . . . 81

6.2.5 Conflicting Documents . . . 82

6.2.6 Communication Misbehaviour . . . 84

6.3 Replication Analysis . . . 84

6.3.1 Simplified Model . . . 85

6.3.2 Applicability of the Simplified Model . . . 87

6.4 Empirical Cost Analysis . . . 89

6.4.1 Experimental Environment . . . 89

6.4.2 Measurements and Expectations . . . 90

6.4.3 Data Extraction and Plotting . . . 91

6.4.4 Strategy Evolution . . . 92

6.4.5 Naïve Flooding . . . 93

6.4.6 Fan-In . . . 97

6.4.7 Variable Degree of Parallelism . . . 99

6.4.8 Directed Flooding . . . 100

6.4.9 Biased Flooding . . . 104

6.4.10 Alternative Termination Criteria . . . 105

6.5 Summary . . . 107

7 Conclusion **109**

A Specifications and Source Code **111**

A.1 Policy Language Specification 111

A.2 Protocol and Message Specifications 112

A.2.1 Communication Overview 112

A.2.2 Communication Startup 112

A.2.3 Termination of Communications 113

A.2.4 Message Formats 113

A.2.5 Protocol Commands 113

A.2.6 `peerlist` Format 114

A.2.7 Signature Data Representation 114

A.3 Source Code Availability 115

Bibliography **117**

Chapter 1

Introduction

Intrusion detection systems (henceforth abbreviated to IDS) are generally accepted tools for successful defence in depth against malicious unauthorised actions on computer systems. Most IDS mechanisms involve comparison of actual activity against indications of known-good activity, and Host-based IDS perform this comparison on the actual computer that is subject to potential intrusions. This makes the IDS software itself and all data it uses for detection subject to tampering by a successful intruder.

One well-established component of Host-based IDS is file integrity checking (abbreviated to FIC): periodic comparison between the content of files on the computer against a previously saved set of untampered contents. This comparison provides a good indication of intrusions, because successful intruders commonly alter some system files to provide persistent access for future malicious activity.

However, generation, storage and handling of the reference data set presents substantial challenges for efficient deployment and successful usage of FIC applications. The problems caused by the introspective nature of FIC have been countered mainly by greater centralisation which introduces its own set of problems. Until now, effective deployment of FIC systems was an unsolved problem especially in large, decentralised environments.

1.1 Motivation

The initial inspiration for this research was provided by the following computer security incident witnessed by the author:

Imagine you (and some friends of yours) have your personal Internet servers in co-location facilities somewhere. You discover that an intruder has managed to break into your machine. What can you do (except panic)? What data has been compromised? What information on that computer is still trustworthy?

After cleaning up, you start thinking about preventing the problem (or at least mitigating the consequences) in the future. File integrity checking comes to mind immediately, but how do you store the baseline data safely? On your machine? On a CD? You have one server, as does each of your friends. Cooperation among this group of equals would be a good idea, but how exactly can this be achieved? You trust each of them *somewhat.* Somehow you would like to use the collective others to improve your local data integrity assurance, but Defence in Depth is required: no single compromised server should keep you from discerning between clean and tampered data. As you've been bitten once already, you *have* a reason for paranoia.

A similar problem was faced by the Debian Project, a large group of software developers that provides a free operating system distribution. In November 2003, an intruder managed to subvert a number of

Debian Project machines[1] including some of the project's software repository servers. After detection of the intrusion, the administrators had to ascertain the integrity of thousands of software packages manually using data comparison with separate, trustworthy mirrors. As Debian's machines are spread globally, physical access is infeasible for day-to-day operations, and deployment of a classic file integrity checker would not have helped much. Also, the project does not have any central machine that might serve as a trusted data repository – in fact, the most central systems were compromised by the intrusion. Administrators of Debian's systems form a very loose and non-hierarchical team of globally dispersed persons, which complicates matters further.

These incidents – and especially the lack of readily deployable safeguards against future occurrences thereof – demonstrated that implementing file integrity checking in a distributed environment is not trivial, especially when the environment is decentralised and without a single controlling entity. None of the currently available systems work well in the decentralised case as outlined in the following sections.

1.2 Current Practices

The basic technology of file integrity checking is simple, well established and documented[2]. But there are shortcomings in most available implementations[3, 4, 5], especially concerning scalability in large or decentralised environments.

The basic file system integrity checker initially stores cryptographic fingerprints of all "known good" files on a system in a local database. Later, files on the system are periodically checked against the database by computing the current fingerprint of the file and comparing it with the stored fingerprint. In case of a security breach this information can then be used to assess the damage with a granularity of single files.

The main threats are that the fingerprint database and the checking application themselves are vulnerable and both must be protected from malicious modifications to ensure trustworthy integrity assessments.

1.2.1 Introspective Operation

In general the checking application runs *introspectively*, i.e. on the computer system where the files are stored, scanning for changes hinting of an intrusion. This implies that there can be no ultimate guarantee that the checking application or its runtime environment are not sufficiently altered to prevent the detection of changes (cf. [6]). As long as the data to be verified is physically attached to a particular system, i.e. the storage is not networked, there is no workaround for the introspective nature of the application.

Protecting the application executable is the easier part. As the executable is not volatile, seldom changed or updated, the current practices – while not very scalable, cumbersome to administer or unsuitable for decentralised environments – can be considered sufficient. They include:

- making the application as much self-contained as possible,
- storing the application on read-only media,
- embedding self-verification measures like cryptographic signatures into the application[7, 8],
- hiding[1] the existence of the checking application[6, 3],
- or doing *very* disruptive integrity checks using a known good runtime environment (i.e. booting the system and running the application from trusted read-only media).

[1] Ironically this uses the very same technologies intruders use to disguise their back-doors or *rootkits*.

1.2.2 Fingerprint Database Storage

Safeguarding the database is the most complex task, mostly because on a well administered system file contents change on a regular basis – and thus the fingerprint database needs to be updated as well.

Simply allowing the database to be updated locally drastically weakens the integrity assurance provided, as a successful intruder could easily change the database to reflect the tampered state of data on the system.

Most current file integrity checking systems suggest

- to store the database on read-only media,
- to embed self-verification measures into the database,
- or to store it on a central server and make it available read-only via some network protocol.

Storing the database on read-only media obviously makes updates cumbersome at best, and infeasible for distributed environments: for machines which are administered remotely across continents, changing media physically is quite impractical. It is also counterproductive as security-minded administrators are encouraged to apply security patches frequently.

The notion of a pseudo-read-only database by storing the data on read-write media but making it immutable and verifiable with a cryptographic signature by an administrator, improves the handling of updates somewhat, but does not provide any guarantees regarding the availability of the database.

The use of a networked central server to serve the data to the individual machines via a network makes updates very efficient and simplifies administration greatly, but also introduces new limitations and security threats into the system:

- The communication between the client and the server has to be safeguarded to ensure privacy and integrity of data exchanges. This is of special importance if the implementation allows for direct updates of the database from client to server.
- The server is a single point of failure and needs to be made highly-available. Redundant backup servers need to be synchronised securely, which increases overall complexity dramatically.
- The server is a convenient single point of attack.
- The server has to be trusted absolutely by all clients. This implied hierarchical trust model is not a suitable setup for most decentralised environments[9].

1.3 Problem Scope

Our goal was to employ cooperative techniques to extend file integrity checking systems beyond a merely local or organisational scope. The main new requirement would be a means of reliable distributed data storage, with the additional characteristics of not requiring overly much implicit trust: the potential corruption of a small number of involved parties must not corrupt the whole system, and problems should be easily detectable by individual parties.

Applying Peer-to-Peer (P2P) technologies seemed appropriate, but making do without the guarantees that a tightly controlled local installation provides brings up a lot of new questions ranging far beyond simple algorithmic issues.

The most central questions we would like to explore are as follows:

1. Is it possible to design a distributed mechanism whereby an individual computer system can determine whether its stored data is untampered? Can such a distributed detection mechanism be extended to also support reconstruction of damaged data?
2. Can such a mechanism be made robust in the presence of malicious failures? How many faulty parties can be tolerated? Are there attack vectors related to the distributed nature of said mechanism that ultimately cannot be prevented?
3. Is strict consistency across all systems required at all times? Or can such a mechanism deal with conflicting/inconsistent information present at some parties, and to what extent?
4. What level of trust can or must an individual computer system place in itself? What level of trust can or must the individual computer system place in the other systems?
5. How much autonomy can be given to an individual party? How much altruistic cooperation and trust between parties is required for this mechanism to work?
6. How can such a distributed mechanism be deployed and maintained efficiently in a large environment?
7. What is the computational and communication cost of such a mechanism? How can data transfers be implemented in the most economic manner? Are there group size restrictions, and if, of what kind?

In the remainder of this thesis we present our answers to these fundamental questions. To this end, we have designed and implemented mechanisms and software that offer replicated storage of documents among a group of independent peers, in which no single component of the system is to be trusted a priori and even malicious failure of a limited number of components can be survived.

1.4 Proposed Solution Outline

Our solution provides means for ubiquitous implementation of file integrity checking with minimal infrastructure requirements. It consists of a communication protocol and an algorithmic specification of a distributed service, named `Peeranoia`, that is provided cooperatively between independent peers.

Instead of using local data or a central server as repository for file fingerprint information, a group of independent, mostly untrusted computer systems running the `Peeranoia` application provide data integrity verification services among all the group members. This service provides better integrity assurance for crucial data to a peer, as the probability of all involved peers being compromised at the same time is exceedingly low.

Parties in our environment are widely spread geographically and communicate over unreliable public networks. All data is replicated, stored redundantly and also certified independently by all peers.

The data storage mechanism aims for eventual consistency (i.e. data reaches all peers with high probability), which is sufficient for a practical system as we will show. Overall, our design goals are in order of priority: secrecy and integrity of stored data, simplicity, availability of data, and robustness of the system as a whole.

Peers are independent and autonomous, and are not trusted: any peer decides for itself how far data has to be integrity-certified until the peer is satisfied and willing to use it. This decision and others related to the replication operation are made locally based on independently verifiable information in the form of cryptographic signatures on data.

The network connecting the peers is assumed to be a public network like the Internet, also untrusted and without throughput or reliability guarantees. Communication between peers is safeguarded from tampering or eavesdropping by using symmetric cryptography. We show that the overall `Peeranoia` service is robust and attack-resistant, as long as there is only a limited number of corrupt peers or administrators within a group.

Integrity certification of data is provided by all peers as part of replication procedures and additionally, human administrator signatures can be used to further vouch for validity of specific data items. Essential to practical deployment, `Peeranoia` provides data update mechanisms to peers. This is combined with fine-grained control over what level of integrity and validity assurance a particular datum has to reach until it can be used as basis for file integrity checking decisions.

In the following chapter we examine prior efforts in related areas.

Chapter 2

Related Work

2.1 Intrusion Detection and Countermeasures

The basic mechanisms of file integrity checking[2] are all similar, using a previously established set of baseline file fingerprints in periodic comparisons with current file contents. There are various methods for the generation of file fingerprints, but this is only a superficial difference. Verifying file integrity is an accepted component of IDS and is not subject to any significant research, as far as its basic functionality is concerned.

However, all currently available FIC systems[3, 4, 5] only offer detection of trouble retrospectively as they detect traces of an intrusion rather then the actual event. Network-based intrusion detection mostly works real-time, but comes at the cost of a very much worse rate of false positives and higher computational requirements.

Some interesting improvements for this retrospective nature have been suggested during the last decade and are outlined as follows.

The ITCC project[10] is a system for developing *intrusion-tolerant* applications using threshold cryptography, as a measure complementary to classic intrusion detection. The basic goal is similar to our setting: compromise of a few system components should not compromise sensitive security data. The ITCC system's main design principles are sharing an application's private key among a number of servers and never locating keys in their entirety at in a single location. The first is a straight-forward example of secret sharing (discussed further in Section 2.10); the latter requires threshold cryptography so that a shared key can be used without reconstructing it in any single location.

As security information is never stored in a single location, there is no single point of attack; erasure coding and threshold cryptography provide probabilistic guarantees for detection of corrupt share servers, but arbitrary Byzantine failures are not covered fully. The ITCC has been implemented and tested with some example applications, namely an SSL certification authority and a secure webserver with a private key being shared out. Analytical results showed a moderate cost to the mechanism, but on the downside the ITCC system works for RSA private keys only due to the different sharing and threshold characteristics of asymmetric encryption systems: for example, they noted that RSA keys are easy to share but hard to generate distributively whereas for DSA the opposite is true. For `Peeranoia`, the ITCC system is not directly applicable: we wanted to improve reliability by replicating whole data sets rather than safeguarding secrets by sharing only specific keys. Also, the ITCC system needs to be linked to a particular application and as such is not very generic.

A more generic approach to detecting intrusions early was suggested by Ko et al. in [11]. They show the feasibility of in-kernel intrusion detection using software wrappers to detect and prevent intrusive behaviour. In their nomenclature, an intrusion detection software wrapper is a state machine that is

bound dynamically to a program and gets activated when system calls are invoked. A software wrapper would observe the system state at activation time and look for anomalous events or signatures of known intrusion vectors and block or modify the system call if deemed problematic. Wrappers can be viewed as autonomous agents residing in the operating system kernel, and as such have the advantage of being close to where things happen. However, there are the usual disadvantages listed as well: no portability, dubious robustness by increasing the effect of problems as the core of the operating system is affected, and one element that we think the authors of said paper underrepresented: added complexity in an extremely sensitive area.

Ko et al. also suggest a dynamic mechanism to add and control these in-kernel wrappers, yet at the same time claim that it is harder to tamper with the overall intrusion detection system as kernel protections would have to be surmounted. We believe these two are mutually exclusive: either the kernel is sacrosanct while in operation (thus offering higher tamper resistance), or the in-kernel IDS is tunable (thus allowing the intruder to switch it off). We believe that this all-in-one approach is overly heavyweight and fraught with a danger of providing a false sense of security; other systems like SElinux[12] properly treat any kernel security configuration as an irrevocable operation to be executed at system start-up.

Some researchers have tried to apply immunological principles to intrusion (or rather *change*) detection; Forrest et al. suggest in [13] the construction of an artificial immune system that functions along the lines of real-world T-cells: string-match detectors are randomly generated for some known good data and applied continually; probabilistic detection of *non-self* data is the result. The core idea looks a bit similar to ITCC's agent concept, but with a drastically simplified detection engine. The authors note that the major limitation so far is the computational cost of producing the initial detector set and acknowledge that their idea is not in a practically implementable state yet.

Another possible more light-weight approach towards intrusion tolerance would be to execute only known good programs. Researchers have proposed and implemented a number of signature verification schemes, with varying degrees of success. Microsoft's AuthentiCode is one example with very limited usefulness as signature verification is not done by the operating system core but rather by a user-land application of doubtful trustworthiness [14]: Internet Explorer.

Elfsign[15], a tool for generation and verification of digital signatures for the ELF executable file format used on many if not most UNIX platforms, provides a similar level of assurance as AuthentiCode; like AuthentiCode it works within the X.509 certificate framework. The ELF format is used for stand-alone executables as well as shared libraries, and supports extra sections in the file meta-data header.

At this time, the most comprehensive framework for verifying executables by the operating system consists of the tools BSign and DigSig[16]. BSign is similar to Elfsign, except that it operates within the PGP domain and uses GnuPG[17] to embed RSA signatures into an ELF executable file. Signature verification is handled by DigSig, a kernel module for the Linux kernel which will abort execution of files whose signature is missing or does not check out. In contrast to AuthentiCode, DigSig is meant as a tool for system administrators rather than a mechanism for software vendors to assert the provenance of data. Due to its heavy caching of signatures, DigSig imposes very little runtime overhead: benchmarks on current mainstream hardware show a cost of a few nanoseconds per byte of executable file, which is only incurred on first loading. Based on these figures, the claim of no performance degradation visible to the end user seems justified.

DigSig's assurances are based on a few assumptions, namely that the kernel itself and the signer's private key are not compromised, and that the superuser account is not breached. The latter would allow trojaned kernel modules to be loaded, which could disable DigSig altogether. The authors note that deploying DigSig in combination with SElinux can remedy the short-comings of DigSig being a loadable module; ideally, however, DigSig or a derivate would become an integral non-removable part of the Linux kernel. It should be noted that none of the systems for verifying executable files can protect from many if not most intrusions themselves: most attack vectors work by modifying the execution flow of running programs, at a point in time long past the signature verification.

Signature verification can only work *after* an intruder is successful and leaves Trojan horse programs

behind. Besides the actual operating system kernel (and its modules), the most interesting class of Trojan programs would be language interpreters or compilers as these are fundamentally trusted elements[18] of computer systems. Some aspects of this trust relationship and relevant mechanisms are discussed in Section 2.12.

In conclusion we note that most practical recent developments in the area of host-based intrusion detection involves improved safeguards against tampering for the actual IDS application and the operating system it runs on. The basic FIC mechanisms seem adequate for detection of corrupt file data – if the integrity of the baseline data can be assured.

2.2 Distributed Consensus

Our proposed system uses data replication among multiple parties to provide better integrity assurance for FIC baseline data. As the involved parties are independent, the effects of faults happening during the data exchange must be considered. Obviously we want to assure that all parties receive, store and operate on the same data.

At the core of this issue lies the problem of distributed consensus, also called the "Byzantine Generals Problem" (BG), which was presented for the first time in 1979 by Lamport, Pease and Shostak in [19] and formalised further in [20]. The problem setting is that of independent parties which try to agree on a common datum that is provided by an originating party, while using two-party messages in a synchronous and fail-safe communication network.

The central aspect of the problem is that faults in a distributed environment are not necessarily limited to benign events (like non-availability of processors or communication channels) but could be arbitrarily malicious: a faulty party can *lie*, alter messages that it receives or send out fabricated information to confuse others.

The problem was discovered during the design of a fault-tolerant computer system, where the initial expectation was that simple majority voting would suffice for consensus. The three main design areas affected were identified as clock synchronisation, consolidation of sensor input and agreement on diagnostic test results. It became obvious that the problem is a fundamental one in systems with decentralised control. So-called "Byzantine failure", components behaving arbitrarily malicious, is the most generic fault classification. While various fault type classes are known in the respective literature (cf. [21]) for which providing fault-tolerance is easier, assuming only a particular fault type for a design risks the catastrophic failure of the design if the fault assumptions are not met. For this reason maximally fault-tolerance systems must take Byzantine failures into account.

Lamport et al. define the sought system property, *interactive consistency* as follows:

- All correct parties agree on the same datum.
- If the originator is correct, then all correct parties will agree on the originator's datum.

They provide proof of the problem being solvable if and only if the number of faulty parties is strictly less than a third of the overall number of parties. Solutions for the BG problem are commonly called *Byzantine Agreement protocols* or *reliable broadcast protocols*.

The mechanism for agreement requires successive rounds of information exchanges, where highly redundant messages of the form "party x told me datum y" are transmitted between parties until all parties decide to terminate the exchanges. The originator sends its value to all parties in the first round, followed by a number of rounds in which every party reports to every other node what they received in the previous round, and terminating with a decision on the agreed value after a sufficient number of rounds.

[20] also provides an important result for a slightly weaker adversary. If messages are unforgeably signed, then a corrupt party can refuse to pass on information or fabricate its own datum as originator but it

can not lie or modify data received by others without detection. In this case, the problem is shown to be solvable for arbitrary numbers of faulty parties. The agreement algorithm changes only minimally so that the messages include a sequence of signatures wrapping the originally received datum d like this: $\{\{\{d\}_{p0}\}_{p1}\}_{p2}$, with $\{x\}_j$ denoting a signature over datum x constructed by party j. Note that any party signs and distributes only correctly signed messages. This class of agreement algorithms is usually called "authenticated" or "signed".

Both solution algorithms require up to $m+1$ messaging rounds with m being the number of faulty parties, which Fisher and Lynch[22] have shown to be the minimum amount of rounds. Each party will have to wait for messages from the originator and from up to m other parties to be received until consensus is reached.

The crucial aspects of the BG problem for practical applications relate to synchrony requirements. The messaging network is always assumed to be reliable and fully synchronous, with the ability to detect the absence of an expected message (to counter parties simply not sending messages). In contrast, real communication systems can fail to deliver or delay messages, which has to be taken into account during the consensus building. Communication failure is treated as a faulty party and detecting the absence of an expected message would generally be implemented by using a time-out: this in turn requires known upper bounds for message transmission delays and clocks at sender and recipient to be synchronised to within some maximum error. While the former is straightforward to provide, the latter problem of clock synchronisation is as difficult as the BG problem itself: physical clocks as commonly available do not run perfectly and re-synchronisation would be required periodically. But during those re-synchronisation exchanges, a faulty party could very well maliciously broadcast fabricated timing values to prevent successful agreement on a common clock.

The communication network is also assumed to be authenticated, i.e. every message indicates its sender unforgeably, and originally Lamport et al. envisioned a mesh of fixed lines between parties to be used rather than a message switching network. Applying these assumptions to public wide-area networks would suggest the use of cryptography to identify sender and verify message content at the same time. Cryptographic signatures do not, however, *strictly* guarantee the intactness of data or make forgery impossible: instead they allow us to set the probability of failure at an arbitrarily small value which suffices for practical purposes.

Making do without synchronised clocks would of course be a major improvement for practical deployability, but Fischer et al. [23] proved in 1983 that distributed consensus is impossible for a fully asynchronous environment. Every agreement protocol has the possibility of nontermination, even if there is only a single unannounced failure[1]. This means that without further assumptions about the computing environment and network, no strict guarantee of full consensus is possible. Fischer's system model is quite generic and only assumes a reliable communication system with messages being delivered correctly and exactly once. Parties have no synchronised clocks and no assumptions about the delay of message transmission are made; these last two aspects are the crucial elements for the proof that a "window of vulnerability" exists during which the delay or inaccessibility of just one party will prevent overall consensus indefinitely.

This very fundamental negative result suggested two subsequent trends in research of the consensus problem: accepting synchrony or relaxing the problem scope to probabilistic agreement. The consensus problem has been studied extensively over the last three decades, with an emphasis towards synchrony and strict agreement in the earlier stages; more recently, improvements in the field of asymmetric cryptography and the rise of peer-to-peer technology have changed that bias towards probabilistic agreement again.

[1] The fault does not even have to be malicious; a simple processor halt is sufficient.

2.3 Variants of the Consensus Problem

Over time many approaches to the consensus problem have been studied, emphasising different goals and assuming different environments: pretty much much every single aspect of the problem has been questioned, with varying results.

For example, consider the dichotomy between authenticated and non-authenticated agreement algorithms: authenticated protocols can tolerate a larger number of faulty parties and are more message-efficient but require signature generation for all messages which is computationally expensive. Borcherding shows in [24] how an interesting blending of this distinction can improve matters: In his example protocols, authentication is used only in certain rounds. Results suggest that if signature generation is the dominant cost factor, a mixed protocol with a low number of authenticated rounds provides a good trade-off between fault-tolerance and cost.

Another important area of improvement is the termination of the agreement protocol which especially affects the case of a smaller number of faults than expected. While Fischer and Lynch proved in [22] that $m+1$ rounds are required to deal with m faulty parties, the less pessimistic case of $f < m$ faulty parties suggests adjusting the agreement protocol to leverage redundancies. To that goal, "Early Stopping" algorithms for Byzantine agreement have been proposed. Krings and Feyer[25] provide a taxonomy and discussion of early stopping and give an algorithmic definition of one non-authenticated algorithm (directly derived from Lamport's algorithm) which is optimal in terms of required rounds.

Early stopping requires one minor adjustment to the definition of Byzantine agreement: the original specification required all correct parties to stop the protocol in the same round. This so-called "immediate" agreement conflicts with the notion of early stopping, therefore this requirement was relaxed to allow correct parties to stop in arbitrary rounds. For such "eventual" agreement with early stopping, a lower bound of $min(f+1, m+1)$ rounds has been established.

The goal of stopping short of the full $m+1$ rounds is reached by observing the information gathered in the past rounds, to infer the agreement value early (if possible). The received information includes the value and a history of parties the message has travelled through. This can be visualised in a "EIG" or *exponential information gathering*[2] tree which can then be used in a special voting process to check whether it already reveals the true agreement value in any given round: if so, the detecting party will stop propagation early in that round.

The interesting aspects for our work include the locality of control in the propagation mechanism that early stopping implies, as well as the representation of received information in a historical tree, on which we will focus further in Section 4.5.

Another interesting modification of the underlying problem is suggested by Goldwasser and Lindell[26]. If one is willing to relax the definition of distributed consensus to include the possibility of correct parties aborting, then deterministic protocols with low round complexity and without the requirement of a broadcast channel exist. After a short classification of the state of the art regarding distributed consensus, they present a rationale and example classes of protocols for their scenario: a correct party can either agree on the correct value that is being distributed or it can abort the agreement protocol altogether without a result value. It is stated that this relaxation still allows for meaningful secure distributed computation, and this recent work certainly suggests an interesting direction for further study as well as providing an introduction of how point-to-point communications relate to the consensus problem.

Finally, it is possible to leave the realm of deterministic agreement protocols and endeavour to solve asynchronous agreement using probabilistic algorithms. Especially in a setting like the Internet, reliable synchronisation of parties is impossible to guarantee, thus the asynchronous scenario has to be accepted. Cachin[27] notes that any protocol which requires cryptographic means for its operation inherently includes a non-zero probability of failure (of the underlying cryptographic mechanism to provide

[2] Lamport's algorithm is also known as an EIG algorithm.

the assumed property), and he argues that this is sufficient cause to abandon deterministic protocols in favour of randomised agreement protocols. Example protocols in [28] make extensive use of threshold cryptography and coin tossing protocols and have properties for which there are only heuristic justifications.

2.4 Distributed Services

As pointed out previously, the design of a fault-tolerant computer system was the original trigger for research into the consensus problem, and it appears that most related work during the 80's strongly centred around the resultant distributed state-machine approach which is very precise and powerful. During a survey of some literature on this topic, it appeared to us that the intended application domain would allow for a vastly simpler approach, namely replicating data instead of operations – a notion that numerous other researchers have pursued for similar reasons. These data replication mechanisms are covered in Chapter 2.7.

Returning to state-machine replication, the initial work studied was Schneider's standard paper[29] which describes the state-machine paradigm to provide fault-tolerant distributed services and also presents a short taxonomy on practical systems that use state-machine replication. Schneider's work covers services that use a client-server architecture and suggests that fault tolerance for such a service is easiest to achieve by extending the service infrastructure from one central server to a group of separate replicas, each of which can provide the service. In a nutshell, providing a fault-tolerant distributed service requires

- replicated service state machines, and
- replica coordination.

Replica coordination requires that all replicas receive and process the same sequence of requests, which can be decomposed into *agreement* (all correct replicas receive all requests) and *ordering* (all correct replicas use the same relative order for the requests processed). Provided proper replica coordination and all replicas starting in the same initial state, the distributed service will be fault-tolerant up to a (configurable) number of faults.

The agreement aspect of coordination is related to the Byzantine Generals problem, which covers only a single value agreement. Ordering is obviously an issue for services where individual requests do not commute. Most types of distributed services fall under this category: requests rarely are idempotent and causally independent, therefore replicated execution will only work if all the replicas handle the requests in the very same order. Request ordering requires the assignment of unique identifiers to requests and that replicas process requests in the correct sequence according to a total ordering relation of the identifiers.

This in turn means that a replica needs to determine whether a request is *stable* and ready for execution: Schneider defines request stability at a processor as the time when no correct request with a lower identifier value can be delivered to this processor. Schneider then presents some example implementations of a request ordering, each of which consist of a method for generating said unique identifier and an associated stability test. The examples discussed use logical clocks, approximately synchronised real-time clocks or replica-generated ordering identifiers.

The concept of logical clocks in distributed multiprocess systems is due to Lamport[30], where it is shown that the mapping of events to local integers in order of occurrence is an irreflexive partial ordering which can be extended to a total ordering. Logical clocks on single systems can be simple local counters, whereas the complexity of operation in a distributed system stems from interaction between systems: communication messages. Lamport states that if messages carry the timestamp of the sender, two simple implementation rules define logical clocks: a processing party must advance its clock counter between events, and upon reception of a message and its timestamp, the local counter must be advanced beyond the maximum of the current counter value and the received timestamp. To produce a total ordering, ties

between parties must be broken but any arbitrary total ordering of parties is sufficient. A simple way of achieving this is to have every party construct its logical clock value by appending the aforementioned counter with a fixed-length bit string that uniquely identifies the party.

Schneider expands on the consequences for a request ordering if Byzantine failures are possible: Devising a stability test for logical clocks does not make sense in the asynchronous case due to Fischer's impossibility result[23]. However, if one adds the reasonable restriction that for non-faulty parties the relative processor speeds and network delays are bounded, it is shown that this suffices to generate approximately synchronised real-time clocks. This restriction does not seem excessive for real-world application.

The final contribution of Schneider's work is a discussion of the long-term behaviour of a distributed system that suffers faults: faulty parties need to be excluded from the overall system, failures are remedied and repaired parties need to be reintegrated gracefully. Schneider suggests various mechanisms to manage the overall system's configuration and discusses the effects of a changing processor population on request ordering and processing.

Schneider's work clearly sets the ground rules for survivable distributed services, and state machine replication is the basis for many established systems as outlined in the following sections. We note, however, that most of the complexity stems from the problem of non-commuting and causally dependent requests to be handled. As our system replicates only independent documents and has no operations of finer granularity than complete documents, full and strict state machine replication is not required in our case: we have therefore used data replication mechanisms, which are covered in Section 2.7.

2.5 Group Communication Systems

As survivability of faults was becoming more and more of a requirement due to the advent of massively distributed computing systems, the goal of developing practical and efficient algorithms for distributed agreement became more and more pressing. Various advanced secure communication mechanisms were developed to meet global security requirements for secure distributed computing, mostly based on state-machine replication. In the following sections we will discuss some notable examples of work in this area.

State machine replication requires a reliable group multicast mechanism for distribution of messages between parties which also delivers messages totally ordered. In [31], Reiter presents some example protocols for what he calls "reliable and atomic group multicast" and describes their role within the framework of the "Rampart" toolkit for building high-integrity distributed services. This work of 1994 was the first successful effort to demonstrate feasibility of reliable and ordered multicast protocols for building distributed services in an asynchronous environment in the presence of malicious faults. But this was achieved at great cost: it is stated that tolerating malicious adversaries raises the cost of the protocols (for both latency and throughput) by an order of magnitude over protocols dealing with benign failures only. Reiter circumvents the asynchronous impossibility result by allowing the correct parties to remove unresponsive parties from the group in order to make progress during a multicast operation: communication delay or failure is deemed a fault of the unresponsive party. Reiter acknowledges that the risk of removing unresponsive but otherwise correct members wrongly does exist and that such wrong expulsion could lead to inconsistencies becoming visible to clients of the service.

In Rampart's environment parties are connected pair-wise by first-in-first-out communication channels which are authenticated and integrity-protected (provided by mechanisms external to Rampart). Parties each have a private key and are known to each other by their public keys which are used for signing messages. The system operates on *group views*, which are sequences of group membership lists provided by a secure group membership protocol to reflect the currently known fault status. Malicious activity of up to 1/3 of the members in each view is tolerated. The atomic multicast protocol is built on top of a reliable multicast protocol (which is basically a signed Byzantine agreement protocol), with the atomicity and total ordering property provided by designating a *sequencer* in every view which determines the order

of requests to be executed. Reiter closes his paper with an analysis of Rampart's performance, noting the need to optimise for the common case of no faults being present and suggesting some optimisations like temporary short encryption keys and combining multiple data elements into a single message.

Recently, Cachin[27] has proposed another architecture for asynchronous state-machine replication, in which the impossibility result for distributed consensus is circumvented using randomisation. The system is not view-based (whereby progress is made by removing corrupt elements) but requires a static server population and relies on outside means of recovery after corruption has occurred. Also, the threshold cryptography techniques used require an honest dealer for setup, and all parties need to exchange information (initially and to a slightly lessened extent during latter rounds), but on the other hand the protocol provides deadlock-freeness. This paper also provides a broad if slightly biased overview of other failure-tolerant systems developed over the last two decades.

Cachin suggests a digital distributed notary service as an example application; we note that there is some similarity between the goals of a notary and our system which notarises *and also stores* copies of documents. However, the Cachin infrastructure is not employed in `Peeranoia` because it is heavy-weight and requires lots of point-to-point communication to achieve strict sequencing of operations – a property which is not required in our scenario and whose absence allows us to reduce the communication overhead incurred.

Other well developed secure group communication systems include SecureRing, which was presented by Kihlstrom et al. [32] in 1998. This system imposes a distinctly different logical communication structure on the group: a Token Ring, where a wandering token is used to control reliable and ordered delivery of messages among the ring members. SecureRing's message delivery mechanism has a message complexity of $O(n^2)$ in a group of n parties, and requires limited synchrony for correctness. SecureRing uses public key cryptography and digital signatures for message authenticity checks, which are cited to be the main factor of the high computational cost of secure group communication systems. SecureRing introduces some useful optimisations, including the capability of multi-message signatures: messages are not authenticated in real-time; instead, a number of message digests can be covered by a single signature in a subsequent digitally signed token. This was implemented to improve message latency and throughput, but apparently did not meet with full success, as systems like Castro and Liskov's are faster by an order of magnitude [33].

In contrast to many other group communication systems, SecureRing allows dynamic group membership, but does assume out-of-band availability of party's public keys. Due to the ring structure it is necessary that the membership protocol detects failures and reconfigures the group membership to exclude faulty parties before a distributed computation can make progress. The use of unreliable fault detectors is cited as a mechanism to circumvent[34] the impossibility result[23] for asynchronous environments. It results in consensus being achievable, provided that there are less than $n/3$ faulty parties. As unreliable detectors can misclassify parties as faulty and thus have them excluded from the group, there are concerns that an intruder could keep SecureRing from operating correctly by impeding communications until sufficient replicas have been classified as faulty.

The SecureRing system is geared towards a local-area network, as it uses multicast message exchanges as the basic communication primitive and requires a network that never exhibits partitions. It also does not address message privacy, which severely limits its use for wide-area networks. As SecureRing is geared towards state-machine replication, it is mentioned as an example related mechanism more than a direct inspiration for `Peeranoia`. We also note that the delaying of signatures for the sake of throughput optimisation is not exactly in line with our goals, nor are automatic membership updates appropriate in our case where identity is coupled with some degree of trust.

A team of researchers around Agarwal developed a Secure Group Layer (SGL)[35], which approaches the secure group communication problem from a different angle: noting that shared group key agreement is hard, they suggest the use of a normal group communication system together with SGL as an extra layer on top to add Byzantine-fail-safety to the overall system. The resulting system works with shared group keys established by a group variant of the Diffie-Hellman key agreement scheme and is biased

towards state-machine replication: Agarwal et al. note that denial-of-service attacks are an unsolved problem of SGL. Based on their practical results they postulate that a process group over a WAN has a practical limit of around 40 members. Again, the state-machine approach results in high complexity and timing constraints which SGL alleviates by dividing the universe in group and non-group, with only communication among group members being safeguarded. Our application scenario makes this approach infeasible as group members cannot be trusted a priori, and we do not need real time process replication anyway. The SGL paper, however, references a number of related systems that fit our requirements more closely.

2.6 Practical Byzantine-Fault-Tolerant Services

Many researchers have commented on the limited practicality of earlier work in the area of distributed agreement: most techniques were designed to demonstrate theoretical feasibility at the cost of low efficiency or require strong synchrony assumptions for correctness.

Castro and Liskov presented an improved, practical replication algorithm [33] in 1999. Their system works in asynchronous environments (like the Internet) and excels at computational efficiency. Their main contributions are in the area of practicality: their system is highly optimised for the best-case scenario and avoids expensive operations wherever possible. As a proof of concept, they also presented an implementation of a real-word service, a distributed file system that is Byzantine-fault-tolerant and implements the NFS protocol. They provide analytical results for this service, which show excellent practical performance: the service is only about 5% slower than a (particular) normal NFS implementation.

Like everybody else operating in an asynchronous environment, Castro and Liskov had to work around the impossibility problem[23]. Works by Lamport[36] have introduced the properties of *safety* and *liveness* to study and define the components of concurrent execution. In Lamport's terms, safety is defined as what a program must not do (or what it may do), whereas liveness defines what the program must do eventually. Herlihy and Wing[37] suggest a slightly different correctness condition, *linearisability*, meaning that the program globally behaves like a centralised implementation that operates atomically. As a replicated service using the state-machine approach can be considered as one global program with parts executing concurrently, therefore both Lamport's and Herlihy's correctness concepts are applicable here: Safety from the perspective of the replicated system means that all non-faulty replicas execute the same sequence of requests, while liveness means that all requests are executed eventually. Together, these two implement distributed consensus – which we know is impossible in the asynchronous case, so one of the two properties can only be guaranteed with some synchrony assumptions.

Other practical systems[38, 32] require synchrony for safety, which Castro and Liskov comment critically upon: the safety of the overall service may be compromised by simple denial-of-service attacks like delaying non-faulty parties until they are wrongly tagged as faulty and excluded from the group. Castro and Liskov state that their algorithm provides safety without synchrony assumptions and they present a formal proof for this property in [39]. Their algorithm requires only a weak synchrony assumption for liveness, which they claim is likely to be fulfilled in a real environment as long as network faults are eventually repaired.

Their algorithm is an example of state-machine replication with the usual requirements for replicas as outlined in Section 2.4. Notable differences include the algorithm being based on a succession of discrete configurations called *views*: in a view, one replica is master and all others are backups, with clients sending their requests only to the master replica. The master multicasts the request to the backups which respond directly to the client. After sufficient consistent responses have been received, the client continues its operation with the result from the responses. If the master replica fails, progress is still possible (cf. liveness): after timing out, a replica triggers a view change which results in the selection of a new, non-faulty master. A number of measures were taken to ensure that view changes, which involve distributed consensus, do not happen too often or can be triggered by faulty parties to effect denial-of-service attacks. Some interesting optimisations were used to improve their system's behaviour

specifically in the failure-free case. Both of the two main problematic areas, computational cost and communication efficiency, were considered.

In Castro and Liskov's system, results are provided to the client by one replica only; others only supply a digest of the result. This reduces the amount of data exchanged at the end of a computation, but not the number of messages. A fall-back mechanism is employed to request full responses in case of failures. Another optimisation allows requests to be executed *tentatively*, with a reduced number of request preparation phases among replicas; such a tentative request may require a fall-back to a previous checkpointed state if a view change happens. A third related measure offers faster read-only operation by responding early in a tentative state as well.

In the area of computational cost, their optimisations are very much more extensive. Most methods of providing digital signatures are very expensive as they require public-key encryption and thus one would like to avoid such encryption operations where possible. Tsudik[40] shows various ways of employing one-way hash functions to provide message authentication codes (MACs). As hash functions are by about three orders of magnitude faster than public-key encryption[41], using MACs would be very beneficial for performance. However, MACs have their disadvantages as well: they require a shared key between principals and they can not provide proof of message authenticity to third parties, which digital signatures can. Most signed consensus protocols rely on this last capability, while Castro and Liskov managed to reduce that cost by using digital signatures only where absolutely needed and replacing them with MACs everywhere else. It turns out that in the normal failure-free case, all operations can be covered by MACs, because messages have single recipients only; digital signatures are required only during view changes which are infrequent. Thus, their system requires shared session keys between all replicas and clients but the detailed description of their algorithm[42] states that this extra session key management introduces no new major restrictions; instead it is shown that as long as no faults occur, their system has very low overhead and good throughput. In a related paper[43], they present their ideas on long-term operation of such a system and suggest an automatic proactive recovery strategy to reconcile faulty replicas and synchronisation of replicas that might have missed transactions temporarily. This strategy is similar to "anti-entropy" as discussed in Section 2.8.

2.7 Replicating Data and Quorum systems

A substantial body of work exists for quorum systems and benign failure scenarios, which prompted Malkhi and Reiter to develop Byzantine Quorum Systems[44]. Their architectural paper [45] criticises state machine replication as generally not scaling well because all servers must receive and process every request. Quorum systems on the other hand allow requests to complete with only a small fraction of servers being involved. This notion also lends itself to peer-to-peer situations where general availability of parties is a limiting factor.

In their seminal paper[44] they developed the conceptual extensions to quorum systems for tolerance of Byzantine failures: masking quorums, where the intersection of every subset of servers consists of a sufficient number of correct servers for data replication to succeed. The main points of interest here are the reduction of the consensus problem to operations on shared data objects rather than distributed computation, and the limited interactions between the client and a small quorum of servers, without any kind of centralised locking and also without server-to-server communication. Example quorum constructions were given, as well as cost measures. They also note that in the case of self-verifying data, the quorum can be simplified to a dissemination quorum which has less requirements. Also, quorums can be quite small at $O(\sqrt{n})$, where n is the size of the group.

Subsequent work led to the development of Phalanx[46, 45], a software system for building a persistent survivable data repository that offers shared data abstractions to clients. Phalanx is meant to scale to hundreds of servers and therefore employs quorum systems. Phalanx uses a protocol similar to Kerberos mediated authentication to offload most work to the clients, which occasionally has been critically remarked upon by others[27]: critics say that Phalanx is merely helping correct clients to

execute distributed protocols instead of performing computations distributedly. However, we feel that this criticism is not applicable as Phalanx's intended application is to safely share data rather than computational resources.

In Phalanx as well as its successor "Fleet", clients do not interact with other clients, nor is there any interaction among servers themselves. This architecture is similar to mediated authentication systems like Kerberos, where self-contained "tickets" transport all the information required for granting access. In Phalanx/Fleet, tickets serve the purpose of containing all the information that servers require to replicate data. A client contacts a quorum of servers, gets some tickets back, prepares an update using these tickets and sends them out to some quorum (which is not necessarily the same quorum as before). The servers in the quorum verify the received tickets and operate on them locally. Writes thus never directly span the whole system, but as all server subsets overlap sufficiently, updates to a data object diffuse through the whole system over requests (and time). Malkhi and Reiter note that this low number of involved systems per operation is the main benefit of using quorums. Phalanx provided a number of inspirations, as it introduces the notion of embracing temporary inconsistencies of data among the servers and probabilistic completion of operations; the idea of "timed append-only arrays" as data abstraction suggested the implementation of supersede-only documents. However, Phalanx's strict distinction between clients and servers and its lack of server-server interactions was felt as being too restrictive for our intended peer-to-peer setting.

2.8 Epidemics and Diffusion

Phalanx/Fleet and quorum-based systems are far from the only approaches to shared data; one of the earliest examples was Grapevine[47], a distributed and replicated system for delivering email. Grapevine originated at Xerox PARC, where it was deployed corporation-wide during the early 1980s. Grapevine implemented novel solutions for name resolution and email handling, predating and influencing the Domain Name Service[48] and today's Internet email standards[49]. However, Grapevine covers benign failures only.

The primary contribution of Grapevine was the practical demonstration of the benefits of weakly-consistent replicated systems for the first time together with introducing the notion of epidemic style replication algorithms, most notably *anti-entropy.* In Grapevine, email is submitted, routed and delivered controlled by information in a registration database which is replicated across servers. The database uses a partitioned namespace to assign administrative responsibility and is managed in a decentralised manner: an administrative client inserts a change at any available server, all of which periodically synchronise pair-wise which eventually leads to full distribution of the update.

The Grapevine developers noted that demanding full atomicity of database updates was not necessary, as the problem domain that Grapevine operates in allows weaker, eventual consistency for its operation. Obviously, temporary inconsistencies are possible and need to be taken care of in the design. Grapevine uses timestamps to order updates to entries totally, with the server address providing the tie-breaking component that completes the partial order scheme that simple timestamps offer; using these and various other mechanisms to limit the amount of information to be exchanged allows efficient periodic pair-wise synchronisation.

Grapevine was for some time a candidate technology for the Internet name resolution system but eventually got dropped in favour of a more light-weight solution; however, research work at Xerox PARC in the area of weakly-consistent systems did not stop: in 1987, Demers et al. [50] published the classic work, analysing epidemic algorithms for replicated database maintenance. This is mainly based on the authors' experiences with Grapevine and its commercial variant "Clearinghouse".

Adopting the epidemic disease paradigm, they developed and studied simple randomised algorithms for distributing updates among replicas: Each replica periodically *touches* a selected *susceptible* peer and

infects it with the update. Infective denotes a site that does have a particular datum, while a site that does not is called susceptible.

The two main methods studied are anti-entropy and rumour mongering. Anti-entropy is the mechanism employed by Grapevine, and consists of all replica regularly doing a pair-wise synchronisation with a randomly chosen other party. Rumour mongering works by having an infective site periodically select another site at random and offering the update to it, until the rumour ceases to be "interesting".

Following established epidemiology literature, simple epidemics like anti-entropy can be shown to reach the whole population eventually as long as connectivity is given. But because every replica makes random choices based on local information only, there is a low probability of updates not reaching all sites. However, as this probability decreases exponentially over time the danger can be minimised sufficiently for practical purposes. Also, one must weigh in the advantages of randomised protocols, namely their extreme simplicity and that they need few guarantees from the underlying communications infrastructure for robust performance.

As Demers et al. demonstrated, anti-entropy is reliable but too expensive traffic-wise to be used frequently. The other main method discussed by them, rumour mongering, allows a replica to be in one of three possible states: susceptible, infective or *removed*, the latter meaning that the replica has stopped propagating a particular update. Rumour mongering avoids the heavy overhead of anti-entropy and still allows the originator to inject the update at one replica only, but at the cost of an additional non-zero probability of stagnation and thus failure: it is possible that all infective replicas "give up" and enter the removed state while there are still susceptible replicas around; this is unavoidable because replicas have no global knowledge.

With rumour mongering, new updates or "hot rumours" sent to a replica make it periodically attempt to offer the update onwards to random others until some termination condition holds, which makes the rumour go "cold". The choice of termination criterion for ending this procedure is of major importance for the robustness of the scheme; the original paper suggests stopping propagation probabilistically after unnecessary offers are made or making this decision based on round counters or coins. Rumour mongering does not require as much resources as anti-entropy and therefore can be used with higher frequency.

The paper also presents a workable compromise to address the problem of stagnating rumour mongering by using both anti-entropy and rumour mongering where appropriate. In this scenario, rumour mongering would be used initially and be backed up by infrequent anti-entropy cycles to ensure full diffusion. It is observed that rumour mongering has better start-up characteristics, i.e. until most sites are infective, while anti-entropy performs better in the latter stages. Other works (eg. [51, 52]) expanded the coverage of the characteristics of such hybrid push-pull schemes and influenced our choice of mechanism greatly.

An additional contribution by Demers' paper is the notion of "death certificates" to deal with replicating the deletion of data items; in Section 3 it will be shown how our system relates to most of the concepts discussed here, with the addition of greater robustness in the face of Byzantine failures which was outside the scope of the Xerox PARC research efforts discussed in this section.

Among others it was Malkhi, Mansour and Reiter who addressed the missing link between diffusion-style replication and Byzantine failures in their analysis[53] of epidemic-style protocols in a Byzantine environment. Their paper discusses the characteristics of diffusion in a large system of data replicas, some of which might exhibit arbitrary faults. The analysis suggests *not* to use digital signatures for message authentication because it would imply assumptions about the computational power of faulty replicas. Rather, their basic assumption is either a network that intrinsically provides correct sender identification or that one uses pair-wise authentication based on shared-secret cryptography. We believe that this argument is of theoretical interest only, as in most practical cases a computationally unbounded adversary will have lots of attack opportunities outside the scope of the diffusion mechanism; furthermore, pair-wise authentication does not establish a transitive integrity assurance for messages at all. Their point about digital signatures being an expensive component of course stands.

Thus based on unauthenticated Byzantine agreement, their paper describes the scenario of an update being inserted in parallel at $\alpha \geq t$ replicas (of which strictly less than t are faulty), after which all

correct replicas engage in a diffusion protocol. It is stated as a requirement that only updates sent out by the correct fraction of initial replicas is accepted by others; in our opinion, this is quite impractical as single-source injection of information is the usual case. In their setup the initial group of active parties is treated as a single entity, the trusted originator, and because messages are not authenticated transitively, every replica that receives the update from an intermediate has to accumulate t copies of the update until it begins propagation itself.

The scenario is synchronous and rounds-based, however the paper mentions analytical results with relaxed synchrony that are comparable to the synchronous case. In every round there is at least 1 correct replica which will contact α replicas in the following round, thus ensuring that an update is not lost.

The difference of their scheme from classical Byzantine agreement is that in every step the update is only sent to a small number of replicas rather than the whole group. Obviously this produces temporary local inconsistencies, which are eventually resolved by diffusion driving the state towards global consistency. Malkhi et al. provide analyses of random selection of target replicas (which is quite similar to Demers' anti-entropy) as well as a topological approach that selects candidates using some tree arrangement, with random selection outperforming the tree variant if the degree of parallelism α is high.

The analysis considers two main measurements: the number of rounds until an update is accepted by all correct replicas and the number of messages a replica expects to receive from correct peers in a round. An inherent trade-off between minimising message numbers and rounds is proven, and it is noted that tolerance to Byzantine failures in the manner implemented carries quite a high cost in terms of rounds due to the time spent gathering enough good messages by intermediate replicas; this behaviour also causes bursts of activity when many replicas enter the active stage. This observation prompted us to develop our solution using signed messages so as to spread efforts over time, and assuming computationally bounded adversaries.

2.9 Eventual Consistency

Another very interesting system for relaxed-consistency replicated data is called Bayou, again part of Alan Demers' works at Xerox PARC. While Bayou does not take malicious failures into account and is by itself therefore insufficient for Byzantine environments, the architectural paper[54] outlining the goals and design decisions underlying the Bayou system was very intriguing and influential.

Bayou strives to support data sharing among mobile clients, with the explicit assumption that clients may experience extended and involuntary loss of communications. To provide high availability despite such outages, the Bayou system employs weakly consistent replication of data. The architecture paper provides a discussion of why most other, more strictly consistent, mechanisms fail to meet the goals especially when dealing with partitioned networks.

Bayou tried to minimise assumptions about the available communication infrastructure by using peer-to-peer anti-entropy for propagating updates. This infrastructure-less approach has become a recurring theme with the recent commonality of peer-to-peer systems[51].

To maximise a user's ability to read and write data, Bayou allows clients to write to any replica; this of course creates the possibility for conflicting operations being inserted at different peers which have to be resolved somehow. Acknowledging and embracing the floating nature of data updates, Bayou deals with such update conflicts by allowing application-specific automatic merge procedures.

Interestingly, OceanStore, one of the major global storage projects, is partially based on Bayou's conflict resolution system. The notion of tentative versus explicitly committed writes is introduced: committed writes are stable, with no non-committed writes being ordered before them. This can be achieved by distributed consensus among the parties (which is where the Byzantine Generals problem awaits the implementor), or by restricting operations slightly as in Bayou's case: here each database has a "primary" server that is responsible for committing writes. Bayou's authors decided on this simpler operation in

order to get away with minimal communication guarantees. In Bayou, clients can read tentative data if they so desire; this is allowed to keep Bayou simple yet flexible and to allow applications to select their own appropriate point in the trade-off between consistency and availability. Bayou also supports session consistency guarantees so that clients can access more consistent data during a session. These are implemented using version and timestamp vectors.

In a subsequent paper[55], Petersen et al. expanded on the details and resulting characteristics of the Bayou system and also present one of the few light-weight practical application examples: they implemented BXMH, a version of the EXMH[56] mail user agent which manages emails stored using Bayou rather than a local file system. Experimental results show practically sustainable performance with a very small code base (under 30k lines of POSIX C).

Bayou comes pretty close to a minimal practical system for replicated shared data except where security is concerned: in that area, only an untrusted network infrastructure is covered. How our work extends that scenario to cover arbitrary failures of any involved party is presented in Section 6.1; for now we need to discuss some further aspects of epidemic distribution systems.

At the beginning of this section, we covered Grapevine and other systems that primarily rely on anti-entropy, and mentioned the other potential mechanism for epidemic distribution of data, rumour mongering. Rumour mongering has the advantage of being less costly than anti-entropy in communication terms, but at the cost of lower reliability.

Karp et al. [52] provide an excellent coverage of the rumour mongering problem domain. In their paper they discuss the very generic setting of spreading rumours in a distributed environment where communication is randomised and parties are independent. Synchronous parallel rounds of data exchanges between parties are assumed. The goal is distributing rumours from one originator to all parties using as few rounds and messages as possible. Some exchange protocols are suggested and analysed, allowing for what is called “adversarial node failures”. We believe the naming of said term to be slightly misleading as it does *not* encompass Byzantine failures: in Karp’s scenario, an adversary can specify a set of faulty peers which *fail to exchange information* in some of the communication rounds. This falls short of Byzantine safety as exchanging misleading or false information is not dealt with.

Nevertheless, the insights provided are substantial, among which is a proof that rumour mongering algorithms inherently involve a trade-off between optimality in time and communication cost. Also, the combination of pushing and pulling exchanges is studied, with a practical algorithm being suggested. Push transmissions (sending hot rumours from the calling party to the callee) have the useful property of increasing the set of informed parties exponentially over time until about half of the group is involved; thereafter, the residual set shrinks at a constant factor. On the other hand, pull transmissions (asking the callee for news), are advantageous if and when most parties are involved and if almost every player places a call in every round, as then the residual set shrinks quadratically. This dichotomy suggests combining push and pull mechanisms, or at least switching between them. The decision of when to switch algorithms (and in general when to terminate the rumour mongering altogether) is the crucial aspect, controlling the robustness of the overall scheme. This is complicated by parties being independent and only having access to local approximations of the global situation at any particular time.

[52] presents some discussion of the simplest possible push-pull combination, namely sending rumours in both directions in every round; rumours now need to include an *age* counter to be transmitted with the data, which is used to estimate the set of informed players and controls the termination of the distribution. There are some issues with this round counter: first of all, an adversary could lie and adversely affect the distribution. Second, having a fixed maximum age is not robust – but setting this limit higher than strictly needed wastes communications. Karp suggests a scheme which transmits not just an age counter but also some state between parties, where each party uses the state messages received over time to control its switching between push-pull to pull-only and subsequently to silence, i.e. termination of the rumour spreading.

The notion of rumours carrying some meta-information about the rumour in question is also used in [51], and we have adopted a more failsafe variant of this meta-information, as in our environment information

passed by some party can not simply be taken at face value. Therefore, verifiable rumours are used, as detailed in Section 4.5.

Another item of interest is that [52] covers only address-independent algorithms, i.e. ones that use truly random selection of communication partners based only on the sizes of informed and residual set but not on the actual identities. Address-dependent algorithms are more capable due to the extra information that they can base decisions on. Karp also shows that selecting random communication partners is fundamentally more costly than a deterministic broadcast scheme which can require as little as $n-1$ messages for a group of size n; of course without any robustness.

Following these results, our mechanisms were designed to employ random communications and be address-independent, but fine-tuned address-dependent variants are discussed in Section 6.4.

The last surveyed work on epidemic distribution is [51] by Datta, Hauswirth and Aberer. It aims to extend epidemic algorithms towards a peer-to-peer paradigm by suggesting algorithms that tolerate highly unreliable, ephemeral peers. The paper studies the problem of updates in decentralised and self-organising peer-to-peer systems, where progress is hampered by both low probability of peers being reachable as well as the lack of global knowledge to base distribution decisions on. Their overall goal was a distributed storage scheme, with the lack of global knowledge and the need to self-organise locally making up for most of the challenge. In their paper, Datta et al. present a hybrid push-pull strategy which was influenced by the previously discussed variant due to Karp[52], but which is drastically simpler and able to deal with large numbers of replicas as well as the expected frequent disconnections of a large peer-to-peer environment.

Their problem domain and solution approach centres on these cornerstone assumptions:

- Peers are totally autonomous and independent, with no global knowledge or control available.
- Individual peers have a low probability of being online, but the population is large; replication and connectivity factors are high.
- Probabilistic consistency is sufficient, and update conflicts are expected to be rare and not needing automatic resolution.
- The expected failures are of the benign "fail-stop" kind.
- Communication overhead is the main measure of performance.

While not attempting to cover Byzantine failures, this work was still very much applicable to `Peeranoia`'s development and some design decisions parallel our setup. Datta et al. realise that rumour mongering would have quite limited efficiency in an environment where most systems are expected to be unaccessible at times, therefore a push-pull mechanism was adopted: the main distribution model is a push model, but peers who miss an update can use the subsequent pull phase to catch up with developments. We have adopted this approach for similar reasons.

In addition, their algorithm uses an extra mechanism, besides the usual feedback to control the spreading of information: in their system, every message carries a partial history of peers who have been offered this datum previously. This information is used to direct updates more precisely but also as a local metric of the "reach" of the distribution, which in turn controls probabilistic parameters of the rumour spreading; overall it reduces duplicate messages greatly. This *feed-forward* mechanism allows peers to collect a rough approximation of the global state over time; this being a novel contribution for epidemic algorithms as well as their analytical modelling for rumour mongering.

In their algorithm, the push-phase is the predominant means of distributing information: as explained in [52], exponential growth of informed parties allows a rapid spread among online parties. The push phase is meant to allow resurrecting parties who missed information during their downtime to resynchronise at will; after the push phase, most nodes will have the information so pulling in by the remaining nodes is vastly more efficient than prolonging the first phase. The pull decision is made locally, governed by

a variety of heuristic means (e.g. realization of having been off-line, not receiving any updates for a prolonged period or receiving a pull request for a dormant update).

The push algorithm exchanges messages, round counters and a list of peers that the update had been offered to by the sending party, but that does not necessarily mean the message has been received by said peers. Duplicate updates (i.e. all aspects of the message including the history list) are ignored; otherwise the receiving peer makes a probabilistic decision on further propagation of this message and if positive, increments the round counter, selects a number of candidate peers at random from the uncontacted remainder of the population, and sends updates to them. The number of push rounds controls how closely the probability of reaching the whole online population will approach unity.

Our problem scenario is subtly different, as information received by peers requires closer scrutiny. In our design the feed-forward idea was extended, using *verifiable* history information (see Section 6.1). Another important difference is that in our environment, peer identities need to be more permanent because there is an association between trust and a peer.

2.10 Distributed Storage

One of the more well-known global storage system is OceanStore, a research effort at UC Berkeley. Its approach to security- and scalability-related problems is quite interesting, but it was deemed too complex and heavyweight for an FIC application.

In [57, 58, 59], the team around John Kubiatowicz outlines the goals and design decisions that define OceanStore. The goals are fairly ambitious: a truly global maintenance-free data storage system, which is envisioned as the one service *everybody* would entrust *all* of their persistent data to. We see this is a potentially dangerous step towards yet another information technology mono-culture.

OceanStore is meant to be highly scalable, potentially consisting of millions of individual servers that cooperate to provide the overall service. However, these servers are untrusted and a fraction of them is expected to be arbitrarily faulty at any time: redundancy and measures to deal with malicious operations are therefore paramount.

Data migrates freely between servers and is stored in a fragmented fashion using erasure codes[60]. Erasure codes, or m-of-n data encodings, provide data robustness at the lowest levels of OceanStore, with the rationale being that full replication is too costly on a large scale, both storage-wise as well as concerning overall robustness.

However, fragmentation like this obviously requires a means to check the integrity of recombined objects, especially when the data comes from untrusted servers. In [61], Weatherspoon et al. explain how they manage to prevent malicious adversaries from presenting corrupt fragments as valid. The approach works by making data self-verifying and is outlined as follows: for every fragment, a secure hash is computed, the hash is joined with a sibling hash and a second-level hash is computed over that and so on. The hashes thus form a binary verification tree, at whose root a single hash emerges. This is hashed again with another hash of the complete datum[3] to produce what Weatherspoon et al. call a self-verifying globally unique identifier (GUID) for the datum. Every storage element includes a data fragment, the hashes of the whole datum and of all other fragments and intermediaries that are required to recompute the GUID. Every fragment therefore can be verified locally and independently. [61] also points out that such self-verifying data lends itself to application in a peer-to-peer storage infrastructure where repair mechanisms are essential, but claims replication as less desirable and robust than erasure coding - a view which we do not subscribe to.

The GUID also serves another purpose: it immutably names the datum in question; this fixed relationship between content and name effectively mandates the use of some versioning scheme to deal with mutable data (cf. [62]), or a multi-level setup which associates fixed names with sets of changing GUIDs.

[3]This extra hash was introduced to allow efficient coexistence of non-fragmented storage as well.

OceanStore does the latter, but has to rely on cryptographic signatures for verification of that higher-level binding.

Both OceanStore and our setup adopted some ideas related to Mazieres' and Kaashoek's work on self-certifying pathnames in SFS[63], a secure and global decentralised file system that extends the widespread Sun NFS[64]. Mazieres' work assumes the scenario of a very decentralised infrastructure and points out in detail how central control in such a scenario is detrimental. He suggests an egalitarian approach to this issue, using the help of cryptography to deal with such a global namespace without central control. Our research is based on the same desire to limit central control, but we have gone further by also not trusting anybody with our data: Mazieres' SFS does not include replication or other robustness-increasing mechanisms.

Self-certifying pathnames in SFS invariably include a hash of the serving host's public key, thus efficiently partitioning a shared global name space into slices, each of which has one responsible server associated. This host identifier component in the path thus provides a safe way to establish a secure connection to the correct server and to verify the provenance of data exchanged. The SFS system does not support specific integrity assurance mechanisms for the served data itself, although it supports serving cryptographic signatures for read-only file systems. Mazieres is mostly interested in the global decentralised nature of the SFS namespace, and the self-certifying nature of pathnames strives to prove the correct filesystem-to-server binding in an efficient client-verifiable manner. Practical experiments have shown that SFS performs comparably to Sun's NFS which provides no security whatsoever.

In OceanStore itself, we note that the self-verifying nature of GUID together with erasure coding allowed a two-tier hierarchy of the OceanStore servers: one inner ring consisting of very few servers which coordinate strictly using a Byzantine agreement protocol, and an outer ring which merely caches self-verifying data. Every object has its own inner ring, whose crucial service is to provide the mapping between the persistent GUID of the object over time and the actual self-verifying GUID of any immutable version of it: these need to be regenerated whenever a document is modified.

The agreement protocol used by OceanStore is derived from the one developed by Castro and Liskov (Section 2.6). Extensions had to be made to allow for caching replicas, which the Castro-Liskov protocol did not provide for: for efficiency's sake the inner ring typically consists of less than 10 servers which use a threshold signature scheme to collectively sign the results of their agreements to allow verification by outer caches; the inner ring is also modified dynamically to further limit the involvement and effects of corrupt servers. The inner ring executes a modified Bayou-style conflict resolution (with the whole inner ring playing the role of Bayou's primary server).

In OceanStore, data is *meant* to be highly nomadic. Information and location are to be divorced[58] to avoid centralisation and its problems. Pervasive replication would achieve this goal, but was deemed to be too expensive and was replaced with erasure coding and routing infrastructures to locate data. Also, OceanStore includes some proactive measures to manage the amount of replication over time and changes of server availability.

Conversely, in our case we would like to know *precisely* and authoritatively which servers are hosting a replicated document. We do not envision the same global scope for our system, therefore the KISS principle was deemed of higher importance than storage efficiency as in OceanStore - whose efficiency comes at the price of high overall complexity.

2.11 Secret Sharing

Many variants of replicating or threshold-scheme fragmented storage have been developed in the last decades, with most directly or indirectly following up on Herlihy and Tygar's early work on securing replicated data[65]. In their paper, they tackle the problem of protecting the confidentiality of physically distributed data against a number of Byzantine faulty servers. Replication protocols were analysed and extended with quorum agreement mechanisms so that corruption of up to a threshold of parties

does not disclose information to an adversary. These protocols mainly use secret sharing schemes[66], a mechanism to distribute secret data in a way that leaves it completely undetermined unless sufficient parties combine their shares. The difference between erasure or error-correcting codes is that with secret sharing all possible values for the secret are equally probable even if only one share is missing. One such mechanism is polynomial interpolation with modular arithmetic[66].

Herlihy and Tygar note that achieving secrecy is not a problem and is computationally inexpensive even in the face of Byzantine failures, whereas integrity is not; they also see data security as a characteristic that is shared in a group. In contrast, in the FIC scenario we approach data security from an ownership perspective: data is not shared but owned by a distinct entity, possibly made opaque by means of cryptography, and is expected to be of value mainly to the owner[4]. This owner shall be able to retrieve the content, and therefore we encrypt all data asymmetrically for the intended reader. Subversion of parties other than the owner will not affect the document's secrecy (postulating sufficiently unbreakable asymmetric encryption) and is thus outside of our scope.

A fairly recent practical development in the area of distributed storage based on singular ownership is DIBS, the Distributed Internet Backup System [67]. DIBS allows multiple peers to share storage facilities for backup purposes: every peer would altruistically and blindly store other peers' files and in turn deposit his backup data with them. In case of a local catastrophic loss of data, one would recover the data from surviving peers. Global distribution of cooperating peers provides a probabilistic guarantee of a sufficient number of peers for data retrieval.

The main critical aspects of such an approach are, of course, secrecy of the injected data, verifying the integrity of material returned by other peers and robustness in case of failure of peers. Regarding robustness, DIBS uses erasure codes (specifically Reed-Solomon codes) to provide maximum robustness for any given amount of redundancy. Files beyond a certain size (by default 10 megabyte) are split into chunks, extra fragments are generated and all fragments are distributed to other peers. To ensure secrecy and integrity, DIBS encrypts and signs all transactions with GnuPG. For recovery from scratch, a DIBS user will need to store two data items offline: the GnuPG private key and a list of peers storing the backup data.

DIBS is not resilient against Byzantine failures for a variety of reasons, but in most cases (short of total local loss of data) problems with lying peers would be detectable. Another interesting aspect is that DIBS provides for a simple economy of reciprocal trading of space: peers interested in using DIBS can find each other using a peer finder service which exchanges posted *contracts* detailing the capabilities and needs of a peer. Given that DIBS expects no knowledge of or trust in the peers involved, this is a required measure to reduce abuse by parties trying to only consume but not provide. The setup is reminiscent of GNUnet's economy[68], but very much reduced. There is also no mechanism to independently verify compliance with the rules, thus providing another attack vector for malicious peers. Overall, DIBS is interesting on the technical level but lacks critical coverage of trust issues. In [69], a separate group of researchers suggests some modifications to the DIBS system that improve performance, but these suggestions do not address issues of trust.

Nevertheless, DIBS excels in one important measure: simplicity; it is very lightweight with a code base of a mere 6200 lines of Python code, with GnuPG being the sole external software dependency. DIBS covers most of the common replicated storage problems well but excludes some more paranoid scenarios like our FIC setting.

In summary, our overall critique of secret sharing, erasure codes and similar fragmented storage schemes is thus: Putting availability foremost, integrity assurance second, we think that requiring cooperation between multiple untrusted parties merely to retrieve content belonging to a single entity is not a robust proposal. In extremis, we would like a single isolated peer to be able to operate, even on potentially outdated data.

[4]Owner and originator are not necessarily the same entity; for example an administrator alert might originate at one party but be encrypted to all known administrators.

2.12 Trust Issues

Obviously any system that performs operations on our behalf needs to be trusted (to some extent) to perform properly; in a peer-to-peer scenario with independent peers and where no single component can be trusted, the issue of whom to trust what with and how far is a major concern.

Blaze et al. have introduced the term *Trust Management* [70] for this problem, decomposing it into management of security policies, assignment and verification of credentials and trust relationships. It is suggested that the existing specific solutions for problems in that domain are too narrow in scope, and therefore a more comprehensive approach is proposed. To that effect, Blaze introduced PolicyMaker, a tool kit for policy formulation and verification in a programmatic manner.

Reviewing the most common asymmetric cryptographic infrastructures as of today (X.509 and PGP) in [71], Blaze et al. point out their limitations from a distributed systems perspective: both systems primarily provide a binding between the identity of a principal and a key, and do not provide any reasonable mechanism for authorisation or policy. While differing in the level of centralisation, neither system is expressive enough to scale efficiently in either dimension of systems involved or application scenarios.

PolicyMaker tries to fulfil three fundamental requirements for trust management as Blaze identifies them: support for delegation, extensibility and locality.

- Delegation is required for scalability, to decentralise administrative tasks. X.509 is criticised for being too monolithic in that regard; the last decade has shown succinctly that commercial pressures have kept the goal of one global top-level authority unattainable (which would have improved X.509's capability for efficient delegation). PGP, offering the utmost extreme of delegation (as it has no central authorities at all) is commented upon by Blaze as being too ad-hoc and anarchic for commercial use; a point of view we do not share.
- Extensibility is what PolicyMaker offers to general-purpose applications, and admittedly where most previous authorisation frameworks offloaded most of the work to the application designer.
- Locality of control is necessary for any generic system to scale: different applications have different notions of access control, and should not be expected to conform to a uniform policy that is implied or required by the security mechanism.

Considering current authorisation systems in light of these three requirements, Blaze et al. dismiss identity-based asymmetric cryptosystems (even when combined with access control lists) as inadequate solutions for distributed systems. Blaze explicitly includes AuthentiCode in that list of failures.

PolicyMaker (and some other trust management systems as outlined in [71]) combines mechanisms for specifying and verification of security policy, and security credentials into one system and provides a simple assertion language in which applications would express the conditions under which an individual or authority is trusted or under which trust might be delegated. In this language, keys are bound directly to predicates (rather than to the owner's identity) which describe the authorised actions that a key should be trusted for.

The combination of this very expressive policy language with application-independent proofs of compliance provides high flexibility. However, it also introduces a new cost factor: complexity is shifting from the application to policy design.

We see PolicyMaker as an interesting but very heavy-weight approach to authorisation. Therefore we do not employ it directly in the `Peeranoia` system, but Section 4.4 shows how Blaze's ideas of binding keys to permissible actions has influenced our (relatively limited and application-specific) policy system.

Regarding the issue of trust in a computer system correctly observing its own state, we have to acknowledge that any introspective operation on an untrusted computer system is subject to prior undetected

intrusions that might modify the operation. As Ken Thompson has put it in his award-winning lecture on trust[18]:

> "You can't trust code that you did not totally create yourself."

This fundamental fact obviously limits the assurances that any local intrusion detection system can provide. Even to simply run a program of our choice we must make certain trust assumptions, e.g. in the hardware, the operating system and the intrusion detection executable itself (and the compiler, and its compiler and so on).

Thompson showed that compilers can bear interesting gifts in form of self-perpetuating Trojan horse code; such an attack could yield complete control over a vast number of systems and thus is very appealing to an intruder with long-term goals. A very interesting recent work by David Wheeler[72] shows an approach to counter this problem; his paper describes and analyses a method called *Diverse Double-Compiling* (DDC), whereby the suspect compiler is compiled twice to produce a bit-identical copy for comparison. The first compilation step requires a trusted compiler and environment, the second compilation is done using the result of the first. Wheeler notes that complexity of current computer systems practically precludes absolute proof of trustworthiness (both of the trusted compiler environment as well as the result of DDC), but he shows that the assurance can be made as strong as required: diversity can be introduced in various areas, whereby the likelihood of non-detection of malicious code can be made arbitrarily small.

As an example approach to building a trusted compiler environment, Wheeler briefly discusses TCC[73], the Tiny C Compiler by Fabrice Bellard, which is small enough to be verified on the source code level. TCC is standards-compliant and fast enough to be used as an on-demand compiler of the whole operating system during the boot process. Using this procedure provides a high degree of assurance that the operating system is untampered.

The issue of trust establishment is prominent in many of the recent peer-to-peer network projects, especially the ones aiming at anonymity and censorship-resistance like FreeNet or GNUnet. Grothoff presents in [68] the economic model used in GNUnet, a P2P framework for anonymous distributed file-sharing, which uses trust as currency. GNUnet is a collaborative network consisting of untrusted hosts and requires some mechanism to control resource allocation, because not all hosts are worth spending resources on. The economic system attempts to detect nodes that abuse the network by not contributing back.

A very simple idea lies at the core of GNUnet's model: every peer keeps track of transactions that it has been involved with and learns which other peers behave well. Consistently contributing nodes can earn the trust of their peers and get preferential treatment in times of resource scarcity. Nodes gain and lose trust by their usage of the network: both requests and responses (interpreted as consumption and contribution of resources, respectively) carry a priority number. The priority tells the recipient by how much the trust in the sender can be reduced, and a successful response will increase the trust in the responder at the sender by that amount. This system allows for decentralised operation without any overarching authority, an essential trait given GNUnet's anonymity goals.

While GNUnet's development was concurrent with our own design efforts for `Peeranoia`, nevertheless there are some parallels worth noting, most importantly localised operation and behaviour-based trust.

Localised control: In both scenarios, nodes can not be trusted individually a priori and therefore locally available information is the only available base for making decisions. Information about a node is thus stored at other nodes. This has the effect of introducing some forced altruism into the system: there can be no guarantee that a service provided by a node in the past will *buy* the node anything in the future. The nodes having consumed the service might just not cooperate and dishonour their "debts". To balance this possibility of vanishing creditors, behaviour- or evidence-based trust is used as a regulator.

Behaviour-based trust: In GNUnet, nodes form an opinion of each other based on usage history, which is then used to prioritise and triage incoming requests. The more a node knows about its peers, the better it will be able to make good decisions. In our case, nodes generally mistrust each other but collect *evidence* of each other's activities to figure out if the mistrust is justified; if so, the miscreant is not served at all. A key difference is that in our case authentication of nodes is controlled globally and we can therefore use indirect information exchanges to judge peers as all information bears independently verifiable signatures.

But in either case, a node must prove to others to be worthy - by behaving within the limits of the protocols.

2.13 Summary

In the area of intrusion detection, the consequences of the introspective operation of host-based IDS' have been explored extensively in the past. The two main components of such an IDS are the application itself and the data describing the initial baseline, and both are exposed to tampering by a successful intruder. While a lot of effort has been spent on mechanisms to make applications and operating system components tamper-proof, the data aspect has not been covered extensively. Especially in large, decentralised settings, there are no mechanisms for efficient, secure *and* scalable management of the baseline data. Using distributed services to provide such data management is an obvious approach to address this shortcoming, especially given the growing reliance on decentralised computer networks of the last decade.

Distributed services and the fundamental issue of distributed consensus have been studied at length, and the main original goal was strict coordination between the involved parties. This led to state-machine replication systems, where operations are replicated among peers and strictly coordinated to provide a consistent global state. Such strict coordination requires some synchronicity guarantees for the communication infrastructure between peers which are not attainable when public networks are used. There is also a high communication overhead associated with safeguarding against Byzantine failures.

The recent rise of P2P systems has resulted in some refocussing of research into distributed services, with autonomy of peers and decentralisation becoming more important. As a consequence it was suggested to relax the strict coordination goals of state-machine replication to eventual consistency. Ground-breaking early works like Bayou and Grapevine have shown the great potential of this relaxation, as it still allows to construct practical services but is drastically simpler than strictly coordinated systems.

From a IDS perspective, the most interesting distributed service is one for storing security-critical data safely and there is a lot of research that focuses on developing distributed storage systems. Some of these cater to paranoid applications like an IDS, others are geared towards simplicity and yet others aim for efficiency on a global scale, but none provide a good balance the three criteria we deem most important: simplicity, autonomy and mistrust.

After reviewing these related works, it is already clear that the first fundamental question of this thesis can be answered affirmatively: it is indeed possible to construct a distributed service which provides peers with integrity assurance for local data by cooperation between peers.

It remains to show what cost, robustness characteristics and trust requirements are associated with such a service. As the various distributed storage systems have shown, there is an inherent conflict between maximising robustness and minimising cost and trust requirements.

It is therefore necessary to decide on certain design principles to control the trade-off between these goals. In the next chapter, we use the needs and expectations of an FIC application to develop one possible set of such design criteria.

Chapter 3

Design Philosophy

3.1 Introduction

The last chapter has demonstrated that while there are various methods for construction of distributed services, none of them are fully appropriate for use in an IDS/FIC setting. As this research focuses specifically on file integrity checking in decentralised large environments, the need for mistrust and ongoing data verification that verges on paranoia is clearly present. On the other hand, a certain amount of altruistic cooperation is required for any distributed service if there is no global authority.

It is clear that a number of such conflicting design goals must be properly balanced to produce a practical, safe and efficient FIC application.

In this chapter, we present the overarching goals of the `Peeranoia` system and continue to derive from them a set of design criteria which in turn lead to a detailed discussion of the `Peeranoia` protocol and algorithms in Chapter 4.

3.2 Design Goals

One major goal of the `Peeranoia` system is to improve the data integrity assurance of a party's data over what can be achieved by purely local means.

Any FIC application bases the detection of intrusions on a baseline set of known-good file contents, and as discussed in Section 1.2 local storage of this baseline is either insecure or has bad scalability properties. Therefore, cooperation between multiple peers is used to replicate storage of FIC baseline data redundantly and thus make it less susceptible to being compromised. This goal also implies mistrust of *individual* peers and the communication infrastructure, and generally results in costly operations to verify integrity. Another implication is the need for built-in robustness to provide survivability of a limited number of intrusions.

A separate goal is to provide efficiency in two separate contexts: firstly, the cost of cooperation between peers is to be minimised. This includes communication overheads and computational costs. On a macroscopic, global level, the system also must be deployable and maintainable efficiently without sacrificing security. This global efficiency goal is strongly connected with relaxed consistency requirements for data, and with the decentralised environment and the need for human oversight in an FIC setting.

To balance these conflicting goals, we have identified three fundamental design principles from which `Peeranoia`'s design was derived: Autonomy, Mistrust and Simplicity. None of the related works previously discussed balance these three criteria even moderately well, which provides our motivation.

3.3 Autonomy

In terms of the stated goals, centralisation of security-critical operations appears to increase the associated risks and introduces more problems than it solves. Therefore, we try to maximise autonomy of the systems involved. This leads to adoption of a symmetric peer-to-peer environment of parties which provide a mutual service. Such a distributed service must be made robust, especially given untrusted parties, therefore we use replication and redundancy to increase the survivability of the service in the presence of intrusions or failures. Replication of operations as well as data replication would fit that need but in a decentralised FIC scenario, data replication is superior because requiring parties to operate in lockstep would imply high synchronicity which we cannot guarantee on public networks. Also, localising control is strongly suggestive of relaxed, eventual consensus.

Maximising autonomy and local control rules out any mechanism by which data that belongs to one party would be visible to others; this includes any verification mechanism that is executed on a trusted server separate from the party the data belongs to. This suggests separate treatment of data items that belong to different parties. In the `Peeranoia` design, this is achieved by treating a party's data items as a number of separate *documents*, each of which is addressed by a unique name.

Such autonomy has the side-effect of requiring some cooperation and optimistic altruism on part of the involved parties: a good deed done today on somebody else's behalf might not pay off tomorrow, as nobody can coerce the others to honour their "obligations" to reciprocate. We counter this, similar to GNUnet's trust economy, by being suspicious of others and collecting evidence of their behaviour.

As a less desirable consequence, autonomy also implies that the introspective nature of testing data on a system that might become subverted is unavoidable. However, we note that no mechanism short of providing independent physical access to storage among all involved parties can counter this introspection.

Maximising autonomy implies that such a distributed service should degrade gracefully, which implies that in the extreme case of a totally isolated peer, this peer should still be able to perform its integrity validation (if with reduced levels of assurance). As a subsequent implication, no single indication of data integrity coming from outside sources should be used by itself.

Secret sharing, erasure codings and threshold cryptography schemes fall short in providing this autonomy[1], as reconstructing data from multiple sources clearly fails without connectivity.

Therefore, our system design is more akin to a distributed backup than a partitioned distributed storage system, and we aim for local copies of the data in question to be present at the owning system most of the time, with the peer group to provide an *assurance backup* for this local data when possible.

3.4 Mistrust

In our system, no single entity is to be trusted. To the extent possible, not even the computer system on which the `Peeranoia` software runs can be trusted. This stance influences not just the cryptographic primitives employed but also the kinds of operation the distributed service supplies (e.g. abolishing modifications of existing documents in favour of versioning).

A precise distinction must be made between entrusting somebody with a peer's data and trusting some other party to adhere to a protocol in order to provide a service or execute an operation. `Peeranoia` does the latter only, using a simple form of evidence- or reputation-based trust management (similar to GNUnet's trust economy).

Evidence of conforming to the minimum expectations for cooperation in the distributed service is exchanged as part of providing the service; peers use this evidence to form a local impression of the others' trustworthiness. We are not attempting to infer trust globally. Instead, we use a vastly simpler scheme

[1] The mistrust principle also contraindicates sharing or farming out of secrets to multiple parties.

of reducing access locally for violators: all parties that misbehave are excluded from participation. As a consequence, a sending party is required to do its utmost to stay within the acceptable framework. Section 3.7 details the minimal criteria defining cooperation. Note, however, that only *independently verifiable* evidence can be trusted: digital signatures are used to provide such evidence.

Not trusting single entities suggests a semi-static group set, with some safe method to update memberships. Also, being unable to fully trust the computer system on which the application runs on indicates the need for an separate supervising party (or more precisely, a *class* of multiple supervising parties that are not trusted individually): in an FIC scenario, human administrators need to provide the intelligence and the final arbitration for some of the parties' attempted actions.

Digital signatures require some form of Public Key Infrastructure (henceforth abbreviated PKI); in our case this is provided by one set of group meta-data that identifies the involved parties fully. This document must be provided to a newly joining party by an administrator, before that party can participate in the system. As such, this administrator plays the role of an implicitly "trusted dealer". While this may seem to conflict with the mistrust principle, it must be noted that human involvement is not completely avoidable during the initial installation phase of any computer software. Furthermore, an FIC application absolutely requires the establishment of a known-good baseline of file contents which also implies administrator actions.

Mistrusting individual components also requires that communications are safeguarded against undetected tampering and, for practicality's sake, eavesdropping. Having a PKI makes agreement on shared keys between any two parties very easy and thus allows them to perform efficient symmetric encryption on all traffic.

Not fully trusting the computer system on which the `Peeranoia` software runs is a fundamental aspect of any local IDS, and thus also of an FIC application: at best, the time and extent of an intrusion can be pinpointed by detecting changes to files. To achieve this, a distributed history of activity or snapshots of local data is required. Obviously, after an intrusion has been successful, no trust whatsoever in subsequent activities of this system is possible. This implies that retroactive modification of old data must be prevented.

In the `Peeranoia` system, peers by themselves usually cannot create documents that are immediately ready for use: because a peer is always suspected to have been compromised, important data created by the peer must be verified and signed off by a human administrator. While the autonomy and simplicity principles dictate that peers can write to the replicated storage by themselves, mistrusting the peers requires the distinction between tentative and known-good data.

To this end, data replicated in the `Peeranoia` system normally does not become *active* immediately upon creation and replication but only after certain *policy* requirements have been met.

Activeness is a subjective measure of the degree of replication and the level of integrity assurance provided by the distributed storage system. The policy, as detailed in Section 4.4, specifies data authorship restrictions and replication requirements for data to be considered active. By using such a policy, multiple separate levels of (administrator) privileges can be implemented flexibly: not all data is of the same level of importance and needs the same degree of integrity assurance.

Returning to mistrust, we note also that a *single* human administrator does not necessarily have to be trusted either; a policy specification can impose arbitrarily complex restrictions on administrator behaviour (but of course only within the `Peeranoia` system).

In the FIC scenario, a peer would create a new set of file fingerprint data when changes of content are detected. This fingerprint data would be replicated by the other peers, but must not be used immediately by the originator peer as the baseline for FIC decisions: a human administrator must determine if the represented changes were intentional, for example caused by administrative work, or indicative of malicious activity. In case of benign changes, the administrator would certify the fingerprint document and it would subsequently enter the active state and therefore be accepted by the originating peer.

3.5 Simplicity

It is a well-known tenet of (security) engineering that Simplicity breeds Robustness; following this in form of Pike's Rule #4[74] and the KISS principle diligently, we strive to limit complexity wherever possible. Obviously, this involves certain trade-offs like the degree of synchronisation and consensus among peers. When in doubt, we went for simplicity over perfection (cf. [75]).

As pointed out earlier, state-machine replication schemes are powerful in their ability to replicate operations across a group of peers, but at the cost of high complexity and certain synchronicity requirements. In light of the autonomy that we want, state-machine replication and its strict lock-step coordination is not necessarily desirable. The need for strict consensus and total ordering of operations can be avoided, if the access semantics and operations that the distributed system must support are selected carefully. This results in the `Peeranoia` system being only a data replication system (as opposed to state-machine replication).

Simplicity suggests treating parties as autonomous and independent, and non-guaranteed public networks fall into that field of suggestion as well. As such, no timing, connectivity or liveness guarantees for peers and communication infrastructure are available, and the impossibility result for strict consensus in fully asynchronous systems applies. Embracing the probabilistic nature of eventual consensus seems to be the simplest natural answer here. As Bayou, Usenet and the Domain Name Service among others have shown, sufficiently powerful distributed services can be achieved with such reduced consistency.

Also, in an FIC scenario, any system usually has access to a local copy of any crucial data and uses the distributed service as a backup mechanism to reinforce the local interpretation of that data. Therefore, temporary loss of communications or limited inconsistencies are not fatal for a system's operation. It is merely an inconvenience and a situation of temporarily limited trustworthiness, rather than a complete failure as it would be with systems based on state-machine replication, secret sharing or threshold cryptography.

For these reasons, eventually consistent data replication was chosen as the basis for `Peeranoia`. To counter Byzantine failures, data is made self-verifying, using well-known cryptographic algorithms and implementations of asymmetric cryptosystems. Such public key cryptosystems have the advantage of making key management relatively simple; however, there is the need for a distribution and certification infrastructure, or PKI. Most current PKI solutions are overly complex for our purposes; they also attempt to enforce stricter central control than our autonomy objectives would allow. Our design therefore uses a flat, hierarchy-less model without delegations of authority over cryptographic identities and where human administrators certify all cryptographic keys in one operation.

While it would be simplest to assume a static group composition, this would very much limit the practical maintainability of such an FIC system. Therefore the design does allow for sporadic membership changes, combining group membership management with the key management aspect. Both of these functions are executed in-band using the distributed service itself.

As to communication and data replication, simplicity suggests the adoption of epidemic push-pull mechanisms, using an asynchronous protocol with point-to-point connections. Rumour mongering (as pioneered by Grapevine and Bayou) is the simplest replication mechanism available, but as pointed out earlier it has a fundamental (albeit low) probability of stagnation before reaching all parties, so we cannot offer a strict guarantee for consensus among the group. This disadvantage is outweighed by the gains in algorithmic robustness, as in an FIC scenario, most data is authored and stored by the one party, for whom the data is of importance. For data of a more global scope, our design uses anti-entropy to enforce stricter synchronisation.

To ensure that such a system is able to survive a limited number of arbitrary, malicious failures, it is necessary that data is not just distributed among the group members but also that information *about the distribution process* is exchanged between parties. For simplicity, the design combines the consensus gathering with integrity assurance and the actual distribution mechanisms into one simple procedure

which favours the receiving party with a high degree of autonomy and flexibility: in `Peeranoia`, a sender can only *offer* something to others, never push forcibly. This combination allows us to streamline the mechanisms involved substantially. The data that is collected by a peer during the consensus protocol is used locally to control the distribution process and guide the peer's activities, as well as to judge the level of replication and consensus reached.

Another trade-off involving autonomy and simplicity relates to the lack of global status information: replication by rumour mongering inherently produces some inefficiency as no global state is available to guide the process and in addition to that, most of the possible feedback mechanisms for improvement do introduce new problems as the recipient commonly can not trust the feedback provided; various heuristics using locally available information and verifiable rumours have been evaluated for fine-tuning this trade-off as described in Section 6.4.

These trade-offs lead to a system that offers read-often-write-seldom semantics. Write operations are quite complex, and global consistency takes some time to achieve because of the autonomy sought and the lack of communication guarantees available. On the other hand, the operation of retrieving data from others is very much straight-forward and does not impose high computational or communication costs.

An FIC application can work well with such access semantics because most important data will have to be vetted by human administrators anyway. This introduces high document latencies and there is no need for real-time or distributed operations *on* data; instead a simple repository for a sequence of self-verifying document versions with relaxed consistency across the group suffices.

3.6 Relaxing Consensus

Lamport et al. base their work on the Byzantine Generals problem on the assumption that full and strict agreement on the distributed value must be reached at a particular point in time. But is this really necessary for a practical distributed system? It must be noted that in fully asynchronous systems, there can be no solution for the BG problem that can distinguish between merely delayed or incommunicative and truly faulty peers.

We argue that a relaxation of the level of consensus makes sense for practical networks and applications. For real-world public networks like the Internet, the asynchronous communication case must be assumed, because of both their packet-switched nature as well as the lack of central control. Considering the worst possible case, any non-local communication medium (apart from one that consists of physically separate distinct channels between parties) must be assumed to be asynchronous.

The crux of the impossibility result is to distinguish between faults and communication breakdowns. If one is willing to relax consensus to allow for cut-off correct parties, then the asynchronous situation becomes a lot less problematic. Of course, not all distributed application scenarios can co-exist with localised temporary inconsistencies, but in a data replication situation for FIC purposes, perfect consistency is not a sine qua non. Especially for security-related applications, authenticity is usually of higher value than full distribution.

A relaxed goal would be eventual agreement: given a fully connected set of peers, all honest parties will eventually agree on the same version of a replicated document with high probability; at worst an honest party may have an outdated version of the document or no document at all. In other words, there is no fake data in the system and almost everybody has the same data. This kind of agreement is sufficient to provide a meaningful service, as long as a minimal amount of cooperation between parties takes place.

Relaxing the consistency expectations thus is a simple measure, but nevertheless works in an FIC setting for two main reasons: self-verifying data with strict ownership and caching. Using digital signatures to make data self-verifying reduces the powers of an adversary mostly to passiveness, i.e. not propagating or propagating data only partially. Thus, at worst, data fails to reach a party; but if it is received, its integrity is verifiable.

In an FIC environment, data is generally created and "owned" by one party and is of importance mainly to that owner. Similar to Bayou, the Domain Name System, Usenet and Datta's P2P protocols, we cache data heavily (or more precisely, replicate locally originated data). The owning party will usually store a copy of said data locally, and *forcing* all other parties to have the same data at any particular time is immaterial: what is required is an appropriate level of assurance that the party's data is untampered.

This, however, is easily achievable by limited replication. The level of integrity assurance is provided by involving other parties in the replication process. Active involvement is verified by requiring every party to cryptographically sign every replicated document.

The goal of replication in the `Peeranoia` design is to have a document reach the active state by replicating it and collecting signatures covering its content. Controlled by the policy settings, activeness can (but does not have to) require global distribution of the document. This adjustable level of replication/assurance allows a relaxation of the strict, global agreement of the original BG problem.

The precise aim of document replication and verification in `Peeranoia` can be specified as follows:

1. If a document is available, then it is genuine: the originator's signature matches the content (but the document may be outdated).
2. If the document originator is honest, then eventually the same document will be stored at and certified by a number of honest peers that suffices for the document to become active[2].

This means that there can be temporary periods of inconsistency, e.g. if the originator has injected multiple conflicting documents or if honest parties are not reachable. But as it is usually the owner/originator of data who needs it for operation, the fact of some other honest parties not having received this datum does not threaten the overall function. As a side-effect of accepting such temporary inconsistencies, even a simple system can deal gracefully with network partitions.

3.6.1 Versioning

Obviously, non-strict sequencing of consensus operations means the degree of freedom for providing operations *on* data is very limited: all allowable operations must be idempotent, or temporary inconsistencies will cause permanent global failures and loss of data. Documents therefore are treated as write-once read-often and document versioning is used to approximate read-write semantics. As a consequence, the `Peeranoia` communication protocol supports the creation of new objects and additive updates of the meta-data of an existing document, but no deletion or modification of a document's content.

As we are not ordering multiple requests on data but only distribute independent versions of data, we can make do without strict synchronisation, but we need to deal effectively with said multiple versions.

In our environment, data originates at one party who bears the responsibility of naming and versioning. In general, namespaces belonging to parties would be disjoint and thus any document name and version combination would provide unique conflict-free addressing, but in the case of global documents a global ordering of versions is required to ensure uniqueness: we use a combination of local timestamp at the originator and the originator's cryptographic identity to provide this (somewhat arbitrary but sufficient) total ordering. As long as the originator is honest and does not produce conflicting variants of a document with the same name and version identifier[3], we can deal with any (lack of) message delivery ordering very easily: first, existing documents can not be modified - except for acquiring more signatures in their meta-data. This means that no non-faulty parties will attempt the injection of a conflicting document because of the document naming and versioning rules. Second, we use the document status to control out-of-order situations: peers that are sufficiently out of sync that they attempt to offer a superseded document are reflexively told of the existence of an already active document with a higher (and thus

[2] Assuming the policy is non-conflicting and fulfillable.

[3] A dishonest originator is also dealt with easily, but with different detection mechanisms.

newer) version. Besides this reflexive notification of the newest available document version, a peer can also explicitly query for a document's versions at any other peer (see Section 4.8).

The only aspect where such relaxed consensus is *not* sufficient is the group meta-data itself, which is kept more strictly synchronised by forcing every communicating peer to use anti-entropy.

In summary, the question about the requirement of strict consistency that was posed in Section 1.3, can be answered negatively: for an integrity assurance application that can use read-often-write-seldom access semantics, relaxed eventual consistency is very much sufficient.

3.7 Minimal Cooperation and The “Good Citizen” Criteria

Other fundamental questions in Section 1.3 refer to the possible degrees of autonomy versus any trust assumptions that are required, and this section provides our answer – which is not claimed as being the only possible answer but rather one that follows directly from the controlling design principles discussed at the beginning of this chapter.

There is an interesting dichotomy between paranoidly mistrusting everybody and cooperation between autonomy parties providing a mutual service. As discussed in Section 2.12, there can be no ultimate guarantee that cooperation actually pays off. A small amount of altruism is required for every party involved: *generally*, providing services to others will lead to reciprocal services provided by others. Parties that behave within the expected parameters are observed in doing so by the others, and thus gain “credit” or “goodwill” and are co-operated with in turn because of this evidence. But somebody will have to start this cycle, when no prior knowledge is available. For this reason, in the `Peeranoia` design peers are supposed to warily assume trustworthiness of others until proven otherwise. This allows the necessary altruism and forward cooperation to cover initial request-cycles, after which a more solid base of observations is available to guide future peer behaviour.

One of the goals of this thesis was to identify a minimal set of trust and cooperation assumptions that are required for such a service; Verifiability, Redundancy and Conformance were identified as minimal axiomatic criteria and are detailed in the following sections.

These cooperation criteria are used by all peers to gather evidence of others' behaviour and to determine if they appear to be “Good Citizens”: this provides a local and subjective measure of trust in others' ability and willingness to cooperate (and thus the basis for a party's willingness to cooperate with them).

Note also that none of the criteria specifies how a party is to use documents themselves: it is up to an individual peer to determine what documents to use and what integrity assurance to require. In `Peeranoia` and for reasons of practicality, this local decision is controlled by administrators via the policy, but the function of the replicated storage does not inherently require such control.

3.7.1 Protocol and Format Conformance

This rule is specified as “All parties must conform to the protocol and message rules and formats when communicating with others.”

Format conformance is an obvious requirement: parties not talking the same “language” can not expect to provide any service to each other. If a party violates this requirement (in an attributable manner), then others will eventually stop communicating with it and thus exclude it from the peer group. Not all breaches of conformance are easily distinguishable from real transmission errors. While the `Peeranoia` communication protocol does include this distinction, no local fault detector can be perfectly accurate; a seriously degraded communication infrastructure can therefore lead to peers being misclassified as untrustworthy. As Section 6.2.1 shows in more detail, such attacks against the communication infrastructure at worst result in temporary loss of service.

This criterion is also intended to cover such basics as conformance to the agreed semantics of message types, replying honestly to requests and cooperating in the replication process, as well as conforming to the semantics regarding document versioning and lifecycle (see Section 4.2.2).

3.7.2 Communication Redundancy

The redundancy criterion simply states: "Communication operations must be made redundantly to counter Byzantine failures, until the communication goal is reached."

This rule controls `Peeranoia`'s communication behaviour to balance efficiency, robustness and survivability in the case of faults. The original solutions to the Byzantine Generals problem involve direct communications between all parties, which is clearly quite expensive communication-wise.

The `Peeranoia` design aims for a more flexible and efficient treatment of distributed consensus, which requires a delicate balance between the principles of Mistrust and Autonomy (see also Section 3.6). If a-priori trust in the involved elements is increased, a reduction of the "second-guessing" data exchanges during the distributed consensus gathering is possible, thus improving efficiency. On the other hand, being self-sufficient and not willing to trust any single entity requires high amounts of redundant verification and mostly direct communications, and thus reduces efficiency.

The overall goal is to allow the `Peeranoia` system to tolerate up to a fixed number of subverted parties, but without requiring fully meshed communications. Tolerance in the `Peeranoia` environment means that a document reaches all honest parties intact and with high probability. To enact this balance between between perfect consistence and minimal cost, the `Peeranoia` design uses the notion of a *survivability threshold*: this parameter describes how much parallelism is used during communications, and thus governs the number of subverted parties that can be tolerated.

To achieve such tolerance, all crucial messages must be sent redundantly and randomly to at least one more peers than the threshold. Otherwise random choice of peers can result in targeting only subverted peers who might all conspire to not forward the message.

It is worth noting that this parameter can be used very flexibly to adjust to one's needs. If utmost perfection is required, then the parallelism parameter can be set to the size of the group: thus the `Peeranoia` mechanism degenerates to one of the original solutions of the Byzantine Generals – with all the associated cost.

The redundancy criterion also implies the existence of a distinct communication goal, which when reached, allows peers to terminate operations or continue them with reduced redundancy. For document retrievals, this would be receiving a sufficient number of answers to provide the level of integrity assurance that is sought. For replication operations, the goal is either the document reaching the active state or having reached all peers. Other variants of these communication goals are analysed in Section 6.4.

3.7.3 Verifiability

The verifiability criterion states: "All data distributed within the peer group or stored at the individual peers must be *independently* verifiable."

This criterion covers the secrecy and integrity of individual documents and subsequently defines what actions are to be taken to achieve these. In short, it requires peers to "stand up for their actions".

From the verifiability criterion we can derive the following corollaries:

Signatures: All data passed on to others must be signed by the party handling it. All data must also be signed initially by the originator, thus claiming ownership and responsibility.

Signature Verification: Every party must check all signatures of a document before passing it onwards and must only pass it onwards if all signatures check out.

Policy: All material passed on to other peers must conform to the policy requirements related to document origination.

Penalties: A peer that sends a broken document to another peer must be treated as faulty and is excluded from further communication by the recipient.

The signature verification and the penalty corollaries are essential for a simple yet crucial reason: evidence is not transitive. Only an intact signature conveys any proof of the binding between data and signer identity. Conversely, a broken signature does not prove anything; lack of positive proof definitely does not equal a proof of the negation.

This means that receiving a document which does not match the (globally synchronised and agreed) policy does not allow the receiver to precisely determine the perpetrator of the tampering: is it the last party handling the document, or the one who *allegedly* produced the broken signature? The broken signature can not simply be accepted at face value because that would condemn the alleged signer unfairly, as *anybody* could have produced the broken signature.

Consider the example shown in Figure 3.1, where peer `A` receives a message from peer `X` that includes, among other valid signatures, one by peer `F` which does not match the content. `A` can act locally as the broken nature of the data is clearly evident to `A` and of course strongly implicates the sender `X`.

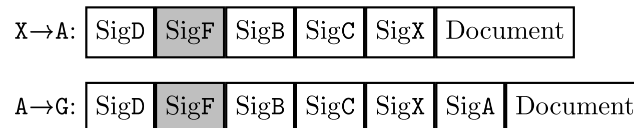

Figure 3.1: A message with broken signatures

However, `A` *cannot* forward this broken message any further, as that would produce rumours that recipients cannot verify. In the example, `A` attempts to send the message on to `G`. `A` duly adds its signature to the broken message, which unfortunately does not help `G` at all: if `A` is honest, this new message would indicate that `F` is faulty. However, if `A` is dishonest, it could have fabricated this "evidence" simply by tagging some random garbage with `F`'s cryptographic identity to cast false suspicion on a possibly honest party `F`. In either case all that subsequent parties like `G` can be certain of is that `A` has sent this message, but whether it was fabricated by `A` or genuinely forwarded is no longer distinguishable.

The autonomy and mistrust design principles provide us with a simple but very strict solution for this situation: only independently verifiable material can be allowed to exist within the system, therefore only material that passes verification must be sent. If unverifiable material is received, then the sender is at fault and can be excluded from the peer group as in violation of the Good Citizen criteria.

The policy corollary deals with the responsibilities of a document originator: technically anybody can inject documents into the peer group[4], but it cannot be allowed that anybody overrides or replaces data belonging to another party. The policy therefore provides guidelines about who owns what (in the sense of who may author or supersede documents, based on the document name). Also, versioning of read-once data requires that the version identifier be totally ordered; violating either of these requirements results in material on the network that is not verifiable (or at least not independently verifiable) and is therefore treated with exclusion of the violator.

[4]Actually everybody *has to* be able to offer documents onwards on behalf of others for efficient replication.

3.8 Communication Fabric

The Peeranoia design, as described so far, resembles a federated altruistic cooperative with all decisions made locally and based on locally observable criteria only, which is very similar to classical P2P systems. Most other P2P designs take this ad-hoc federated localism to the extreme, with the actual fabric of the peer group not under any central control. As a consequence, in such systems a large amount of effort is spent dealing with the ever-changing composition of the group.

Such complete lack of control over the peer fabric is unsuitable in an FIC or other security-related setting, where some degree of trust is directly associated with particular peers' identities. A certain degree of persistence of the group membership and peer identities is required, because the services offered between peers are not anonymous and fully interchangeable.

As stated in Section 3.2, the overall goals of the Peeranoia system are two-fold: providing replicated storage and integrity assurance among peers. Examining these two components from a group membership perspective, it is clear that redundant storage could be implemented simply by replication only. There are mechanisms to allow the originator of a document to learn who else stores that document, and this is roughly the level of services provided in most classical P2P systems.

As such, the storage component of the Peeranoia service could migrate automatically between peers, but integrity assurance ties the identity and reputation of the assurer to the data, and is clearly not identity-independent.

A newly joining peer B cannot *immediately* stand in for peer A as the integrity assurances given by A cannot be transferred[5] into B's situation. It is therefore required to create a more tightly-knit group of peers that can form some level of trust in each other's cooperative operation; furthermore, an FIC setting implies the requirement of some level of human oversight.

Combining these observations, the design adopts global control of the fabric of the peer group, to be exerted by human administrators.

3.9 Document Scope

As mentioned earlier, in the Peeranoia environment all data is replicated and stored by many or all peers, which explicitly includes the originator of the data. Most of the replicated documents are of importance to the originator only, which is a consequence of the FIC setting that the design is geared towards: file baseline data is created and used by the same party. This data is of no concern to anybody but the involved party and its administrators, and thus of local scope.

However, group membership must be controlled globally, which implies one set of information of global scope. For simplicity, the design accomplishes this with a document that combines the list of cryptographic identities of all peers with the policy rules. This document provides all of the *group meta-data*.

That document must be kept in sync across the whole peer group, because it is part of the data that local decisions are based on. The simplest way to achieve such strict synchronisation would be to administer them out-of-band and manually, but considering long-term maintenance, this is neither scalable nor flexible enough.

As the design aims for higher flexibility without sacrificing overall robustness, document replication and integrity certification is applied to the group meta-data as well, treating it very similar to documents of local scope.

There is one crucial difference, which provides the necessary stricter synchronisation of the group meta-data document: on every connection startup between two peers, both are required to perform an anti-

[5] Over time, B can *duplicate* another peer's work and thus become a replacement.

entropy synchronisation of the group meta-data. This ensures that global-scoped documents converge strictly and in minimal time.

For documents of local scope there is no provision for such forced synchronisation, because eventual consistency across the peer group is sufficient for such documents. In this manner, maximum flexibility is gained from in-band handling of the group meta-data while limiting any negative effects that temporary inconsistencies of the meta-data could pose for the overall operation of a peer group.

3.10 Summary

`Peeranoia`'s overall goal is to improve FIC applications by extending the local horizon to using a group of others for better integrity assurances. This is achieved by introducing cooperation among independent parties to provide a service among the parties, but conflicting design directions must be balanced. For example, maximising robustness comes at the cost of high complexity, communication overhead or extensive trust requirements.

To perform this balancing, the requirements of an FIC application have been explored to isolate a set of design principles that supports the overall goal. As design principles we identified Autonomy, Simplicity and Mistrust, and a discussion of the consequences of each was given.

A strong need for autonomy is a consequence of expecting to operate in decentralised environments and using public communication networks. Extensive Mistrust is a requirement for any kind of IDS functionality, and mistrusting data and others implies the extensive use of cryptography and ongoing verification of every observable activity that is related to the overall function. Simplicity is a fundamental guideline of successful engineering, and permeates almost all choices made in the development of `Peeranoia`.

From these three principles, we derived behavioural specifications for all the elements of the `Peeranoia` system: independent peers provide replicated storage and data integrity assurance to each other, human administrators are in charge of controlling peer group membership, restrictions and privileges of peers and extra integrity assurances for important information.

These specifications form a minimal set of “Good Citizen” criteria and describe the minimal amount of trust a peer has to place in others for the overall `Peeranoia` service to work. The criteria mandate independent verifiability of any replicated data, redundancy of communications to provide robustness in the face of arbitrary failures and protocol and format conformance. While not claiming that these specifications are the only way of achieving the overall goals, the Good Citizen criteria were direct consequences of the particular design principles.

Subsequently, more precise requirements for the data that is exchanged between parties have been developed. Most importantly, the use of digital signatures created by every peer handling a document is mandated. There are some limitations related to what information can be exchanged, because negative evidence is not transitive.

Finally, the need for balancing autonomy and human oversight has been discussed, especially in the context of providing operations that involve a notion of trust. Contrasting classic P2P systems and the `Peeranoia` communication infrastructure has shown that human administrators should be in charge of group membership changes. Flexibility is gained by using the same document replication mechanisms for group membership information, with the crucial difference of requiring stricter consistency for this group meta-data.

It remains to be shown, how these high-level characteristics can be translated into detailed specifications of the interactions between all the elements of the `Peeranoia` system. In the following chapter, the foundations provided by the chosen design principles are refined into a precise architecture of our design.

Chapter 4

Design Details

After exposing the design philosophy in the previous chapter, we now turn to the details of Peeranoia's design. While the main characteristics follow directly from the overall goals and the trade-offs depicted in the previous chapter, there are some very important aspects of a more concrete nature which require a detailed coverage.

4.1 Environment and Principals

Conceptually, the Peeranoia system consists of the following elements: the Peeranoia software application, a group of peer computer systems, each of which runs the application, a communication protocol used for data exchanges between peers, and human administrators who control some aspects of the overall behaviour remotely.

The over-arching single entity is the peer group, consisting of any number of individual, independent peers. A peer in our environment is defined as a computer system which has the Peeranoia application installed and active, and which cooperates with the other peers to provide replicated and integrity-assured storage to every member of the group.

Administrators are required for purposes of general oversight and, in the course of that control, need to interact with the peer group occasionally. For this purpose, they are supposed to use computers that are *not* part of the peer group: as administrators have privileges beyond the capabilities of peers, their cryptographic identities are more threatened by a potential intrusion and their key material should therefore not be stored on machines that are susceptible to security breaches. Nevertheless, the computer systems used by administrators will need to have the Peeranoia application installed and available, because administrative oversight is performed in-band, using the same communication mechanisms and protocols as the P2P exchanges between peers.

The Peeranoia application executes within the context of an existing operating system: the application expects primitives for file access, network communication and cryptographic operations to be provided by the operating system. The application design is not specific to any particular operating system, but depending on which asymmetric cryptosystem is employed, may require a multi-tasking operating system.

From the operating system's perspective, Peeranoia is a non-privileged, non-interactive, permanently running application which needs to accept and initiate network connections with others, and which requires access to and control of a certain amount of local storage. The application performs a single peer's part in managing replicated storage among peers, and provides access to the integrity-assured data to a consumer, which is generally an FIC program that also operates on the particular computer system.

4.2 Documents

The object of the overall Peeranoia system is the replicated and integrity-assured data that peers inject into (and request from) the peer group for cooperative storage. Data is controlled at the granularity of an individual document: data that is created by any single peer which chooses a unique name and version for this document.

There are two basic classes of documents: the group meta-data and *ordinary* documents, which encompasses all other documents (including administrator alerts). The difference is the global scope of the group meta-data versus the originator/local scope of others.

All documents are fixed-size and write-once, immutable once generated. The content of documents is unrestricted and for ordinary documents, it is suggested that such documents are made opaque to all peers replicating them (except for the intended user of the data): as discussed in 3.9, most data is of importance only to the originator and thus would be asymmetrically encrypted for this originator. Administrator alerts would be either encrypted to all administrators or be in clear text. The group meta-data clearly has to be available to every peer and is not security-sensitive; therefore it is not stored encrypted. The document opacity for ordinary documents is optional and does not affect the operation of the distributed storage system at all.

Every document has a single content blob and one set of document meta-data. The meta-data consists of addressing information (name, version), size and a signature block. The signature block of every document must at least contain the originator's signature, which spans the document and all its meta-data (except the signature block itself). This mechanism is employed to provide a strong binding between document content and its name and version.

The document meta-data is bounded in size by the size of the peer group. Because of the strict binding between document meta-data and content, most operations related to document replication can be designed to utilise the meta-data only. This has resulted in improved communication efficiency, because the actual document content (which is not necessarily limited in size) only needs to be exchanged very rarely.

During the replication operation, documents acquire signatures from all peers replicating the document and, optionally, from administrators vouching for the document's provenance. This is the only supported operation that modifies a document or its meta-data.

4.2.1 Namespace

Addressing documents by name obviously relies on unique names. Unfortunately, determining the global uniqueness of a name for a group of independent parties involves executing a distributed consensus operation and is therefore prohibitively expensive.

The Peeranoia approach to this issue was driven by practicality and simplicity: in most cases, cooperative heuristics allow for a reasonable compromise between full local independence and global uniqueness requirements.

For ordinary documents, the name needs only to be interpretable to the user of the data. Therefore, a simple naming scheme that includes the peer identity in a document name would suffice. Global data obviously needs to be associated with permanent global names, but in our system there is only one document of this kind: the name peerlist is reserved for the peerlist and policy document (see Section 4.3).

Apart from that reservation, the Peeranoia design itself does not rely on any particular naming scheme. However, it is suggested that a namespace partitioning scheme is used, which incorporates the originator's identity in some form into a document name. This is most beneficial for practical formulation of a policy (discussed in Section 4.4).

4.2.2 Versioning

In Peeranoia, every name is associated with a sequence of immutable versions of a single document. A document name clearly associates exactly one object with the name, but versioning suggests multiple data objects per name: therefore, globally unique addressing is performed using a combination of document name and version identifier.

In order to provide robust integrity assurance, it is necessary to limit the possible operations on a document: for example, it cannot be allowed for anybody to "rewrite history" by modifying old documents or inserting documents that predate current ones as that would be a major vulnerability, which allows an intruder to wipe the traces of the intrusion. Section 4.9.2 describes the exact anti-entropic mechanisms used to counter these scenarios.

Therefore, the design of Peeranoia was deliberately limited to write-once documents, but practicality of course dictates some means for keeping related documents in a time series to share part of the identifier (the name).

The version indicator of a document is numeric, with at least a total ordering defined local to the originating node for ordinary documents and a total ordering that holds globally across the group. In other words, the version must be unique and reflect the creation time of a document version (but only in relation to its prior versions); conceptually it falls between a timestamp and a revision identifier.

The version identifier consists of two components, the local time and the originator's cryptographic key identifier (or any other unique id of the originator). The local time component serves as monotonic increment to ensure that newer documents are actually accepted by the storage system, whereas the peer-identifier component serves to break (unlikely) ties between multiple parties choosing to inject conflicting documents with the same name at the same time. As Peeranoia does not require any particular naming scheme scheme, namespace partitions could be overlapping, which would make inadvertent conflicts possible if only the local time at the originator is used for versioning.

The versioning scheme requires that local time at all parties must increase monotonically; we refer the reader to [76] for a discussion of how to implement negative time adjustments without violating this requirement. Note, however that there is no need for a globally synchronised time in the Peeranoia environment: the vast majority of documents are ordinary ones, i.e. of local importance only, and therefore only locally consistent time is required for correct versioning. On the other hand, there are but a few documents of global nature, all of which are administrator-originated and controlled. Furthermore, even those only need a new version greater than the old one, so that choosing the maximum of observable, consistent timestamps and last version would clearly suffice[1].

4.2.3 Document Lifecycle

Subject for policy restrictions, any peer can create and offer documents to the group for replication, but storage space is limited and historic data, in particular in an FIC environment, is of little interest.

For obvious reasons, direct overwriting of any existing document is not supported in the Peeranoia design, but obviously some means for phasing out data over time is required. Also, not all data can be used immediately after replication: human supervision is necessary for material of crucial importance for an IDS application.

These considerations clearly suggest three distinct states of a document: pending, active and superseded.

pending: A document that has been offered to peers for replication, but which does not yet fulfil (all) policy requirements for an active document.

active: An active document fulfils all policy requirements and can be used in an actual operation on data.

superseded: A document is superseded when it has been replaced by a newer active version.

[1]Note that human oversight is crucial here, as an adversary could trick a fully automated procedure into a runaway clock situation, if the maximum of observed timestamps is used for forming a new version number.

Figure 4.1 gives an overview of the possible state transitions.

To effect these state transitions, only three operations are required in the `Peeranoia` design: a new document version can be created, it can be offered for replication and gain integrity assurances, and it can be superseded by a newer version.

The document states are controlled by a document's replication and signature history and the policy. This data is used as a *local* metric of integrity assurance: every peer uses its set of signature information of a document to determine the document state. Local interpretation of such a metric is sufficient for dealing with ordinary documents because of their local scope.

A newly created and inserted document is in pending state until it has acquired sufficient signatures for the transition into active state, after which the intended user of the data is free to operate on it.

In a security-critical FIC environment, where file fingerprint databases are replicated using `Peeranoia`, human confirmation is required before that new fingerprint data can be used to classify files as untampered: the document therefore must not become active unless a human administrator has vouched for its safety. This control of the transition to activeness is performed using a suitably selective document policy.

In general, most documents will require more than just the signature of the originator to be considered ready for use, and therefore a mechanism for upgrading a document's signature set is required. For simplicity, this is combined with performing attack-resistant replication: whoever receives a document verifies its integrity and then generates a signature of his own; afterwards the document is offered to others with the augmented signature block.

This mode of signature acquisition works also for administrator confirmations: any administrator tasked with the control of a document individually verifies the authenticity and legitimacy of the document and confirms by signing and redistributing. In time, the new information will have been distributed sufficiently through the peer group to allow all peers to classify the document as active.

`Peeranoia` is meant to be an application that is running for long periods, and a means of phasing out old data safely is required. To this end, the design incorporates but one destructive operation on a document, which is to supersede it with a newer version. To replace a document version, a document with the same name but a newer version is generated and inserted into the distributed storage system. Replication and integrity certification takes place, and as soon as this newer version becomes active, the old one is no longer interesting and enters the superseded state.

Superseded material can be removed safely by any peer that is storing it; the overall goal is to have only one version of a document in storage most of the time (i.e. when not transitioning between versions), but there is no mechanism for enforcing this, nor is this enforcement desirable: instead it is expected that not every peer will do this garbage collection and thus a service of ever-ongoing history like Publius or the Eternity Service can be provided easily. This lack of explicit deletion mechanism can also improve robustness in the face of administrator corruption: if independent "watchdog" peers do not purge old versions, then traces of administrator activities *within the system* stay available and provide an audit trail.

The superseded state must only be entered when a newer version of a document reaches the active state, not just when the document springs into existence. This is to ensure that no untrusted or marginally trusted version can overwrite a document. Until a version of a document is superseded, a peer must offer all available versions of the document on request. This means that at any time there can be multiple pending versions in circulation, but at most one active version.

Eventually, some version will become active and at that point all previous versions (active or pending) will enter the superseded state and can be removed to free storage resources. Orphaned documents, where the owner has been removed from the peer group[2], are always treated as superseded documents.

[2] As peer removal is strictly controlled by human administrators, this handling of orphaned documents does not pose any denial-of-service threats.

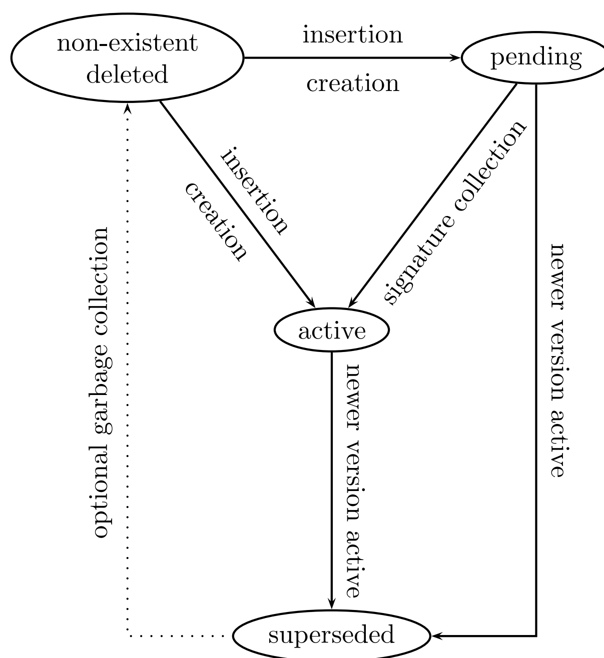

Figure 4.1: Document Lifecycle

4.3 The Peer List

An infrastructure with semi-static group membership was chosen for the design, and therefore every party in the `Peeranoia` system needs knowledge of the involved parties and the document rules; this group meta-data comes in form of a peerlist and a policy document, the latter of which is described in Section 4.4. The peerlist consists of the cryptographic public keys of all peers and serves three purposes:

First, it strictly describes the group membership: only parties whose keys are present in the peerlist can partake. Second, it provides a rudimentary PKI in form of the list of keys themselves. Third, it binds network identities to keys so that peers can authenticate each other and communicate securely.

For our research implementation PGP keys are used, but the design does not mandate a specific class of asymmetric cryptosystem at all: any cryptosystem that supports self-signed keys and attaching a minimal amount of free-form information with the numeric key data is suitable. All keys are required to be self-signed to avoid tampering with the key data.

The peerlist document consists of the list of keys, with each key identifying the respective owner via an attached comment text: a permanent peer is listed by its network identity, while administrators are listed with their email addresses. The reason for this distinction is that administrators should not be doing their work from permanently involved peer machines because of their higher privileges; they are therefore treated as temporary peers who can initiate communications with other peers but are not partaking actively in the document replication procedures (see section 4.7).

4.4 The Policy Language

Not all types of data have the same level of importance and require the same stringent amount of replication and integrity assurance. To cater for such different levels, we incorporated a policy system into `Peeranoia`.

This policy system allows the expression of the administrator- and peer-signatures that a document must acquire to reach the active state. Besides controlling the transition into active state, the policy also serves as a means of regulating document authorship: every document in the system must have a valid originator-signature, which can be restricted to a set of particular peers in the policy. An exact specification of the policy language is given in Appendix A.1.

The policy expresses requirements in terms of three elements: a target specifier, a set of acceptable originators and a clause describing required subsequent signatures. The overall policy document consists of multiple such triples, with the first triple that matches the target document being applied.

The target specifier is a regular expression that is applied to a document's name. This aspect of the policy is responsible for the suggestion of namespace partitioning with peer name prefixes, as it simplifies specifying just what documents a peer "owns" (or has write-rights to).

The originator restriction is optional and gives a list of acceptable originators (by their key identifiers). This serves as a "write-protection" flag for documents. If present, then only documents with an originator signature from this list are accepted for replication by any peer.

In general, this restriction would be set to include the intended user/originator of a document plus some administrators, to allow them to supersede the peer's documents (e.g. in case of the peer being unreachable or dysfunctional for a long period of time). Insertion of an object by malicious adversaries would thus be prevented.

For alerts there would be no restriction, whereas the peergroup meta-data documents clearly require authorship by an administrator: a normal peer should certainly not be allowed to modify the group membership or the policy itself.

The third element of any policy rule are boolean formulas which operate on sets of peer key identities. The sets of keys can have an optional numeric quantifier to offer more precise control of evaluation of the sets: without quantifier, presence of any signature from the set constitutes logical truth; if quantifier n is present, then truth is defined as existence of at least n signatures from that set.

Using these elements, the policy language allows the expression of statements like "this document must reach 70% of all peers and be signed off by at least 3 of these human administrators or this single other administrator".

Clearly, such flexibility comes at a price: for example, it is trivial to formulate policy requirements that are absolutely unsatisfiable; being overzealous and requiring all peers' signatures prevents a document's activeness in the case of even a single unreachable peer and thus destroys robustness. On the other hand, letting crucial data become active without sufficient checks and balances could produce unintentional privileges for peers and so on.

The policy language is fairly crude and possibly over-expressive, and burdens the administrator with the need for precision, especially as it is at the centre of all decisions in `Peeranoia`. For example, the negation operator is very likely unnecessary and mostly included for completeness' sake and to provide flexibility for structuring a pool of administrators. Also note that the policy sets the *minimal* requirements for state transitions, but during the replication process usually most if not all peers' signatures are acquired.

The policy language provides support for controlling all three kinds of documents[3] flexibly. Some possible guidelines for specification of a policy are outlined as follows: ordinary documents like file fingerprint databases or a peer's configuration files can be set to be be less stringently guarded, i.e. with just one

[3] `Peeranoia` distinguishes between ordinary, global and alert documents.

administrator signature necessary for activeness. Alert documents should require only the originator signature.

The signature rules for global documents should require at least some administrator signatures, so that a rogue peer cannot supersede them. However, these documents should not *only* require administrator signoff but also a high degree of replication: if there was a dishonest administrator, he could create different "network pockets" that receive different documents. With a rule like "at least 70% of peers must have signed" the probability of such an administrator-induced network-split can be minimised. On the other hand raising the bar too high makes denial-of-service easier.

Conceptually, `Peeranoia` treats the policy just like any other ordinary document, to be transported in-band, stored at every peer and subject to the different document states. However, the group meta-data documents like the policy must be treated slightly differently from from ordinary documents where the transition from pending to active state is concerned: unlike ordinary documents, where activeness on a global scale is not crucial, for the policy this is a more complex transition.

Updating a policy or a peerlist document presents some subtle problems: does the new policy have to be active *in terms of the old policy* or merely be self-consistent? If the old policy was broken and unsatisfiable, how can we ever override it? And which of the peerlists shall be used? What about adding new peers?

Because of this problem, the policy and peerlist are coupled to form the whole of the group meta-data in one document, and activeness is judged by the pending policy/peerlist's rules itself. As the policy and peerlist define who may do what, they need stricter synchronisation across the peer group: the communication protocol enforces this via anti-entropy on connection startup.

4.5 Trust Levels and Signatures

Reviewing related works in the areas of distributed consensus and integrity protection suggested an initial design where the document replication and any trust assurances are treated separately. In this scenario, replication would have been a very simple and inexpensive flooding operation similar to Usenet News[77].

Using such a history-based flooding mechanism, every peer that receives a document would add his entry to the history, retransmit that and stop further replication when the history shows sufficient diffusion of the document. The rationale for employing a history mechanism was to allow for an indirect flow of information, which can improve efficiency. With only direct exchanges between parties, every party must talk to every other party because nobody would know who else is still in need of this document. No information that is exchanged between two parties directly could have any transitive meaning to subsequent others.

A path header can provide a partial travel history of a document, and in a scenario with only benign failures this would be sufficient to control the replication. However, in the `Peeranoia` setting, arbitrarily malicious failures must be expected, and a simple, linear path listing does not provide us with trustworthy information if dishonest parties are involved. On the other hand, there is a definite benefit of minimising the required data exchanges, if a transitive transfer of knowledge is taking place. A means of integrity verification of the replication history is clearly required to resolve this conflict.

There is one aspect of `Peeranoia` where integrity verification is involved extensively already: providing *document* integrity assurance requires the involved parties to vouch for the document's integrity with some form of digital signature. While it is possible to treat replication and document integrity separately, it is obviously beneficial to combine signing documents for integrity verification with signing the *document's history* to prove that a party have received it, and this is exactly what the `Peeranoia` design evolved into.

This verifiable replication history allows fine-tuning the control of the replication process heuristically by targeting parties that have not yet signed (as far as a peer knows locally) and who is assumed to need this document; the details of this feedback mechanism are covered in Section 6.4. This also allows the detection of inactive or uncooperative peers over time: if a peer does not appear in the path history of a document despite having been contacted and offered the data successfully, then a problem may very well be assumed and an administrator would be informed. A similar benefit is gained when conflicting documents are received: the replication history tells a peer *exactly* who needs to be notified of said conflict.

The combination of distribution operation and integrity assurance collection into one procedure allows a vastly simpler communication protocol, which needs to provide only a single main operator that is used almost exclusively for all of the replication work. With this operator, a peer *offers* a document's meta-data to another party.

Overall this means that every party handling a document will have to sign it at least once. Also, in an FIC environment there is the need for human administrators to further certify important data as being acceptable, which the design provides by having administrators sign documents just like peers involved in the replication would do.

The replication history itself must be integrity protected to be trustable and thus of any value. Otherwise, any handler of a document could modify the path information undetectably and thus affect the decision basis of subsequent peers. In general, any method of cryptographically signing a document must take its meta-data into account to be useful. Following this approach, the `Peeranoia` design specifies that any party's signature of a document is to also cover the replication history *as much as possible*. In the next sections we discuss the challenges and details which pertain to this binding of document content and replication history.

4.6 The Signature Tree

The linear nature of the originally envisioned path history is at odds with redundant distribution which involves multiple disjoint paths for a document to travel. On further reflection, the term "Path" is rejected as imprecise because the information resembles more a tree.

If the transmission history of a document is collected as a series of peers who record their handling and who they got the document from, then initially a number of disjoint linear list of peers are created. Bearing in mind that data exchanges must be conducted redundantly for robustness and that peers decide on subsequent targets autonomously, at some point these lists will converge because of cross-talk between the lists. At this point, a merging of this converging list of lists into some compound data structure is necessary.

The most natural data structure here is clearly a tree that is rooted at the originator, with the parent-child relation defined as the initial reception of the document by the child, transmitted from the parent. This notion of a signature tree was influenced by OceanStore's mechanism for fragment verification (see Section 2.9), where integrity verification for storage fragments is required. In both scenarios, the stated and required outcome is that the receiver of a datum does not need to trust the sender but instead has a means to independently verify the correctness of the datum.

Consider this example scenario: `R` creates a document and replicates it to `A` from where it travels to `B`, then `F` and `G`. The originator also sends the document out via `C` in parallel, where an administrator `D` picks it up and certifies the document. This signature is sent onwards to `F`, where the two distribution paths converge for the first time.

Figure 4.2 shows this situation and the signature knowledge at peers `F` and `D` before the distribution paths converge. `F` now sees this secondary path of signatures, all important and disjunct from what it received so far. `F` must accept and use that path, otherwise the essential administrator signature will

be lost, and F also must pass this signature-path onwards so that other peers can gather the uptodate certification information. These requirements clearly indicate the need for a tree data structure.

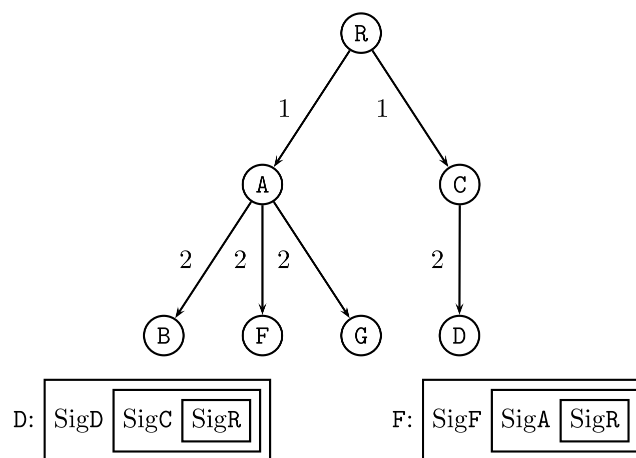

Figure 4.2: Example Distribution and Signature states

The next interesting question is, what the individual records within the tree should contain: cryptographic signatures of the document, but what else? And how often does each peer have to sign?

The original signed solution to the Byzantine Generals problem requires that multiple linear overlapping signature paths be transmitted between parties and that the exchanges must include all parties, point-to-point and synchronously. In an unreliable real-world environment, this requirement is hard to guarantee and we are therefore looking for practical simplifications.

We would like to minimise the number of messages exchanged (and therefore to be verified), as well as the number of signatures a peer must produce, as signing cryptographically is a very expensive operation.

Ideally, we would like to sign a document once only, and from this, derive "verifiable rumours" for propagation which contain all the signatures received in the past in one exchange, and with every exchange as complete a snapshot of the signature state at the sender as possible.

Unfortunately we face a substantial problem here, as we would have to detect and counteract the possible *lack* of evidence: using cryptographic signatures limits the potential mischief of an adversary to not communicating data in total or part, but absence of a signature does not prove anything by itself.

In the classic Byzantine Generals problem, the absence of messages is detectable because of the strict timing guarantee for the communication medium and can be used as decision criterion, but this kind of guarantee can not be expected to hold for general-purpose packet-switched public networks: not receiving a message from a peer does not necessarily and immediately indicate malicious failure of the sending party. Also, the collapsing of signature exchanges we seek is necessarily in conflict with the mostly full mesh communication paradigm of the BG problem.

Therefore it is necessary to look for an alternative means of efficiently exchanging signatures with as much integrity protection as feasible. Unfortunately, the following negative result limits our efforts in that direction to a mere compromise.

Claim 1 *In an environment of autonomous parties there can be no ultimate guarantee that the sending party is transmitting its known signature set* completely*.*

Proof: The sending party has control over its communication, and can of course modify data in transit. Modification of data which is covered by the sending party's signature, is not detectable as the sender can always re-generate the signature to match the data. Material that is covered by signatures of other parties is not modifiable without detection, as the sender cannot fake other parties' signatures; however, the signatures themselves are only *inert* data. The lack of synchrony requirements in the `Peeranoia` environment results in unordered events, and thus without means of inferring from local knowledge only that a peer must have received information previously which it does not propagate onwards. □

In short this means that feigning ignorance consistently cannot be detected without a governing superstructure. If absolute certainty was required for a peer not sending curtailed data onwards, then distributed consensus would be necessary *for every single signature exchange*. This would effectively reduce `Peeranoia` to strictly sequenced state-machine replication, negating all benefits of relaxing consensus.

In the following we discuss a few possible approaches to binding signatures more closely and finally present the detailed design chosen for `Peeranoia`.

4.6.1 Signing Once Only

Signing a document just once at initial reception leaves the problem of how to distribute the signatures attached to a document. The simplest way of representing such a set of signatures is a list, but this does not bind the signatures together transitively and it is very much prone to tampering: a corrupt peer can simply strip individual signatures from the list before propagating the document onwards, and such behaviour can not be proven independently if it is done consistently.

Therefore the simple list is extended to generating overlapping signatures, once at first reception, which offers increased tamper-resistance: signatures within the overlapping sequence can not be removed by themselves, as the enveloping signatures later in the sequence would be invalidated. A malicious peer would have to remove *all* signatures down-sequence of a signature it wants to remove.

But how much overlap is feasible? As mentioned earlier, the ideal result would enclose a complete snapshot of a peer's state if possible.

Claim 2 *The goal of signing only once is not compatible with a signature representation that contains the full signature state at the sender.*

Proof: We construct a trivial counter-example that starts out with a linear forwarding of offers and leads to unresolvable conflicts.

Assume an originator `R` sending a document to `A`, who signs it and sends it onwards to `C`, who hands it to `D`. A linear distribution is not in any way fail-safe because it is not redundant enough, therefore `R` also sends to `F` in the first step, who happens to forward the offer to `C` where it is received after all previous offers. This state of affairs is shown in Figure 4.3, together with the signature states at peer `C` after the second and fourth offer.

Up to now, everybody could easily verify the chain of signatures, but now `C` has the problem of deciding what to do with the offer from `F`. This offer is consistent by itself and the signatures it contains (Sig`R` and Sig`F`) verify correctly. `C` needs to merge in these signatures which were learned after it created its one and only signature. Also, `C` is supposed to provide the strictest possible binding of all signatures for further offers that it sends out.

The goal is to keep the signature block small, containing just a single signature by every peer. To this end, we assume that only one signature per peer is present in a signature set.

Assuming that `C` has a method to re-sign the merged tree after incorporating `F`'s offer, the signature SigC' as shown in Figure 4.3 is created. This is passed to `E`, who signs and offers it back to `D` where the conflict becomes evident. `D` has received two signature sets, each in itself consistent but incompatible and unmergeable: different signatures by `C` are present in each offer. Figure 4.4 shows this situation.

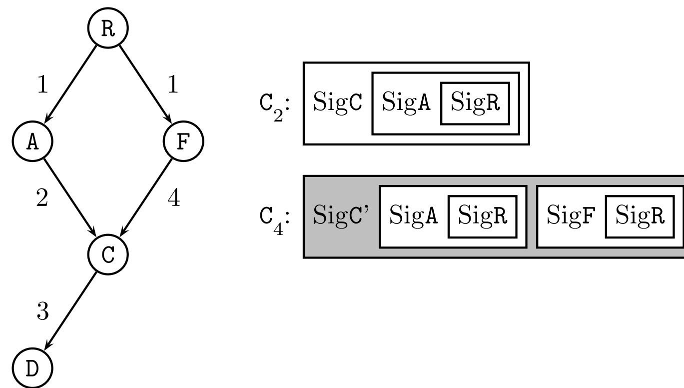

Figure 4.3: Signature Knowledge at Peer C after second and fourth offer

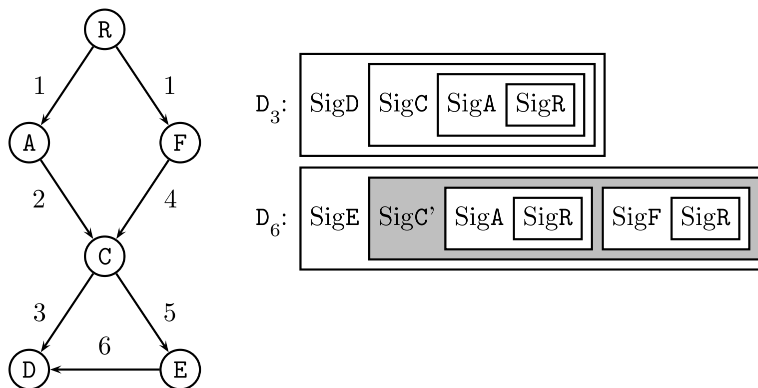

Figure 4.4: Signature Knowledge at Peer D after third and sixth offer

This situation can not be reconciled, unless multiple signatures by a single peer are used. The first signature made by C is used and wrapped by D's signature and thus must stay available or nobody will be able to verify D's signature. On the other hand, the newer signature by C that was presented by E is part of and required by E's signature. □

In more abstract terms the problem is this: knowledge of others' signatures at a peer increases over time as it collects offers from others. If every new offer is to reflect *precisely* the full state of knowledge at the sending peer, then new signatures for every such state must be generated. Replacing one signature over and over does not work as others need to include a peer's signature in theirs for binding reasons.

Merging a newly discovered tree into the local state is impossible, if the receiver's signature must cover the new tree: the old signature cannot be replaced for the reasons explained above and the peer cannot sign again.

Thus, with a single signature per peer, the signature can not always span everything known locally when the peer offers information onwards.

4.6.2 Multiple Signatures over Leaves

As an alternative, the 'sign once only' restriction can be relaxed. For this variant, it is assumed that everybody signs every leaf of the tree before passing it onwards, in an attempt to guarantee that what a peer forwards is the latest knowledge at the peer.

Using the sequence of exchanges from the previous example, C now offers the initially gained set of signatures to D, later learns from F and adds its signature to that branch as well, and offers both to E, who signs twice beneath C and offers this to D. D now has three branches to sign: the first learned path from A to C and the two paths that E has offered in the last stage. Figure 4.5 shows the signatures at D and E.

It is indeed possible to do this and merge paths back together, but is anything gained? Unfortunately the answer is negative; the multiple signing has only reduced efficiency dramatically. As proof we consider D to be dishonest: D could simply decide not to propagate the branch R→A→C→E (the first listed in Figure 4.5), and nobody except E (who knows that it offered that to D) would be any wiser.

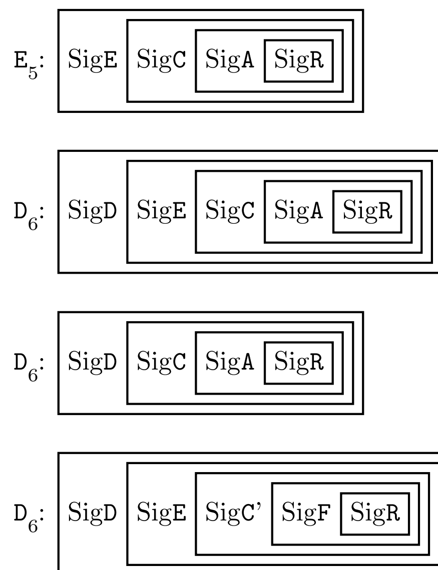

Figure 4.5: Signature Knowledge at Peers D and E

4.6.3 Multiple Onion-shell Signatures

Clearly, multiple *discrete* signatures do not provide any benefits, but what about signatures that envelope the state in the style of an onion shell?

Again using the exchanges of the previous examples, the actions of peer C are considered. After the fourth offer, C has the paths R→A→C and R→F→C to merge and sign. Assuming that C can do that, C binds the branches together with SigC, offers that to E who signs the tree and offers that to D, who has received the linear branch R→A→C→D earlier. D would have to sub-sign in a way that covers both its original signature state and E's offer, which appears possible if not exactly simple.

However, this is still extremely inefficient as every offer now requires a new signature creation and these efforts have not produced much gain: some signatures can still be removed without a trace.

Assuming full signatures over the whole tree whenever it is sent onwards, a dishonest party can peel the onion layer by layer until the one signature that it does not want to pass on is outermost: strip it,

then sign the reduced onion as "genuine" and pass it onwards. If this is done consistently, the unwanted signature and all binding signatures will not travel via this dishonest peer. Clearly this would result in massive reductions of signatures and might be detectable over time, but multiple signatures by themselves cannot prevent this peeling. A peer downstream simply does not have any means of knowing that some peer has signed three times and that a dishonest party has peeled off two of these later.

In our example, `D` could just ignore the branch that involves `E` and keep propagating the older, linear path `R`→`A`→`C`→`D`.

Overall, this variant results in a horrible processing cost, an ever-growing signature state that must be stored and transferred and lots of extra complexity, specifically to figure out when to stop sending further offers. This basically degrades the setup to the non-signed, flooding Byzantine agreement solution and is clearly non-viable.

4.6.4 Hierarchical Signature Tree

As shown so far, neither signing once nor multiple signatures can provide an ultimate guarantee of an offer containing the full, latest knowledge at the sending peer. This leads to a trade-off in the design for signature representation and binding, which combines the benefits of signing only once with a certain degree of tamper-resistance for the transfer of signatures. This is achieved by hierarchical signatures, which span a subset of others' signatures as present at signing.

In `Peeranoia`, every peer must sign once upon first reception of a document. The signature covers the document content, name, version and size as usual, and also the first observed distribution path, as given by the signatures "up-tree" of the receiving peer. Every peer's signature covers all the parents' signatures up to and including the originator signature. Henceforth the terms "originator signature" and "root signature" will be used interchangeably.

Figure 4.6 shows the accumulation of signatures at a peer `F` over time: after the second round of offers, `F` has received the datum from `A` and wrapped it in its signature. Step 3 sees `D` offering its knowledge, consisting of signatures by the originator, `C` and `D`. This new information is verified and then merged into an overlapping tree, without `F` signing again.

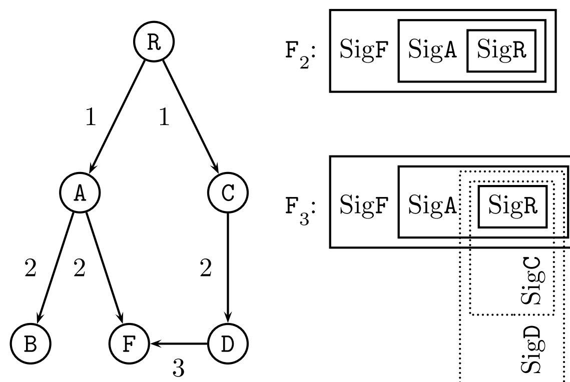

Figure 4.6: Accumulation of Signature Tree at Peer `F`

As subsequent document exchanges are not embedded in new signatures, the design gains the benefits of a signature block that is bounded in size by the group, as well as a caching capability for signature verification (as peers' signatures are not modified). Additional benefits are moderate but still significantly compared to a simple list of signatures: for example, the topology of the signature tree allows for heuristics

to fine-tune the propagation based on this verifiable information. As the signature tree does not *precisely* correspond to the sequence of data exchanges, these heuristics can not be perfect, but nevertheless provide substantial improvements as shown in Section 6.

Tampering with the replication process by removing signatures enroute is made harder (but not impossible as pointed out earlier), and in section 6.2 we argue that the possible negative effects of such tampering are very much overshadowed in severity by outright non-communication anyway. Also we feel that while being a trade-off, this improvement should be considered good value as it incurs no extra cost: every peer must generate at least one signature anyway, and the composition of the material to be signed is straight-forward and does not involve any cryptographic operations.

4.7 Communications

Assuming the utilisation of a public network like the Internet, clearly we cannot trust that communication infrastructure.

All communication between peers is made secure and private: public key cryptography is used for authentication and agreement on a session key and private key cryptography is used for efficient communication thereafter. Communication is one-to-one only (in contrast to many other distributed systems that use multicasting heavily), using a very simple protocol whose precise specification is given in Appendix A.2.

Apart from the connection initiation phase, the protocol is fully symmetric and no distinction between the two parties is made. The protocol is intentionally kept very simple and, as a consequence of this design goal, does not implement any error recovery mechanisms: communication failures or timeouts are dealt with by dropping the connection. This reduces possible timing dependent race conditions and other deadlocks.

From a security perspective, the protocol is fairly straight-forward: asymmetric cryptography is too slow for bulk traffic. Furthermore, the peers' asymmetric keys are expected to be long-lived and cannot be used directly for this reason. The standard solution for this is a hybrid cryptosystem with ephemeral symmetric session keys. This has the added benefit of providing perfect forward secrecy as the session keys are randomly chosen for every session.

As every peer knows every other peer's address and public key it is easy to safely agree on a session key. In our scenario, the connection initiator chooses a random session key, signs and encrypts it to the contacted party at connection startup. As only the responder can decrypt the message and as only the initiator can have signed it, this is clearly as safe as the underlying asymmetric cryptosystem.

We aimed for highest efficiency for the connection startup with the minimal amount of overhead in the protocol. Therefore the protocol was designed without any elaborate hello/acknowledgement phase during the startup. Instead the connection initiator is required to supply the selected session key to the responder in the very first message. As `Peeranoia` has to deal with "roaming" peers (the human administrators) who are connecting only temporarily and whose network identity is not static, the responder is not able to securely propose a session key as it does not know the identity of the initiator at that stage. Therefore the initiator must supply the initial session key. Incurring the extra complexity of a challenge/response scheme to negotiate a key in multiple stages was rejected, because it would add overhead to the startup and introduce a lot of potential for timeouts and mischief.

As soon as the responder has decrypted the session key and verified the initiator's signature, the distinction between the two parties is abolished. The connection is deemed ready for operation and all subsequent messages are symmetrically encrypted using the session key.

4.7.1 Low-level Protocol

`Peeranoia`'s communication protocol is expected to work over any bidirectional bit-stream, but works best with a robust protocol like TCP. The protocol has two layers, a low-level "wire" format that simply consists of a fixed-size message length identifier and a binary message, with the length sent in clear-text and the message encrypted with the session key. Figure 4.7 shows the protocol message formats.

Every message (except the initial submission of the session key) transmitted via the underlying communication stream is in this format. Peers exchange such messages unidirectionally, without any notion of state. A connection is terminated by simply directing the underlying communication mechanism to disconnect; there are no quitting notifications of any kind. Any communication problem on this level simply results in the connection being dropped.

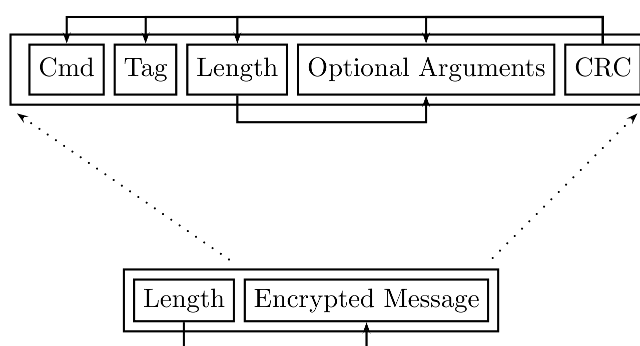

Figure 4.7: High- and Low-level Message Formats

4.7.2 High-level Protocol

The higher-level protocol implements document-oriented operations. It is symmetric in nature, with tagged messages (similar to the IMAP protocol[78]) to allow interleaving of multiple requests. Requests and responses are associated by their shared tag. This allows the protocol to be full-duplex and makes it very timing-insensitive. The format of protocol messages is very simple as shown in Figure 4.7, consisting of a message tag, a command, an optional argument block and its length and a cyclic redundancy checksum (or CRC) that covers the previous message fields.

The checksum is included on the higher protocol level in order to detect messages that were garbled during transmission or other problems related to the symmetric encryption on the lower level. Because the checksum is only used for error detection and not involved in integrity protection, it does not have to be of great strength and common algorithms like CRC32[79] are quite sufficient. If the checksum does not match, `Peeranoia` drops the connection: the data block has proven to be not decryptable which might be a key agreement gone wrong or an indication of a unrecoverable network problem. In either case there is no simple way to reconcile.

After a connection has been established successfully, the two parties must exchange the group meta-data within a short time window. If this procedure fails or is not executed at all, the connection is dropped with extreme prejudice as the other party has clearly demonstrated its being in breach of the Good Citizen criteria.

The reason for this requirement is that peers' cooperation decisions are made based on this group meta-data, and therefore this information must be kept as consistent as possible. In this step, anti-entropy (as discussed in Section 2.8) is performed on the meta-data, which provides the strictest synchronisation and

also fastest convergence upon changes that can be achieved in an environment of autonomous parties. The procedure is performed using separate, mutual offers of the `peerlist` document.

Subsequently, peers exchange messages in the depicted format at will, interleaved using the same tag for request and response (where the command elicits a response). As on the lower protocol level, most communication problems simply lead to a connection termination (with the possible exception of timeouts for responses to requests, which the requestor may reissue with a new tag).

On the higher protocol level, `Peeranoia` offers operations for requesting documents or their meta-data from the other party, for offering document meta-data for replication, and some minor connection-related maintenance and debugging mechanisms. In the following, the term *signature block* is used for a *document*'s meta-data, which consists of its name, version, size and the document's signature tree in a binary, linearised form.

4.7.3 Protocol Message Types

The `Peeranoia` protocol is implemented using nine different messages, of which three are minor maintenance provisions and two more are responses.

REKEY: To switch to a new session key, either peer can issue a REKEY command to the other side at any time. Rekey messages bear the same payload as the initial key exchange (which is a new session key, asymmetrically encrypted and signed), but as the rekey message is part of the higher protocol level, the message is symmetrically encrypted with the old session key.

The issuer switches to the new session key immediately after transmission, while the receiving peer does so after verification of the payload. To avoid simultaneous rekeying (which is not supported as it leads to a race condition), the protocol specification suggests that peers rekey after a configurable period plus or minus a random fudge factor[4].

It is also suggested that the receiving peer initiates one PING/PONG sequence (see below) upon connection establishment or rekeying, in order to detect problems as early as possible before actually using the communication channel.

PING and PONG: These diagnostic helper messages are used to verify that communication sessions are still alive and can also be used to produce filler traffic to disguise actual operations. Upon reception of a PING message a peer is expected to respond with a PONG message. As with ICMP Echo Requests, the payload of either message is ignored.

GET and GETANSWER: These two messages provide read access to documents. The GET message addresses the requested document by name and timestamp, and the peer servicing the request must answer with a GETANSWER message which can be empty to indicate a negative result. If not empty, the GETANSWER payload will be the document and its signature block.

HEAD and HEADANSWER: Similar to GET and GETANSWER, these two messages are used to retrieve just the signature block of a document. One difference is that HEAD supports addressing of documents by regular expression, and HEADANSWER can contain multiple signature blocks.

This wildcard addressing feature was included so that a peer can request a quick state overview from another peer; this allows for fast pull resynchronisation upon a peer's rejoining the group after extended outages.

IHAVE: The IHAVE message is used to offer a document to a peer. An IHAVE message contains the signature block of the offered document, not the whole document itself (to avoid easy denial-of-service).

[4]Note that the risk of simultaneous rekeying is not completely removed. To do so, the protocol would require slight modifications, e.g. allowing only the connection originator to initiate a rekeying.

IHAVE messages do not have a direct response message associated, but if the receiving peer is interested in this document, it will follow up with a GET.

4.8 Document Retrieval

In a `Peeranoia` group, peers can acquire documents in one of three ways: documents can be generated locally, they can be received as offers during a replication run or they can be actively requested from another peer.

Usually a peer that is well-connected would learn of new data during replication runs (as detailed in the next section); if, however, there is a network split or downtime or other problem causing this peer to miss updates, then it can still easily resynchronise with the peer group by using the HEAD and GET commands.

The protocol offers retrieval commands for both meta data and full documents. The semantics of GET and HEAD are simple: from the request receiver's perspective, they provide remote read access to locally available, named documents on request. The addressing of specific documents for HEAD and GET commands includes both document name and version, but there are situations when a peer needs to get information about just any version of a document: for example, an administrator may very well have to query for all alerts or all documents pertaining to a particular peer, or a peer is trying to catch up as noted above.

To support this, both GET and HEAD support two extra (reserved) version identifiers to retrieve all versions or just the active version of a document. Furthermore, HEAD supports regular expressions in the document name argument. It was decided that the full expressivity of regular expressions is warranted when querying names, as name spaces are likely partitioned. For the numerical version identifier, the simple "all available versions" wildcard is sufficient.

The need for the GET command is obvious: retrieving a document and all its meta-data is essential for verifying that local information matches the distributed storage. If the document asked for exists and is up-to-date, the behaviour of the requestor is straight-forward.

The rationale for the HEAD command is a bit more subtle: retrieving information about existing material without transferring potentially huge amounts of data is valuable for flexibility, but is it *necessary* to respond with the whole signature block?

The reason for this behaviour is efficiency: a HEAD command that only responds with document names and version would indicate the existence of a document but that indication would not be *directly* verifiable. The requester would have to follow-up with a full document retrieval to ascertain the document state at the other party. Therefore it makes sense to contain everything but the actual document body in HEAD's answers. It all comes down to mistrust again: a signature block contains positive proof of integrity (if the document body is available), while a mere name does not.

For robustness and resilience in case of arbitrary faults, a peer will always request data from multiple sources (when possible) in parallel and compare their responses with local data.

In the case of GET, the serving peer must respond with the document body and the signature block if the requested document is available (and verified) or with an empty response indicating non-existence. In the case of retrieving documents, the document state is of interest only to the requester, not the suppliers: the requester being the intended user of the data must have its correctness criteria fulfilled.

The interesting aspect is how to deal with differences between states at requester and supplier. If the document does exist on the requestor's side but the requestee does not have it, then a reflexive offer via IHAVE (see the next section) is triggered in order to improve convergence. The other party will fetch the full document; it also may start a new replication run if the document checks out but is not already fulfilling all replication requirements. Similar activities take place if version discrepancies become

apparent; either reflexive offers will be issued to bring outdated peers up to date or the requester itself acquires the newest active document version and continues the assurance gathering with that version.

4.8.1 Handling Requests for Non-existent Documents

From the requestee's perspective, if a request is received for a document that does not exist locally, the possibilities are to deny knowledge or to query other peers for this document. The `Peeranoia` design uses the first choice as being simpler but at the potential cost of slower convergence. While considerable thought was given to the benefits of proactive "read-ahead", it was decided that the disadvantages outweigh the benefits: an adversary could simply flood peers with requests for non-existent documents and thus send peers hunting these ghosts, which could potentially exhaust all available resources.

To avoid this, the requestee first ascertains that the request is genuine, which requires inspecting not just the signature block but also the document content in question. This can be accomplished most easily by following the security tenet of the suspicious party always generating a challenge first before doing anything else: After receiving the unserviceable initial GET request, the requestee sends a negative response. This negative response challenges the requestor to prove its legitimacy by reflexively offering a new version of the document, which the requestee would retrieve and start replicating if signatures check out. This flow of events can be handled without any extra decision code and is therefore simplest and most efficient.

Generally, peers requiring documents or integrity assurance will check the local cache first and verify its integrity. If the document is not available locally, the peer does a GET request to another peer at random, thus retrieves the document once only (unless the response is negative or unverifiable), and verifies the consistency of document and signature block. Subsequently, only signature blocks will be requested to back up local interpretation.

If any of the retrieved data is corrupt, the connection is dropped and the supplier of the corrupt information is blacklisted locally. Note that "corrupt" in this context means that the data in question includes broken signatures or similarly unacceptable data, but has passed the CRC verification of the lower-level protocol. This proves conclusively that the actual sender has transmitted the corrupt material as received, which in turn indicates that the sender is unreliable and untrustworthy. The peer therefore should not be cooperated with until administrative intervention resolves the problem.

4.9 Document Replication

The conceptual basis of the `Peeranoia` document distribution protocol is a hybrid push/pull epidemic algorithm, similar to rumour mongering as pioneered by Bayou but with the crucial difference of transporting *verifiable* rumours.

Any peer can produce a document and offer its signature block as an *abstract* to other peers. The recipient of the offer then decides how to proceed, based on policy and local criteria: every offer received triggers a number of new offers sent out by the recipient, until the replication is considered finished and terminated (see Section 6.4 for detailed termination criteria).

Exchanging an abstract of a document instead of the whole document is beneficial for efficiency, because the abstract is limited in size and transports all information that is required after an initial document retrieval. As highly redundant message exchanges are required to counter Byzantine failures, offering the whole document would be a vast unnecessary cost and also simplify denial-of-service attacks. The flooding mechanism shares some properties with Usenet News and the NNTP Protocol[77], where a "Path" header is part of every document and provides autonomous sites with a history of the article's travels so that the recipient can determine the route the article has taken and base further distribution activities on that knowledge.

From a protocol perspective, this all revolves around the IHAVE command (whose name and semantics were borrowed from the NNTP protocol), which is asynchronous, reply-less and has two main purposes:

- in the initial phase of document injection: notification of the existence of a new document (version) at the sender
- in the subsequent consolidation phase: transfer of a known document's signature block to spread new signatures

This simple expedient of combining all necessary mechanisms into one provides substantial advantages, as the sending party does not need to know anything about the recipient: the offer is self-contained as much as possible without being susceptible to denial-of-service attacks.

4.9.1 Initial Injection Phase

When a peer `B` receives an offer for a previously unknown document from peer `A` as shown in Figure 4.8, `B` will first briefly inspect the signature block and verify that the document in question *can* conform to the policy, i.e. that the root signature matches the authoring restrictions for the document name. Note that the signature block at that time is inert and not sufficient to fully verify correctness: while the document content is covered by every signature in the block, it is not yet available.

If the recipient `B` is interested in the offered document (which it generally should be, see 6.4 for exceptions), then `B` will reflexively issue a GET request to peer `A` as a follow-up, in order to acquire the data that was referenced in the offer. Note that this request is only transmitted to the sender of the offer, not any other party: as discussed previously, it is sound security practice to delay acting until the suspicious party has verified legitimacy. In this case, the GET request serves as the challenge to the original sender to prove that it has the document which it claimed to have. Until that GET request has been completed successfully and the peer is in possession of a document that can be verified with the signature block offered earlier, no further replication efforts are made and judgement of the offering party's correctness is suspended. When the document has been completely received and verified, peer `B` continues the replication process by assuming an active role in sending out offers, as depicted as step 4 in Figure 4.8.

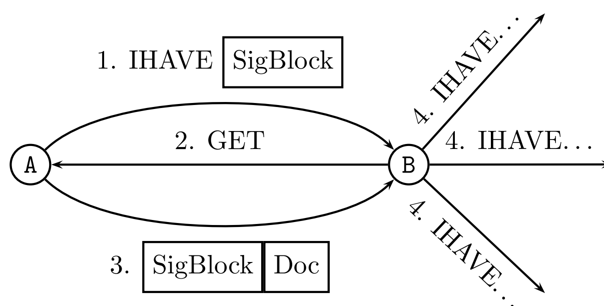

Figure 4.8: Initial Document Injection Phase

This multi-step procedure is the consequence of the design not including a request type to push full documents, for reasons of autonomy and efficiency. The autonomy principle in the `Peeranoia` design implies that operations should rely on cooperating parties instead of trying to coerce peers into cooperation. Also, while detecting a dishonest originator would be simplified with self-contained offers, the higher communication efficiency of signature-only offers is of higher importance. Also, it is shown in

Section 6.2 that dishonest originators can be detected almost as easily with signature-only offers. This leads to the adoption of leaving it up to the recipient of a "polite" IHAVE offer to react accordingly and appropriately.

4.9.2 Consolidation Phase

After a peer has gone through the initial document acquisition phase, the reception of further offers does not trigger GET requests anymore. Instead the consolidation phase is entered, where peers try to maximise signature knowledge by repeatedly exchanging offers. Providing efficiency in this phase is the second purpose of the IHAVE command.

When an offer for a locally known document is received, the versions are compared and first some anti-entropic, reflexive activity takes place: parties with outdated versions (i.e. older than the active version or older than the newest version if none is active) try to synchronise to the newest state. If the receiver has more recent material, it will offer it back to the sender; if the sender's is newer, it will try to acquire that version and continue propagating the newer version.

If the versions are identical, the document's replication and integrity assurance state is updated: the signatures in the offered signature block are verified and both local and offered signature blocks are compared and merged into a new block.

The merged information is used in generating a subsequent new set of offers, which are sent out to a number of peers in parallel. These recipients repeat the offer flooding cycle until the termination criteria for the replication process are met.

Various termination criteria are studied in detail in Section 6.4, but an overview of the simplest initial design should be given here: the information differential between sender and recipient of an offer is used to determine whether the offer is a "duplicate" (no signature differences) and thus should be ignored or whether further offers should be generated. As replication activity is offer-triggered, the convergence towards a globally consistent state of knowledge causes an increase of duplicates, and the replication process thus ends gradually.

How many peers to send parallel offers to, is controlled in part by the policy; in general, a high degree of parallelism is used until the document is active. After that transition, this degree is very much reduced (or even zero, depending on the specific variant of termination criteria). This mechanism is similar to the one that Datta et al. (see Section 2.9) use in their P2P system.

This initial high degree of parallelism is employed to make the flood distribution mechanism robust in case of Byzantine failures, so that no small number of rogue peers can keep updates from reaching other peers and thus stall the process. The specific parallelism parameters for pending and active documents are part of the information of global scope which all peers need to agree on for reliable operation of the replication.

Offers that contain corrupt data lead to the connection being severed and the sender being blacklisted locally. Note that in most cases it is impossible to *reliably* convince other peers of mischief, therefore malicious peers will be cut off only gradually.

The behaviour of an IHAVE offer is similar to the answer to a HEAD request in one particular regard: the content is in both cases a full signature block. The question arose whether IHAVE should contain only the name and version, but upon further review this was rejected as inefficient: without including the signature block, such a reduced IHAVE command forces the the recipient to always follow-up with a HEAD or GET request. As a signature block is very small and bounded in size, there is no benefit to not transmitting it with every offer.

IHAVE messages always carry the latest signature state of the sender and thus commonly transport multiple new signatures in a single message. Peers are also allowed to delay transmission of subsequent offers for a random, short period of time so that all signatures, which are received within this period can be consolidated.

During the design process the question arose, whether always carrying the full state is actually required: IHAVE could transfer only "interesting" signatures, the ones which the other party does not (seem to) have. This would reduce the amount of data that must be exchanged during reflexive offers, where the current state at the recipient peer is known exactly.

In the end, such partial updates were not included in the design, because the dominant communication case is sending offers *without* good knowledge of the state at the recipient, which means that all signatures would have to be assumed to be interesting. Also, withholding information from any peer is detrimental to utilising transitive knowledge transfers, which is the actual reason for exchanging document signatures.

4.10 Group Management

Of course the `Peeranoia` system must address efficiency of deployment and maintenance (see Section 1.3), which this section will elaborate on. One of the fundamental issues of maintenance is how much administrative interaction must be provided over time. But is full automation suitable for a security-critical setting?

In `Peeranoia`'s FIC context, it is recognised that full automation of introspective integrity checking systems poses the threat of not capturing an intrusion *as it happens*. After that, none of the checker's actions can be trusted. Therefore, an autonomous file integrity checker should not be able to modify its fingerprint databases on "auto pilot", but rather rely on this crucial information being signed off by a human administrator before it is used as indicator of integrity.

Thus the notion of human administrators being in supervisory charge of a group of peers was made a core component for the system, but without relinquishing flexibility: certification by administrators is an optional requirement that the policy may stipulate for a document. Administration is also designed as a remote operation that can be actioned from any computer system that carries the `Peeranoia` software and which is capable of contacting at least one member of the peer group.

In-band administration involves three kinds of activities: retrieval of alerts, certification of documents as being correct and legitimate, and management of group membership and policy rules.

4.10.1 Membership Control

In the `Peeranoia` scenario, peer group membership is not ad-hoc as in ephemeral P2P systems. In most P2P systems, emphasis is placed on autonomy on a global level and ad-hoc, infrastructure-less communication structures: data is often routed hop-by-hop, potentially anonymous, and peer groups are expected to shrink and grow continuously because of the lack of single supervising entity.

As `Peeranoia` serves a slightly different purpose, it is necessary to concentrate more on permanence of identity and group membership. After all, what value can be assigned to a data integrity assurance given by a totally unknown party that may or may not be there at a given time and might change identities at will? Another difference from other P2P systems is that, in `Peeranoia` a peer cannot simply be replaced by another and still perform the same part of the overall service. In most other P2P systems much of the functionality is location- and provider-independent and allows automatic load transfer between peers.

Therefore group membership in our system is strictly controlled: administrators decide on membership changes by editing and injecting the `peerlist` document. This document contains the current list of peers and links network identities and public keys. Without an entry in the `peerlist`, no communication with any peer in the group is possible. As such, membership information is communicated simply and efficiently, in-band using the same replication and assurance provisions as available for ordinary documents.

4.10.2 Administrator Capabilities

Simplifying matters further, all administrators have a peer identity known within the group and are treated as temporary peers: the main difference between permanent peers and administrators is that administrators are not partaking in the replication scheme *as recipients of offers*. Because as they are connected to the group in a dynamic and temporary manner, no permanent peer can initiate connections to them.

An administrator can use the system software in a location of his choice, and for the duration of his connection to other peers interact with them by issuing any of the commands the `Peeranoia` protocol offers and is thus able to access and update documents remotely.

Administrators are expected to sever their connections with the peer group upon completion of their task and not to operate from a single permanent peer. This temporary peer nature was chosen for the following reasons:

- Administrators have extra privileges and capabilities; their cryptographic keys are especially threatened by intrusions and should therefore not reside on permanently connected computers.
- Administrators should have the flexibility to work from anywhere.

Administrator capabilities in `Peeranoia` can be layered in order to provide different classes and levels of privileges. This is done by using the policy to specify signature requirements that involve multiple administrators. This allows for decentralised, solo- or team-oriented operation by multiple (not necessarily hierarchically organised) administrators. It can also be used to limit the actions of rogue administrators (*within* the `Peeranoia` environment, thus only to a certain degree).

4.10.3 Document Certification

Commonly, the decision as to whether a document is valid and legitimate is outside the scope of an automated system. Administrators will frequently be required to verify that a document that has been injected by a peer is indeed valid. In `Peeranoia`, this is achieved in a decentralised manner by the administrators adding their signatures to the document in question.

An FIC application would rely on this feature when it detects changes to its tested files. The application can simply inject an updated version of the fingerprint database into the peer group and alert an administrator. The policy would usually not allow this fingerprint document to become active immediately but mandate an administrator's signature. The inactive version of the fingerprint database would be replicated, certified by other peers but still be inactive for the originator.

To certify this document, the administrator in charge would use the `Peeranoia` software to connect to the peer group, retrieve the document from any peer and verify its integrity, appropriateness, conformity to corporate policy and so on. Afterwards, he would create the necessary signature on the document, which is then attached to the signature tree just like any other peer signature.

Finally, the updated signature block would be reinjected into the peer group by offering it via IHAVE to at least one peer (but preferably to some more in parallel to counter malicious failures), and the new signature block would be replicated throughout the group and eventually reach the originator.

If multiple administrator signatures are required (because a document is of a very critical nature or because some administrators cannot be fully trusted), other administrators would do their certification in the very same manner. Eventually the document will have reached the requirements the policy mandates, thus become active and the integrity checking application will start operating on it.

Document origination by administrators works similar to the procedure above; as with any other document a valid originator signature is required which in this case would be the administrator's.

While certifications by administrators have an implicitly higher value than the merely automated replication, we decided not to embed the assignation of value into the technical components of our system but rather leave this as an option when formulating a policy. Thus we implemented administrator certifications with signatures just like peer-generated assurances.

4.10.4 Adding and Removing Peers

Adding a peer to the group is a relatively simple procedure: the computer system to become a peer has to be primed and the remainder of the group needs to be told of the peer's existence and capabilities. The former requires actions taken on the new peer, while the latter can be done remotely at any location.

1. First the new peer needs the `Peeranoia` software to be installed, a new cryptographic key pair is generated for the peer, the private part of which resides on the peer computer only while the public key is incorporated into a new peerlist document.
2. The administrator creates an updated `peerlist` with the new peer's public key and an amended policy that includes the peer's rights, and signs that. The administrator then offers the updated `peerlist` document to the group for replication. As soon as it has acquired sufficient signatures and distribution, it will become active and the peers in the group will start using it. From then on the new peer will be recognised by the other peers.
3. The administrator can now retrieve the new active peerlist, supply it to the joining peer and start the `Peeranoia` software on the peer computer.

It may seem useful to an administrator to short-circuit this procedure by implementing step 3 immediately after step 1, but this is fraught with complications: even if the joining peer after step 1 has a `peerlist` that it considers active, none of the other parties have any knowledge of the new peer yet and will therefore not accept any communication attempts from it.

Also, it is possible to enter a deadlock situation in step 2: The new `peerlist` must become active without the new peers' involvement or it will not become active, ever, because the new peer can not be integrated until the policy has become active which requires the peer to communicate and so on.

This indicates that it is not possible to *immediately* tighten the policy to require the new peer in the `peerlist` update of step 2. If this happens inadvertently, an administrator can easily remedy the problem by supplying a properly consistent `peerlist` that restores function. A two-step procedure suggests itself here: First new peers are added without changing the policy, the `peerlist` is distributed in the original group (without the new peers being involved). Then this active peerlist is installed at the new peers, and now it is finally possible to safely tighten the policy requirements.

The `peerlist` document is the crucial cornerstone for peer group detection and decisions and therefore is synchronised as tightly as possible: peers must exchange their `peerlist` s (or more precisely, reciprocally offer the signature blocks) at every communication startup. This is to ensure fastest convergence of the `peerlist` throughout the whole group.

Removal of a peer works in the same manner, by generating a new peerlist that lists exactly the desired peers; managing of administrator memberships follows the same procedures.

If a peer is removed, then the documents that it has authored become unverifiable as the key the originator signature was made with is no longer available. In this case, peers storing such documents are allowed to garbage-collect them. On removal of a peer, signatures that it has made on other documents lose their value as integrity assurances and are not entered into the activeness calculations.

Note however, that there is no support for automatic removal of peers that are subverted or become otherwise unreliable: the reason for this is again the problem of lack of *verifiable* evidence in most cases. It would be very much counter to our robustness aims if peers could cause others' exclusions

by distributing hearsay. Instead of such a global mechanism, Peeranoia uses local blacklists for acting on locally observable but not independently provable malicious activity. Blacklists are not used for problems caused by lack of communication reliability; in such cases, only the current connection with the unresponsive peer is dropped.

The Peeranoia design suggests that local blacklists are cleared when a new version of the peerlist becomes active, the reason being that human intelligence will most likely be required to determine the reason of a conflict and remove the problem; the easiest decentralised way of signalling this process having been completed is to do something that everybody can observe but which only administrators can effect, hence the release of a new peerlist.

4.10.5 Alerts

The distributed document storage mechanism is not only useful for peer-owned data but can also be used to safely inform administrators. This can be accomplished very simply by a combination of policy and naming provisions.

A peer that wishes to inform any or all administrators within the system, generates a document with the desired information and gives it a name outside the normal name space partitions, for example a name like alert:machinename. The document is then encrypted collectively to the targeted administrators (which is a mechanism that all currently deployed hybrid cryptosystems provide) or not encrypted at all, depending on the nature of the information, and then injected into the peer group for replication.

Assuming a flat hierarchy of administrators cooperatively sharing the burden, the policy should not put any originator restrictions on documents from the alert: namespace and allow activeness after a single administrator signature. An administrator would collect the list of current alert documents using the HEAD command with a regular expression, retrieve the pending ones, act on the information and finally supersede the alert document with an empty version which would immediately become active, and being empty, allow peers to delete the document (and its earlier versions).

The situation for compartmented groups of administrators is similar, but the policy would have to take compartment boundaries into account: for example, the alert: namespace might be subdivided further for simplicity, and alerts within each sub-partition would be made subject to whatever administrator sign-off requirements there may be for this particular compartment.

4.11 Summary

In this chapter, the conceptual framework of Peeranoia's design philosophy has been expanded into a detailed specification of the system component's behaviour. These components include:

- Peers are individual computer systems, each of which runs the Peeranoia application on top of an existing operating system. They form a uniform group, and provide the integrity-assured storage service mutually to members.
- A specific Peeranoia communication protocol is used for data exchanges between group members.
- Human administrators are used to control some aspects of the overall behaviour remotely, as well as to supply extra certification for important data.

These elements together provide the Peeranoia service: peers can inject and request data, which is replicated and integrity-assured by other peers. Data is organised as separate documents, which originate at individual peers. Documents are versioned and are immutable once created: they can only be superseded with newer versions, but not altered in place. The majority of documents are of local scope,

and of importance to the originator only; a small number of global documents are required for group administration.

A policy language is used to describe how well a document must be distributed and certified, before its content can be used. This policy also allows control of document origination, based on document names. In an FIC setting, the policy mechanism would be used to specify who must sign off on a file fingerprint document, before the FIC application on the respective computer system is allowed to use the fingerprints as a known-good baseline.

To provide integrity assurance and robustness for the replication process, every document is associated with some meta-data, whose most important element is a signature block. This signature block reflects replication and integrity certifications that a document acquires during the process of distribution among the peer group. A discussion of possible modes of representation and composition of the signature block lead to the `Peeranoia` design adopting a hierarchical tree format, which provides a certain amount of resistance to the removal of signatures enroute by a malicious peer. All signatures are produced using asymmetric cryptography, and can be verified independently: every peer has access to all others' public keys.

The communication protocol used between peers is a simple, asynchronous protocol which uses only nine different message types. All data is symmetrically encrypted with ephemeral session keys, which can be established easily because peers know each others' public keys. One command, IHAVE, is used extensively in every replication process and some optimisations for that purpose were developed.

With these building blocks in place, a research implementation of the `Peeranoia` system was developed, and in the next chapter, an account of this development and the resulting application is presented.

Chapter 5

Implementation

To demonstrate the practical feasibility and to provide quantitative measures for the computational and communication cost of our design, a prototype implementation has been developed[80, 81] and examined.

The scope of the prototype implementation is to provide experimental support for the replicated storage and integrity assurance aspects of the `Peeranoia` environment, but without integration into an actual FIC application: the novel contributions of this thesis are related to the distributed service rather than to the well-established methods of file integrity verification itself. For source code availability see Section A.3.

In this chapter, an outline of the architecture of the prototype will be given, together with a sketch of the development process itself.

5.1 Technological Considerations

The prototype was developed in Perl[82, 83], which was chosen for all the usual reasons that apply to interpreted high-level languages: extreme conciseness of written code, availability of high-level data manipulation operators and the abundance of existing Perl packages to interface with other software. These benefits come at the cost of potentially reduced performance when compared to compiled languages, but this was not a major concern because the `Peeranoia` design does not require real-time operations.

The aspect of code conciseness and expressivity was of high importance, because of the intention to experiment with various different propagation strategies and to evaluate their robustness and cost characteristics. The good availability of packaged Perl modules was also very beneficial, especially as rather mature packages exist which cover all the cryptographic aspects of the `Peeranoia` system.

The prototype runs on UNIX platforms, but its operating-system-dependent aspects are trivial (e.g. the character used for separating directory components in a file path) and can be adjusted easily to any platform that is supported by Perl, and which provides TCP/IP networking, multi-tasking and some kind of hierarchical file system.

One issue with Perl, though, is its considerable resource requirement in the areas of disk storage, memory and computational power. This conflicts with expectations for a paranoid, security-critical software application, which needs to be self-contained as much as possible. Therefore we cannot classify the current prototype as being deployment-ready.

The experimental nature of the prototype does, however, not extend to the specifications of the `Peeranoia` design: the future option of integrating the functionality into one of the free FIC applications like Samhain[3] has been taken into consideration from the very beginning. Therefore, great efforts were taken to ensure that the protocol and format specifications (see Appendix A.2) were developed with

portability in mind. For example, the network protocol was designed using binary message formats instead of textual ones to allow for an efficient reimplementation in compiled lower-level languages, such as C.

Applying a software engineering perspective, the following main elements were identified as implications of the `Peeranoia` design: the prototype is an event-driven application, its functionality is provided by a hierarchy of software modules and objects, and it makes extensive use of existing software.

5.1.1 Event-driven Operation

Because of the asynchronous communication design of `Peeranoia`, the prototype requires some mechanism to deal with parallel and asynchronous handling of communication messages.

The core Perl language environment provides only `select()`-based support for asynchronous communications, but implementing the prototype using this mechanism appeared to be overly complex and difficult to experiment with. A similar argument kept us from implementing the communication elements as separate processes, which would also have introduced a significant inter-process communication overhead as a lot of state information must be shared between such processes. Furthermore, there is the requirement for the handling of other time-based events like timeouts and periodic maintenance tasks, which suggests using a more general event dispatching mechanism for both communications and handling of periodic tasks.

A number of different event frameworks are available for Perl, ranging from very minimalistic special-purpose packages to extremely broad and powerful environments. Because of previous positive experiences in other projects, one particular system was a definite candidate: POE, the Perl Object Environment[84].

POE is a framework for cooperative multitasking and event handling, which offers an extensively layered environment and which comes with a variety of pre-packaged, readily available functionality. An initial evaluation showed very promising results: it took less than 200 lines of code to implement a simple hello/echo peer-to-peer application.

5.1.2 Code Reuse

`Peeranoia` clearly requires support for both asymmetric and symmetric cryptographic operations, both of which are well catered for by existing Perl modules.

For symmetric cryptography the Blowfish cipher was chosen, provided by the package `Crypt::Blowfish`. To extend this block cipher for secure operations on multi-block document, the cipher is used in "cipher block chaining" (or CBC) mode (see [85]). The module `Crypt::CBC` supplies this functionality. Both of these modules perform satisfactorily, but a deployable FIC application should likely employ the AES cipher suite, as AES has been evaluated more extensively than Blowfish.

The requirement for asymmetric cryptography was a bit harder to fulfil; not because there are no candidate implementations, but rather because of the intention to keep the prototype as small and simple as possible. It was decided not to use any of the low-level modules like `Crypt::RSA`, which provide only the basic cryptographic algorithms: because the `Peeranoia` system also needs mechanisms for key generation and management, it would have been necessary to develop and test our own key management infrastructure, together with the necessary stand-alone tools for signature verification etc.

To avoid this extra implementation effort, it was decided to use GnuPG[17]. GnuPG is a stand-alone application which implements asymmetric and symmetric cryptography for files, and which also includes support for an extensive key management framework. The prototype benefits especially from the readily available key management functionality: all peer's keys are standard GnuPG keys, and use the key comment facility to identify the key's owner. For permanent peers, their name, IP address and listening

TCP port are stored in the key comment; for administrators, only name and an email address are given. Internally, the prototype uses the numeric key identifiers for peer identification everywhere (except for logging, see Section 5.2).

However, the integration of GnuPG functionality into a Perl program proved to be problematic, as there are no Perl-native GnuPG interfaces: all currently available modules involve dealing with separate GnuPG processes in one way or another. After some evaluations, the `GnuPG::Interface` module was selected because it is the most complete and programmer-friendly variant; but as it is based on inter-process communication with separate GnuPG processes, its lack of performance is likely to be unacceptable for a realistically deployable implementation.

The final area where the prototype made extensive use of existing technology is the communication infrastructure. The prototype operates on any Internet-like network. From a TCP/IP perspective, the `Peeranoia` protocol is part of the application layer, as it uses the TCP protocol as the underlying communication mechanism.

While the Perl core comes with the necessary functions to manage TCP communications, the POE environment provides more abstracted access to similar functionality. After some initial evaluations, the submodules `POE::Component::Server::TCP` and `POE::Component::Client::TCP` looked especially promising for efficiently providing the basic communication infrastructure. Unfortunately, the POE environment turned out to have some serious shortcomings also, which will be discussed in Section 5.4.

The prototype also requires good sources of randomness for strategical decisions. The kernel of the operating system we used, Linux, provides the software device `/dev/random`, which is a secure random number generator. It gathers environmental noise from device drivers and other local sources into an entropy pool and produces random numbers of high quality; the prototype implementation makes extensive use of this software device.

5.1.3 Modular Design

The prototype implementation was developed with fine-grained boundaries between functional units, in order to facilitate the evaluation of different components. This development was done bottom-up, beginning with helper modules like document storage interfaces up to a single object that collates and coordinates all activities on a high level.

Following Perl's *very* relaxed approach to object-orientated software engineering, the prototype's architecture defines object classes only for the core components; helper modules with more limited scope were implemented as simple sets of functions. The module layout and boundaries are detailed in the following sections.

5.2 Functional Hierarchy

Figure 5.1 gives a broad outline of the module hierarchy of the prototype application, after adjusting the design to the requirements and expectations of the external Perl packages that are used. The figure does not display *all* interactions: certain limiting aspects of POE's layering necessitated the implementation of a number of inelegant work-arounds, most of which violate the boundaries of the module hierarchy to some extent (see Section 5.4).

The main object class is `Peeranoia::Peergroup`, which encapsulates all of `Peeranoia` as far as a single peer is concerned. Every peer computer system runs an instance of the prototype application, which instantiates exactly one such object. The `Peergroup` object deals with documents and delegates tasks to subsidiary objects when necessary: it is responsible for all high-level decisions and strategies, accepting and dispatching inbound connections from other peers, scheduling of data exchanges with others and it also manages user interaction (see next section).

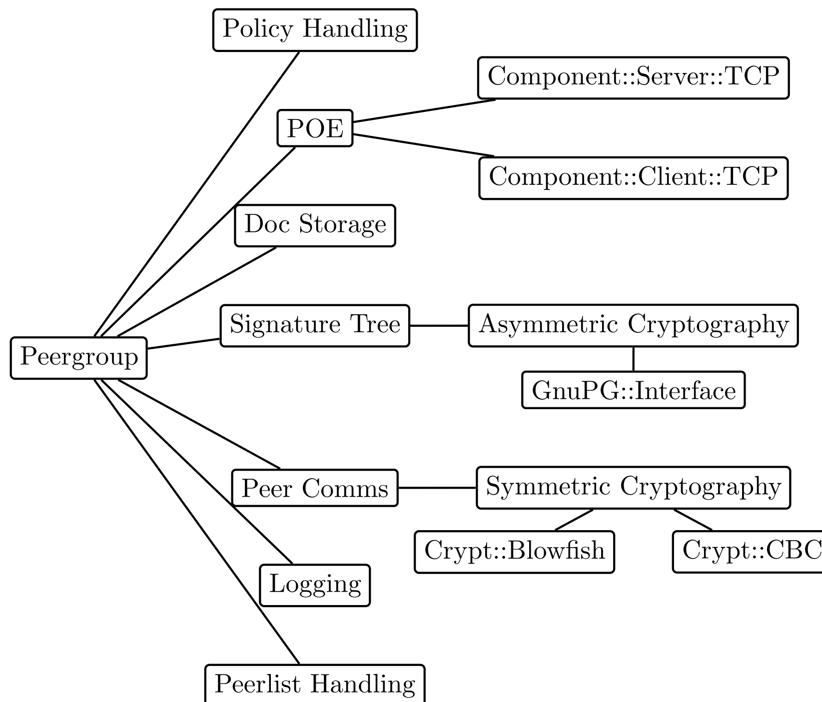

Figure 5.1: Module Hierarchy Overview

To perform these tasks, the `Peergroup` object creates and supervises a number of `Peeranoia::Peer` objects (labelled "Peer Comms" in Figure 5.1): there is one such object per connection with another peer. These `Peer` objects manage all aspects of a single connection between peers: establishing connections and session keys, exchange of protocol messages and decoding these using symmetric cryptography and so on. The `Peer` objects report back to the parent `Peergroup` object using call-backs whenever a message that involves documents must be handled; simpler operations (PING, PONG, REKEY) are handled by a `Peer` object by itself.

These two object classes perform the majority of high-level operations, while tasks of more limited scope are delegated to specific helpers as follows:

- The `Peeranoia::Storage` objectclass deals with access to and control of the local storage area for documents. In the prototype, this is merely a convenience wrapper that manages documents as separate files in a particular directory.
- `Peeranoia::GPG` wraps and extends GnuPG::Interface in order to provide consistent, implementation-independent access to asymmetric cryptographic functions.
- The `Peeranoia::Peerlist` class handles the management of the `peerlist` document, which encapsulates the policy and the list of peer public keys. As the prototype uses GnuPG for asymmetric cryptography, this list of peers' public keys simply uses GnuPG's "keyring" format.
- Objects of the `Peeranoia::Sigtree` class perform operations on document's signature trees: creation of new trees, adding of signatures and merging of whole different trees, and conversion between an internal tree representation and a flattened, linear form for storage and transfer.
- The `Peeranoia::Policy` objectclass provides all functionality related to the policy: parsing and import of the human-readable policy format into an internal binary representation, and application of the policy rules to a document's meta-data to test its activeness.

As the task of verifying policy rules against a document's signature state occurs frequently in the `Peeranoia` environment, the implementation uses a binary format for efficient storage and evaluation of policy rules. This format would be very cumbersome for human administrators, therefore the policy is formulated in textual form, according to the specification given in Appendix A.1. The `Policy` module provides functionality for conversion between those two formats.

The prototype uses GnuPG keys and the numeric GnuPG key identifiers to uniquely address peers for policy purposes. As a concrete example, consider a group with peer key identifiers `a` through `k`, administrator key identifiers `M` to `P` and the following policy rule:

```
^localtest: rootsig M,N,P (| {6}* {3}f,g,O)
```

This rule specifies that documents whose names start with "`localtest:`" must be authored by administrators `M`, `N` or `P`, and then have to acquire either at least 6 signatures from all known peers or alternatively signatures from `f`, `g` and administrator `O` before being considered active.

- The `Peeranoia::Log` class provides logging of activity to a file, which is required to collect information for measuring the effectivity and cost of the `Peeranoia` mechanisms. To allow efficient, automatic extraction of such information, a structured log format has been developed. This format is used for all logging, regardless of whether debugging messages are handled or whether statistical or simply informational data is to be recorded.

  The log format is purely textual; it contains both a timestamp with a precision of $\frac{1}{100}$ of a second, and the peer's name (for convenience, as the key identifiers used elsewhere are numeric and long). These elements were incorporated to allow the collation of multiple peers' logs concerning a single experiment into one file[1].

  Furthermore, every log message carries a type code from a pre-set list of known codes, an optional comment and (where applicable) name and version of the document in question, as well as the other peer's name. Depending on the type code, document name and peer may be optional: there are some codes for free-form informational log messages, but most messages are meant to be taken in the context of operating either on a document or communicating with another peer.

  The order of elements in a log message is time stamp, logging peer, code, document and peer context and finally the comment. An example excerpt from such a log is given in Figure 5.2, to be interpreted as follows: peer `p0` opens a connection to `p1`, in order to offer document `r3-1`. Immediately upon successfully establishing a TCP connection and agreement on a session key for symmetric encryption, the two parties offer each other their versions of the `peerlist` document; furthermore, a diagnostic PING/PONG exchange is initiated. Interleaved with these maintenance operations, the actual offer of document `r3-1` is transmitted: the receiving peer `p1` doesn't have the document content yet, and therefore follows up with a GET request to retrieve the whole document.

5.3 User Interface

The top-level `Peergroup` object only provides a programming interface, which exposes some methods to initiate requests and offers. It is up to the developer of the encapsulating program to supply facilities for interaction between a consumer and the `Peergroup` programming interface. In the prototype implementation this is provided in a very rudimentary fashion, which nevertheless suffices easily for the purpose of automating experiments; it must be noted, however, that a realistically deployable application would need major improvements over the prototype.

[1] This implies the need for reasonably synchronised clocks on all computer systems that are involved in such an experiment.

```
1458.62|p0|newout|||p1|opening new conn to 192.168.0.11/9001
1458.63|p1|newinb||||new inbound connection
1458.66|p1|Irekey|||p0|inbound rekey tag
1458.66|p1|Oihave|peerlist|44bf1fc1f9d88849|p0|dequeueing
1458.70|p0|Oihave|r3-1|44c0baf2f9d88849|p1|dequeueing
1458.71|p0|Iihave|peerlist|44bf1fc1f9d88849|p1|handling ihave
1458.71|p0|Iping|||p1|inbound ping tag 212
1458.71|p0|Oihave|peerlist|44bf1fc1f9d88849|p1|dequeueing
1458.73|p0|ckdsig|peerlist|44bf1fc1f9d88849|p1|checking sig delta: 0/0 new h/t
1458.74|p0|valid|peerlist|44bf1fc1f9d88849|p1|1 10 signers
1458.75|p1|Ipong|||p0|logonly inbound pong tag 212
1458.75|p1|Iihave|r3-1|44c0baf2f9d88849|p0|handling ihave
1458.75|p0|Iget|r3-1|44c0baf2f9d88849|p1|inbound request get
```

Figure 5.2: An example `Peeranoia::Log` output

The prototype application is controlled by two different means: command line arguments given at start-up, and files being dropped in particular directories.

Command line arguments are used to set various parameters that do not change over the time that the application is active. For example, this includes the name of the peer and the location of its document storage directories, but also strategical parameters like the degree of parallelism to use for offers and retrieval requests. For a complete description of directives the reader is referred to the source code (see Section A.3), as that is of minor relevance for the functional description given here.

After the application has started up, it enters the main event handling loop and awaits one of two kinds of events: inbound connections from other peers, or files appearing in a particular request directory.

The administrator can initiate document injection and replication by saving a document as a specifically named file in this request directory; the application detects its existence, interprets its name and triggers the relevant method of the `Peergroup` object. Document retrievals are handled in a similar manner, with the result of the retrieval becoming available in the peer's document storage directory.

Besides this main application, the prototype implementation includes some separate standalone utility programs which are used to generate and verify signature blocks, extract and analyse log information, and manage policy and peer lists.

5.4 Practical Experiences

Overall practical experiences with the prototype were good, with the following few exceptions:

- POE has a rather non-orthodox way of extensively layering unfortunately-labelled concepts. For example, POE calls a task or thread of execution a "session" instead of using a more standard terminology. Such POE idiosyncrasies were some of the reasons for not having been completely successful in strictly separating the tasks of the per-peer communication `Peer` objects and the top level object.

- Some of the abstraction boundaries that certain POE components use are inconsistent and needlessly complicate code reuse. The choice of employing the `POE::Component::*::TCP` modules proved to be especially suboptimal, as their abstractions clashed with our module hierarchy in numerous places.

  For example, a newly created TCP client object can not access *any* caller-supplied information when the connection is successfully established, but only earlier at the object's startup. To solve

this, we had to introduce glue code that copies data around. A lot of our problems with POE stemmed from such design shortcomings.

- POE also lacks a fair number of crucial mechanisms. There is, for instance, no (reliable) mechanism to terminate a session and no way of finding a session's identifier or keeping track of sessions.
- Under heavy load, POE would often misfire and duplicate events and "zombie" sessions would linger (due to deeply buried references to data). This also was a constant problem to code around.
- Debugging of POE applications is hideously difficult, in part due to the extensive layering.

It was also found that the prototype's paranoid approach to verifying integrity *of local data* was was over-zealous: it would not trust any on-disk data, and recheck all signatures every time a document was handled. This level of paranoia and the comfort of having a key infrastructure came at the price of enormous runtime costs: about half of the CPU time was spent starting up and tearing down external GnuPG processes. For the prototype, this problem was remedied by introducing more intelligent caching of signature verification status and reducing the number of re-verifications of locally stored data. Despite this measure, a realistically deployable system would have to use a native implementation of asymmetric cryptography.

In the next chapter, we will examine the characteristics of the `Peeranoia` system further: after covering theoretical aspects, we will return again to the prototype implementation and discuss empirical properties and cost measures that were obtained in extensive experiments.

Chapter 6

Analysis

After elaborating on the design principles, the details of the Peeranoia system and its research prototype in the previous chapters, we now focus on the characteristics of the resulting service.

In this chapter, we answer the remaining core questions of Section 1.3, and provide our arguments and answers in relation to the questions that are still unanswered: how much robustness can be designed into an autonomous distributed mechanism like Peeranoia, and what is the associated computational and communication cost?

Four different perspectives are covered: the expected benefits and design motivations are discussed first, followed by a survey of the handling of various security threats. Next, an analytical model is developed to provide robustness measures and finally, we discuss the cost measures and benefits that were obtained using the research prototype, together with an examination of certain variations of the replication procedure.

6.1 Design Influences

The Peeranoia framework as detailed removes the need for a central party for storage or integrity assurances and can be managed efficiently even in decentrally administered environments. The reliance on multiple, suspicious and minimally cooperating parties for backing up local information and integrity verification greatly improves the robustness and resilience of the overall system.

Conceptually, the design borrows from a number of prior efforts, notably the Bayou system (outlined in Section 2.9). Some of the parallel goals of Bayou and Peeranoia are quite visible in the resulting design:

- Peer-to-peer environments with little if any communication infrastructure guarantees are the basis of both Bayou's and Peeranoia's field of operation.
- Eventual consistency and different needs for different applications prompted us to adopt Bayou's distinction of tentative versus committed writes to some extent, in the form of pending versus active documents.
- In Bayou, replicas know the state of affairs with regard to write operations at other replicas (to drive the anti-entropy algorithm) and updates work on partial data with conflict resolution mechanisms. In contrast, Peeranoia operates on opaque content (like OceanStore, see Section 2.10), and therefore supports only superseding full documents. Also, Peeranoia treats all conflicts as the originator's fault and no conflict resolution mechanism is required. However, certified knowledge of what other replicas know or store is the basis of Peeranoia's assurance quality measure, and drives the propagation process here as well.

- Bayou uses versioning for writes and allows pruning of old writes when stabilised; individual replicas may choose how aggressive to be about pruning. `Peeranoia` uses versions for documents and allows similar garbage collection of superseded information.

- Bayou does not guard against Byzantine faults but is extremely simple and small. `Peeranoia` strives for simplicity as well, but needs to take such failures into account and thus can put less trust in other parties' deeds. For example, where Bayou accepts updates from peers unconditionally (after authenticating the peer), `Peeranoia` needs to take a more suspicious stance until provenance and acceptability of data can be proven.

As `Peeranoia` aims at robustness in the face of arbitrary failures, information about the document distribution process must be exchanged between parties. In this regard `Peeranoia` extends Datta's P2P system (see Section 2.9), which exchanges untrusted and partial replication histories with every message. In our case, integrity assurance and replication history are combined into signature blocks, which provide a *verifiable* partial topological history of data exchanges. The rationale in both cases is similar: this information is used for speculation and fine-tuning of the distribution procedure. Another similarity is the strongly decentralised paradigm that underpins most design components.

The use of asymmetric cryptography however requires us to consider efficiency measured not just in the number of messages but also in computations per message. OceanStore faces similar challenges, and has provided quite some inspirations: both systems distinguish precisely between trusting somebody with data *content* (which does not happen) and trusting some party to *handle* data while adhering to a protocol to provide a service. The notion of a verification tree has proven useful, and versioning is an obvious, common answer to the need for opaque content that changes over time, but most of the other elaborate mechanisms in OceanStore were rejected for reasons of simplicity. Neither erasure codes nor secret sharing are fully compatible with `Peeranoia`'s autonomy goals.

Security-wise the `Peeranoia` system is superior to any centralised variant, because to falsify (fingerprint) data one would have to subvert many members of the peer group simultaneously, but the high degree of decentralisation makes this infeasible. Also, a technology like `Peeranoia` that is simple and platform-independent allows multiple different implementations to be used, which further increases variety and decreases the chances of a single attack vector being sufficient for widespread successful intrusions.

The `Peeranoia` design relies in diversity and redundancy to provide trustworthy integrity assurance. The protocol aims to balance strict assurance against flexibility, but as a consequence of the trade-offs involved, global consistency can be provided with high probability only, not absolute certainty (as other solutions to the Byzantine Generals problems do). On the other hand, the combination of consolidating multiple information elements into a single offer message and the verifiable nature of the content allows `Peeranoia` to distribute a document to all peers with significantly fewer messages than other mechanisms.

Individual elements are untrusted by default, but cooperation is effected using simple evidence- or reputation-based trust, which is a mechanism similar (but simplified) to GNUnet's trust economy (see Section 2.12).

Our approach also scales well in large environments because of the remote administration capabilities. Overall, this limits any administrator activity that has to be executed on the peer computer system itself to the initial setup phase only. While geared towards file integrity verification, this storage system is useful for other applications where read-often-write-seldom semantics with eventual consistency are sufficient. Specifically, this system lends itself to being applied in the context of a *computer immune system*[86], where it can offer regeneration of damaged or lost data from a known good tamper-resistant source.

6.2 Threats and Countermeasures

The overall goal of employing rampant mistrust and digital signatures in `Peeranoia` is to limit the amount of mischief an arbitrarily malicious adversary might perpetrate. Cooperative techniques are used to ensure that no outside actions, nor any small number of subverted peers within the system can cause more than a graceful degradation of the system's services.

As shown in Section 3.2, our design goals can not be all maximised at the same time. For example, utmost tamper prevention conflicts with high efficiency. The design thus includes certain trade-offs which limits the attainable degree of some goals.

From the perspective of an FIC application, data security and robustness in case of attacks must be given high priority. Therefore, the most important criterion in `Peeranoia` is to ensure data integrity, followed by availability of integrity assurances to cooperating peers. As a consequence, the worst-case attack scenario results in denial-of-service only, which we will show in the remainder of this section.

However to do so, certain basic assumptions for the environment must be given: First, both asymmetric and symmetric encryption employed in `Peeranoia` are expected to be *practically* unbreakable. As the design is not centred on any particular algorithm, the reader is referred to [85] for algorithmic details and methodologies for currently available cryptosystems.

Using asymmetric cryptography and cryptographic hash functions already introduces a non-zero (but minuscule) probability of an adversary undetectably forging data. Therefore, we assume that adversaries are computationally bound and of comparable power to the other parties; the discussion of threats covers this situation only.

6.2.1 External Threats

Parties external to the peer group cannot affect `Peeranoia`'s operations much: all communication is encrypted. Also, new communication sessions are authenticated using the cryptographic identities which every peer has. Connections are only accepted from known peers (or more precisely, connections are dropped immediately by the contacted party unless the initiator can prove with the very first message to be part of the group).

There is also the provision for locally blacklisting such external adversaries. Thus an outside adversary can not effect more than a little waste of computational resources by pretending to be a peer: after the first such failed connection, the responding peer would blacklist the perpetrator.

Passive attacks like traffic analysis can tell an eavesdropper who the group members are but not much more, as the protocol makes provisions for decoy traffic using PING and PONG messages.

Deduction or acquisition of a single session key does not leak any information beyond commands exchanged by the respective two peers and the referenced document names and versions. Furthermore, the rekeying provisions ensure perfect forward secrecy: the acquired session key will be replaced by an independent new key after a configurable period within a connection; also all connections use totally independent random keys.

If an intruder was capable of modifying messages enroute, then any modifications would be caught by `Peeranoia`'s lower-level communication protocol that checks for communication garbles using a CRC on the inside of the symmetrically encrypted message. It is exceedingly improbable that common, well-researched symmetric encryption algorithms (e.g. AES in CBC mode[85]) would allow an attacker sufficiently fine-grained control to both modify the unseen payload as well as keep the CRC intact. Any garbles detected by a message's CRC are not treated as indications of active misbehaviour by the sender; the connection will be dropped as being unreliable but no blacklisting of the other peer happens.

The effect of replaying collected messages blindly is also limited, even without the low-level protocol being designed with strict replay protection. First, the protocol specification suggests that the contacted

peer issues a PING after connection establishment. As message tags are chosen randomly by the sender of a command, the probability of a replayed response message matching this random tag is low. Secondly, even if the intruder replays a whole session from the session key establishment onwards, he faces the problem that there is no visible indication of what messages are requests and which are responses. If the intruder manages to guess correctly, then he will still not be able to affect the overall system negatively. Because the protocol does not have any commands that can force the receiving party into doing something outside the framework of the Good Citizen criteria, the worst possible outcome is that the intruder might offer superseded documents: recipients of such offers will reject these offers.

If an intruder disrupts communication temporarily, then peers will fall back to using their locally available versions of crucial documents. This leads to less stringent integrity assurance but can be argued to be favourable over non-operation. Also losing communication does not affect the existence of the signature blocks on these local documents which include multiple signatures by others. This definitely provides better assurance than a peer's own signature only.

Such communication disruptions can, however, result in peers temporarily misclassifying unreachable others as untrustworthy. In no case can such temporary misclassification affect the safety of data, but progress in exchanging document assurances is of course impeded. In the common benign case, unreachable peers are simply disregarded and allowed to catch up upon restoration of connectivity; the `Peeranoia` system thus resumes full operation seamlessly. If a peer `A` happens to misinterpret impeded communications with peer `B` as malicious, it would blacklists `B` *locally*: blacklists are local only and will not affect anybody else's data exchanges with peer `B`. Given the redundancy built into document exchanges in `Peeranoia`, `B`'s signatures will still reach `A` (if indirectly). Blacklists also can be cleared explicitly by administrators (see Section 4.10.4): by injecting a new `peerlist` document, even this worst case result of communication outages can be resolved easily.

Overall an outside adversary is limited to denial-of-service attacks, and only if he is in control of the underlying communication infrastructure. Otherwise an adversary will only be able to determine the extent of the peer group.

6.2.2 Malicious Administrators

Any tolerance of corrupt administrators can only apply to activities *within* the `Peeranoia` environment, and is thus quite limited. However, depending on the actual policy requirements in place, the actions of a small number of rogue administrators can be tolerated.

Obviously, a document is threatened if a rogue administrator is expected by policy to certify it as correct: he could insert a forged document and certify it to be correct. Among these, the most interesting target would be documents of global scope, i.e. the group meta-data. If the policy requires only the rogue's signature (and no other administrators' signatures) for the `peerlist` document to become active, a group split can be produced. An administrator could also (re-)introduce known corrupt peers, and there are some other, slightly contrived scenarios where manipulation of rogue administrators do negatively affect honest parties.

Generally, we assume that `Peeranoia` administrators are also the system administrators of the actual computers; in this case, the above threat scenarios are insignificant compared to the amount of damage an administrator can cause outside of `Peeranoia`'s environment. A rogue administrator will usually have other, more direct methods of destroying such a distributed system, so trying to guard too extensively against a person who has access to the underlying operating system anyway is rather futile.

If the administration of `Peeranoia` and the actual computers is separated in a particular deployment, then the hierarchical administrator capabilities of the `Peeranoia` system can be very beneficial: the administrators can formulate a policy which limits the actions of "junior" administrators to contain potential mischief or mandate a bigger quorum of administrators to sign off updates to documents of higher importance.

6.2.3 Local Threats

Threats internal to a peer group are more interesting and potentially far-reaching, but as we argue in the following, they can still be countered fairly effectively by the application of general mistrust and redundancy to defend in depth. First, we examine threats local to an individual computer system.

The crucial components of our system are the individual peers' private keys. An intruder who has successfully subverted a peer is assumed to have access to that peer's private key and as such has the peer's privileges. However, this does not jeopardise the secrecy of data that the subverted peer stores on behalf of others: all important data is expected to be encrypted for the intended user anyway. Data integrity of these documents is also not threatened, as every document has multiple separate signatures covering it, most specifically the originator's signature which the intruder can not fake or replace without detection.

Subverting a peer will also give the intruder access to the policy document, the list of peers and their public keys, but assuming practically unbreakable asymmetric encryption none of these is a major concern. From a local perspective, the worst case scenario is indeed denial-of-service: documents stored locally can be destroyed or removed but not tampered with undetectably.

6.2.4 Malicious Insertion of Documents

After a particular peer has been subverted, the intruder could use this peer for subsequent malicious activity, locally or affecting the whole peer group. A more promising attack vector than the destruction of locally stored documents is the subversion of the document replication process: *partially* partaking in the communications allows an intruder to affect certain results globally.

A corrupted peer can insert documents into the peer group successfully, if and only if all its behaviour is within the expected specifications as far as other peers can observe it.

Unless the policy is overly lenient, the malicious peer cannot affect other peers' documents negatively: modification of existent documents is not supported, and superseding documents belonging to others is easy to disallow using originator restrictions in the policy. It is also not required for peers to remove superseded document versions, thus a total history service (cf. [62]) can be provided easily.

A corrupt peer also cannot simply send garbage: any deviation from the Good Citizen criteria leads to disconnection and blacklisting by the detecting peer, therefore the corrupt peer would lose access to the peer group after just a few attempts.

Inserting well-formed documents is of course possible and allows denial-of-service by overwhelming other peers. However, there are a few simple countermeasures that limit this threat vector greatly.

- First, the intruder is at a disadvantage: *all* the inserted documents to make up the flood must have originator signatures that match the actual data, but signature generation is quite expensive computationally. Assuming roughly comparable computational capabilities for the peers, the intruder is very unlikely to succeed.
- The intruder can, however, attempt to flood the group with bogus IHAVE offers. It is free of cost for an adversary to produce a garbage signature block, and therefore the adversary is at advantage – until it is required to supply a consistent document, which is exactly the mechanism that `Peeranoia` uses to counter this scenario. Offer handling requires the document in question to be available, as signatures without data are not verifiable and until the document is available, no computing resources of the receiving peer are used at all.

  The standard response of any contacted peer will be to send a GET request for the document that the intruder claims to offer. Unless the intruder has such a document with matching signatures, the other peers will deal with no answer or no forthcoming document by simply blacklisting the

intruder as being fraudulent. Therefore, the intruder cannot expect to be able to send more than one bogus offer per target peer (which also will not consume resources on the targets).

- Furthermore, a rate-limiting mechanism for accepting new documents originating at any particular peer is trivial to implement and would allow to blacklist peers that originate more documents per time interval than expected. While neither the specification for `Peeranoia` nor the research prototype include such a limiting device currently, it would be trivial to add to the policy language. The same argument holds for denial-of-service by insertion of overly large documents.

In these cases the originator would be blacklisted quickly across the entire peer group (very much in contrast to the case of a peer transmitting faulty material, where always the sender is blamed): every document carries an infalsifiable proof of origin which every peer can act upon, regardless of whether it is in direct contact with the perpetrator or merely receives the information from intermediaries.

6.2.5 Conflicting Documents

An intruder who is allowed by the policy to insert documents can insert multiple *conflicting* documents. “Conflicting” in this context means that multiple different documents share the same name, version and originator signature[1] and which are consistent and correct when examined separately.

Such behaviour is detected and countered during the document replication procedure, when a peer receives an offer for a locally available document where the signatures contained in the offer do not match the document content. This detection will result in global blacklisting of the perpetrator in short order. Figure 6.1 shows such a situation, where a dishonest originator `R` offers two conflicting versions of a document to different parties. These parties propagate “their” versions onwards, and the conflict is detected when peer `B` sends an offer which connects the two groups for the first time.

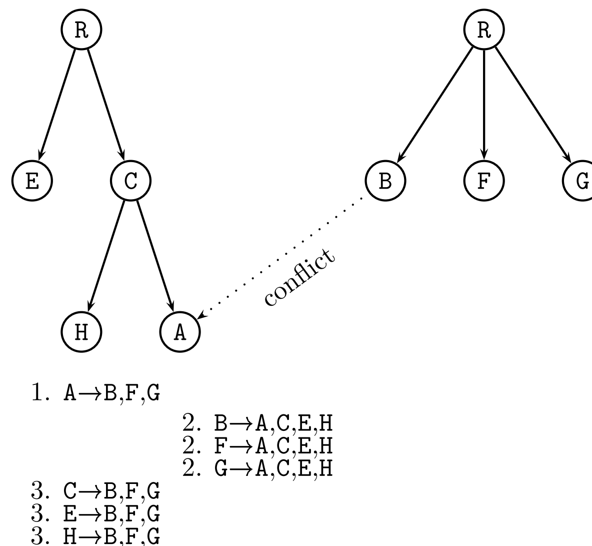

Figure 6.1: Detection of a Corrupt Originator

Dealing with this kind of misbehaviour is made a bit more complicated by the replication process not transmitting the document bodies with an IHAVE offer, while the verification of signature requires that document body.

[1] Multiple originators can not accidentally produce collisions of name and version because the version identifier is by design totally ordered.

The problem arises when peer A is offered a document by B with the signature block not matching the locally available document of the same name and version: if A just blacklists the offering peer B, it might hit an innocent party. The otherwise rigid policy of always immediately blaming the messenger for transmitting broken data must be adjusted for this scenario: it is impossible to determine whether the transmitting peer or the document originator is at fault if only a signature block is available.

Using the example given in Figure 6.1, we assume that peer A has a signature set SigA matching the document content DocA, and that A subsequently receives an offer from peer B that contains SigB which is entirely unlike SigA. Without the data that apparently underlies the signature set SigB, A can not reliably determine who is to blame: B could have received a document DocB which is consistent with its signature block SigB.

But from A's perspective it is impossible to determine whether the problem stems from a corrupt originator that injected two different versions, or whether B has sent corrupt data. It is thus necessary to distinguish more precisely between a bad messenger versus a bad originator, by doing a follow-up retrieval of the offending document.

A must therefore request the whole document DocB from B: if the request fails, B is implicated as being faulty. The same happens if the document is returned and it turns out that DocB is equal to DocA, or if there is any inconsistency in SigB when checked against DocB.

However, if the root signature in SigB verifies and the originator of the document is the same (and if all other signatures in SigB are correct as well), then A can safely conclude that the originator has produced conflicting documents.

Note that merely attempting to verify the root signature from SigB against DocA is *not* sufficient: B could have tampered with that, and as the root signature is part of all other signatures, none would match the data DocA anymore. This looks exactly the same as a dishonest originator from A's perspective.

Once the conflict has been detected, it is necessary to inform others of the problem to effect fast convergence to the same state: the faulty originator is blacklisted. However, the peer A can not sign (or store) the document DocB after detecting the conflict, because that is against the Verifiability requirement of the Good Citizen criteria. Much less can A offer an unsigned document to anybody. How can the detecting peer A inform others, if it cannot send the conflicting document DocB?

The solution is to perform a recursive "ping-pong" among the peers involved in the deception: using the signatures present in signature block SigB, peer A sends offers of its SigA to all the peers that are part of the "other propagation branch". The recipients of this offer will have to perform an operation analogous to the one A has just completed, that is ask for the document belonging to SigA, detect the conflict and send notificational offers of their view to everybody who needs to know – which involves A, thus looping badly.

To break that loop, a peer must flag a document version as bad when such a conflict is found and stop accepting offers (regardless of the provenance of the signature block) thereafter. It must, however, still accept GET requests for this document so that others can get this peer's version and thus determine the conflict themselves. The offer exchanges listed in Figure 6.1 show the knowledge of the conflict travelling through the whole group.

This recursive scenario is very simple but inefficient, with a peer being part of one branch receiving lots of duplicate offers from peers on the other branch until everybody is convinced of the deception. However, we judge this inefficiency to be preferable over the risk of stale information.

So far, such conflicting documents were assumed to have the same originator, and in this case that peer clearly can be blacklisted, as it is definitely the responsible party. However, if the conflict turns out to involve two different originators, then the one peer whose document version was not conforming to the design specifications should be penalised (but administrator alerts should be generated also).

After a conflict is detected, neither version of the document is propagated any further, but all versions are still available for retrieval. In order to clean up such situations, an administrator is expected to supersede the conflicting version with an empty document, thus allowing all peers to delete the conflicting data.

6.2.6 Communication Misbehaviour

As outlined above, *activities* within the framework of `Peeranoia` are fairly easy to counter. Instead, it is the *lack* of actions that proves to be more problematic, as the remainder of this chapter will show.

As all data to be exchanged is signed, a corrupt peer cannot send garbage more than once before being cut off; as signatures are organised in a (hierarchical) tree, they can also not be tampered with, and a non-verifiable signature leads to blacklisting of the sender as per the Good Citizen criteria. The same goes for not honouring document requests when connected: all of these are detectable and thus easy to counter. In all cases of corrupt data or inappropriate activity, the connection is dropped by the detector and depending on the severity of the problem, the other party is blacklisted locally.

Misbehaviour affecting the retrieval of previously replicated documents can be dealt with in a straightforward manner: a faulty peer can always answer a request for a document with a negative indication, or with valid but outdated data; it can also ignore subsequent reflexive offers by others, who attempt to inform the faulty peer of a newer document version to be used. However, retrievals are always performed with a certain degree of parallelism, and thus other peers will be queried as well.

A peer ignoring newer information thus must be counted as neutral: not providing full service but also not actively trying to impede progress as being made by others. The same neutral classification applies to a peer that does not communicate with others at all by neither accepting nor initiating connections, because this failure behaviour is visible to others who can easily work around it by choosing alternative, operational communication partners.

The most problematic malicious activities that a peer can exhibit are related to the replication of documents. These are, in ascending order of severity, tampering with signature blocks enroute and intentional ignorance.

- Tampering with signature blocks is the less problematic activity. Because of the construction of the signature tree, the only undetectable manipulation is removal of some signatures. This, however, does not keep the offer from being propagated further. As detailed in Section 4.6, a peer attempting to remove signatures will not be able to remove all of them, nor to remove them very selectively due to their overlapping hierarchical nature: there must be an originator signature and a signature by the sending peer to stay within the Good Citizen requirements. Any offer – no matter how extensively pruned – is a positive and helpful contribution towards reaching the distribution goal, as it is not wasted: any offer informs the recipient of a document's existence and triggers further action by the recipient, which eventually and with high probability will lead to full distribution.

  Also, the effects and possible extent of tampering with signature trees are not overly interesting in the face of the fact that a saboteur would gain more disruption by the simpler expedient of being willfully ignorant.

- "Intentional ignorance" is interpreted as a peer which *appears* to others to be functional, but without ever contributing: offers are ignored, documents are not signed, requests are answered negatively or with outdated data. The effect of such a "silent" peer for the replication process is that of an informational black hole: others send it offers, in expectation of cooperation and thus furthering of the document distribution, but waste their efforts on this peer, thus decreasing the probability of the data reaching all honest peers.

In the following sections we provide our arguments showing that `Peeranoia`'s replication mechanism is sufficiently robust to counter silent peers.

6.3 Replication Analysis

As pointed out in Section 3.6, there can be no ultimate guarantee of a document reaching all peers during the replication procedure. This is caused by using rumour mongering as distribution mechanism, and by

the relaxation of the consensus goals, which was chosen to improve efficiency and reduce the necessary timing guarantees.

From a practical perspective, the cause of potential stagnation of the replication without reaching everybody, is that communication choices are probabilistic and independent: it is therefore possible (though unlikely) that by random chance *all* other peers choose not to contact a particular peer.

Therefore, the best attainable property for the offer-based replication in `Peeranoia` is to reach all peers *with high probability.* Our claim is that `Peeranoia` reaches this goal, and in the remainder of this section, we will provide our reasoning to support that claim. Furthermore, it can be shown that the probability of failure can be made arbitrarily small, which suffices for the `Peeranoia` scenario. We will also outline some extra pull measures that can be used as a backup of offer-based replication, in case a strict guarantee of global consistency is required.

Our main argument can be outlined as such: correct peers forward every interesting offer to multiple other peers in parallel. While a single message can certainly be lost, or be "uninteresting" to the recipient, no *single* message is crucial to overall success. As peers are autonomous, the likelihood of *all* messages reaching only a subset of the peer group diminishes sufficiently to make our system viable and safe in practice.

6.3.1 Simplified Model

To examine the offer-based replication analytically, we have developed a simplified model for the propagation of messages, which provides a lower bound for the probability of successfully reaching all peers. As the model necessarily includes simplifications, we discuss the main differences between model and practice and the applicability of the model in the next section.

For modelling, we focus on a single message that is distributed by an originating peer among a group of correctly operating peers. In this simplified model, correct operation is defined as "every non-duplicate message received is forwarded to multiple other peers".

Let G be the number of peers in the group and N be the degree of parallelism in the replication. The simplified model does not use the notion of different document states and does not vary this degree of parallelism through the propagation process.

Let t be a global logical time or round counter, reflecting the number of processed offers. This reduces the model to a synchronous model which is the standard for analysing epidemic algorithms[52]. Every offer a peer receives and handles increases t and exactly one message is handled at time t. As messages are sent in parallel and asynchronously, the choice of the next message and peer to be considered is arbitrary (but unimportant for this simplified model). $t = 0$ denotes the time when the originating peer is given the message to distribute.

Let $i(t)$ denote the number of informed peers at time t (i.e. the peers that have seen the update at this time); $i(0)$ is defined as 1 (the originating peer knows the message but has not sent out its offers yet). Peers that are target of a message at time t are counted towards $i(t)$, even if they have not yet handled the message themselves. Let $q(t)$ be the number of "queued messages" at time t: the number of peers that have received the message before or at time t but have not processed it yet. q must be considered as we are dealing with multiple messages being generated at the same time, of which only one is disposed of in a single round. $q(0)$ is defined as 1 (the originating peer is waiting to operate).

Let $\delta(t)$ be a number between 0 and N inclusive that denotes the number of new peers that are targeted in disposal of the message at time t. $\delta(0)$ is fixed at 1; $\delta(1)$ is fixed at N (the originating peer *can* only choose new peers at this time). At any time t, exactly N peers will be chosen as targets of the newly created message; however, there is no guarantee that we choose only new peers (or any new peers at all). For all other t, $\delta(t)$ is a probabilistic function with values in the range $[0, N]$. In the simplified model its values are chosen according to the probability of randomly choosing peers out of the group (but excluding the operating peer itself).

$i(t)$ is recursively defined as $i(t+1) = i(t) + \delta(t)$: the number of informed peers at a time depends on the state of the previous round and how many new peers are contacted in this round.

$q(t)$ is recursively defined as $q(t+1) = q(t) + \delta(t) - 1$: every round handles one message, and generates up to $\delta(t)$ new messages to be handled in the future. If we do not choose any new peers in a round, then we start working off the queue.

The model has two termination criteria: $q(t) = \delta(t) = 0$ means we have run out of messages to process and $i(t) = G$ which means we have reached our goal of informing all peers. Figure 6.2 shows the model in C-pseudocode.

```
t=0; i=1; q=1; delta=1;
while(1)
{
  t=t+1; q=q-1;
  /* compute delta as delta(t) */
  i=i+delta;
  if (i==G)
    /* terminate: success */
    break;
  q=q+delta;
  if (q==0)
    /* terminate: failure */
    break;
}
```

Figure 6.2: Simplified Model in C-pseudocode

Obviously our goal is to show that $i(t)$ equals G at some time with high probability.

$i(t)$ is obviously monotonically increasing with t: if a peer receives a duplicate of a message it has seen earlier, the message will be discarded and the number of informed peers stays constant. If a new message is received, it is sent out to N others. At best, $i(t+1) = i(t) + N$; at worst $i(t+1) = i(t)$ as these N offers could be perceived as duplicates by all the recipients. Thus $i(t)$ is not strictly monotonically increasing.

To prove that $i(t)$ reaches its maximum G with high probability, we look at the overall probability of success and show that it increases strictly monotonically with N. The only way for the model to terminate with failure is to select duplicate peers in multiple subsequent steps until the queue is exhausted.

Increasing the parallelism to $N+1$ means that every message reaching a new peer now produces up to $N+1$ queued messages (up from N). Looking at this from the perspective of an automaton, we gain another positive transition in every step t and the probability of the one non-contributing, potentially bad choice (of reaching no new peers) goes down. Thus higher parallelism drowns the probability of bad choices.

The probability of function $\delta(t)$ producing a value k is finite and computable, the tree of possible transitions is finite (if large) and thus the number and probability of positive terminations versus bad terminations can be computed.

As $\delta(t) = k$ with a probability equal to that of choosing exactly k new peers out of $G-1$ peers, we can express the probability in terms of t, $i(t)$, N and G as such:

$$P(\delta(t) = k) = \prod_{m=1}^{k} \frac{G - i(t) - m + 1}{G - m} \prod_{m=1}^{N-k} \frac{i(t) - m}{G - k - m} \binom{N}{k} \tag{6.1}$$

The first term represents the probability of choosing k previously uninformed peers from the group, the second term specifies that $N-k$ informed peers are chosen as well and the binomial covers the number of ways to pick k out of N.

The sum of the values of $\delta(t)$ over time t is the cardinality G of the peer group, if a sequence of target choices is successful. As there is a finite number of sequences, we can compute the overall probability of success as the sum of the compound probabilities of the decision sequences where the model terminates successfully; each such sequence has as probability the product of the probabilities of the individual choices at each time t.

With increasing P, the number of possible sequences rises for $N=2$ and 3, then decreases monotonically down to a minimum of 1 (with $N=P-1$, there can be no bad termination). With increasing N, the possibilities of failure decrease as more messages are available which all have to terminate as duplicates for overall failure. Figure 6.3 shows the results; we note that the probability of success follows $1-\frac{G}{2e^N}$ closely.

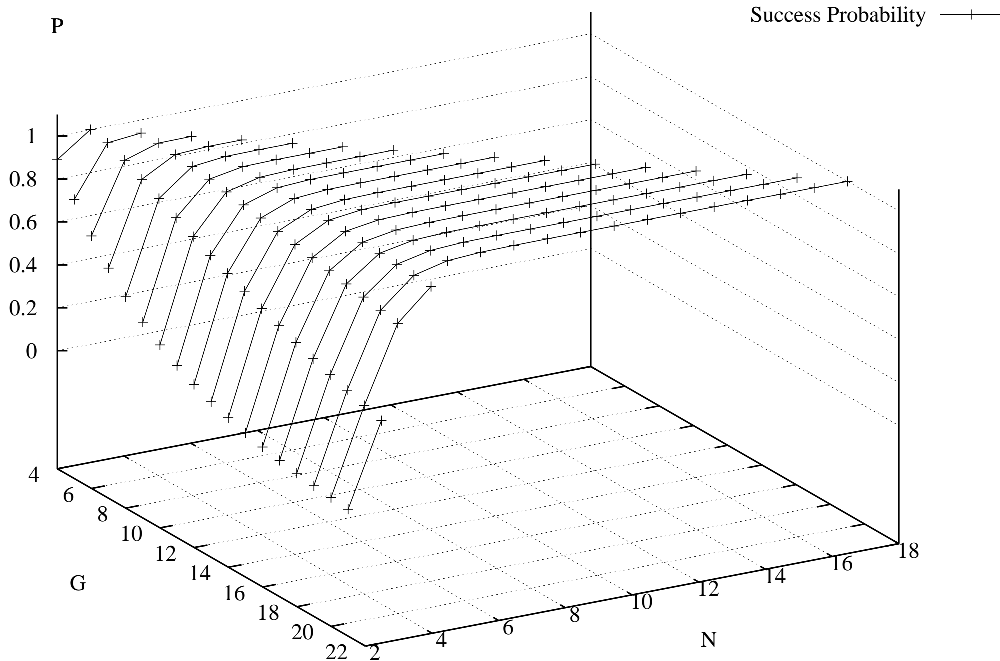

Figure 6.3: Simplified Model Performance

As an alternative, the model also can be interpreted as describing a non-deterministic finite state machine (or FSM) with ϵ-transitions. The nodes of that FSM are all the possible pairs (i,q), describing the overall state of involved peers and queued offers. The edges model the transitions that the handling of one offer can produce: as such, the edges correspond to the various values of $\delta(t)$. The start node of the FSM is $(1,1)$, and the terminal nodes are the nodes (G,x) for all values of x (indicating success), and $(x,0)$ for all $x<G$. All transitions are ϵ-transitions, but each has the probability of the event $\delta(t)=k$ associated. Figure 6.4 shows the knowledge state transitions for a group of size $G=5$ and degree of parallelism $N=2$.

6.3.2 Applicability of the Simplified Model

The above model is representative for the real system insofar as it establishes lower bounds for probability of success; the differences between the model and the real system ensure substantially better performance.

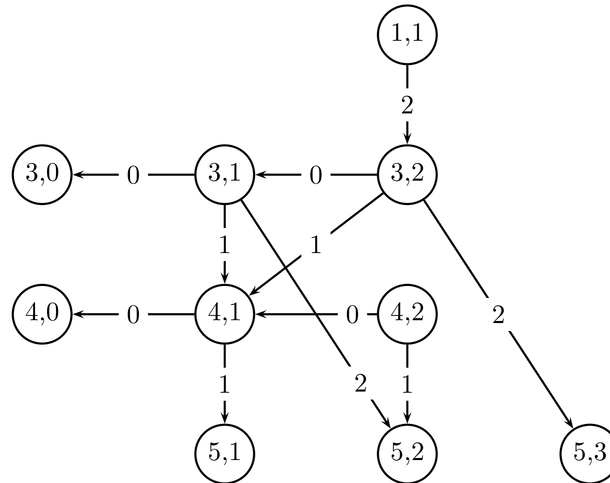

Figure 6.4: An example FSM for $G = 5$, $N = 2$

The real system follows roughly the same program as the simplified model: whenever a message is received, verify it, select a set of candidate peers and distribute the message to them as required. Ignore messages after everybody is fully informed or when the message does not indicate an information differential[2]. Therefore, the real system has at least the same probability of success as the simplified model.

However, the simplified model only covers a single message being distributed among the peers. In the real system, each peer contributes its own signature upon first reception of a document and thus produces another distinct message. Therefore we have G *related* messages to distribute (with G being the size of the group), and a failure scenario requires that *none* of these distribution runs reaches a particular peer which is of course much less likely than that of failure of a single run.

A more important difference, however, is that in the model all messages are equal and free of content whereas in the real system, messages have an incremental payload of signatures: every document acquires a history of prior exchanges which is self-verifying and allows peers to transitively infer information about the document's distribution state.

This has major effects on the classification of duplicate messages: in the simplified model, every message that reaches a peer multiple times is considered a worthless duplicate that does not contribute towards the goal of informing all peers. In the real system reception of a message is coupled with the payload that the message carries, and the payload *is* taken into account: only if the content of the message is exactly the same as known locally, is the message discarded as duplicate. In all other cases new information is exchanged and distributed further onwards and thus provides a contribution towards informing everybody of everybody else's signatures.

The history payload reduces the number of real duplicates (and thus the probability of failure) drastically. For example, in the simplified model an exchange from peer `C` back to originating peer `A` is a duplicate. In reality, the first communication reaching `A` from `C` is assuredly not a duplicate as `C`'s signature is communicated newly. Even subsequent exchanges from `C` to `A` can still convey new information, as long as either `C` or `A` have accumulated different message states in the meantime.

The time-and-path-reflecting nature of the messages' payload in the real system also allows us to design various candidate functions which use mechanisms other than simple random choice. Some of these provide substantial improvement to maximise information flow and minimise the probability of non-contributing duplicates. A discussion of such strategies and their effects follows in Section 6.4.

[2]This is a simplification of the strategical decisions required, which are discussed in more detail in Section 6.4.

The model does not take corrupt peers into account, but the behaviour in case of up to B corrupt peers can be incorporated with only minor changes and the main applicability argument remains valid. From the perspective of the candidate function $\delta(t)$, the bad peers are part of $i(t)$: talking to them does not result in any progress towards the goal as they will (at worst) disregard the message. The probabilities of the individual scenarios thus worsens; $\delta(1)$ is no longer certain to be N as we could choose up to B corrupt peers and so on. The termination criteria and transition mechanism stays the same. Of course, the likelihood of choosing duplicates or corrupt peers is now higher, but as long as the value of parallelism is sufficiently large with $N > B + 1$, at least one message per time t reaches non-corrupt peers and thus results in furthering the distribution.

6.4 Empirical Cost Analysis

To answer the remaining fundamental question of how much the `Peeranoia` mechanisms costs in terms of computation and communication overheads, the prototype implementation was used extensively to obtain analytical performance measurements.

In this section, we will describe the framework for these experiments, and the expected outcomes. As a number of interesting observations are made, some successful refinements of the (initially very simple) propagation strategy will be discussed as well.

6.4.1 Experimental Environment

The test environment included a number of separate computer systems, all of which ran up to 10 instances of the `Peeranoia` prototype application to simulate that number of peers.

Using different numbers of such physical machines, it was possible to simulate group sizes between 10 and 100; this upper limit was chosen arbitrarily, not because of technical restrictions but rather because we do not expect a larger *single* `Peeranoia` group in a practical situation.

The computer systems used for these experiments are of current standard PC specification (Pentium IV CPU, about 5500 MIPS, 1 GB main memory) and are connected using a private, switched 100 Mbit Ethernet. The communication protocol is TCP, with each peer listening on a separate IP address and port.

In early experiments we attempted to simulate larger peer groups on a single machine, to eliminate any negative effects that the network data transmission might cause. These experiments included groups of up to 40 peers, but were abandoned for two reasons: first, a number of memory leakage problems with the POE event handling framework became apparent and severely affected the reliability of the obtained results. The second and more important reason for reducing the amount of simulated peers per real computer was that the timing differences between only local and networked communications are negligible. Most of the workload of the application and incurred delay is related to cryptographic operations, and the extra latency of document transmission is insignificant in comparison.

Experiments were conducted in series with varying group size and strategy parameters. Each series included numerous repeated replication runs with documents with different policy requirements. A separate controlling computer system provides automatic remote administration of all simulation machines (using SSH), so that the experiments could be performed unattended.

Each experiment was conducted as follows:

1. First, the simulators are primed with the necessary initial information: the controlling computer transfers the pre-generated `peerlist` documents to all simulation machines. These `peerlist` documents are made to contain just the GnuPG keys of peers for the intended group size.

The accompanying policy document is simple, but sufficient for the experiments: it associates a number of document name prefixes with an increasing number of peer signatures for activeness. For the experiments no human administrators are used, as the goal was to establish measurements of the automatic replication components only.

The `peerlist` document is provided to every simulated peer with a full signature set, i.e. pre-signed by every peer in the group. This was done to avoid extra efforts to complete the integrity assurance process for this document, so that no factors besides the ones related to newly injected documents affect the performance observations.

2. After priming the simulation computers, the controller remotely starts the prototype application on the simulation computers. The application is given the appropriate command line arguments to select a propagation strategy, parallelism parameters etc., and its log output is collected on the controller.

3. The controller then injects documents for replication at one peer. These documents are very small (less than a kilobyte in size), and their names are chosen to match the given policy rules in turn, so that every number of required signatures is tested.

4. After document injection, the controller observes the combined logs from all simulation machines to detect the termination of the replication process. As the controller has access to the simulation computers' logs, it simply looks for new log entries; if activity has ceased, the experiment is considered completed.

5. When the controller determines that the replication process has finished, it repeats the experiment with further document injections. The experiments were repeated with multiple separate documents for every number of required signatures, in order to avoid artefacts in the observations that are caused by the random aspects of the `Peeranoia` procedures.

6. After the planned experiments with one set of strategy parameters and group sizes is completed, the controller remotely terminates the applications on the simulation computers. The application's start-up parameters persist for the life time of the application instance, therefore it is necessary to start the application again for different sets of parameters: the complete cycle is repeated for every propagation parameter set and group size.

The collated logs on the controlling computer are then evaluated. Timing and cost measurements are extracted and plotted to aid our analysis of the effects of different parameter sets, which will be given in the following sections.

6.4.2 Measurements and Expectations

The goal of the experiments was to verify the viability of the `Peeranoia` replicated storage system, focussing specifically on computational cost. The predominant component of computational expense is performing asymmetric cryptographic operations. As the main measure of cost, we used the number of offers that are exchanged, with the following rationale.

Clearly, any storage system's two main aspects are document injection and retrieval. The latter is an unsophisticated and cheap operation, whose cost is easy to determine: every retrieval operation requires a (fixed) number of parallel requests, followed by the same number of signature block verifications. The governing factor is the degree of parallelism, which is in a linear relationship with the cost.

The document injection case is more complex, with two factors contributing to the computational cost: first, every document's content must be retrieved (after an initial offer is received, see Section 4.9). Every peer performs exactly one such reflexive retrieval requests per document; thus this factor's cost is linearly related to the group size.

The second factor is the number of offers that are exchanged overall. As the handling of each offer requires multiple signature verifications, this is the predominant factor of computational expense; especially as offers are flooding the network in an epidemic manner.

The absolute minimum number of offers required for *fully* informing a group of size G is $2G - 2$ which is achieved by two rounds of offers in the fashion of a linear loop. The first round "infects" peers and the second round serves to distribute remaining signatures to everybody. Figure 6.5 shows this transfer for a small group of size $G = 4$, with the edges of the graph depicting the flow of new signature information *only*[3] Because such a linear transfer does not provide any redundancy whatsoever, this is clearly insufficient for a robust service and is thus a theoretical lower bound only.

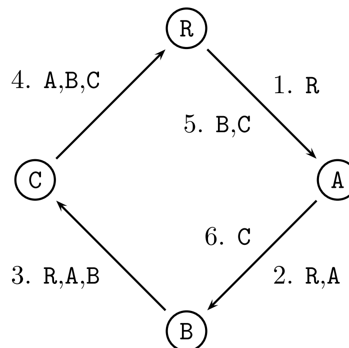

Figure 6.5: Linear Offers and the Flow of New Information

A more realistic expectation of mean message numbers would be G^2 in a group of size G: every peer must sign once, and every other peer most learn all others signatures. Clearly this can not be claimed to be a strict upper bound, because distributed consensus requires a degree of redundancy of messages. A second reason is that peers in `Peeranoia` usually have no up-to-date global knowledge, and therefore cannot synchronise termination of the replication process *precisely*. This leads to a certain degree of superfluous offers.

Therefore message numbers are expected to grow *roughly* squarely with the group size, but the construction of the offer payloads allows for a substantial reduction of messages: every offer can carry multiple new signatures. Therefore any particular amount of new information can be exchanged either with many offers carrying just one new element each or with a smaller number of combined offers. As we show in the following sections, with the best replication strategies that were examined, a reduction of about 50% of the required messages could be reached.

6.4.3 Data Extraction and Plotting

All propagation strategies were subjected to similar test schedules, varying all involved parameters: group size, parallelism and other propagation parameters, and number of signatures required until active.

- Tested group sizes ranged between 10 and 100, with the earlier strategies being tested only in groups of up to 40 peers.
- For the parallelism parameter, the evaluated choices were 2, 3 and 4; these correspond to a survivability threshold of 1, 2 or 3 corrupt peers, respectively.
- Different policy requirements were simulated, using documents with various numbers of required signatures for activeness. This number of required signatures varies between 3, 5, 8 and 10 for a

[3] Every offer contains the full signature knowledge at the sender, some of which is already present at the recipient.

group of size 10, and otherwise ranges from 5 to the group size, in increments of 5. For each such signature requirement, five separate runs were executed and the results averaged.

From the experimental results, four separate measures were extracted and used for comparison:

- the number of offers required until the document reaches activeness first,
- until the originator classifies it as active,
- until it is active at all peers,
- and finally the total number of offers.

Clearly these measures are monotonically increasing in the given order. On certain plots, these states are labelled in an abbreviated form as "f","o","a" and "d".

Separately, the timing behaviour of offer exchanges was studied. Plots of offers being received and sent, and of contributing versus duplicate offers over time have been included where appropriate. In such time plots, the state transitions for first, originator and global activeness of a document are also shown, with key "state" and small circular symbols (e.g. Figure 6.10 on page 95).

6.4.4 Strategy Evolution

The two major elements of a propagation strategy are the means of selecting candidates for offers and the criteria for termination of replication efforts. The main consideration is how to *intelligently* dispose of an offer that has been received at a peer, with two conflicting goals in need of balance: diffusion versus efficiency. On one hand, everybody should be informed as quickly as possible of the latest signature state but on the other hand we would prefer not to send overly much redundant information and thus waste bandwidth.

In pursuit of this balance, `Peeranoia`'s design aims for having a document reach activeness everywhere first, and operating efficiently afterwards. Applying this goal to the document propagation process, an offer must be sent out with a high degree of redundancy while the document is only pending, so that malicious peers can not keep the crucial information from being distributed. As soon as the active state is reached, pushing as hard is no longer required and thus efforts can be reduced (but not terminated immediately so that full distribution can be reached).

This strategy is similar to the one suggested by Datta et al. in [51], where the probability of sending further offers is reduced as the propagation progresses over time. The rationale in both scenarios is that rumour mongering loses efficiency the fewer uninformed parties are left; the reduction of offers allows a graceful tapering off of the efforts.

In the `Peeranoia` design all evaluated propagation strategies operate along similar outlines:

- If a received offer does not indicate an information differential between sender and recipient, then it is considered to be a duplicate and is ignored.
- Otherwise, the offer triggers the selection of candidates for subsequent offers to be sent. The exact mechanisms of the candidate selection functions differ, but all simply return a list of other peers.
- If the document is pending, a high degree of parallelism is used: this number of peers are chosen from the candidate set and offers are sent to them. Most strategies use a fixed number for this parameter, which we will denote as H henceforth.
- If the document is active, a lower degree of parallelism L *can* be used. Most tested strategies use the lowest practical parallelism parameter, 1, for this stage, but some variations were examined.

Clearly, the accumulation of knowledge across the whole group leads to an increase of offers being classified as duplicates and dropped, which in turn lets the propagation run out slowly.

As document activeness is only observed locally, *every* propagation strategy will introduce unwanted duplicate offers and fall short of absolute precision. However, we argue that robust trade-offs do exist and present some possible approaches in the following sections. First, we focus on the candidate selection function and examine some variants thereof in Section 6.4.5 to 6.4.9. This is followed by a discussion of alternative termination criteria in 6.4.10.

6.4.5 Naïve Flooding

The first successfully implemented strategy followed the outline above very closely. A fixed, high degree of parallelism is used until activeness is reached, after which parallelism is reduced to 1, i.e. linear forwarding of offers only. This simple strategy uses purely random candidate selection, excepting the peer which runs the selection function, and with the addition of offering newer information back to the sender reflexively when needed.

Experiments with this strategy were conducted for group sizes 10, 20, 30 and 40 and with parallelism 2, 3 and 4. The observed performance in different group sizes was comparable. Total offer numbers were observed to be growing almost linearly with the required signatures, and remained below $0.65G^2$ for all group sizes and $H = 2$.

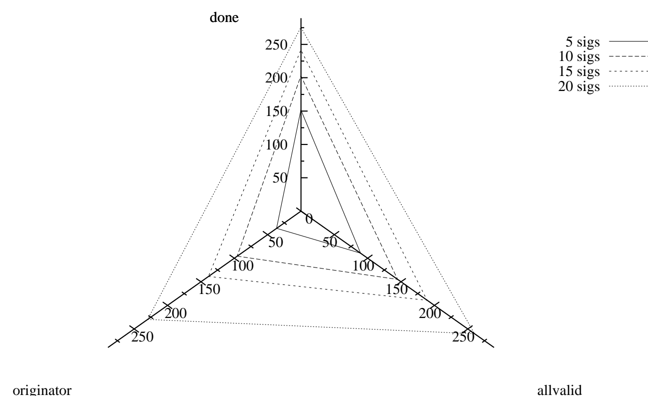

Figure 6.6: Offer Numbers, $G = 20$, $H = 2$, naïve flooding

Figure 6.6 displays the number of offers required for the state transitions from pending to activeness at the originator, further on to activeness everywhere and to final completion of the document propagation. The numbers of required offers are plotted for documents requiring different levels of integrity assurance (i.e. different signature requirements). The offer numbers span almost isomorphic triangles, indicating that the *relative* cost of reaching a particular state does not change much for higher signature requirements. For low parallelism, $H = 2$, the propagation strategy is clearly viable.

Figure 6.7 elaborates on the increasing costs for higher levels of assurance: the number of offers required for any particular state transition rises almost linearly with the number of signatures required for activeness. The unguided, naïve nature of the data exchanges is also visible in the extensive difference between the first peer reaching activeness versus the time when the document becomes active at its originating peer.

The almost linear cost increase is less pronounced in larger groups, as shown in Figure 6.8 which contrasts the largest tested group of 40 peers with the small group of Figure 6.8. For low assurance levels, it can be observed that many peers become active early, all peers are active after about half the overall offers

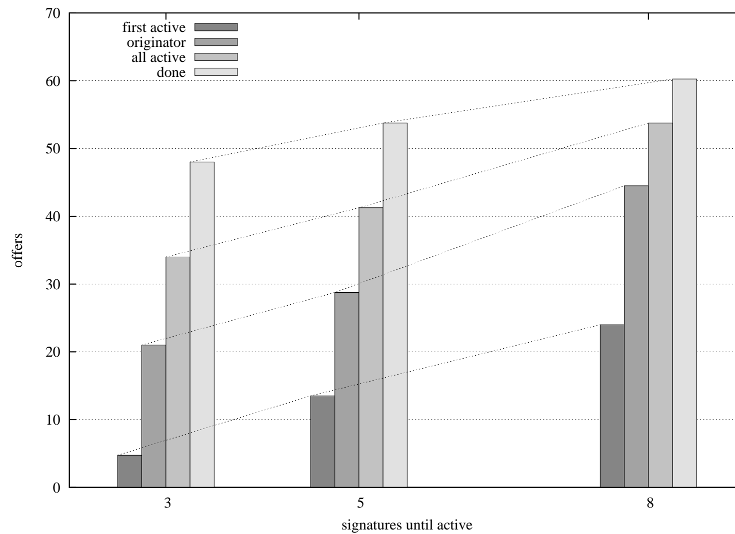

Figure 6.7: Offer Numbers, $G = 10$, $H = 2$, naïve flooding

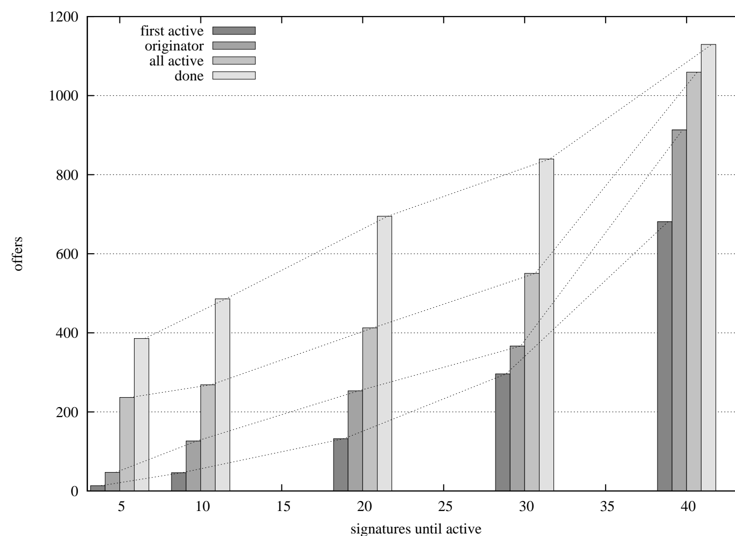

Figure 6.8: Offer Numbers, $G = 40$, $H = 2$, naïve flooding

and full synchronisation is reached with a long trailing phase. With increasing signature requirements, the trailing phase is reduced down to a minimum — but which is still greater than zero: even when all peers' signatures are required for activeness to be reached, some offers are generated *during* that transition which are received post-transition (and discarded as duplicates).

These previous three graphs clearly show that with low parallelism $H = 2$, the offer cost is below $0.65G^2$ for every group size and even the highest number of signatures required before activeness.

Unfortunately, a very much reduced performance has been observed for higher degrees of parallelism. Figure 6.9 plots the required offers for the various state transitions as surfaces, for a medium-sized group of $G = 20$. The almost linear increase of cost with higher signature requirements is again evident in the right-to-left slope of the surfaces, but the required number of offers do increase less favourably with the degree of parallelism. The total cost doubles between $H = 2$ and 4 (for the group of 40 peers, the

required offers actually *quadruple*); with degrees of parallelism H above 2, the expected offer number of G^2 is exceeded drastically.

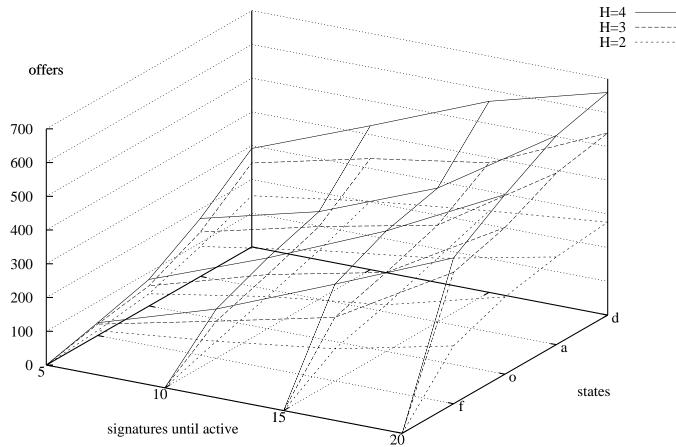

Figure 6.9: Offer Numbers, $G = 20$, naïve flooding

To assess the reasons for this disappointing decrease in performance, the timing of offer exchanges and transported information were examined. In Figures 6.10 and 6.11, we plot the number of offers being sent against time, and also discern between offers that transported new information and those that the recipient discarded as duplicates. The plots also include the state transitions for first activeness, originator and all active (which occur always and necessarily in this order[4]).

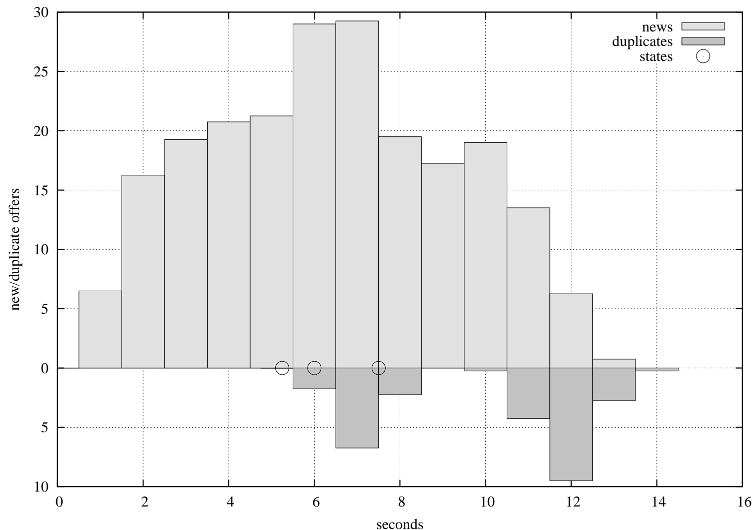

Figure 6.10: Offer Timing, $G = 20$, $H = 2$, 15 signatures, naïve flooding

Figure 6.10 shows that for low parallelism, the ratio between contributing and duplicate offers is very good: the first and few duplicates occur only after the active state is reached at some peer, with the majority of the duplicates happening in the final trailing phase where some duplication is inevitable.

[4]Because of certain technical restrictions, only a single symbol is available for all state transitions in these timing graphs.

However, it is also apparent that within a very short period of time, a lot of offers are generated: after an initial exponential ramp-up, every second more than G offers are sent, each of which triggers another $H = 2$ further offers.

While almost every offer contributes *some* new information to the overall goal of global synchronisation, the information density of an individual offer is not very high – and every offer requires expensive cryptographic verification measures to be performed. The large number of offers exchanged at the beginning of the distribution run becomes more pronounced with higher degrees of parallelism, and constitute veritable "offer storms", with lots of messages being exchanged quickly but for very little informational gain.

Figure 6.11 depicts this behaviour for $H = 3$ in a larger group of 30 peers: the initial exponential ramp-up, where new peers are involved for the first time, occurs during the first 4 seconds, after which all are busy flooding the group, and every second sees almost $2G$ new offers being sent. This rate of offers is sustained until just after everybody has reached the active state; the ratio of contributing versus discarded offers starts to deteriorate a lot earlier, after the first peer classifies the document as active.

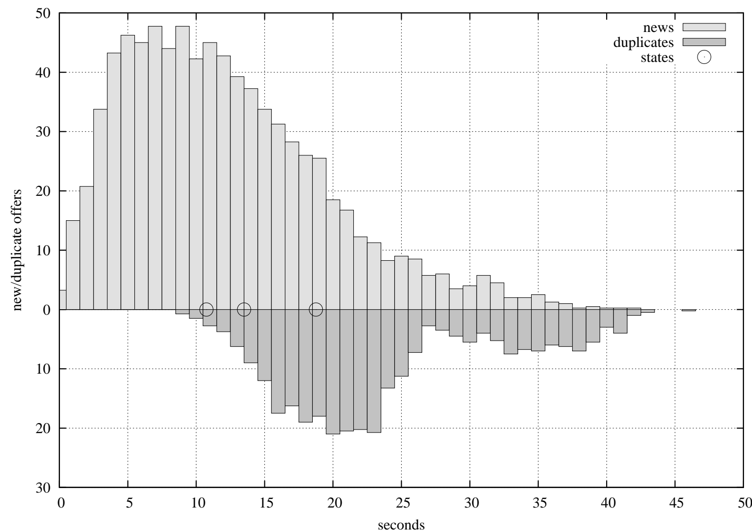

Figure 6.11: Offer Timing, $G = 30$, $H = 3$, 30 signatures, naïve flooding

After about 20 seconds, we can see that all peers have reached the activeness transition and now more data is discarded than positively incorporated. In this larger group the random nature of choosing peers for offers reduces efficiency especially badly in the later stages of distribution, where *most* information is already replicated everywhere: the likelihood of randomly targeting one of the few peers still needing a particular item of knowledge is low, thus the amount of duplicates increases.

From these timing graphs two main causes for this inefficiency can be identified: random, unguided choice of targets and bad offer timing. Both deficiencies were addressed in different strategy variants, as the next sections detail.

The first problem tackled is the suboptimal offer timing that the naïve flooding strategy exhibits. A single offer can carry potentially multiple new signatures (and thus reduce the number of offers required to distribute the full set of information). But with the naïve propagation strategy, offers are sent out *immediately* upon reception and completion of the candidate selection process. Peers do not wait long enough to actually accumulate multiple signature sets before generating offers.

This is visible in Figure 6.11 where large number of messages were exchanged quickly, most of which indeed carrying new information; subsequent examination of the logs showed that during the early stages of the distribution, each message generally carried only a single new signature. The higher the degree

of parallelism, the more pronounced this problem becomes as every offer triggers another H subsequent offers. On the other hand, the propagation is reasonably fast.

6.4.6 Fan-In

To reduce the unnecessary duplication of efforts during these offer storms, the idea of a fan-in period was considered and eventually implemented with good results. The idea is to allow peers to accumulate and combine more knowledge into individual offers (at the cost of some artificial delay), thus reducing the overall number of required offers.

Every received offer that indicates an information differential triggers subsequent offers to be sent by the recipient. With the fan-in variant, a random delay is introduced between reception of an offer and the transmission of any triggered offers[5]. During the delay period, receiving additional offers for this document leads to more signatures being available at the peer in question; all of this knowledge is incorporated into one offer just before it is finally sent. Furthermore, during the delay period no extra offers of the same document will be scheduled to be sent to any particular peer.

Experimental results suggested that a delay uniformly distributed in $[0, 30]$ seconds is practical. Figure 6.12 succinctly shows the resulting average phase shift of 15 seconds between offers being triggered and being sent out; the slowed down ramp-up of involving multiple peers at the start is also clearly visible. During the pronounced trailing consolidation phase post activeness everywhere, fewer and fewer new offers are created, as less and less new information is available to be distributed. The main negative aspect of this variant is very obvious in this figure, too, with the distribution now requiring about 90 seconds.

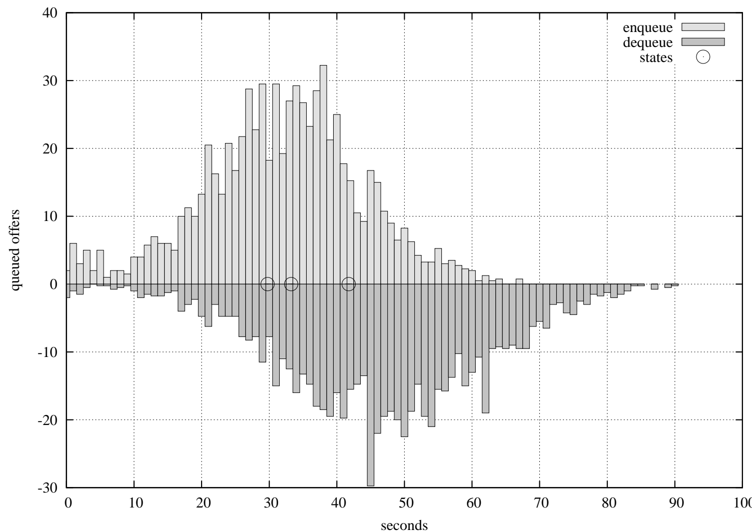

Figure 6.12: Queue Timing, $G = 40$, $H = 4$, 20 signatures, fan-in $[0, 30]$ seconds

This trailing phase is illustrated quite clearly by Figure 6.13: there are very few duplicates until all peers reach the active state. After this transition, most of the remaining offers that are dequeued over time, fall in the category of discardable duplicates and do not contribute to progress towards global synchronisation. This high number of duplicates is related to the second problem with the naïve flooding strategy, random choice of offer targets, and an improved strategy is described in Section 6.4.8.

Figure 6.14 provides an overview of the behaviour of the fan-in strategy with regard to different degrees of parallelism (in stark contrast with Figure 6.9 on page 95). It is clearly observable that the number of required offers differs very little between low and high degree of parallelism (visible as the close proximity between the surfaces). The delay helps very much to spread the load of more parallel offers over time.

Comparing the first activeness transition in Figures 6.14 and 6.9, we also note that the number of offers required for that transition is disproportionately lower with the fan-in variant. This is caused by each

[5]The initial offer by the originator is not delayed, because there is no potential for accumulation of knowledge yet.

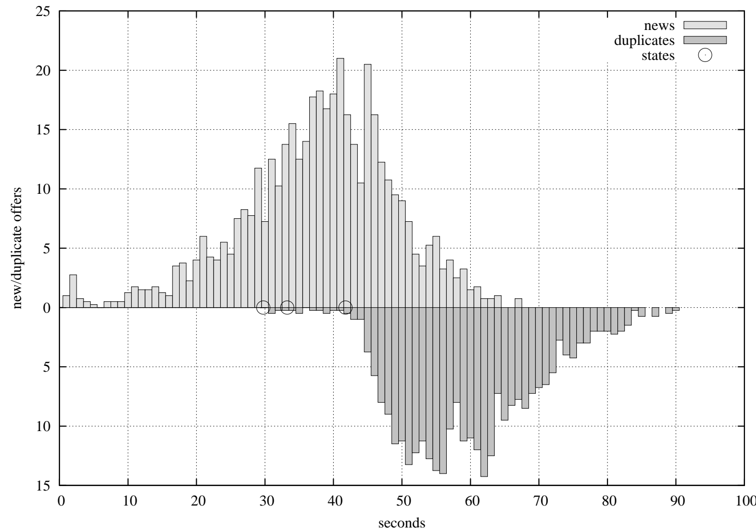

Figure 6.13: Duplicates, $G = 40$, $H = 4$, 20 signatures, fan-in $[0, 30]$ seconds

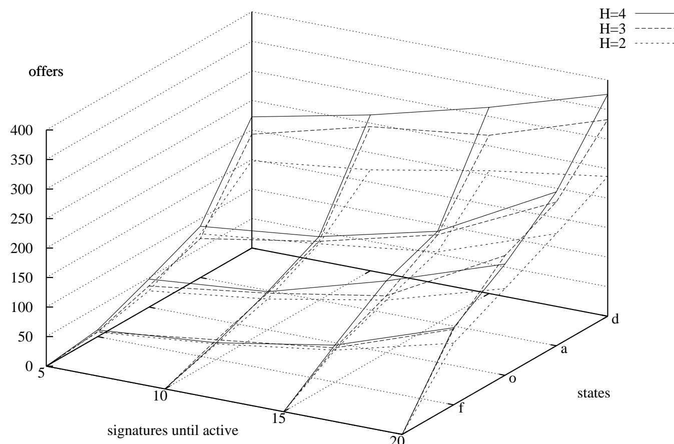

Figure 6.14: Offer Numbers, $G = 20$, fan-in $[0, 30]$ seconds

offer carrying much more information on average.

Overall, the addition of the fan-in delay resulted in a substantial reduction of required offers, but at the cost of increasing the time required for replication. As `Peeranoia` does not aim for real-time updates, we consider this longer replication time much less important than the reduction in computational cost.

Experiments with different fan-in values showed that the potential savings effect of the delay is proportionally related to the number of peers that are informed and actively propagating a document. This means that a fan-in delay in the earliest stages of document insertion has less effect than later. This suggests another potential measure for cost reduction in the form of an adaptive fan-in period: depending on the group size and how many peers are involved, the fan-in could be increased until the document is valid and then decreased again for the trailing consolidation phase.

However, this variation was not pursued further for two reasons: from the duplicate offer plots it is

already clear that very few offers up to activeness everywhere are duplicates; this is the phase where fan-in provides most benefits. Second, this variation does not address the reduced efficiency caused by the large amount of duplicates during the consolidation phase.

6.4.7 Variable Degree of Parallelism

The trade-off between efficiency and redundancy has been visible quite distinctly in the propagation strategies discussed so far, with redundancy being favoured by the naïve strategy. But adding a fan-in delay is clearly just one possible modification for this balance, and some others have been examined as well.

An attempt was made to control the amount of redundancy *adaptively*, using feedback from the propagation process in form of the document state: while the document is pending, a very high value of H is used. Subsequently, this number of parallel offers is reduced gradually, controlled by the number of signatures the document has already acquired. This adjustment is performed independently by every peer which is sending offers, as activeness is observed locally. The expectation was an improvement of the speed of convergence by getting most peers involved quickly and thus earlier state transitions.

This variation was tested with the degree of parallelism tied to the number of *further* signatures required for activeness: Starting with 10 offers at the onset, the number of parallel offers is ramped down to 2, until activeness is reached. After that, both variants behave the same with $L = 1$.

Figure 6.15 shows the effects of this modification, contrasting it with the fixed parallelism variant (both employing a fan-in in $[0, 30]$ seconds). We observe that initially a large number of offers per second are generated, which leads to a quick activeness transition; state transition times are reduced by more than half, and so far the expectations were met. However, the time until overall completion does not decrease markedly, because the overabundance of offers during the early phase causes a large number of queued offers that prolong the trailing phase of post-activeness consolidation.

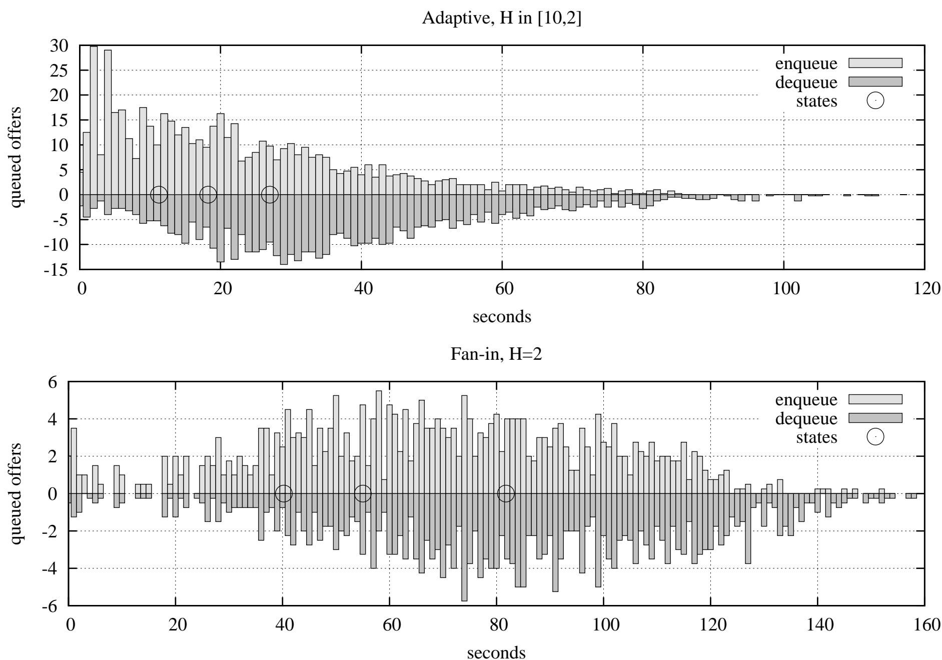

Figure 6.15: Timing Comparison of Adaptive and Fan-in Strategies, $G = 30$, 10 signatures

Figure 6.16 demonstrates the excessiveness of the initial distribution phase even better: the high degree of parallelism at the onset produces substantially too many offers. All the state transitions require more offers with this adaptive variant than with plain fan-in, with the extended trailing phase between all active and completion being a major cost factor. Also, the number of offers until overall completion occasionally exceeds G^2 — while the fan-in variant stays below $\frac{G^2}{2}$.

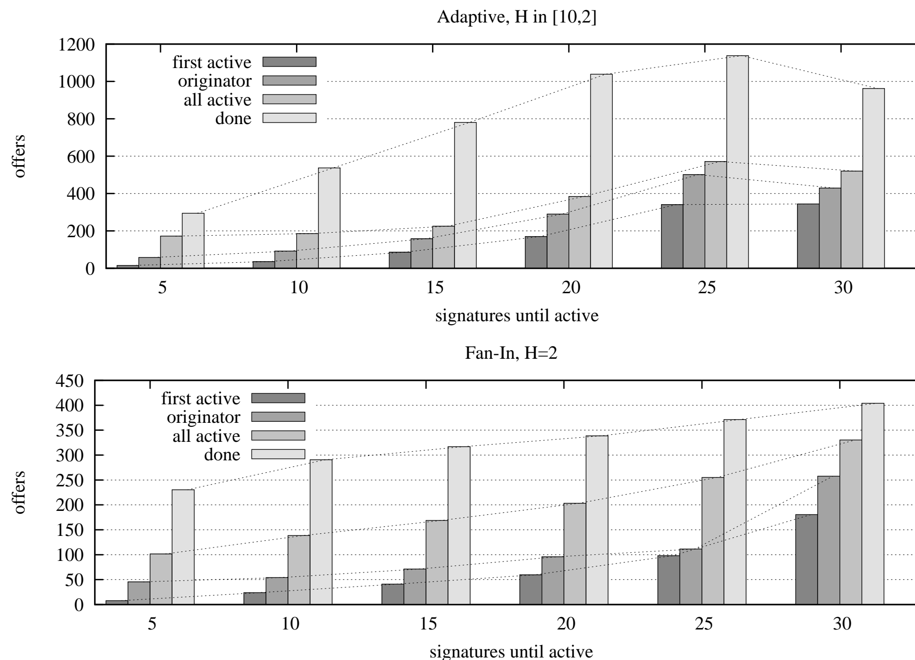

Figure 6.16: Offer Comparison of Fan-in and Adaptive Strategies, $G = 30$

The reasons for this increase of offers are that the candidate selection is still purely random, and that the high parallelism during the sending of the very first offers lowers the amount of transported new information per offer. This degraded ratio completely offsets the benefits of fan-in and makes the adaptive strategy actually perform *worse* than the naïve flooding strategy. In our opinion, the increase in required offers is of higher importance than the reduction in time, and therefore this variation was not developed any further.

6.4.8 Directed Flooding

After examining possible timing and parallelism parameters in the previous two sections, more focus was given to the candidate function: purely random choice clearly does work, but can there be improvements? Does the pool of candidates for an offer really have to be the whole peer group?

The time-and-path-reflecting hierarchical nature of the offers' payload suggests that a candidate function could use this extra information. The expectation is that choosing candidates with the exchange history factored in should allow the peer to detect that certain information is already known to particular other peers; these others could therefore safely be excluded from the candidate set, and the offers be sent where they still can "do some good". This more informed directing of offers towards those in need of information should maximise the information flow and reduce duplicates.

Unfortunately, there is no way to precisely infer the *complete* global state of knowledge from the locally available information and without direct communication with every other peer. Therefore, any heuristic mechanism trying to target peers in need of information will necessarily be imperfect, and a certain degree of duplicates is inevitable. Nevertheless, a lot of obvious duplicates can be avoided by examining a signature tree's topology and the local history of exchanges. To support this claim, a candidate function was implemented which peruses all locally available information, in order to reduce the candidate set to include only the peers that seem to need this information. This "directed flooding" strategy still closely follows the overall structure given in Section 6.4.4. Its main difference to other strategies lies in

the candidate selection function; the strategy can therefore be used with or without fan-in or variable parallelism.

The selection function requires that a history of past exchanges is kept: every peer must remember which signatures were directly offered to or received from other peers. This information is used during the selection to reduce the candidate set from all peers to only the ones that are “interested”. A peer being interested is interpreted as one where there is no local indication that the peer knows a particular signature yet. From these interested candidates, H or L are selected randomly and offers are sent to them.

The propagation termination remains mostly unchanged: duplicate offers are ignored, and after activeness the degree of parallelism is reduced to $L = 1$. There are two differences worth noting:

- Offers which indicate less available information at the sender are not reflexively answered (but of course the sender is definitely part of the interested candidates). This is a minor difference, which only limits the number of triggered further offers strictly to the degree of parallelism. With the naïve strategy, reflexive offers did not count towards that number.
- Offers are ignored if there are no more interested peers. This is of higher relevance, because of its potential for negative effects: if the decision about being interested is based on guesswork, then the peer might mistakenly exclude peers from the candidate set.

  For this reason, this decision is based only on verifiable strict knowledge. Every signature currently available to the deciding peer is compared to what is known about the candidate peer's knowledge. Only if the candidate definitely knows every one of these signatures is it excluded as uninterested. A peer is expected to know and remember a signature under the following circumstances:

  - if the candidate has been offered this signature by the deciding party in the past,
  - if it has offered this signature directly to the deciding party,
  - or if the candidate has signed the document in question, enveloping the signature being tested (see Section 4.6.4).

There were two main expectations for this modification:

- a reduction of duplicates (but not complete avoidance, as the signature state at the candidate peer could have changed between the last direct communication and the time of the propagation decision), and
- easier detection of corrupted peers, as a history of their actions is kept everywhere.

For comparability, the implementation of directed flooding was tested with the same choices for parameters H and L, and the beneficial modification of a fan-in delay in $[0, 30]$ seconds was also retained. Interestingly enough, the expectations were met only partially: the fan-in variation performs about as well as the directed strategy, both offer-wise as well as from a timing perspective. There is also quite some variation in most plots, with offer numbers fluctuating up to 10% between repeated experiments: offer timing and choice of communication peers are heavily affected by randomness, which produces extremely different signature knowledge structures. This in turn affects the amount of state information that can be derived from the signature structures, which can reduce the effectiveness of the candidate selection: candidates are only excluded if it is *certain* that they have the same knowledge as the deciding party. In all other cases, the candidate must be given the benefit of the doubt.

As an example scenario, we present again a group of size $G = 30$, and compare fan-in and directed strategy, for $H = 2$. Figure 6.17 contrasts the timing and amount of duplicate offers of directed and fan-in variants for one representative signature requirement (but averaged over a number of experiments); it shows that there is only a slight reduction of duplicate offers. As expected, almost no duplicates occur

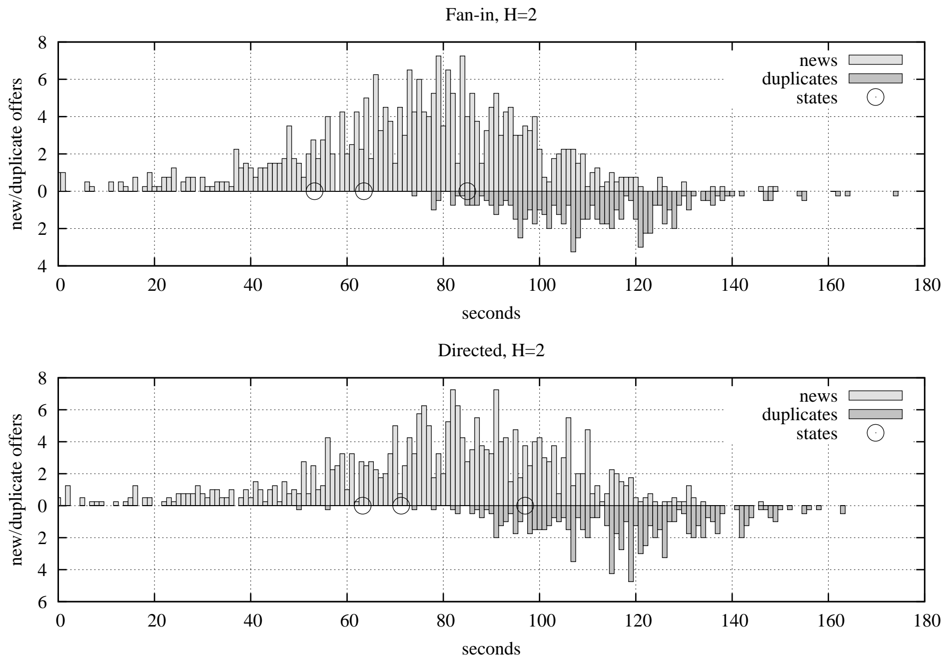

Figure 6.17: Duplicate Comparison of Directed and Fan-in, $G = 30$, $H = 2$, 15 signatures

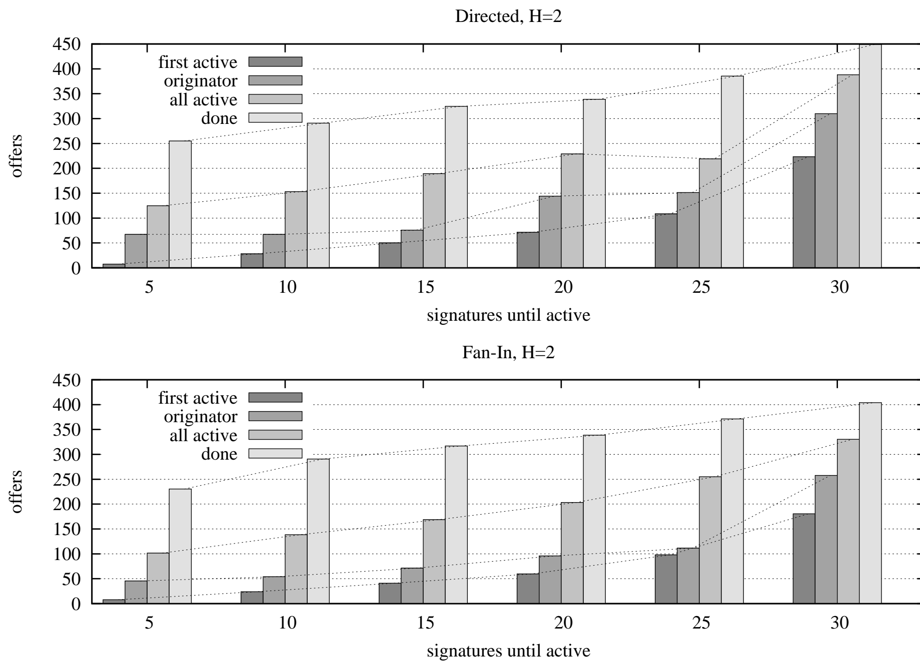

Figure 6.18: Offer Comparison of Directed and Fan-in, $G = 30$, $H = 2$

during the early stages of the distribution, because there is still much undistributed knowledge; during the last consolidation, the quality of the local estimates of still interested peers clearly deteriorate and duplicates happen more often. The graph also shows that the times for the particular state transitions are comparable, with the directed variant requiring a few seconds more for activeness but about the

same time for completion.

The number of required offers are compared in Figure 6.18 and again we note the lack of major differentiation between directed and fan-in strategies, at least for the low degree of parallelism $H = 2$.

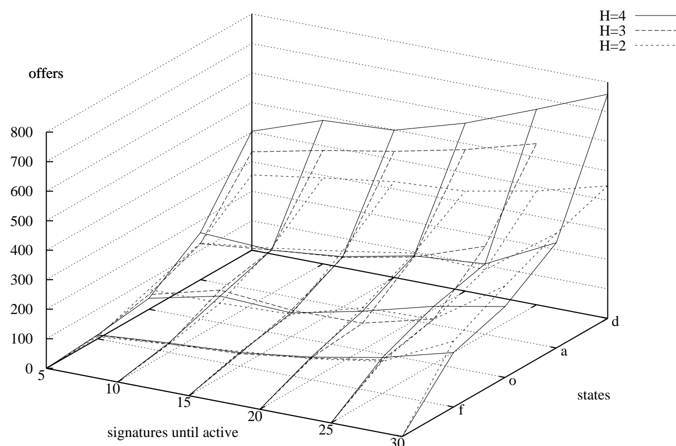

Figure 6.19: Offer Numbers, Directed Strategy, $G = 30$

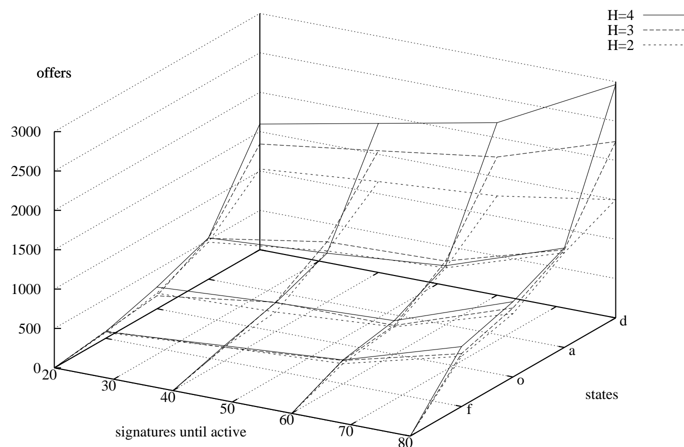

Figure 6.20: Offer Numbers, Directed Strategy, $G = 80$

A more positive outcome concerns this strategy's performance with different parameter sets: We noted that there is very little sensitivity to changes in the degree of parallelism.

As Figure 6.19 shows in the minimal distance between surfaces, the offer cost is very similar for all state transitions and degrees of parallelism, except for the offers for overall completion: here, the highest degree of parallelism produces surplus offers that are scheduled for transmission after global activeness is reached. As the prototype implementation does not purge offers once queued, these extra offers are transmitted during the final consolidation phase and degrade the otherwise good overall performance.

Figure 6.19 also shows the differences for various levels of integrity assurance clearly, with a mostly linear relationship between the number of signatures for activeness and the required offers to reach that state. This behaviour is evident across all the tested group sizes; a second demonstration of this for a group of 80 peers is given in Figure 6.20 .

Overall, the directed strategy performed sufficiently well, requiring at worst about G^2 messages for full distribution (with $H = 3$). This worst case was observed in the smallest group ($G = 10$), and the behaviour actually improves substantially for larger groups (down to $0.3G^2$ with $G = 100$). This clearly demonstrates the feasibility of the method.

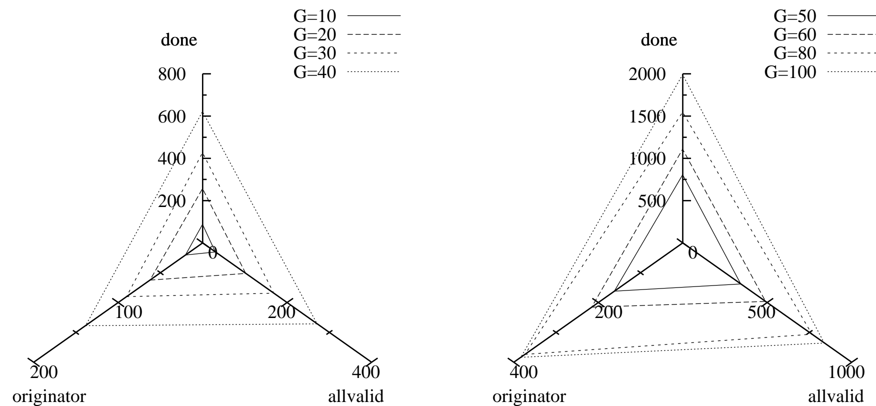

Figure 6.21: Offer Numbers, Directed Strategy, $H = 3$, $G/2$ signatures

Figure 6.21 displays an offer comparison for all group sizes that the directed strategy was evaluated for; the parallelism parameter was fixed at $H = 3$ and $G/2$ signatures were required for activeness: these were chosen as practical, realistic requirements. Higher degrees of parallelism do not substantially worsen this performance, nor does requiring more signatures for integrity assurance. We note that validity at the originator is usually reached after about a third of the overall offers, and validity everywhere after about half. The remaining half of the offers are expended during the final consolidation process; this characteristic prompted the consideration of strategy variants which stop earlier in Section 6.4.10.

6.4.9 Biased Flooding

As further variation of the candidate selection, it was considered to build a bias towards particular goals into the directed selection. The expectation was that in order to speed up activeness, it would be useful to concentrate signatures where they are needed, i.e. at the peers already involved. This should result in activeness at the originator fastest, with the remainder of the peer group being updated at a more leisurely speed and without a high degree of parallelism.

Such an "inward" bias was implemented on top of the directed strategy: the candidate selection function was modified to choose peers from the known signers preferably, and to fall back to contacting others only if there are too few involved parties.

After some experimentation it was realised that this strategy is flawed: document offers do not provide feedback to the sending party. In this inward-biased scenario, peers generally continue sending duplicate offers to the small group subset of already involved peers. As duplicate offers are ignored on the recipients'

side, the result is commonly complete stagnation of the propagation process: documents do not reach enough peers to become active at all.

The underlying reason that causes the dismal performance is the lack of knowledge of the global distribution state. The local state known at a peer represents the global state insufficiently well to be used for such biasing: a peer could effectively determine that stagnation is impending and decide to contact peers outside the involved parties, if and only if the peer had access to trustworthy indications of the signature state at all others. Unfortunately, information about others' knowledge does not filter back to each and every peer – by design: the transitivity of the exchanged information allows indirect knowledge transfers, which `Peeranoia` uses for efficiency. Contacting every peer directly to gain such a snapshot of the global state of affairs is clearly non-viable: it reduces the mechanism to the low efficiency of the original solutions to the Byzantine Generals problem.

With polling the global state being infeasible, stagnation can not be predicted precisely by a peer, because there is no such prediction mechanism that operates on local knowledge only and does not involve unwarranted trust assumptions. As an alternative attempt of salvaging the situation, other modifications to the candidate selection function were considered in order to avoid stagnation altogether.

Unfortunately, any such modification either introduces more duplicates than the bias ever could prevent in the first place, or evolves into performing further distributed consensus operations *within* the document distribution process. As an example of the first problem, we examine the idea of using duplicate offers as indicators of the need to contact peers outside of the involved group instead of dropping the offer. This produces lots of unnecessary offers because some duplicates are necessary for robustness: not every duplicate is indicative of impending global stagnation.

The second problem is closely related to Claim 1 of Section 4.6: It is not possible for a peer to prove a local *absence* of knowledge in a transitive, independently verifiable manner to others. If mere rumours are used to guide the replication process, either the number of duplicates increases due to the bad quality of the propagation decision, or worse, an intruder can negatively affect the global distribution by crafting suitably misleading rumours.

Of course, such indications of lacking knowledge *can* be integrity-protected and made verifiable, just not in a transitive manner: trying to communicate a peer's state to other peers more extensively (i.e. maximising local knowledge at the expense of extra offers) is basically distributed consensus yet again and thus prohibitively costly.

For these reasons it is futile to attempt to widen the audience adaptively when stagnation is impending, because only local information is available. That local information will either rarely reflect the global state *reliably* or would be too easy to manipulate for an adversary.

As a variation of the original biased variant, an outwards-biased selection was also examined: this selection function prefers uninvolved peers over involved ones. The original expectation was that the bias towards uninvolved peers would maximise the speed of document distribution. A similar argument regarding locality of knowledge versus global state applies and renders this strategy mostly unusable. With an outward bias, it is activeness at the originator that suffers because too little feedback makes it back to the originator, once peers are involved and the bias excludes them. But while not as absolute a disaster as the inward-biased variant, the outward-biased strategy certainly provides no improvement over directed flooding without bias; neither biased variations were pursued any further.

6.4.10 Alternative Termination Criteria

After discussing variants of the candidate selection function in the previous sections, the second major strategy aspect must be examined: the termination criteria. With all strategies presented so far, this was simply defined as ignoring offers when they indicate no information differential: because all propagation

activity is triggered by offers, the convergence towards full signature knowledge everywhere increases duplicates, which do not trigger further offers, and thus the propagation process tapers off gradually.

Obviously this is not the only possible way of dealing with when and how to cease replication efforts; because of the autonomy goals of the `Peeranoia` environment, any termination criteria can use only local information as decision basis. Taken together with the mistrust and efficiency design principles, this implies that any such criteria will be necessarily imprecise in their decision: repeatedly polling all others to get their precise state is prohibitively expensive and contrary to the efficiency goals. On the other hand, using indirect assurances of somebody's state (see Section 4.6) either requires trust in the messenger (which the mistrust principle forbids), or peers must repeatedly sign their state (which is extremely inefficient).

This imprecision can have two effects: stagnation short of distributing the whole signature set to every peer, or inefficient continuation of efforts after everybody is informed. The latter is clearly not a major concern. The former can be problematic, depending on the overall replication goal: unless the document has become active, it can not be used for FIC purposes.

But reaching activeness everywhere and full distribution of all signatures are not the same. As a reduced goal we propose to aim for activeness only in a subgroup centred around the document originator. With this goal, replication can be allowed to cease once an active document state is reached at the originator, and in the "compact" subgroup of peers that is spanned by the originator's initial offers.

We argue that this kind of limited replication can be useful where documents of local importance are involved. As discussed in Sections 3.9 and 3.6, most data in the `Peeranoia` environment is of local importance only and uses the replicated storage service only as a backup. The policy designates the number of peers that must be involved in storing and certifying the document, and with the reduced goal it would be expected that the document becomes active only at a subgroup of this size.

To examine this proposal practically, the propagation strategy structure was reviewed. It turned out that the only necessary adjustment for this reduced goal was to set the degree of parallelism for active documents, L, to 0. With this change, peers are allowed to cease sending any further offers once the document has reached the active state.

This early stopping was combined with the directed selection strategy and tested in groups of up to 40 peers. Interestingly enough, the independent random choice of interested peers still allows most documents to reach global activeness: the whole group is spanned by the offers. But past activeness everywhere, the propagation efforts cease very quickly (but not instantly, as offers that were queued before the state transition are sent when their fan-in delay expires).

Figure 6.22 shows a comparison of the directed strategy and the early stopping variant in a group of size $G = 30$ and with parallelism $H = 3$ until active; as expected, not a lot of difference is observed before activeness is reached everywhere and the state transitions are of comparable cost. The difference in the consolidation phase is very pronounced, of course: with the directed strategy, offers after activeness still produce a single further offer as long as new information is transported. The early-stopping variant simply stops generating new offers after activeness is reached, and only previously queued offers are processed during this late stage, thus the consolidation stage is a lot shorter and less costly. However, this comes at the disadvantage of potentially not fully informing every peer.

While positive results were obtained with this termination criterion, this moving away from the original goal of full distribution is a major change. We are of the opinion that this variant should not be the only termination mechanism in a practical deployment of `Peeranoia`; instead, it is suggested to extend the policy language so that an administrator can flexibly decide on the intended replication scope of documents. The performance of handling documents of local importance would certainly benefit from using this compact replication only.

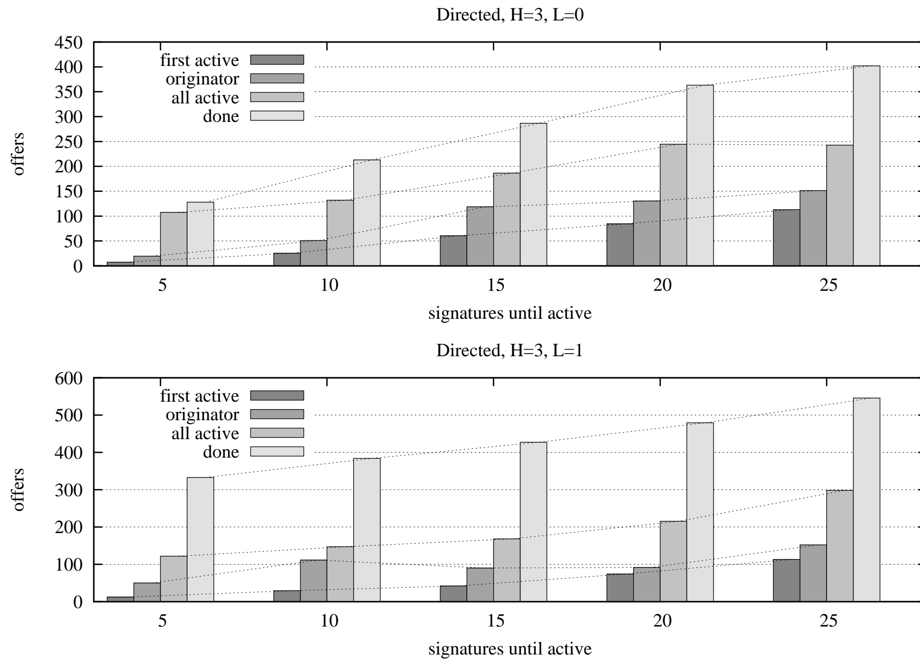

Figure 6.22: Offer Comparison of Directed and Early-Stopping Strategies, $G = 30$, $H = 3$

Such a policy extension could also provide a combination of early stopping and full distribution: if the parameter L is allowed to be chosen so that $0 \leq L \leq 1$, the result would closely resemble the probabilistic propagation decision that Datta's P2P protocol employs (see Section 2.9).

6.5 Summary

After developing the mechanisms and detailed functionality descriptions of `Peeranoia`'s components, this chapter focussed on analysing the resulting system, its expected benefits and associated costs. We covered the theoretical behaviour with regard to security threats, and also presented empirical cost measurements for a variety of document propagation strategies that evolved over time.

`Peeranoia` borrows conceptually from a variety of prior efforts. As is evident from our design principles and the balancing trade-offs that were chosen, the simplicity of systems like Bayou was very influential. `Peeranoia`'s minimal trust infrastructure resembles GNUnet's extensive trust economy to a certain extent, leavened by our desire to maximise autonomy. This resulted in data propagation protocols that share certain traits with Datta's P2P system. Overshadowing all these influences, however, is `Peeranoia`'s embracing of paranoid mistrust.

To limit the amount of mischief any small number of adversaries might produce, `Peeranoia` uses layered encryption and overlapping signatures for all documents. The overall result is that the `Peeranoia` service is robust in the face of arbitrarily faulty peers. The main threat that `Peeranoia` is susceptible to is denial-of-service: if an intruder controls the underlying communication infrastructure, or if numerous peers decide to sabotage the document propagation process by willful ignorance, then the service quality will degrade.

Note, however, that the service will degrade *gracefully* and in one specific regard only: full replication of data cannot be *strictly* guaranteed (unless inefficient replication parameters are chosen by the administrators), and in case of corruption of a majority of peers, the probability of having every party replicating everything is reduced.

Apart from full replication, all other security properties of the `Peeranoia` service remain intact regardless

of corrupt peers: document integrity is still verifiable and better than any local-only assurance could provide, data secrecy remains guaranteed and due to caching, a peer normally will be able to proceed with its FIC work even if *all* other peers are faulty.

For the case of no faulty peers, the simplified model shows that even with a very moderate degree of parallelism used in the document propagation process, the probability of reaching all parties is sufficiently close to unity to make the service viable in practice – especially as the model is necessarily much simpler and very much more inefficient than the actually implemented mechanisms.

This probabilistic robustness of the `Peeranoia` service is the result of balancing computational and communicational efficiency against mistrust; the `Peeranoia` service also allows an administrator to adjust this balance for every single document; thus, flexibility and adaptability for a wide spread of application requirements is given.

In the second half of the chapter, the practical usability of the service was examined and a cost analysis of a number of propagation strategies was presented. This analysis emphasises the predominant expected situation without faulty or subverted parties: inefficiency in this case would hurt practical deployment very badly. As `Peeranoia` does not need to provide real-time semantics for using in an FIC setting, the main measure of cost that was analysed is the number of required offers for full distribution: offer handling requires multiple signature verifications which accounts for the majority of the computational expense.

Showcasing the development of a number of improvements beyond the originally implemented naïve propagation strategy, it was shown in numerous experiments that a roughly square growth of messages with the size of the group must be accepted. However, heuristical exploitation of locally available knowledge at a peer can be used to fine-tune the propagation decisions. This introduces a significant reduction factor into the overall cost: the best observed strategy requires only about $0.3G^2$ offers in a group of size G for full distribution (and with a moderate parallelism of $H = 3$).

These observations also led to the examination of a variant which relaxes the distribution goals and allows for stopping short of reaching everybody, just after the document has become active. This variation provides further cost reductions, but at the cost of shifting away from the more desirable goal of full replication. Overall the empirical cost measurements clearly proved practical viability.

Chapter 7

Conclusion

The aim of this thesis was to evaluate whether it is possible to combine decentralised cooperation of untrusted parties with the security-critical needs of an FIC application, in order to provide improved data integrity assurance. Given the introspective nature of FIC (and most other IDS) systems, locally stored data is not overly trustworthy, and other commonly used approaches to remedy this problem leave a lot to be desired in the areas of scalability, ease of deployment and robustness.

In this thesis, we have evaluated the resulting problem domain extensively and identified conflicting design goals at the core of any such cooperative mechanism. Certain trade-offs must be made in the design of such a service. These are mainly related to the conflict between efficiency and robustness, but the goals of simplicity and autonomy also have some impact.

Building a service using cooperation between autonomous, independent parties is not exactly trivial, but it is definitely feasible. Introducing the extra requirement of not being able to trust any individual component complicates matters very much, especially when a decentralised environment is assumed: there are various systems for (strict) coordination of independent parties, some of which are robust in the face of malicious adversaries, and others are exceedingly simple – but none manage to combine all elements suitably.

We examined the requirements of an FIC application setting to identify a set of guiding design principles and trade-offs. The three principles used are Autonomy, Simplicity and Mistrust. After finding a workable balance between them, we used this to define our cooperative storage and integrity assurance mechanism. The resulting mechanism clearly demonstrates that a practical trade-off between independence and cooperation is possible, and that such a trade-off results in beneficial characteristics for an FIC application.

Apart from this more philosophical examination of the contrast between altruism and mistrust, we derived a behavioural specification of a *practical* service from these design principles: the `Peeranoia` service. The design details of this system clearly reflect the intended application domain, but sufficient flexibility was built in to cover a wide range of uses.

The resulting service provides tamper-resistant replicated data storage, utilises peer-to-peer techniques and provides robustness against malicious failure of a limited number of peers. At the core of the `Peeranoia` service lies the idea of using digital signatures to provide a trade-off between efficiency and mistrust: signatures provide positive proof of data integrity which can be verified by everybody independently. By exploiting this transitivity of the integrity assurance, we achieved a substantial reduction of the communication overhead compared to other distributed systems.

We then developed the specification into a prototype implementation and examined it analytically. This prototype was used to evaluate the communication and computational cost of our system for a variety of replication strategies and environmental scenarios. These experiments have shown the system to

scale into hundreds of peers. Besides proving the practical viability of the Peeranoia service cost-wise, the evaluation of the prototype's strategies also demonstrated some interesting possibilities for further research.

The chosen balance between the goals of autonomy and mistrust necessitates that the Peeranoia service relies only on locally available and verifiable information. This property, taken together with the efficiency goals, introduces the relaxation of consensus to eventual consistency, which is one of the few shortcomings of our system: there is no ultimate guarantee that data reaches every party for replication.

This is partially a consequence of the push-only data distribution model, and some works in the area of epidemic flooding[52, 51] suggest potential modifications to provide a better compromise between the simplicity of randomised (push) flooding and the reliability of anti-entropy (push-pull). The distribution mechanism could be adapted to switch to a periodic pull phase after the document has become active: the consolidation phase, when most parties are already involved, is where push flooding is least efficient. Another useful suggestion would be to have all parties periodically poll others for status, so that a peer can detect easier that it has been missed during a previous distribution procedure. While the Peeranoia specification currently does not enforce any such fall-backs to pulling, the protocol certainly already includes the required functionality: the HEAD command allows regular expressions in its arguments. In effect, such modifications would improve the attainable distribution guarantees to strict consensus again, but at the cost of extra communication overhead.

But even without strict consensus the integrity assurance provided to independent peers and the flexibility of our system already exceeds the capabilities of other (more centralised) FIC systems. Peeranoia does not require special-purpose cryptographic hardware or trusted networks for its operation, supports decentralised control by multiple independent administrators, and offers benefits to any application which needs tamper-resistant, secure, replicated storage with read-often-write-seldom semantics.

Based of the results presented in this thesis, we believe that Peeranoia can be classified as a successful practical example of a mechanism for improving security applications in decentralised environments.

Appendix A

Specifications and Source Code

A.1 Policy Language Specification

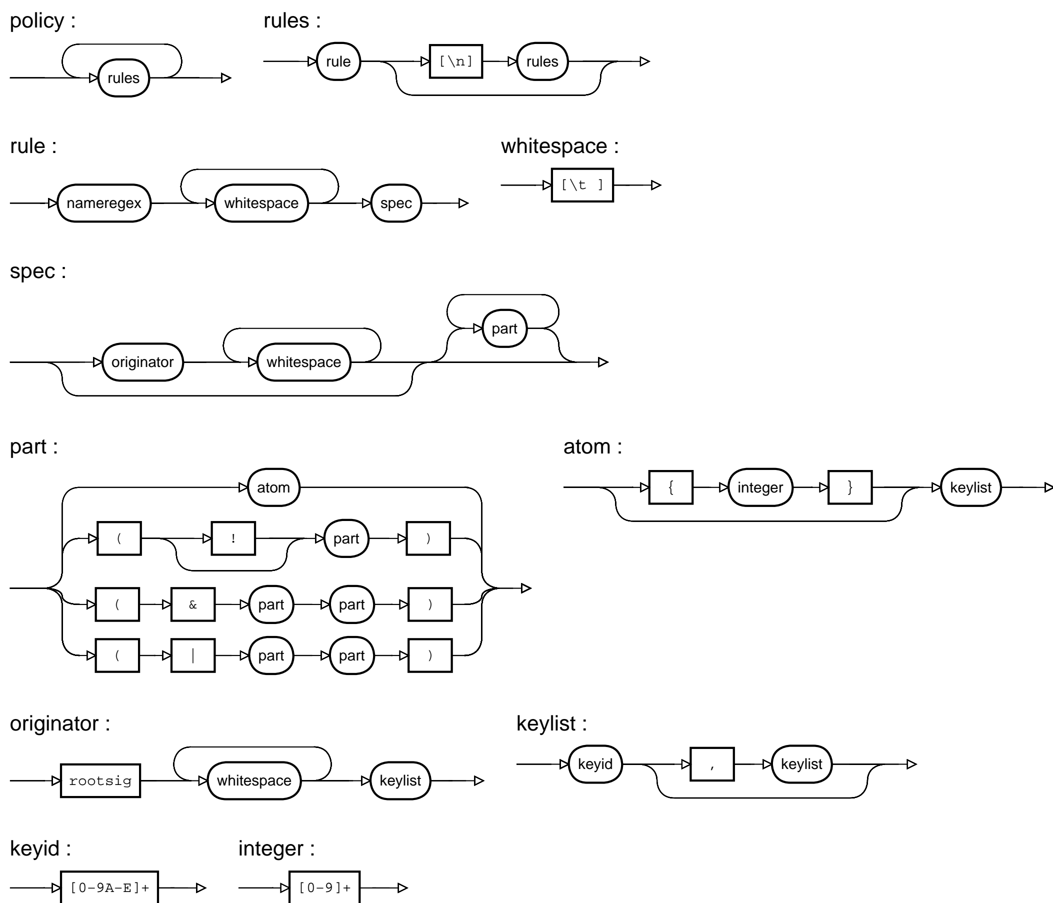

Figure A.1: Policy Language Syntax Diagram

The policy is expressed in clear text (see Figure A.1), with one rule per line and the lines being separated with the ASCII LF character.

Every rule specifies which documents it is applicable to, using a Perl-compatible regular expression that is matched against the document name. Only the first matching rule is applied.

Every rule controls whether a document is valid and active by specifying which peer signatures the document must bear. This is provided by giving arbitrarily complex boolean formulas in pre-fix notation. The operand in these formulas are lists of (public) key identifiers, with an optional integer quantifier which controls how many signatures by members of the list must be present. A non-existent quantifier is treated as having the value 1. The quantifier is interpreted inclusively: value n means "n or more signatures from the list must be present for logical truth".

A rule can also contain an optional originator restriction, which is also a list of key identifiers, but here only the originator signature is checked against the list. No quantifier is possible for an originator restriction.

The key identifiers used in the key lists are the numeric key identifiers of the underlying asymmetric cryptographic infrastructure; in case of the prototype implementation, these are hexadecimal GnuPG key identifiers.

A.2 Protocol and Message Specifications

A.2.1 Communication Overview

All communication is executed point-to-point between known peers, with the transport protocol being TCP. Data is exchanged in variable-size messages, all of which are encrypted.

There is no general acknowledgement of messages (some elicit responses, though). Messages can be sent by both parties, and there is no explicit client-server distinction (except during communication startup).

A.2.2 Communication Startup

The connecting peer establishes a TCP connection with the other peer, with address parameters extracted from the `peerlist` document.

The initiator then generates a session key, encrypts this session key with the contacted peer's public key and signs the result with its private key. This encrypted and signed session key is transmitted by the initiator as the first message to the contacted peer.

The session key should contain random data and incorporate the current timestamp at the initiator as a replay protection. The contacted party must verify the signature on the session key and drop the connection if the key is not suitable. Also, if the timestamp difference is more than 10 minutes, the contacted party should drop the connection.

After accepting the connection, the contacted party should acknowledge the startup by sending a PING request. From this point onwards, the roles are symmetric and there is no client-server distinction.

Both peers must then inform each other of their respective versions of the group meta-data (the `peerlist` document) by exchanging IHAVE messages. A peer detecting a newer version of the meta-data object must retrieve this newer version.

The communication session should be kept alive for a prolonged period of time to minimise the startup overhead cost. There is no maximum allowable session duration.

Once every hour, a new session key should be established (via REKEY, see below). Both parties must keep a counter of elapsed time since the last rekeying and on exceeding the interval, a peer must initiate

a rekey operation. After successfully establishing a new session key, both parties reset that counter. Simultaneous rekeying is not supported, therefore the interval must be adjusted using a random fudge factor of up to 10%.

A.2.3 Termination of Communications

There is no explicit protocol shutdown mechanism; a peer can shut down at any time by simply dropping the TCP connection.

If any low-level data corruption is detected, then the detecting party should terminate the connection. The same holds for timeout problems on the transport level.

A.2.4 Message Formats

The low-level, "wire" format consists of only two elements: a message length (4 bytes) and a argument block (variable length). All numeric message components are in network byte order (big endian representation).

For all messages except the very first, the argument block is encrypted symmetrically with the current session key (using AES in CBC mode with a 128 bit key); with the first message, the argument block contains the asymmetrically encrypted and signed session key.

The decrypted argument block of the low-level format is of fixed format, containing the following components: a tag (1 byte), a command (1 byte), the length of the optional argument block (4 byte), the optional argument block (variable length), and the CRC (4 bytes).

The tag is set randomly by the issuer of the request; if the request elicits a response, the response message must carry the same tag. For requests that do not elicit responses the tag is ignored. An issuer must keep track of unanswered, pending requests and must not reuse the same tag (with the particular communication partner) before the request is conclusively handled.

The CRC must span all other fields of the message. For some commands, the argument block is optional. The command must be one of the following list.

A.2.5 Protocol Commands

REKEY, 0x01: Rekey commands do not elicit a response (but a PING/PONG cycle is suggested). This command can be issued at any time by either peer. Peers should modify the base validity period with a random factor of up to +/- 10% of the base period to avoid concurrent rekeys.

The argument block for this command must contain the asymmetrically encrypted and signed new session key. The issuer of a rekey request sends this new session key, and stops accepting the old session key.

The accepting side of a rekey should afterwards run a PING against requester, to make sure that connection is live again. Concurrent rekey requests result in undefined behaviour, and the connection will be dropped.

PING, 0x02 and PONG, 0x03: PING commands are used to check connection status and produce filler traffic. The argument block is optional for both PING and PONG, and its contents are simply ignored. PING commands must be answered with PONG commands with the same tag.

IHAVE, 0x04: The IHAVE command elicits no response and is used to offer a document (version) to the other peer.

Ihave commands must have an argument block identifying the document and transporting the signature state at the sender. This requires the following components: length of the document

name (1 byte), document name (variable), version (8 byte), length of the signature block (4 bytes), signature block (variable).

If multiple versions of a document must be offered concurrently, then IHAVE commands should be issued in order of their version, newest first.

GET, 0x05: The GET command asks the recipient to respond with a particular document. The GET command must have an argument block that identifies the the document: name regexp length (1 byte), name regexp (variable), version (8 bytes).

The version identifier with all bits set is reserved and is used as a wildcard meaning "any version of this document". The document name regexp must be a Perl-compatible regular expression.

A GET command must be answered with a GETANSWER command with the same tag, regardless of success.

GETANSWER, 0x06: A GETANSWER command transports (potentially multiple) documents in its argument block. If there is no matching document available, a zero-length argument block is sent.

Otherwise, the argument block has the following format: length of the document name (1 byte), name (variable), version (8 bytes), length of the signature block (4 bytes), signature block (variable), length of the document data (4 bytes), document data (variable).

Multiple documents can be contained in a single GETANSWER: the length indicator for the argument block is compared against the sum of the lengths of the individual elements.

HEAD, 0x07: The HEAD command retrieves the signature block of a document. Its argument block is the same as for the GET command, and it must be answered with a HEADANSWER command.

HEADANSWER, 0x08 : The HEADANSWER command is similar to GETANSWER, but transports only signature blocks (not document bodies). Therefore, its argument block is identical to IHAVE's argument block.

A zero-sized argument block in HEADANSWER indicates that no matching document is available.

A.2.6 `peerlist` Format

The group meta-data document, `peerlist`, is the only object with a specific format; all other documents are opaque and interpreted only by the originator.

The `peerlist` contains the policy specification and the list of public keys of all involved parties (peers and administrators). All keys must bear some information identifying the key owner; in case of the prototype implementation, GnuPG's key comment facility was used. For permanent peers, the peer's IP address and listening TCP port must be given in text form, separated by a colon. For administrators, an email address must be used. The actual key material must be self-signed, and must be collated into one (binary) object.

The format of the `peerlist` is as follows: length of the policy text (4 bytes), policy text (variable), length of the key data (4 byte), key data (variable).

A.2.7 Signature Data Representation

Signatures on documents are stored in signature blocks, which contain a depth-first linearisation of the signature tree. The storage format consist of multiple instances of the following elements: length of the signature (4 bytes), number of child signatures (4 bytes), signature data (variable).

The very first signature must be the originator's signature, and every document must have at least this signature before being injected into the peer group for storage and certification. The originator signature must cover the document name length, the data length, the name, version and the document data.

All signatures downstream of the originator must cover the same elements as the originator's, and also the concatenated parent signatures (up to and including the root signature).

A.3 Source Code Availability

The prototype implementation described in Section 5 consists of roughly 15000 lines of Perl code. This size and the experimental, unrefined nature of the implementation make the inclusion of the source code in this thesis infeasible.

For these reasons, the complete sources are made available upon request: if you wish to experiment with `Peeranoia`, please contact the author via email at `az@bond.edu.au` or `az@snafu.priv.at`.

The source code is available as "open source", licensed under the GNU General Public License Version 2.

Bibliography

[1] Wichert Akkerman and James Troup. Debian Server Compromises. http://wiggy.net/debian/, 2003.

[2] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM Press, 1994.

[3] Rainer Wichmann. Samhain. http://la-samhna.de/samhain/index.html, 2002.

[4] Rami Lehti and Pablo Virolainen. AIDE (advanced intrusion detection environment). http://www.cs.tut.fi/~rammer/aide.html, 2000.

[5] Tripwire Inc. Tripwire manager. http://www.tripwiresecurity.com/products/manager/, 2002.

[6] Dave Dittrich. Root Kits and hiding files/directories/processes after a break-in. http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq, 2002.

[7] Michael A. Williams. Anti-trojan and trojan detection with in-kernel digital signature testing of executables. http://www.trojanproof.org/sigexec.pdf, 2002.

[8] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th Usenix Security Symposium*. USENIX Association, 2000.

[9] Carl Ellison. SPKI/SDSI and the Web of Trust. http://world.std.com/~cme/html/web.html, 2001.

[10] Thomas Wu, Michael Malkin, and Dan Boneh. Building intrusion-tolerant applications. In *Proceedings of the USENIX Security Symposium*, pages 79–92. USENIX Association, 1999.

[11] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and Countering System Intrusions Using Software Wrappers. In *Proceedings of the 9th USENIX Security Symposium*. USENIX Association, 2000.

[12] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[13] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri. A change-detection algorithm inspired by the immune system, 1995.

[14] Secunia. Vulnerability Report - Microsoft Internet Explorer 6.x. http://secunia.com/product/11/, 2006.

[15] Matt Miller. Elfsign. http://www.hick.org/code/skape/elfsign/, 2003.

[16] Axelle Apvrille, David Gordon, Serge Hallyn, Makan Pourzandi, and Vincent Roy. Digsig: Runtime authentication of binaries at kernel level. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 59–66. USENIX Association, 2004.

[17] Werner Koch. Gnu Privacy Guard. `http://www.gnupg.org/`, 1999.

[18] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[19] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] Philip M. Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems*, pages 93–100, 1988.

[22] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[23] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[24] Malte Borcherding. Partially authenticated algorithms for byzantine agreement. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, pages 8–11, 1996.

[25] A. W. Krings and Thomas Feyer. The byzantine agreement problem: Optimal early stopping. In *HICSS*, 1999.

[26] Shafi Goldwasser and Yehuda Lindell. Secure computation without a broadcast channel. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, 2002.

[27] Christian Cachin. Distributing trust on the Internet. In *Conference on Dependable Systems and Networks (DSN-2001)*, 2001.

[28] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th Annual Symposium on Principles of Distributed Computing*. ACM Press, 2000.

[29] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[31] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 68–80. ACM Press, 1994.

[32] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS)*, volume 3, pages 317–326. IEEE Computer Society Press, 1998.

[33] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999.

[34] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[35] D. A. Agarwal, O. Chevassut, M. R. Thompson, and G. Tsudik. An integrated solution for secure group communication in wide-area networks. In *2001 IEEE Symposium on Computers and Communications*, 2001.

[36] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[37] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26. ACM Press, 1987.

[38] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 68–80. ACM Press, 1994.

[39] Miguel Castro and Barbara Liskov. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Massachusetts Institute of Technology, 1999.

[40] Gene Tsudik. Message authentication with one-way hash functions. In *INFOCOM (3)*, pages 2055–2059, 1992.

[41] Michael J. Wiener. Performance comparison of public-key cryptosystems. *CryptoBytes*, 4(1), 1998.

[42] Miguel Castro and Barbara Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Technical Report MIT/LCS/TM-595, Massachusetts Institute of Technology, 1999.

[43] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 273–288. USENIX Association, 2000.

[44] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578. ACM Press, 1997.

[45] Dahlia Malkhi and Michael K. Reiter. An architecture for survivable coordination in large distributed systems. *Knowledge and Data Engineering*, 12(2):187–202, 2000.

[46] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Symposium on Reliable Distributed Systems*, pages 51–58, 1998.

[47] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: an exercise in distributed computing. *Commun. ACM*, 25(4):260–274, 1982.

[48] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 123–133. ACM Press, 1988.

[49] J. Postel. RFC 821: Simple mail transfer protocol. `ftp://ftp.internic.net/rfc/rfc821.txt`, August 1982.

[50] Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *SIGOPS Oper. Syst. Rev.*, 22(1):8–32, 1988.

[51] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS2003*, 2003.

[52] Richard M. Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.

[53] Dahlia Malkhi, Yishay Mansour, and Michael K. Reiter. On diffusing updates in a Byzantine environment. In *Symposium on Reliable Distributed Systems*, pages 134–143, 1999.

[54] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, 8-9 1994.

[55] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16)*, 1997.

[56] Brent Welch. Customization and flexibility in the exmh mail user interface. In *Proceedings of the Tcl/Tk Workshop*, pages 261–268. USENIX Association, July 1995.

[57] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[58] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. OceanStore: An extremely wide-area storage system. Technical Report UCB//CSD-00-1102, University of California, Berkeley, 2000.

[59] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201. ACM Press, 2000.

[60] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, International Computer Science Institute, Berkeley, 1995.

[61] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc. of International Workshop on Future Directions of Distributed Systems*, 2002.

[62] Ross J. Anderson. The Eternity service. `http://www.cl.cam.ac.uk/users/rja14/eternity/eternity.html`, 1997.

[63] D. Mazieres and F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proceedings of ACM SIGOPS*, 1998.

[64] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, 1985.

[65] Maurice Herlihy and J. D. Tygar. How to make replicated data secure. In *CRYPTO*, pages 379–391, 1987.

[66] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[67] Emin Martinian. DIBS: Distributed internet backup system. `http://sf.net/projects/dibs`, 2002.

[68] Christian. Grothoff. An excess-based economic model for resource allocation in peer-to-peer networks. *Wirtschaftsinformatik*, Jun 2003.

[69] Faruck Morcos, Thidapat Chantem, Philip Little, Tiago Gasiba, and Douglas Thain. iDIBS: An improved distributed backup system. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, 2006.

[70] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 5 1996.

[71] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[72] David Wheeler. Countering trusting trust through diverse double-compiling. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 33–48. IEEE Computer Society, 2005.

[73] Fabrice Bellard. TCC – the tiny c compiler. `http://www.tinycc.org/`, 2001.

[74] Rob Pike. Notes on programming in C. Technical report, Bell Labs, 1989.

[75] Richard P. Gabriel. The Rise of "Worse is Better". `http://www.ai.mit.edu/docs/articles/good-news/`, 1991.

[76] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992.

[77] M. R. Horton and R. Adams. RFC 1036: Standard for interchange of USENET messages. `ftp://ftp.internic.net/rfc/rfc1036.txt`, December 1987.

[78] M. Crispin. RFC 2060: Internet Message Access Protocol — Version 4rev1, December 1996.

[79] Philip Koopman. 32-bit cyclic redundancy codes for internet applications. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 459–472. IEEE Computer Society, 2002.

[80] Alexander Zangerl and Robert Barta. Perl, POE, Peers and Paranoia. In *Proceedings of the Australian Open Source Developers' Conference (OSDC 2005)*. Open Source Developers' Club, 2005.

[81] Alexander Zangerl. Tamper-resistant replicated peer-to-peer storage using hierarchical signatures. In *Proceedings of the The First International Conference on Availability, Reliability and Security (ARES 2006)*, pages 50–57. IEEE Computer Society, 2006.

[82] Larry Wall. perl – Practical Extraction and Report Language. Usenet mod.sources archives, 1987.

[83] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., 2000.

[84] Rocco Caputo. POE: The Perl Object Environment. `http://poe.perl.org`, 2003.

[85] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[86] Mark Burgess. Computer immunology. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98)*. USENIX Association, 1998.