TrustNet: Trust-based Moderation

Using Distributed Chat Systems for Transitive Trust Propagation

Alexander Cobleigh

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis TFRT-9999
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Media-Tryck.
Lund 2020

Abstract

This thesis introduces TrustNet, a flexible and distributed system for deriving, and interacting with, computational trust. The focus of the thesis is applying TrustNet as a tool within distributed chat systems for implementing a subjective moderation system. Two distributed chat systems, Secure Scuttlebutt and Cabal, are discussed, the latter having been extended with a proof of concept implementation of the proposed system. The concept of ranking strategies is also introduced as a general purpose technique for converting a produced set of rankings into strategy-defined subsets.

This work proposes a complete trust system that can be incorporated as a ready-made software component for distributed ledger technologies, and which provides real value for impacted users by way of automating decision-making and actions as a result of assigned trust scores.

Acknowledgements

I want to start by thanking Johan Eker, my thesis supervisor from the department of Automatic Control at Lund University, for his support, patience, advice and feedback in this unusually long Master's thesis. I am very glad you took the time to listen to the random student dropping by your office to pitch his thesis idea. I also greatly appreciate the frequent discussions we have had throughout the thesis process. In a similar vein, I want to thank my thesis examiner Karl-Erik Årzén, also from the department of Automatic Control at Lund University, for agreeing to take on the responsibility of this thesis—thank you for your patience with the extended process the thesis ended up being, and for your feedback.

Next, I would like to thank my friend Linus for his help and emotional support throughout the thesis process. Your advice around the academic process concerning getting started on the thesis was essential, in addition to all the advice you have provided throughout my time at Lund's University. Thank you.

I want to thank Aljoscha for reading a *very early* draft, back in July of 2019, and for your keen feedback. Thanks to Christian F. Tschudin, professor of Computer Science at the University of Basel, for inviting me to present an early version of TrustNet at the P2P Basel workshop in February 2020, just before the Covid-19 pandemic took over the world.

I also want to thank Erick, Daniel, and Linus for reading drafts and for providing such excellent feedback. I would also like to thank Sara for her illustration advice, without which Fig. 6.1 would have looked *much* worse. Thanks to my friends who were working on their theses at the same time for their excellent company and motivation; thanks Ingrid, Magnus, and Anton.

And a heartfelt thanks to my parents—for letting me pursue my interests freely, despite not always understanding what I was going on about.

Contents

1

Introduction

How do you remove malicious participants from a chat? For a set of participants, what are the steps needed such that the malicious participant is no longer visible by anyone in the set?

In a centralised chat context, there is always someone with the power to remove a participant. Usually, this is the person that started the context itself (i.e. a group chat). There is a special privilege granted to the initiative taker such that they can add and remove participants, as well as grant others the ability to do so. This role is usually known as an administrator, or admin, and the ones they grant powers are known as moderators, or mods.

Even if no administrator exists, maybe they decided to leave the platform hosting the group chat, there is always someone who has credentials to the hardware running the chat software. Thus, if the situation gets bad enough, it is technically possible for the platform administrator to individually intervene and, for example, remove the malicious participant from the database, or decree a new administrator. It is cumbersome and rare that it would come to that, but it is possible.

This is a harder problem to solve in a distributed chat context. How do we know who is the leader in an eventually consistent system, where people may continue to perform actions offline? There is a much higher degree of subjectivity possible in these systems, as compared to a purely centralized context. Causality ceases to be straightforward when participants are allowed to continue participating in temporarily disconnected portions of the system.

In the centralized context, removing a malicious participant is the action of a moderator. Usually it is one or two clicks, and the malicious participant has been removed for all other participants.

In a distributed context, there are many possible answers to this problem. The first and naive solution is to delegate the responsibility of removing the malicious participant to each individual participant. Thus everyone participating has to individually hide offenders. Viewed as an isolated case it works, but repeated instances will risk causing an outsize burden on the participants.

Another solution is to designate someone as a moderator for the entire group, like in the centralized context. Leader elections in a distributed context are however

rather complex (see Paxos [Lamport, 1998]) and sensitive to Sybil attacks [Douceur, 2002], where one actor controls many individual actors inside the system, gaming it and electing themselves as leader.

This thesis explores an alternate approach. What if participants could automatically block the malicious peer, if they discover that the peer has been blocked by someone the participant trusts? This is similar to the administrator from the centralized context, but more flexible. In the centralized context, if the administrator is misbehaving and a participant loses trust in them, their only options are to live with it, or to leave the group. In the system where you effectively choose who can moderate for you, you can also choose to revert that decision if your trust later proves to have been misplaced. This is the central topic of the thesis, and one potential answer is presented in the form of a new system for managing and interacting with trust, *TrustNet*.

The core **problem statement** of the thesis is the following:

> How can we efficiently hide malicious participants in a distributed chat context?

and TrustNet is the proposed answer.

1.1 Overview

The thesis starts by explaining the technical foundations, Chapters 2–4, which mainly constitute the work's backdrop across the areas of cryptography, distributed systems, and (distributed) chat systems—though the second half of Chapter 4 presents the novel concept of *subjective* moderation systems. Then we venture into the chapter on trust, Chapter 5, followed by Appleseed in Chapter 6, the core algorithm. The subsequent chapter on TrustNet, Chapter 7, is the heart of the work and presents the main contributions. Chapters 8–9 contain the evaluation, results, and discussion portions of the thesis.

In Chapter 2, *Public-key cryptography*, we introduce the cryptography knowledge needed to understand that distributed chat systems are possible in the first place, as well as fundamentally secure. We follow this up with Chapter 3, *Distributed systems*, which goes deep on the topic of distributed systems—what they are, what range of distributed systems this thesis is concerned with, and how a distributed system may be put together to enable a cohesive chat experience. Topics such as the CAP theorem, append-only logs (and how they may be secured), and vector clocks are detailed, among others.

Chapter 4, *Chat systems*, presents the topic of chat systems, and more importantly the topic, and causes, of moderation in chat systems. The last half of the chapter details the distributed chat systems we are primarily concerned with in this work. Two distributed chat systems, *Secure Scuttlebutt* and *Cabal*, are briefly presented, and the novel concept of a subjective moderation system is introduced. The

following chapter, Chapter 5, *Trust*, details the topic of trust. We begin broadly and then narrow down to the topic of computational trust, detailing such topics as trust transitivity, human-meaningful labels for trust scores, and the difficulty of interpreting distrust. A survey of related work in the domain of computational trust is presented in Section 5.2.

Chapter 6, *Appleseed*, presents the Appleseed trust metric, which is the foundation of this work. Appleseed allows us to subjectively manage trust across individuals in a single network of trust. The most trusted nodes, as seen from the perspective of a single node, are found by graph traversal. The chapter presents the Appleseed algorithm from an intuitive viewpoint before the details of the algorithm are presented and discussed, and drawbacks with the approach outlined. The following Chapter 7, *TrustNet*, describes TrustNet, the culmination of this work. TrustNet's purpose is described, the system's attempted mitigations of Appleseed's drawbacks outlined, and more.

Chapter 8, *Results*, presents the results and evaluation of TrustNet's efficacy, followed by Chapter 9, *Discussion*, where future work is presented, caveats and potential concerns are discussed, and potential use cases of the proposed system outlined.

1.2 Contributions

The **main contributions** of this work may be found in Chapter 5 (*Trust*), Chapter 6 (*Appleseed*), and Chapter 7 (*TrustNet*).

The Appleseed algorithm has been implemented in two programming languages. It was initially implemented in Python for the sake of prototyping, and afterwards in Nodejs, as a reusable javascript module.

In order to allow easier interaction for end-users with Appleseed's results, as well as extend the algorithm by adding support for trust areas and a distrust mechanism which preserves the properties of the algorithm, the TrustNet system is proposed and implemented.

The notion of a subjective moderation system is introduced, which to, our knowledge, makes this thesis the first academic result to present the concept. Two distributed chat systems are also presented, Secure Scuttlebutt and Cabal, the latter having been used to implement a proof-of-concept subjective moderation system.

The concept of a ranking strategy is proposed as a way to interact with Appleseed's produced rankings. A cluster-based ranking strategy is described and implemented. The cluster-based strategy splits the produced ranking into 3 groups, discards the cluster containing the lowest values, merges the remaining two clusters and promotes the merged result as the trusted peers of the system.

In order to evaluate the TrustNet system, an evaluation framework was built in Nodejs. The framework consists of a scenario generator and a demonstration tool. The scenario generator generates pseudo-random trust graphs, with a configurable

node count, in order to test the results of TrustNet. It also allows setting an average amount of trust assignments for the generated graph, as well as configuring the skew, or spread, of the trust weights for the trust assignments.

The demonstration tool integrates TrustNet with the distributed chat system, Cabal, as a proof-of-concept of a subjective moderation system. The tool has been built to serve the demonstration in the browser, and consists of an HTTP server serving the demonstration webpage. The tool can furthmore interact over WebSocket connections. Finally, it also spawns Cabal nodes in order to simulate a fully featured Cabal swarm, which are coordinated via the HTTP server's WebSockets.

Both the Appleseed algorithm and the TrustNet trust system are intended to be published as javascript modules under open source licensing shortly after the publication of this thesis.

2

Public-key Cryptography

In the context of chat systems, strong cryptographic primitives are important to ensure privacy from eavesdroppers when communicating. In distributed chat systems, the public-key cryptography described in this chapter is essential for the system to function at all. As we will see, the described primitives enable private communication, but also the creation of verifiable identity systems, as well as enable entirely new forms of message distribution architectures.

Public-key cryptography, also known as asymmetric cryptography, is a branch of modern cryptosystems. It was first proposed in 1976 by Whitfield Diffie and Martin Hellman in a seminal article titled *New Directions in Cryptography* [Diffie and Hellman, 1976].

Prior to the proposal, all known cryptography in use were variants of symmetric-key cryptography, wherein a single key is used to transform a plaintext into a ciphertext and back again. Since symmetric cryptography uses a single key for both encrypting and deciphering, the confidential transmission of the key is paramount. Public-key cryptography turns the problem on its head by introducing two keys, one for encrypting, *the public key*, the other for deciphering, *the private key*. Only the private key can decipher messages encrypted with the public key, and the entire paradigm is based on the difficulty of taking the public key and deriving the private key [Diffie and Hellman, 1976]. This enables a message recipient to publicly announce and spread their public key while keeping the private key to themselves, neatly solving the key-transmission problem that troubles symmetric cryptography.

An analogy for understanding public-key cryptography would be the following. Alice sends physical locks to all of her friends. The locks are unlocked until firmly clasped onto something. Whenever Alice's friends want to make sure that something, a bike they want to lend to Alice maybe, is only accessed by Alice, they use one of Alice's locks. Once the lock has been clasped on a thing, the only person that can unlock it is Alice, because she is the only one with the key.

We have established that public-key cryptography enables sending computationally secure private messages. Another interesting property of the Diffie-Hellman article is the introduction of *digital signatures*.

2.1 Digital signatures

Digital signatures can be used to solve the problems of *authenticity*, *non-repudation*, and *message integrity*. Authenticity is the problem of determining if the party you are dealing with are who they claim to be. Non-repudation concerns itself with preventing a party from making a claim at one point in time and later pretending it never happened. Message integrity deals with maintaining the contents of a message from its generation, through its transmission, and until its final receipt—with the assurance of it being tamper-free.

Digital signatures are created with the same type of keypair—but typically *not* the same keypair, as keypair reuse is discouraged by cryptographers—that enable secure encryption and decryption of messages. A message author uses their private key to *sign* the message, proving that they were its author. Recipients *verify* the received message by using the author's signature, attached in the same transmission as the message, in combination with the author's public key, which has been retrieved elsewhere—a secure keyserver, which is a place where people upload their public keys to make them easily discoverable by others, for instance. In the section below, we detail the signature generation protocol in detail.

Signature generation Creating the signature occurs in the following manner. The private-key holder wants to sign a message, m. They hash the message with a cryptographic hash function H, generating the hash h. The signing operation, *sign*, is then applied to the hash of the message, generating the signature *sig*. The signature can only be generated by someone in possession of the private key, $k_{private}$. The message and the signature are then transmitted together to the recipient.

$$\begin{aligned} h &\leftarrow H(m) \\ sig &\leftarrow sign(h, k_{private}) \end{aligned}$$

For the recipient to verify the signature they apply the verification operation $verify$ using the public key k_{public} and the received signature *sig*. The result is the message hash h, initially generated by the private-key holder. The recipient then hashes the received message, generating h'. The recipient compares their message hash, h', with the signature-derived hash, h. If the two hashes are equal, the signature has been verified: the message is authentic and has not been tampered with.

$$\begin{aligned} h &\leftarrow verify(sig, k_{public}) \\ h' &\leftarrow H(m) \end{aligned}$$

The above protocol ensures that any message sent or retransmitted in a system based on public-key cryptography can be verified and attributed, without a doubt, to its original author. As with physical signatures, digital signatures enable the representation of digital *identity*, which we will now discuss.

2.2 Public-key-based Identity

The digital signatures from above can be used to implement a decentralized identity system. However, when we say *identity system*, what do we actually mean? Going forward in this work with mentions of *identity* and *identity systems*, we mean the following: a system for verifiable and unforgeable entity representation. For Alice's ability to conduct transactions without Mallory being able to forge transactions in Alice's name, for Carole to be able to uniquely and unambiguously refer to Alice, and for Bob to be able to verify that a person that claims to be Alice is, in fact, Alice. Clearly, this maps well onto the previously described digital signatures.

When we say *in Alice's name*, in the context of an identity system, we don't necessarily mean operating under Alice's given name—the identifier needs to be unique and unambiguous. Examples of such identifiers include assigned national identity numbers, generated public keys, and domain names.

In a *decentralized* identity system, each identity is represented by a generated public keypair. Thus, any entity can create multiple identities on their own—without needing to interacting with hierarchical third parties, such as nation-states or certificate authorities.

Identity collision As regards two entities accidentally creating the same identity, using the popular Ed25519 signature scheme from the Edwards-curve Digital Signature Algorithm (EdDSA) family [Bernstein et al., 2011] as an example, the probability basically amounts to the entities both randomly choosing the same 256-bit string, or picking the same random number among a space of 2^{256} numbers. It is very, very unlikely.

On naming In any identity system the addressability of entities—naming another individual, for instance—is important for practical use. An entity's outward facing identifier in a public-key-based system is the public key. In order to make the identity more readily usable by people in e.g. text communication, the public key can be encoded from a byte representation into a string representation.

There are a few string representations in popular use. `base16` and `base64` are two popular and accepted engineering standards, defined in *RFC 4648* [Josefsson, 2006]—RFC, or *Requests for Comments*, is a widely accepted set of documents and a process for documenting and establishing Internet standards. `base58` is another encoding scheme, come into prominence thanks to its initial use in Bitcoin, a distributed ledger focused on exploring notions of scarce (as in a finite supply and in opposition to abundant) digital currency. [Nakamoto, 2009]

`base16`, more commonly known as *hexadecimal*, encodes the public key as a text string using an alphabet of 16 characters, represented by the character ranges $[0-9, a-f]$. Hexadecimal allows both upper- and lowercase characters to be used interchangeably. More compact encodings include `base64` as well as the `base58` variants. As there is no canonical standard for `base58` there exists more than one `base58` encoding scheme; we will refer to the set of `base58` encoding schemes as

the `base58` *variants*.

The `base58` variants encode the public key using a set of 58 characters, mainly lower- and uppercase latin letters $[a-z, A-Z]$ and arabic numerals $[0-9]$, while also excluding visually similar characters (e.g. the numeral 1 and the letter *l*) from the permissible range.

The benefit of using larger character ranges, e.g. `base64`, is that more information can be encoded per character. This results in shorter text-encoded identifiers than when using less dense representations, like `base16`. Hexadecimal representations are simpler to implement and less ambiguous—`base58` has a few different implementations with slightly varying character sets. While `base64` is canonically defined its character range overlaps with special characters commonly used in URIs, whereas the `base58` variants exclude those characters from its character set. Using `base64` can therefore cause issues when e.g. identities are addressed as part of hypertext links. There is, however, a somewhat less known variant of `base64` known as `base64url`, which was proposed—in the same *RFC 4648* as `base64`, in fact—to remedy the URI-addressing issue mentioned above. In `base64url`, the character `+` (*plus*) is replaced by `-` (*dash*), and `=` (*equals*) is replaced by `_` (*underscore*)—the remaining character set is unchanged across the two base 64 encodings. [Josefsson, 2006]

In order to visualize what has been described above, the following two strings encode the same information. The first string is encoded in `base64` and the second in `base16`.

```
C6fAmdXgqTDbmZGAohUaYuyKdz3m6GBoLLtml3fUn+o=
0ba7c099d5e0a930db999180a2151a62ec8a773de6e860682cbb669777d49fea
```

As a final note on the topic of naming, so called *pet names* can be employed to make referencing identities easier. A pet name is a human parseable name associated with e.g. a public-key identity. [Stiegler, 2005] Pet names are, for instance, used in Secure Scuttlebutt, a distributed protocol and social network that will be discussed in Section 4.2.

Public-key-based identities will be further detailed in 4.3, in particular their potential *drawbacks*. As regards the cryptographic primitives needed to achieve these identites, we have now described all we will need.

2.3 Uses in distributed peer-to-peer systems

Public-key cryptography is an essential component of the distributed chat systems that are the topic of Section 4.2. The asymmetric cryptographic primitives of this chapter essentially enable a completely decentralized identity system. Any one node can generate an identity, represented by its public-key keypair, entirely on their own.

Messages can be attributed to authors and even passed along between peers until they reach their destination, with the added assurance of its contents remaining untouched. Secure private communications can be established between individuals

without any third parties or intermediary services.

Fundamentally, the briefly described decentralized identity system allows for the emergence of *trust* within this new class of systems. It allows individuals to attribute actions and consequences to others, for friendships to blossom, and new initiatives and projects to be born.

Commonly used implementations of the cryptographic primitives described in this chapter are Curve25519 [Bernstein et al., 2011] for the public-key cryptography, and Blake2b-256 [Aumasson et al., 2013] or SHA256 [NIST, 2001] for the cryptographic hashing functions, in addition to the previously mentioned encoding schemes `base16`, `base64` and `base58`.

3

Distributed systems

Chat systems are a type of highly distributed system. In a given chat system, many nodes communicate with each other and the preserveration and ordering of messages is of the outmost importance, lest context be misconstrued. This is especially important for distributed chat systems, where participating nodes make up the infrastructure in which messages are both created and stored, as well as passed through.

We begin this chapter on distributed systems with a definition of them from the extremely prolific distributed systems researcher, Leslie Lamport:

> A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. [Lamport, 1978]

One of the aims of using a distributed systems architecture is to increase resiliency against *catastrophic failure*, i.e. the system becoming totally unavailable, or losing great amounts of information.

Distributed systems come under many guises. They may be centralized, as distributed architectures are chosen in order to preserve the huge amounts of inflowing information, and to handle the immense computational requirements. [Kreps, 2013] They may also be peer-to-peer, as the massively successful BitTorrent protocol has proven.

The client-server model, e.g. a web browser requesting a webpage from an IP-addressed web server, is a centralized model. Underlying the simple request, a distributed architecture may be activated, and a fleet of servers queried, to produce a timely response among a potential million others. The model often seen in file-sharing applications, where individual computers seek each other out to directly exchange pieces of files e.g. using the BitTorrent protocol, is a peer-to-peer model.

3.1 The CAP Theorem

Modern distributed systems are often reasoned about using something called the *CAP theorem*, initially proposed by Eric Brewer in a distributed systems conference keynote [Brewer, 2000] and later formalized by Gilbert and Lynch [Gilbert and Lynch, 2002]. The CAP theorem is, fundamentally, a tool for analyzing tradeoffs in distributed systems. *CAP* is an initialism created by taking three desirable properties for distributed systems, namely:

- *Consistency*, ensuring all nodes have the same data
- *Availability*, ensuring the responsiveness of the system
- *Parition-tolerance*, ensuring continued functionality despite separate partitions, or, in other words, when communication lines between groups of nodes have been temporarily severed

The three properties describe desirable characteristics of a distributed system in the event of node failure. Furthermore, the CAP theorem states that you can only pick any two factors; i.e. you may have a *consistent* and *available* system, but not also at the same time a *partition-tolerant* system.

Kleppman's critique The CAP theorem has later been critiqued by Martin Kleppman [Kleppmann, 2015] for being vaguely defined and easily misunderstood. Kleppman also asserts that the actual impossibility results are not useful—one part of Gilbert and Lynch's formalism is found to only be valid for network partitions of infinite duration, which violates eventual consistency (discussed in Section 3.2). Kleppman finishes his critique by asserting that the CAP theorem is *"no longer an appropriate tool for reasoning about systems"* [Kleppmann, 2015], and presents a draft of an alternative framework he calls *delay-sensitivity*. The proposed delay-sensitivity framework has more rigorously defined terminology than the CAP theorem. The framework also presents a classification of algorithms into two camps: one in which algorithms are *delay sensitive* in that they are dependent on network latency. The other camp of algorithms are classified as *delay independent*, and as such are unaffected by network latency variability.

Despite Kleppman's critique, we will occasionally refer back to the CAP theorem, as it is an easy shorthand which allows for briefly illustrating principles of distributed systems in a relatively accessible manner. Before we continue, however, we will elaborate further on the topic of consistency and present a few of the variations of consistency and what they mean, as an introduction to talking about eventual consistency in Section 3.2.

Consistency

In 2007 Werner Vogels, CTO of Amazon.com, a multinational corporation and provider of vast storage and computing services, published an article titled *Eventually Consistent*, which was revisited and republished in early 2009 [Vogels, 2009].

The article was part of a broader movement of alternative database systems going under the banner of *NoSQL*, depicting a departure from using databases centered around the SQL querying language [Kempe, 2012]. As Vogels' article title implies, the movement was centered around exploring notions of consistency in databases. After a short introduction Vogels says the following:

> In an ideal world there would be only one consistency model: when an update is made all observers would see that update. [Vogels, 2009]

Vogels then proceeds to unpack the passage and elaborates upon it, which we will now attempt to do as well.

The many ways of consistency Read-your-writes consistency, sequential consistency, causal consistency, processor consistency, monotonic read consistency, monotonic write consistency, eventual consistency. At a first glance of the CAP theorem, it is easy to be fooled that either a system is consistent or it is not. However, as per Kleppman's critique [Kleppmann, 2015], what is often called *consistency* is in fact a spectrum of consistency models [Mosberger, 1993].

One way to view consisistency is that a *consistency model* is fundamentally a promise on what application developers may expect from the underlying model, e.g. regarding memory access across processors in a single computer, or which values can possibly be returned in a distributed network of computers. Consistency models on the stronger side of the spectrum provide more robust promises, with less conditions that need to be filled in order to uphold the promise. The farther you go from the stronger side of the spectrum, the more conditions need to be considered by the application developers in order for the consistency model's promises to remain intact.

Models on the stronger side of the spectrum trade performance gains and latency sensitivity for reduced application complexity [Kleppmann, 2015]. The *sequential consistency* model fits this category, where all processes observe the same sequence of operations, as derived from the order of statements in their executing programs [Mosberger, 1993]. The expectation in a sequential consistency model is the following: when one process writes a value, any other process issuing a read after the write will read the written value. This expectation does not always hold for the weaker side of the spectrum.

Weaker consistency models place more constraints on application developers in terms of which expectations are valid, as compared to stronger models. What weaker consistency models receive in exchange is a greater tolerance of network latency, or increased performance, as, for instance, reordering of operations now becomes allowed [Mosberger, 1993].

Eventual consistency is the weakest consistency model which is still useful for application development [Kleppmann, 2015], which we will now proceed to elaborate further upon.

3.2 Eventual Consistency

Eventual consistency is the weak guarantee that data will be delivered to all nodes of a distributed system if *enough* time has been allowed to pass [Vogels, 2009].

Let us illustrate the principle with an example. Assume we have a node which is using a distributed chat system while offline, i.e. disconnected from all other nodes. The node decides to author new messages, perhaps they are reponses to earlier questions from fellow peers. The authoring node, and their unsynced messages, may be viewed as a temporary partition; they are temporarily disconnected from all other nodes. When that node heals its partition, whether by synchronizing with another node over the local network, or re-establishing internet connectivity and reaching other nodes in that fashion, the authoring node will exchange messages with the nodes they come across, including the offline-authored messages. Given enough time, these messages will spread across the distributed network of nodes as temporarily partitioned nodes communicate with others.

If we view the example's system through the lens of the CAP theorem we see that it is *available*, as new messages can be posted without network access. The system is *partition-tolerant*; old messages can be read and new ones authored irrespective of any temporary partitions. The system is not consistent, but it is *eventually* consistent in the sense that, given enough time, all nodes will have the same set of messages—enough time needs to pass for temporarily partitioned systems to briefly come online and sync with other nodes. This property is a core design characteristic of the distributed chat systems detailed in Section 4.2.

What makes eventual consistency feasible is the representation of operations in the distributed system as distinct messages, passed among computational nodes over time. Representing operations as distinct messages in distributed systems is called *message passing*—somewhat evoking a parallel to students passing scribbled messages to each other in a classroom.

3.3 Message Passing

As mentioned above, message passing is a key component in a lot of distributed systems. In a message passing system, the base unit is, as might be expected, the *message*. As an actor in a message passing system executes actions, messages representing those actions are created. Just like actors may issue any kind of operation, the created messages may represent any kind of action, like an arithmetic operation or a request to store information.

Remote procedure call comparison

Message passing can be contrasted with the Remote Procedure Call (RPC) paradigm. RPCs are synchronous, and thus require the receiver to be online for a sender to perform operations on it. This makes RPCs a a potentially unwise choice

for a distributed systems architecture in which nodes may be offline in unpredictable patterns—thus disrupting any RPC operations. Message passing, however, is asynchronous and therefore better suited to represent delay-independent operations. Receivers process a sent message whenever it arrives, just like a vacation postcard is read whenever it ends up at its destination.

Decoupling

Message passing decouples requests coming in to a system, i.e. from actors generating messages as a result of their actions, from the execution of them. Which, for a distributed system consisting of potentially many nodes in various states of connectivity, is a desirable architectural feature. One of the desirable features of this decoupling is that it allows for a heterogenous collection of computers to function as nodes in a distributed network. This disparateness is manifest in a few ways. First, messaging passing functions more on the protocol level of only requiring nodes to know how to read and write messages, resulting in less strict requirements as to which software needs to be running than other paradigms, such as RPC. Secondly, it is possible to trivially bring any kind of computer, irrespective of its hardware or operating system, as a new node into a distributed system, as long as it knows how to read and write the system's messages—resulting in a net increase in computational resources.

3.4 Append-only logs

Logs are a commonly used data structure in a lot of distributed systems [Kreps, 2013]. A log structure is an ordered sequence of items, often called *records*, which are typically ordered according to insertion time into the log. An *append-only* log is a log where the only allowed operation is the *append* operation, that is, adding information to the end. The append-only log's records may not be moved nor removed.

From the strictness of append-only logs, interesting properties emerge. Records are by definition ordered by time, with earlier appearing records having been created before records which appear later on in the log. Each record also has a valid identifier by definition—its position in the log. Append-only logs greatly simplify state management—keeping track of the current state of a computational node—with respect to distributed systems; synchronizing the log's information becomes a trivial operation thanks to its strict criteria.

Log synchronization example Let us paint an example. Alice and Bob want to sync Carole's append-only log, which contains personal project updates concerning her new and exciting distributed chat system. Carole has published a total of 13 updates, so her append-only log's last record is identified by the id `12` (the first record is indexed at `0`). Alice has a server that is always online, as

such she has received all of Carole's 13 updates—Alice's last known record of Carole's log also has the id `12`. Bob, however, is an avid traveler and only occasionally uses his laptop. He only synced the 3 first updates of Carole's distributed system adventures before leaving for his latest intrenet-free trip, making `2` the last known id Bob knows about concerning Carole's log. To synchronize Carole's log, Alice and Bob exchange their last known ids of Carole's log:

Alice Hi Bob! My last record for Carole has id `12`.
Bob Hi Alice! My last record for Carole has id `2`.

Using the exchanged information, Bob now knows that he is lacking 10 updates and proceeds to request those 10 updates from Alice.

Append-only logs make it efficient to figure out the *delta*, or the difference in two datasets, that needs to be transmitted for two peers to synchronize state. Only two responses are needed to calculate the difference in synced state, regardless of the log size. As in the example, no duplicate information is transmitted either—Bob only requests the updates that he personally does not have on his computer.

Securing append-only logs In the example above we discussed the simplicity of append-only log synchronization. The example, however, is incomplete, as there was no mention of integrity guarantees—how can Bob be certain that what Alice sends him is actually Carole's data, and in the correct sequence. There are two easily realized attacks: either Alice counterfeits all of the data she sends to Bob, or Alice reorders data that Carole has authored e.g. representing id `11` as id `10`. Both of these attacks can be solved by the same conceptual solution: *hash signatures*.

Hash signatures can be used in multiple ways to secure data integrity, we will concern ourselves with briefly explaining two common variants. Irrespective of the particular flavor of the hash signature scheme that is used, a public-key keypair is needed for the signature portion. Like we discussed in Chapter 2, the private key is used to sign the data in the append-only log while the public key is used to verify the signature. This prevents any counterfeiting from taking place. Continuing on the example from above, Carole would generate a public-key keypair and use the private key to sign her 13 updates in some fashion. Bob would use Carole's public key, which has somehow been transmitted to him at an earlier time, to verify the information Alice transmits to him—confirming that the data actually originates from Carole. Merely relying on signed data is still vulnerable to the reordering attack, so we will need to go one step deeper and come up with a *signing scheme*.

The first scheme is something which we will call *hash chaining*. In hash chaining, each record in the append-only log is run through a cryptographic hash function, `sha256` [NIST, 2001] for instance. A finite length identifier for the data is generated, the hash, and if the data is altered in any way its new hash will also be drastically different. The trick to overcome the reordering attack is the following: for each record in the append-only log, hash its contents. If the current record has a predecessor, add a field pointing to the hash of the previous record *before* hashing

the current record. Thus, when we hash the record with `id 2`, the data that is hashed is the main content of id 2 but also *the hash of the preceeding record with id 1*.

Root hash signatures [Keall, 2019], make use of a tree structure known more formally as *Merkle trees* [Merkle, 1987], and is the second scheme we will describe for securing the append-only log. In this scheme, each record of the append-only log is hashed. In addition to the record hashes, there are also parent hashes that hash two record hashes together.

$$\begin{array}{lcll} h_1 & = & hash(record1, record2) & record\ hash \\ h_2 & = & hash(record3, record4) & record\ hash \\ hp_1 & = & hash(h_1, h_2) & parent\ hash \end{array}$$

This hashing structure, which forms a tree of hashes, is repeated until we end up with a single hash—the root hash. The described tree of hashes, with the top-level root hash, is the Merkle tree mentioned above. The root hash is effectively derived from each preceeding hash in the hash tree. If any record were to change place in the log, or have its data changed, the root hash would end up being different. The root hash is then signed by the private key from the generated public-key keypair, as described above. It is trivial to verify the integrity of the append-only log of this structure—any recipient can repeat the process and compare their computed root hash with the log author's signed root hash.

3.5 Kappa Architecture

One can combine message passing with append-only logs, yielding a full history of a node's actions and operations. In practical use, however, it is usually only the latest state that is sought. Thus, it is desirable to reduce the log's history into representations of the current state. One solution to this problem is an architecture called the *kappa architecture* [Pathirage, 2014]. Kappa architecture builds on earlier concepts such as *event sourcing*, and is related to the concept of *materialized views* from traditional databases like SQL. Examples of kappa architecture style databases are, for instance, Apache Samza combined with Apache Kafka [Kleppmann and Kreps, 2015].

As materialized views and event sourcing might be more familiar concepts or more intuitively explained, we will briefly detail them before moving onto the discussion of kappa architecture-style databases.

Materialized views In an SQL database, data exists as rows and columns in one or more tables. To get data from the database, a query is formed and executed against the tables. An example query might look like:

```
SELECT * FROM posts WHERE author='cblgh';
```

SQL queries can get notoriously hairy, so there exists mechanisms in the querying language, and in SQL databases like PostgreSQL, to save queries and their re-

sults. An SQL *view* is one of the ways that a query can be stored, in order to rerun it at a later time. They are called *views* because they are a particular view of a set of tables's data. Views do not cache any data, and instead perform the query each time the view is queried [Brumm, 2018]. This can take a lot of time if the query is complex and repeated often. Another mechanism for saving queries are the aforementioned *materialized views*. A materialized view is basically a combination of a table and a view. It is created as a result of an SQL query, and represents the data that the query would return. The query used to generate the materialized view is stored to disk, allowing for materialized views to be regenerated at a later time. How materialized views differ from ordinary views is that they also *store* the query's data, meaning a materialized view is essentially a generated table which can be queried with minimal time costs. The tradeoff is that, since the data of materialized view is stored, it requires extra storage space, whereas an ordinary view does not.

Thus, a materialized view is a storage-persisted view of a set of data, generated by a potentially complex and resource intensive query, which can itself be queried with minimal time cost after its initialization. One issue with the materialized view approach is that the view can grow *stale*, meaning that the underlying table data has been updated *after* the creation of the materialized view, causing the view to become out-of-date. However, since the query that generated the view is stored, the materialized view can be regenerated once and any subsequent queries against it will be both fast and up-to-date. Usually regeneration of a materialized view is handled automatically, through periodic regeneration or by monitoring the underlying tables for changes.

Event sourcing Event sourcing is a pattern where application state at a specific time can be derived by replaying the historical events leading up to the specified time [Fowler, 2005]. When using event sourcing, application state goes from being history-less to being history-derived. The place where events are stored is called the event log.

Let us illustrate event sourcing with an example. We have two chat participants, Alice and Bob. The state of the chat amounts to the history of messages Alice and Bob have posted, alongside any other changes they may have caused, such as changing their nicknames. Table 3.1 shows the sequence of chat events that occur.

The event history from Table 3.1 would produce the following state, at $t = 5$, for the chat application (newest messages at the bottom):

```
*Alice joined*
*Bob joined*
Alice: Hi
Bob: Yo!
*Bob is now known as Bob-afk*
Bob-afk: ttyl
```

t	**Event**
0	User 'Alice' joins the chat
1	User 'Bob' joins the chat
2	Alice posts message: 'Hi'
3	Bob posts message: 'Yo!'
4	Bob changes nickname to 'Bob-afk'
5	Bob posts message 'ttyl'

Table 3.1: Events of a chat application as seen from an event sourcing perspective, where t is the relative time as counted in event occurrences.

We can see that we have a few different events. Users joining the chat, users changing their own state, as well as messages being posted. These different events can be regarded as having different *types*. Thus, an event log may contain many *streams* of events—a sequence of nickname changes, a series of messages being posted. We now have enough context to be able to dive into kappa architecture proper.

Kappa architecture In a kappa architecture, events are stored in an append-only log. Each event is a message, as discussed in 3.3, and each message has a *message type*, identifying what kind of event it is.

As in event sourcing, the current state is derived through processing the log's historic messages. Going through the log each time to derive state takes linear time, however. To solve this, views are instantiated using the log's events, trading time costs for storage—just like SQL's materialized views. The derived views may use a combination of different event types, e.g. there may be a user-centric view which combines three event types to represent the latest known state of each user: the number of messages the user has written, their newest nickname, and the time of their latest published message. The purpose of a kappa architecture is to combine events spread across multiple append-only logs such that views can be created which can then be queried. Events are combined through *interleaving* the logs of the kappa architecture.

3.6 Interleaving logs

In the type of distributed system this thesis concerns itself with, i.e. those systems which can ultimately be used to model chat systems, each producer typically has its own log. The combined logs from multiple producers are used to reconstruct the system's state. This is accomplished by essentially creating a single *virtual* log through interleaving, or combining, the records from each producer's log, Fig. 3.1, according to some kind of measure. The purpose of the measure is to synchronize events across the different logs, in order to establish an ordering. Events may be ordered in two ways: a total ordering or a partial ordering.

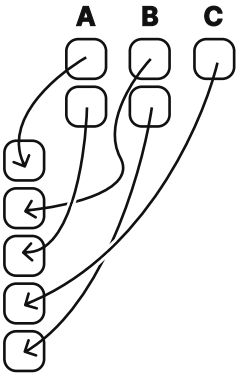

Figure 3.1: An example of interleaving logs. The three logs *A*, *B*, *C* have their individual records interleaved to create a single virtual log.

Total ordering A total ordering is an ordering where all of the events of a system are causally defined and, as such, any two events may be compared with each other to determine which occured before the other. A total ordering may be *arbitrary*, in that two concurrent events are arbitrarily but consistently resolved in terms of which came before the other. An example of such a strategy is given in the explanation of *Lamport timestamps* below.

Partial ordering Partial ordering preserves the order in which related events occured, but does not resolve the order of unrelated events. If we have two events, event *A* and event *B*, and we determine the partial ordering such that we know that *A* occured before *B*, then we say that we have established a *happened-before* relation between *A* and *B*. This is sometimes written as $A \rightarrow B$. Partial ordering is typically used in distributed systems to determine the order of events and messages, due to the difficulty of globally synchronizing clocks.

A partial ordering can be thought of in terms of a tree-like structure, where individual branches of the tree contain partially ordered events, and where two branches may not be compared as they contain unrelated chains of partially ordered events. A total ordering of the tree would definitively and consistently manage to merge all branches of the tree into a single sequence.

Interleaving measures

There are different measures that may be used to interleave the records. Real-clock—as opposed to the *logical* clocks discussed in the next sections—timestamps, e.g. the time reported by the computer at the time of log insertion, may be used. Receive time, e.g. the time at which a consumer receives a record from a remote producer, is another. More complex measures include *Lamport timestamps*, *vector clocks*, and *tangles*, which will be further discussed in subsequent sections. Which measure is used depends on the needs and context of the particular system. Times-

tamps are theoretically unsecure as they may be trivially forged, but are in practice a simple mechanism for interleaving messages—again, the particular application and its context determines the appropriateness of its use. Furthermore, systems using real clock timestamps may also exhibit unexpected behaviour due to unsynchronized clocks. A high-trust messaging app, for example, where everyone in a group is considered trusted, and unknown third-parties have no ability to join, may use regular timestamps without cause for fear, while a publically writable timeline of posts may want to use vector clocks. Distributed systems with streams of subsecond events may likely be better served by the latter approach, due to the increased significance of unsynced real clocks.

We will now briefly detail the more complex measures mentioned. In the next sections we will use the term *process* to mean an independent node in the distributed system. In a chat app, the process would be a person's computer, while in an operating system it would be an actual executing process. Importantly, processes do not have a common realtime clock in which to rely on for synchronization. The reason we use the term is to stay consistent with the literature, which, seemingly, primarily considered distributed systems from the perspective of the day's operating systems research.

Lamport timestamps Lamport timestamps, or Lamport clocks, were proposed by Leslie Lamport in 1978 as a mechanism to establish a partial ordering of events created by processes in a distributed system, as well as a way to determine an arbitrary total ordering the system's events. [Lamport, 1978]

Basically, each process has an internal logical clock; a counter which is incremented. When sending a message, the clock value at the time of sending is attached to the message. When receiving a message, the receiver's clock is set to be greater than the timestamp contained in the message. Thus, if the receiver clock already has a greater value than the message timestamp, nothing needs to be done. If the receiver clock is less than or equal to the message timestamp, the receiver clock is set to a value greater than the received timestamp. Finally, events in the same process occur in such a fashion that the clock is always incremented between two sequential events—or more simply, after an event has occured, the clock has incremented at least once. If the clock time of an event a is less than the clock time of another event b, a is regarded to have *happened before* b.

As regards the mentioned arbitrary total ordering, it can be established by introducing an arbitrary, but consistent, tie-breaking mechanism to handle the case of concurrent events—that is, events in which there is no way to determine causality due to them having the same clock timestamp, making them causally *unrelated*. One simple tie-breaking mechanism: if two events have occurred concurrently, the event stemming from the process with the lowest process ID is regarded as having happened before. Lamport timestamps paved the conceptual way for vector clocks, which will now be detailed.

Vector clocks In a vector clock-based system, each process in the distributed system has a counter, usually called a logical clock. The process increments its logical clock any time an event happens. Events that may occur are:

1. The process processes an internal event.
2. The process sends a message to another process.
3. The process receives a message from another process

Each of the above events would increment the process's logical clock by one.

The *vector* in vector clock comes from the following: in a distributed system with N participating processes, each of the N processes has a counter, or a *logical clock*. Each logical clock will occupy a position in a *vector clock*, a vector of length N, consisting of the individual logical clocks [Fidge, 1987].

The messages that processes send, or receive, each contain a vector clock capturing the *known* state of the system, as seen from the sender at the time of sending. Thus, when a new event arrives at a process, the receiving process can correctly order the event according to the causal information carried in the event's vector clock, establishing a partial ordering of the system's events.

Simply put, a vector clock keeps track of the time. Each position in the vector clock represents the state of time for a particular process, at a particular point in time. Let us represent the vector clock with T, and the vector clock for an event a with T^a. Furthermore, let $T^a_{p_A}$ represent the *clock value* for process p_A at the time of the event a.

To find out if a happened before b, we compare the vector clocks for both events. Specifically, for a to have happened before b, the clock value for each process T^a_{pX}, has to be less than or equal to the the clock value for each process T^b_{pX}, for all processes X. Additionally, there has to be at least one process Y where the clock value T^a_{pY} is strictly less than the clock value T^b_{pY}. This is what Equation 3.1 captures.

$$a \rightarrow b \text{ iff } \forall X(T^a_{pX} \leq T^b_{pX}) \wedge \exists Y(T^a_{pY} < T^b_{pY}) \tag{3.1}$$

Let us make the above more concrete. Let the vector clock T encode processes p_A and p_B as $[p_A, p_B]$. Let the vector clock for event a be encoded by $[2,1]^a$, and the vector clock for event b by $[2,2]^b$. Using Equation 3.1, we can see that $[2,1]^a$ is less than or equal to $[2,2]^b$. We also see that, at event a, the clock value for process p_B is strictly less than its clock value at event b. Therefore, a happened before b.

Fig. 3.2 contains a visualization of a sequence of vector clocks for three processes over a span of time and events. In the figure, time proceeds from the bottom up. The numbers in brackets show a process's vector clock at the time of the event. For instance, $[2,2,0]_{\mathrm{p}_2}$ can be read as the event of process 2 knew that process 1's state at the time was 2, its own state was 2 while no information was known about process 3. We can also see that there are two unrelated, i.e. concurrent, events in $[1,0,0]_{\mathrm{p}_1}$ and $[0,0,1]_{\mathrm{p}_3}$—it is impossible to say which happens before the other.

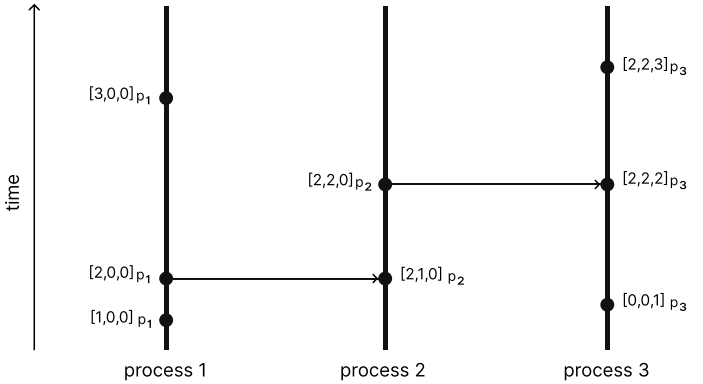

Figure 3.2: Vector clock example with 3 distributed processes. Time progresses from the bottom up. The events $[1,0,0]_{p_1}$ and $[0,0,1]_{p_3}$ are concurrent.

One issue when using vector clocks in a distributed system is that they grow in size with the number of participating processes—there needs to be one entry in the vector clock per participating process. The vector clock is part of every message, increasing the required message overhead. For a long lived system with a lot of participating processes, the overhead quickly adds up and may become a significant problem. The problem is particularly severe in systems with many short-lived processes.

Tangles The tangles data structure, briefly detailed in [Tschudin, 2018], is a new data structure that can be used to establish partial ordering, and which is starting to sprout from distributed ledger technologies such as Secure Scuttlebutt, discussed in 4.2. Essentially, a tangle is a type of append-only, directed, and acyclic graph (*DAG*). Directed, meaning each edge has a direction from one node to another, acyclic that there are no cycles, or loops, in the graph structure i.e. the graph $A \rightarrow B \rightarrow A$ is disallowed.

How a tangle differs from a typical DAG is that there is a ruleset that determines the naming and referencing of other nodes in the tangle. For instance, a tangle has a root, a set of tips, and a set of ancestors. The root is the start of the tangle and which all nodes of the tangle must reference, the tips are the outermost nodes which point to the tangle (but have yet to become part of it), and the ancestor set is how a node points back to previous records in a tangle. A tangle creates an ordering of events by having events causally and explicitly linking to each other, thereby establishing a partial ordering. The simplest tangle is a chain of events, where each node refers to the preceeding node, e.g. $A \leftarrow B \leftarrow C \leftarrow D$, where A is the root.

In a tangle-based system, there are normally many tangles. For instance, in a dis-

tributed chat system, each message thread—with many participating authors, each creating posts in the thread—would be represented by a tangle. Note that the nodes in a tangle are location-independent, such that two nodes may have been authored in two different append-only logs.

One advantage tangles have over using vector clocks, for establishing partial ordering, is that they do not necessarily grow in size with the number of participants in the distributed system, as a vector clock does. Each message in the tangle does not need to keep track of the state of every other node it knows about, which a vector clock does. Tangles merely require referencing the latest known tips at the time of authoring (in addition to the set of ancestors).

As tangles are relatively untreated in the literature, we will leave off their discussion at this point, and refer interested readers to the article draft by Tschudin [Tschudin, 2018].

3.7 Looking to distributed chat systems

We have described the underpinnings of a class of distributed system, shown how messages passed among nodes in a distributed network can be secured and verified, how those messages may be ordered causally to preserve their original context, and described how normal patterns like managing application state can be accomplished in this domain.

If we combine the outlined distributed systems architecture, with the cryptographically-secured identity systems of Chapter 2, we can start to envision how a distributed chat platform might be formed. Before describing such an entity, however, we first need to specifiy what we mean by a *chat system*. In the next chapter, Chapter 4, we define what this work means by a chat system, outline its components, as well as its issues, before proceeding to present the topic of a *distributed* chat system, which builds on what we have presented in the present chapter.

4

Chat systems

Chat systems are widely used across the world. Friends use them to stay in touch, activists to plan and organise, strangers to discuss common interests. They let us bridge space and time, connecting us over vast oceans and timezones in pursuit of exchanging ideas, feelings and banalities.

This chapter will introduce chat systems from the perspective of how interactions between participants can be managed. Common concepts like groups, admins, and moderators as well as the moderation operations of hiding, blocking and banning are discussed. The intent is to set the scene for how moderation can be accomplished in a peer-to-peer, eventually consistent, *distributed* chat system, presented and discussed in Section 4.2. Distributed chat systems, we argue, are a type of chat system with a different set of capabilities and considerations than its centralised counterparts.

A chat system is, for our purposes, defined as a multi-person networked and primarily text-based communication system with clear boundaries, usually with an emphasis on shorter rather than longer responses and near-synchronous communication though not strictly so. Examples of boundaries, or *chat contexts*, are the Internet Relay Chat (IRC) channel [Oikarinen and Reed, 1993], the Facebook group [Facebook, 2020] , the Mastodon instance [Mastodon, 2020], the Slack workspace [Slack, 2020].

That is, a chat system (IRC) can have many chat contexts (channels), and each of the chat contexts may have many participants.

4.1 Moderation

By moderation we mean the act of controlling the content of a chat context by actively managing its participants. This type of moderation is also known as *content moderation*. The end goal of moderation is to handle disruptive participants by reducing their capabilities (i.e. actions they can take) whether by removing them from the chat, temporarily removing their ability to write, or by hiding their responses. In some systems, individual responses and messages may also be deleted.

Moderation can be implemented on an *individual* level: Alice might hide Bob's responses, giving her a respite from having to read Bob's comments, but without affecting anyone else in the chat context. It can also be implemented on a *group* level, where the disruptive participant has their ability to post new messages revoked across the entire chat context. Participants with the ability to manage *other* participants on behalf of the group are usually referred to as moderators, while the creators of the chat context itself, e.g. a particular IRC channel, are often referred to as administrators, or admins. Administrators can grant and revoke the moderator status of other participants, otherwise their capabilities usually tend to be the same as that of a moderator. Henceforth, we will concern ourselves mostly with moderators.

The design and values of a chat system influence the actions moderators and individual participants can take. Some chat systems like Slack disallow participants from hiding or blocking others, while others like IRC allows individuals to hide (known as *ignore* in IRC) others. Moderators may hide participants on behalf of the entire group (*ban*) and remove (*kick*) them from the chat context.

Terminology We will now delve briefly into the terminology that will be used with regard to chat systems in this work.

The semantics and names for the different moderation actions differ across chat systems, but the fundamentals of hiding and removing unruly participants, and delegating moderation capabilities may be found in a wider variety of systems.

The most common moderation actions include *removing*, *hiding*, and *blocking* a participant. To remove a participant is to take away all of their capabilities. To hide them is to render them speechless usually—but not necessarily *exclusively*—from the hider's point of view. Blocking can have a few interpretations, but it most often includes both the removal of the ability to post new messages as well as preventing the blocked participant from receiving new ones. For example: if Bob blocks Alice, Alice can no longer send any messages to Bob. Alice can also no longer see what Bob posts. In distributed chat systems, blocking can also impact Alice's *distribution* of messages to participants other than Bob—Bob may be a unique connection through which Bob's friends receive Alice's messages. This is discussed further in Section 4.2.

Administrators and *moderators* have already been defined above as participants that can grant and revoke the capabilities of other participants.

The *capabilities* of a participant are the set of actions they can take, and stems from the verb *to be capable of*. The capability to read the messages of the chat context, and to post new ones. To grant and revoke the capabilities of others, or to invite new participants.

By *participant* we mean a single entity, usually a person, represented by a unique identifier within a chat context. In Facebook, this is more or less your legal identity (coupled with either your profile url or a unique database identifier), in IRC it is your nickname within a chat network, either registered with the network or combined with your IP address to produce a unique identifier. The identifier allows capabilities to be managed definitively, without any ambiguity as to whose capabilities are being granted or revoked. For a more in-depth description of identity systems see Section 2.2.

The *chat context* is the domain in which communication between participants occurs. A *chat system* may have many chat contexts. We will use IRC as an illustrative example. In IRC, the chat system is composed of the *protocol*, which determines how messages are exchanged and which actions are possible, the *chat network*, a collection of servers that distribute the network's messages between each other, and the *chat channels*—of which a chat network has many. The chat network may be viewed as one chat context, as a participant's identifier is the same across it; the chat channel as another. Moderation actions can be taken at both the channel level, kicking a participant from the channel, as well as at the network level, banning them from the entire network.

A chat context may also be referred to as a *community*, as the participants of a chat context have a complex set of relations to each other. In the section *Participant classification* below, the terms *community* and *chat context* will be used interchangably—mostly to prevent a dull sense of monotony from repeated use of the term *chat context*.

Different kinds of unwanted participants

There are many conceivable reasons for why a chat participant would be the recipient of a moderation action. The action may be initiated to mitigate the harassment of others in the chat context, to remove an automated peer posting a flood of bogus messages—or one participant might have simply tired of seeing the messages of another.

Participant classification Below is a classification of participants which are relevant to the problem of moderation. More importantly, they are the *causes* of moderation. The intent of the classification is to map out the different causes of moderation, in order to be able to discern the types of moderation actions required for the different levels at which moderation may take place.

- **Spammers**
  Participants, usually automated programs, generating large amounts of unwanted messages.
- **Trolls**
  Participants with malicious intents. These can be subdivided into:
  - **Vulgar trolls**
    Vulgar trolls display acts of overt, offensive behaviour. This may include calling people vulgar names, setting display pictures to offensive images and similar behaviour. They will tend to be blocked as soon as they are detected.
  - **Argumentative trolls**
    Argumentative trolls waste time and create strife in communities by debating other participants on inflaming topics, usually without the intent of bridging a gap in understanding.
- **Harvesters**
  Passive listeners which log and exfiltrate as much as technically possible from the chat context. May also map out participants and their social graphs.
- **Bad apples**
  Bad apples [Felps et al., 2006] are not intentionally malicious, but their way of communicating causes strife in the community and they generally cause more damage than good.
- **Controversial participants**
  Participants involved in dramatically more conflicts, and at a higher rate, than others. Usually not malicious in intent.
- **Help vampires**
  High maintenance participants requiring more energy and engagement from other participants than they are the source of.
- **Unwelcome participants**
  Participants with extremist or other kinds of incompatible views which are orthogonal to a particular community.
- **Half-overlap participants**
  Participants whose interest may overlap with another, but which may be considered odious outside the intersection of interests.
- **Breakups**
  Participants which have had a long-term relationship (not necessarily romantic) but where one or both of the participants have now blocked each other.
- **Orthogonal participants**
  Participants with completely orthogonal interests, viewpoints and communication patterns.

Unwantedness spectrum The classified participants may be tolerated according to the following spectrum along the two axes *desirability* and *scope*.

Desirability reflects the potential desire, or interest, one participant has in interacting with another participant. The desirability axis may either be *welcome* or *unwelcome*.

Scope captures the breadth of conflict a particular type of unwanted participant represents. The scope axis may be either *global* or *local*. A *global* scope is taken to mean the entire chat context, i.e. the entire chat context is unanimous in their blocking of spammers and trolls. On the contrary, a *local* scope is in the context of a single participant; an action taken for the sake of that participant, but which may be different for every participant in a particular chat context. One example would be filtering out half-overlap participants from chat channels outside of the common interest; another would be blocking a previous partner due to a break up. It also possible that the scope lies in between global and local; in this case we use the term *partially global*—this can may be the case when there are neighbourhoods of differing opinion, which we will expand upon soon.

The axes become more clear when viewed in context of the spectrum in Table 4.1.

Desirability \ Scope	**Global**	**Local**
Welcome	Breakups	Controversial participants
Unwelcome	Trolls, spammers	Half-overlap participants, breakups

Table 4.1: Table over unwantedness spectrum demonstrating where participant classifications may fit in along the two axes *desirability* and *scope*.

Let us further expand on Table 4.1 by placing additional examples of participant classifications along the spectrum:

- Globally welcome
  e.g. breakups
- Locally unwelcome
  e.g. half-overlap participants, orthogonal participants, controversial participants, breakups
- Partially globally unwelcome
  e.g. bad apples, controversial participants
- Globally unwelcome
  e.g. spammers, unwelcome participants, trolls, harvesters.

We can see that a participant may be *globally* welcome at the same time as they are *locally* unwelcome, as is the case for breakups—where Alice may have broken

up with Carole, making Carole's posts locally unwelcome for Alice. Alice and Carole are both still globally welcome, as none of the two are malicious participants (merely heartbroken).

Partially globally unwelcome signifies that there may be neighbourhoods of opinion considering a certain type of participant, where one neighbourhood may be more tolerant than other neighbourhoods as regards the type of disruptive participant.

Finally, in an ideal chat context, participants which are globally unwelcome will be filtered out from the chat context e.g. spammers, harvesters, and trolls will have no long-term effect on the chat context.

4.2 Distributed chat systems

We have so far described chat systems implicitly in terms of a server-centric model, where there is no ambiguity as to who may or may not moderate for the group—or even who may or may not be *part* of the group. But there is a new class of systems being developed today which, in academic and business circles, go under the moniker of *distributed ledger technologies* [Walport et al., 2016].

The idea of a distributed ledger fundamentally comes down to variations of the append-only log structure that was the subject of Chapter 3, and in particular: how to structure *writing* into the log, how *many* valid logs there are, and how to *transmit*, or *replicate*, logs between participants of the distributed ledger.

Blockchains like Bitcoin [Nakamoto, 2019] are the most prominently featured examples of DLTs today, but there are other projects which do not fit cleanly under the blockchain classification, yet center on the use and replication of append-only logs. Secure Scuttlebutt (SSB) [Tarr et al., 2019] is one increasingly popular technology and community that fits under the DLT classification.

Properties Distributed chat systems have a set of properties that make them different than their centralized counterparts. In a distributed chat system, each node in the chat system typically stores all of the content they can see and interact with. That is, in addition to the content the node itself has produced, it also stores messages on the local machine that have been created by others (often even messages it has not seen yet). This kind of selfless storage sharing increases the availability of the system, increasing the spread of messages, which will be further discussed in the *Replication* section below.

This class of system is also often delay-independent—a notion which was introduced in Chapter 3—meaning that it continues to function without any internet connectivity. Systems with this property have been termed *local-first* [Kleppmann et al., 2019], because the system design prioritizes the software working on the *local* machine, while allowing its functionality to be extended by the presence of internet connectivity.

Indirect communication The combination of the previously described properties, of delay-independent operations and storage sharing, enable a peer-to-peer distributed architecture. This peer-to-peer distributed network has a greatly increased resiliency, as compared to that of a centralized network.

For nodes to communicate with each other in a centralized chat context, all communications normally have to be coordinated by the central actor—typically a Domain Name System-based (DNS) domain name. In the distributed chat context, nodes instead communicate directly with each other. It goes deeper than that, even. The described properties allow nodes to communicate *indirectly* with each other, through other, yet non-privileged, nodes with greater network connectivity. Let us illuminate what we are positing with a scenario.

Two nodes, Alice and Bob, want to communicate with each other but are unable to because of their network configuration (e.g. inability to accept direct connections due to requiring network router configuration). Luckily, their common friend Carole is good with routers and has configured hers properly. This results in that Alice can reach Carole, and that Bob can reach Carole. Since Carole can be reached by both Bob and Alice, and Carole also stores all of the messages of people she interacts with, this means that Alice and Bob can communicate using Carole as a relay. This property is emergent from the previously described properties of creating a storage sharing and delay-independent system. For an alternate introduction to the underlying communication model, see the article by Tschudin [Tschudin, 2019].

It is important to point out that Carole is a *non-privileged* node, i.e. the distributed chat system is not configured to route all messages through Carole. It just happens to be the case that Carole, a node, is easier to reach, and so that is one of the pathways messages between hard-to-connect peers end up taking.

Going forward we will describe two real-life distributed chat systems: the previously mentioned *Secure Scuttlebutt*, and the group-centric *Cabal*.

Secure Scuttlebutt

Secure Scuttlebutt (SSB) [Tarr et al., 2019] was initially proposed by Dominic Tarr in 2014. SSB is a peer-to-peer protocol and architecture for creating applications with a strong basis in a community's *social fabric*—or the real-life connections and relationships between people and how these are facilitated by their computers. Since 2014, SSB has attracted a large community consisting of academics conducting research on the protocol, companies building applications ontop of it, open source developers improving the protocol and its array of open source applications, and everyday interactions by everyday people. It is currently host to more than 10000 identities [Tarr et al., 2019].

This section will present an overview for *some* of Secure Scuttlebutt's details and properties. Interested readers are encouraged to read the scholarly article detailing the protocol [Tarr et al., 2019], or the beautifully illustrated technical documentation, *Scuttlebutt Protocol Guide* [SSBC, 2017].

Identities In SSB, identities are represented by the Ed25519 signature scheme, discussed in Chapter 2. This has the consequence that any identity can be easily created by generating a signature keypair. In the preceding chapters, we discussed the upsides of being able to generate verifiable and cryptographically secure identities, such as allowing for the transport of messages across many nodes without tampering. One of the downsides, however, is that it becomes possible to generate *any* amount of identities. This can become a problematic attack vector.

Social fabric SSB mitigates the described attack vector with its emphasis on the architecture's *social fabric*.

SSB's follow graph, see Fig. 4.1, is essentially a directed network graph, where nodes *pull* information from the direction of the *following* relation. Following is done by posting so called *follow messages*, which has as payload the public key of the person that is to be followed. That is, users only have their log messages shared across the network if others *follow* them. Thus, in SSB, the social graph and the network graph are one and the same—this facet is what we refer to when we use the term *social fabric*. Fig. 4.1 visualizes the follow graph and how information is propagated in the system.

Spammer example Let us say we have a spammer, Eve. Eve spams by posting bogus messages to her log. For Eve's spam to have an effect on anyone else in the network, other people have to follow Eve. If someone, let us say we have a naive person named Bob, follows Eve, Eve's spam will typically have a reach of 3 *hops*. A hop is essentially a traversal in the follow graph, from one node to another. That is, Eve's 3 hops boils down to: Bob (who is following Eve), anyone following Bob, and anyone following one of Bob's followers. If, however, Bob realizes that Eve is a spammer, he can revert his follow, terminating the spam for all subsequent hops that are connected to Eve through him. Thus, if a spammer is blocked, and they create a new identity, they still have to become followed by others in the network for their spam to have reach. This can be thought of as similar to the bottleneck property of attack resistance, which will be discussed in Section 6.2.

Connectivity One detail which causes issues for the concept of the social fabric and the follow graph is the presence of *pubs*. Due to the difficulties in the current architecture of the internet, establishing so called pure peer-to-peer connections is error-prone, with issues stemming from network address translation and the exhausted IPv4 address space. As a mitigation, SSB has introduced a differentiated type of peer called a *pub*. Pubs are, essentially, automated peers that are easy to establish connections with. Typically, a pub can be made to follow another peer. This causes the pub to effectively bridge the peer being followed into the wider SSB network by storing and propagating the messages of the followed peer.

The problem that a pub poses is that, since it is a semi-automatic mechanism for connecting disparate peers together, it may continue to spread Eve's content unless the pub operator individually intervenes. Thus, everyone in the network has to

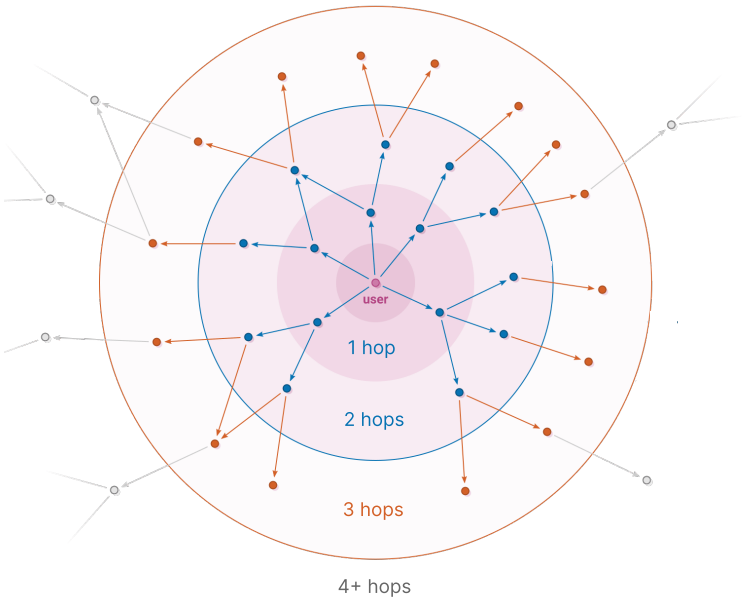

Figure 4.1: Visualisation of a follow graph in Secure Scuttlebutt. The purple center node is the user whose perspective we are viewing from. The nodes within the circle titled *1 hop* have been followed directly by the user. Nodes in the *2 hops*-circle are followed by the nodes that the user follows, i.e. they are 2 edge hops away from the user, or indirectly followed. The orange nodes in the *3 hops*-circle are nodes which are not normally visible to the user in client interfaces, but whose logs *are* stored locally. The *4+ hops*-area contains unknown nodes, representing the frontier of undiscovered parts of the Scuttlebutt network. Image adapted from the public domain-released Scuttlebutt Protocol Guide [SSBC, 2017].

individually block Eve to remove her bogus messages, despite not following her, disrupting the elegance of the follow graph and using unfollows as a type of moderation mechanism.

Finally, we find it important to note that all communication between two peers in SSB is encrypted by using the keypairs of both communicating peers to derive a shared secret, as part of a key exchange protocol. The key exchange protocol, known as the *Secret Handshake* protocol, has been described elsewhere [Tarr, 2015] [SSBC, 2017].

Peer discovery Finding other nodes to synchronize with is an important aspect for any peer-to-peer system. SSB has two main methods of peer discovery.

The first is a Local Area Network-based (LAN) approach. The approach consists of SSB clients broadcasting a User Datagram Protocol (UDP) packet regularly on the LAN connection. The UDP packet contains information about the peer's IP address on the LAN, as well as the peer's `base64` encoded public-key [SSBC, 2017]. This information, then, allows any recipient to connect to, and exchange messages with the broadcasting peer.

Peer discovery over the internet is also present in SSB, and makes use of the aforementioned pubs. We mentioned that pubs could be made to follow users, thus bridging them into the wider Scuttlebutt network. The mechanism for doing so is by an invite-based system. The pub operator can generate an *invite code*, which can be redeemed by another user. Redeeming the invite code causes the pub to follow the user. The invite code contains the IP address—whether as a domain name or as a numerical address—its `base64`-encoded public key, the port the pub uses to communicate, as well as a secret key generated and stored by the pub in the act of creating the invite code. Redemption, then, consists of contacting the pub in the invite code and presenting the secret code. On successful communication, the pub follows the user and proceeds to request the contents of their log.

Replication Sharing data between two peers in SSB is known as *replication*. The term comes from the world of database systems, where it is derived from the act of creating a *replica*, or a copy. Thus, *replication* is the act of sharing copies across nodes in the network—getting each other up to speed on the latest happenings, if you will. Replication in a distributed chat system, then, is the act of sharing messages originating from different peers. The idea behind the message exchange protocol that SSB uses is the same as that explained in the log synchronization example of Section 3.4. For convenience, we will briefly rephrase the contents of the synchronization example with respect to what happens in SSB.

Two peers, Alice and Bob, discover each other and want to know the latest news concerning their groups of friends. Bob sends Alice a list of the peers he is interested in, as well as the id of the latest record for each peer's log; Alice does the same with the list of peers she is interested in. Each peer can then figure out the intersection of the set of feeds the other peer has on their disk, and the set of feeds they themselves are interested in. Importantly, they can efficiently figure out which of the newer records to send to the other party, by considering only those that happened *after* their local copy.

Applications As SSB's protocol and architecture models a social graph, it is natural that there exists a variety of social network-style applications. In Fig. 4.2, we can see an example of a typical social application in SSB. Application views are populated using the messages contained in each peer's log by stitching them together in the manner described in Section 3.5 on *Kappa architecture*. Furthermore, SSB uses the tangles concept to establish partial ordering of messages across different logs, which we previously outlined in Section 3.6. For easily referencing and mentioning other peers, a fundamental action in social interactions, the petname

system described in Section 2.2 is employed, an example of which can be seen in the bottom-most SSB post of Fig 4.2.

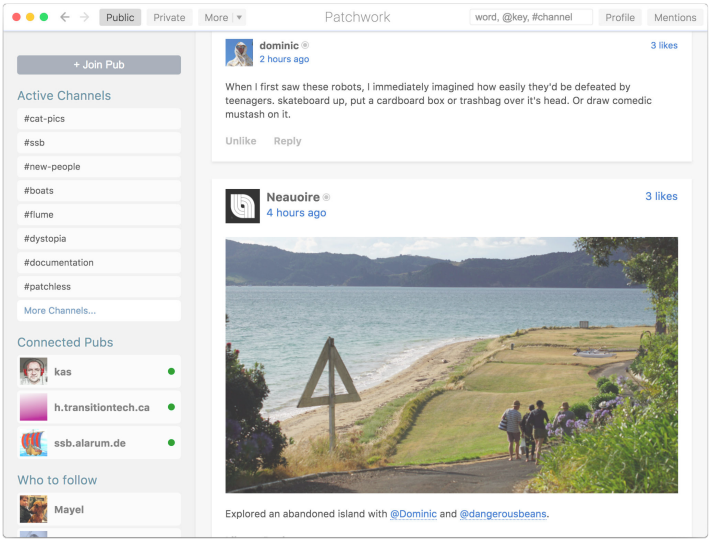

Figure 4.2: A typical SSB application. The image shows the Patchwork client, focused on presenting posts in a forum-style. In the image, we can see posts from two Scuttlebutt users, as well as an example of using petnames to refer to other users. One of the posts features an embedded image, which is also synchronized over SSB but with a mechanism that resides outside of the peer's logs. Binary data, called *blobs*, is synchronized by issuing a *blob request* [SSBC, 2017].

We have now briefly outlined the workings of the Secure Scuttlebutt protocol and architecture. Its design was an inspiration for the Cabal project, which will now be discussed.

Cabal

Cabal [Cabal-Club, 2020], initially proposed by the author in April 2018 [Cobleigh, 2018], is a peer-to-peer application and architecture focused on facilitating group communication that can continue to function in limited connectivity settings. Anyone can start using Cabal by running one of the clients on their computer, sharing the generated *cabal key* with people they want to chat with. The cabal key is a `base16`-encoded Ed25519 public key. The public key is generated for the purpose of using it as an identifier for a *cabal*, the group-specific chat context in the Cabal architecture—the private component of the generated keypair is discarded. Like SSB, there are no accounts or centralized identity servers. Cabal has grown

to encompass a community of 30+ past contributors, in addition to 5 active core developers as of writing. Fig. 4.3 shows an image of Cabal's desktop client.

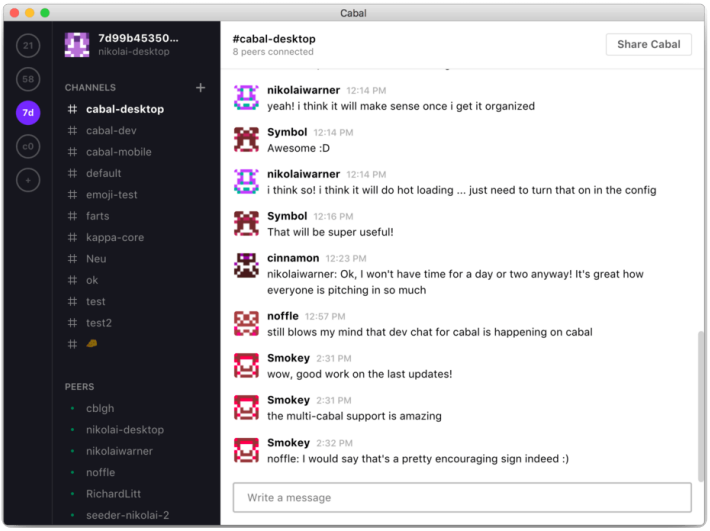

Figure 4.3: Cabal's desktop application.

Similarities to Scuttlebutt Like Secure Scuttlebutt, Cabal also builds on the kappa architecture pattern by combining append-only logs with message-based operations and deriving views for different usecases—whether that be the main chat view for a chat channel in a given cabal, or retrieving the latest nickname of a participant in the cabal. Cabal's architecture is in great part inspired by SSB, as well as IRC [Oikarinen and Reed, 1993] and other chat systems.

Cabal also works on LANs by broadcasting and listening for other cabal clients. Finally, Cabal's identities also build on the Ed25519 signature scheme.

Differences from Scuttlebutt While Cabal has a lot of similarities to SSB, there are also differences. Cabal's clients have a greater real-time focus, favouring inline back-and-forth chat messages, than the existing cohort of SSB social applications, which mainly favour the forum-style seen in Fig. 4.2.

The largest difference, however, is that Cabal has a much greater *group*-focus than SSB. SSB's social fabric puts a large emphasis on interweaving the network graph of Scuttlebutt with the social graph of its participants. It is, however, open for anyone to connect with anyone—as long as they can bridge the social gap. What we mean is that SSB has more of a world wide web-focus, while Cabal is more focused on clearly delineated communities. To explain by metaphor, where SSB is a teeming city, Cabal is the underground punk club only open on odd weekends. It should however be noted that SSB has the option of using a different network key,

which effectively creates a new and unbridgeable Scuttlebutt network; the norm, however, is to use the single, established network key. This is however the exception for SSB, while for Cabal it is the main usecase. That is, it is much simpler and straightforward to create multiple, separate cabals—every one of Cabal's clients implements it—than it is to accomplish the equivalent in SSB.

Cabal also lacks any concept of SSB's pubs at the moment, opting instead for using Distributed Hash Tables (DHTs) to find other participants over the internet [Maymounkov and Mazieres, 2002]. A DHT is essentially a large network of nodes that can be queried. A new node entering the network has to enter from a set of preconfigured nodes, known as *bootstrap* nodes. Having once entered via a bootstrap node, however, they now have access to the entire DHT. Different nodes in a DHT know about different types of information, and the mechanism for finding information is to query nodes, asking them if they know where to find the sought information. For Cabal, this amounts to deriving a *discovery key*, a cryptographically secure hash of the cabal key, and using that to query nodes in the DHT. Nodes that respond affirmatively are identified as nodes part of the cabal identified by the cabal key, and a direct connection is attempted in order to synchronize the logs of the cabal.

Proof-of-concept moderation Cabal features in this thesis both as an example of a distributed chat system, but also because it will later, in Section 7.3, be extended with a proof of concept implementation of a *subjective* moderation system—a topic we will expand on in Section 4.3.

The reasoning behind why Cabal was chosen as the basis for implementing a proof-of-concept, instead of SSB, is as follows. The author has a greater familiarity with the Cabal application stack as opposed to the more mature, and thus complex, SSB ecosystem of code modules. Cabal was simply easier for the author to modify and extend than the extensive SSB stack, simplifying the evaluation process of the proposed trust system. Finally, the need for a moderation system in Cabal was the initial inspiration for the topic of the thesis, so it seemed fitting to conclude the journey where it started, so to say.

Wrapping up

This has been a brief presentation of the Secure Scuttlebutt protocol, as well as the Cabal project. The scholarly article on SSB [Tarr et al., 2019] contains a lot more detail and nuance than what has been communicated here, as our aim has primarily been to introduce the topic of distributed chat systems, as well as outline a few of their important properties with regard to the topic of moderation.

We will now proceed to outline the concept of a *subjective* moderation system, as a conclusion to the entire chapter. We will synthesize aspects of chat systems, traditional chat system moderation, and properties of distributed chat systems to end up with the concept of subjective moderation systems. We will finish the section by arguing that subjective moderation systems are necessarily based in notions of *trust*.

4.3 Subjective moderation

The traditional moderation systems touched on in Section 4.1 are incompatible with distributed chat systems. The main reason behind the incompatibility is that Section 4.2 distributed chat systems have their basis in attempting to remove any centralizing aspect of chat systems; once a chat system is allowed to be delay-independent, the notion of an objective set of moderators quickly becomes difficult to wrangle. Being allowed to delegate the responsibility for mitigating the impact of malicious participants remains useful, however.

The naive approach The simplest moderation system to implement in a distributed chat system is that of everyone independently issuing moderation actions. If Alice wants to remove Eve's spam posts, she has to individually hide Eve. If Bob also wants to stop seeing Eve's posts, he too has to hide Eve. We will call this the *naive* moderation approach. The naive approach can still be effective, however, as illustrated by the Secure Scuttlebutt example, in the *Identities* section above.

The drawback of the naive approach is that it is ineffective, and as such, exhausting to the participants in a given chat context. If participants continually need to *individually* intervene with every bad actor that discovers the chat context, the chat context risks being short-lived.

Delegation What if we could instead have a system of delegation, where Bob could trust Alice such that, if Alice issues a moderation action, then Bob will *automatically* issue an action as a result of his trust in Alice. This kind of system spreads out the burden of moderation along paths of trust, receiving a similar benefit as that of the traditional moderation systems described so far. This combines the positive property of the naive approach, where every participant has been granted the ability to issue moderation actions—a capability not always granted in traditional chat systems—with the positive property of the traditional moderation system's delegation.

The result of such a system would be that a single chat context could have potentially many neighbourhoods of moderation, where each neighbourhood would be based in the relationship of trust that exists within that neighbourhood. The described trust-based system is what we mean when we refer to a *subjective* moderation system.

Traditional vs subjective One hidden aspect of traditional moderation systems is that they grant a special privilege to the initiator of the chat context. This can become problematic for many reasons. The initiator may disappear, leaving the group moderation-less. Similarly, the initiator may be adept at starting new chat contexts, but lacking in skill concerning matters of moderation (e.g. assigning new moderators). Finally, issues may arise where previously good moderators have a falling out and start banning people.

A subjective system, where participants can themselves decide who moderates on their behalf, sidesteps the mentioned problems. As previously stated, the mech-

anism of freely allowing multiple people to moderate also spreads out the invisible care-giving labour required to keep a community free from abuse.

Subjective moderation actions

One question that arises when considering a subjective moderation system is what *types* of moderation actions make sense within it. Clearly, there should be some way to signal a delegation of moderation responsibility, in addition to the ability to hide malicious participants.

One approach that SSB uses is that of *blocking* participants. Blocking in this context works on multiple levels. At its most basic level, it hides any content from the blocked participant. It also signals to other SSB peers not to forward content about the issuer to the blocked participant. Futhermore, it also *removes* the content of the blocked participant from the issuer's local database. This lowers the availability of the blocked participant. If enough people block a participant, they could effectively become hidden in the wider network.

Another approach would be to hide the participant. By *hide*, we mean that the participant issues an action that hides all of the content of the hidden participant, without affecting the availability of the content. It could furthermore act as a signal to others to issue a corresponding hide; we discuss the nuance of this operation in the section *Disambiguating intentions* below.

Disambiguating intentions

A moderation action has a set of semantics that need to be regarded in order for a subjective moderation system to function according to the expectations of its participants. We will illustrate the semantics by looking closely at the *hide* action.

We propose that there are three modes of the hide action for a subjective moderation system: the *personal*, *network*, and *propagated* modes. The three modes can be implemented in various ways, but they need to exist in some form in order to disambiguate the *intention* of the hide. We will now detail the three modes and the intentions they are intended to represent.

Personal A *personal hide*, is issued for the local participant and does not necessarily signal anything other than the participant not wanting to see the hidden participant anymore. A personal hide may be *private*, i.e. not transmitted in a format readable by any other participant (that is, it is encrypted to the local participant only).

The personal hide exists to reflect that, while the hide action is issued for a valid reason, it is purely personal and does not by necessity mean that other peers should issue hides of their own.

Network A *network hide*, is issued by a participant to signal to others who trust them that they, too, should hide the hidden participant. Thus, the network hide is the mechanism by which one participant can signal and warn others of a malicious participant.

Propagated A *propagated hide*, is issued as a *result* of a network hide by another, trusted participant; i.e. it has been *propagated* from a network hide. Propagated actions should not result in further actions—it is propagat*ed*, not propagat*ing*.

The propagated hide needs to exist so that hides that were issued as a result of trust in another participant can be tracked to their origin and also revoked, if the trust has been lost.

Conclusion

We have argued above that a subjective moderation system acts along pathways of trust between the participants of a chat context. That, in order to delegate responsibility, there needs to be a baseline trust in the delegated entity. But what is trust? And how can we capture notions of trust in a computational environment? We explore these questions and more in the next chapter, Chapter 5, *Trust*.

5

Trust

Our lives and societies are built on the foundations of trust. Trust in institutions not to swindle us, in governments and politicians not to waste our money, and, most importantly, in our personal relationships. We trust the people closest to us, and extend that trust to people whom they deem trustworthy. Importantly, we also disregard the advice from those whom have accrued our distrust, regardless of what our friends may say about them.

Trust is foundational for the proper functioning of chat systems. Trust in the fact that the person you are chatting with is who they claim to be, and that they are not swapped out mid-conversation and replaced by some other intervening third-party. Trust in the permanence of actual people behind online monikers is what allows relationships, amorous and platonic, to blossom in the digital landscape.

The digital infrastructure we use daily is also built on trust. Trust of the root and authoritative name servers to honestly answer domain name queries, on the signing authorities of digital certificates, in the companies that are hosting our personal correspondence and private information, and in the tamper-free distribution of software packages and binaries. In the digital sphere, however, there is a pervasive lack of the personal kind of trust we employ in our relationships.

It is, of course, possible to place trust in people and their accounts. To verify fingerprints of secure messaging services, hashes of binaries and so on. This amounts to verifying the *authenticity* of things—this device belongs to that person, that binary has not been exchanged in transit—but speaks little as to any actual degree of trust. There are few translations into the digital realm of the trust and mechanisms which we rely on day to day; that of delegating responsibilities to people that are close to us and whose abilities and confidence we treasure.

5.1 Definitions

The definition of trust this work relies on is one which has been put forward by Bhuiyan, Jøsang & Xu:

> Trust is the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible. [Bhuiyan et al., 2010]

Bhuiyan et al. label the above definition as *decision trust*. The definition beautifully captures the notion of dependence and delegation which is inherent to trust, as well as the fact that trust can be both misplaced and have unanticipated consequences.

The measurement of the extent to which one party trusts another is known as a *trust metric*.

Trust and reputation Trust is also connected to, but separate from, reputation. Abdul-Rahman and Hailes provide the following, somewhat dry, definition of reputation,

> [...] reputation is an expectation about an agents behavior based on information about or observations of its past behavior [Abdul-Rahman and Hailes, 2000]

Reputation is an aggregate. It is an opinion on something derived from a quantity of prior opinions and judgement calls by, chiefly, other entities. Reputation can be used to inform one's trust. If, however, you trust someone, that entails the possibility of maintaining that trust irrespective of any bad reputation the person may have accrued. Trust captures something personal, while reputation is a utilitarian value, condensed from repeated observations by the crowd regarding a particular entity.

Bhuiyan et al. define the essential difference of trust and reputation in two commonplace statements,

> I trust you because of your good reputation.
> I trust you despite your bad reputation.

The first shows trust being informed by someone's accrued reputation. An unknown becoming welcomed into the fold. The second, that the person remains trusted because of prior personal knowledge; a relationship being maintained in stormy weather.

Reputation and trust have their own inherent characteristics which make them suitable for different purposes. Reputation has, for instance, chiefly been used in marketplace scenarios such as ecommerce sites or ensuring the authenticity of files in fleeting peer-to-peer networks [Kamvar et al., 2003]. Whereas personal trust has so far been relatively underused in respect to computer systems.

Trust facets There are a few dimensions, or facets, which make up that which we call trust. Trust is *subjective*. That is, our own trust for a particular entity may differ from that of another person's trust in the same entity.

Trust is, to some degree, *measurable*. Alice may trust both Bob and Carole, while at the same time she may trust Bob *more* than she trusts Carole.

Trust is *scoped*. Alice may trust Bob to take care of her plants while she is away, but she does not trust Bob for financial advice.

Using the above definition of trust in combination with the just described facets of trust, that trust is subjective, scoped, and (to some degree) measurable, we can begin to envision a system of trust able to be mediated by computers.

Trust metrics A trust metric is defined as the way in which trust across entities is measured. The term also extends into the territory of *trust propagation*, or how new trust relations can be discovered given an initial set—trust propagation is discussed further in Section 5.3.

Ziegler and Lausen, authors of the Appleseed trust metric which is the topic of Chapter 6, describe trust metrics in the following manner:

> Trust metrics compute quantitative *estimates* of how much trust an agent *a* should accord to its peer *b*, taking into account trust ratings from other persons on the network. [Ziegler and Lausen, 2005]

5.2 Related Work

This work rests on a body of knowledge from social science as well as computer science. The field of computational trust had a boom around the end of the 1990s to the middle of the 2000s, which was strongly correlated with the advent of popularized and widespread internet usage. Of particular note for the academic community was research focused on the promise of the Semantic Web.

The Semantic Web promised an interconnected network of documents and resources with clear classifications and taxonomies. For various reasons this vision of the Web was never realized—although its promise lives on as the *linked data* movement [Bizer et al., 2011]. The Semantic Web research is becoming increasingly relevant in today's world of social media, and the burgeoning wave of distributed ledger technologies and the alternative propagation schemes they bring.

The now renowned PageRank [Page et al., 1999] introduced the notion of objectively ranking web resources by analyzing directed graph edges, where resources regarded as more interesting receive higher rankings. The authors also identify what they call *rank sinks*, self-reinforcing loops which inflate rankings, and propose a solution for the problem. Their solution essentially amounts to randomly jumping from one resource to another, preventing any prolonged stay in a rank sink. PageRank produces a global ranking for all of its resources, and requires a centralized view of the entire graph.

EigenTrust [Kamvar et al., 2003] uses insights from PageRank to perform a distributed computation of a global trust value for each peer in a peer-to-peer network. The algorithm makes use of peers' own trust assignments of each other to derive each peer's global trust value. According to the definitions in Section 5.1, it would be more correct to call EigenTrust's global value a *reputation score*, as the global value is a derived aggregate that disregards any one peer's subjective trust assignment. EigenTrust also relies on a pre-shared trust base, consisting of known trusted peers, in order to function properly. Mario Schlosser, one of the EigenTrust co-authors, has written an article [Schlosser, 2019] on the conditions and insights that led to the algorithm's inception. It also describes, in plain language, the algorithm itself. It is worth reading.

Abdul-Rahman and Hailes introduce many useful concepts in [Abdul-Rahman and Hailes, 1998]. Conditional transitivity is introduced, the idea that trust is only transitive if certain conditions hold. The authors separate *direct trust*, i.e. trust statments issued from one user regarding another, and *recommender trust*. Recommender trust captures the concept that a party can be trusted to relay and provide recommendations on whom to trust directly. *Trust categories* are brought up, which are identical to the trust scope defined above in Section 5.1. Trust value semantics, mapping trust values to human-meaningful terms, are also introduced. The paper's proposed propagation scheme and trust calculation are admitted by the authors to be too ad-hoc. Furthermore, that users have to assign two different kinds of statements regarding others, direct trust statments and recommender trust, appears well-intentioned but naively optimistic—people are notoriously reluctant to expend energy outside any application's critical path / core value proposition.

In [Jøsang et al., 2003] trust scope, called *trust purpose* in the paper, is mentioned as a core facet of trust. Direct and indirect trust are also touched upon, and used in a similar manner as in [Abdul-Rahman and Hailes, 1998]. Different scenarios are brought up which illustrate the need to properly handle direct and indirect trust, with an important caveat not to issue direct trust assignments using indirect trust as a basis. The conclusion is that parties should only recommend direct trust, as established from their own experiences, to avoid problematic scenarios. Trust calculation is not detailed and neither is trust propagation.

Propagation of Trust and Distrust [Guha et al., 2004] is a treasure trove of trust findings. They empirically test a variety of subjective trust propagation schemes against a dataset of 800k trust scores issued by 130k people. The article makes the claim to be the first to empirically evaluate computational trust. They also introduce interesting variations on trust such as *co-citation*; two people who don't know of, or trust, each other but both trust another third person are more likely to have others in common that they trust. Guha et al. claim to be the first to describe an implementation of computational distrust, and present promising results:

> [..] one of our findings is that even a small amount of distrust information can provide tangibly better judgements about trust informa-

tion. [Guha et al., 2004]

Appleseed [Ziegler and Lausen, 2005], which this thesis builds heavily upon, will be described in detail in Chapter 6.

None of the previous works have as a goal to make the resulting trust scores actionable for end users—whether as peers in a network or as system designers incorporating the work as a software component. The body of existing work has focused chiefly on recommender systems, while this work focuses on enabling automated decision-making in the chat domain. This work is also the first, to our knowledge, in which a trust system makes use of the replication layer of distributed ledger technologies as a propagation mechanism of trust statements. To summarise: this work is, to the best of the author's knowledge, the first to propose a complete trust system that can be incorporated as a software component for distributed ledger technologies, and which provides real value for impacted users by way of enabling automated decision-making and actions as a result of assigned trust scores.

5.3 Computational Trust

Computational trust has many potential applications, which is evident from the literature. Levien's Advogato [Levien, 2003] was used in a real-life internet community which used trust rankings to limit article posting to trusted users.

Ziegler and Lausen [Ziegler and Lausen, 2004b] use trust to improve recommender systems, reasoning that trust and user similarity are tightly coupled. They validate their reasoning by empirically proving the coupling of trust and user similarity for a scraped dataset of book recommendations. Their aim of using trust is to avoid the problems, e.g. the cold start problem, associated with the collaborative filtering techniques used in the majority of recommender systems today.

Collaborative filtering is the idea that you can group users by similarity based on the users' previous scores regarding the same products, i.e. similar users will rate similar products in a similar fashion. Given two users, user A and user B, and a set of products P that user A has not seen but which B has rated favourably. The products P can then be recommended to A if users A and B are similar.

Ziegler and Lausen propose that instead of using only collaborative filtering techniques, one can supplement those techniques by considering *neighbourhoods of trust*. A trust neighbourhood is fundamentally a set of nodes, say book reviewers, that have assigned trust to others nodes in the neighbourhood. A book reviewing community will thus have many neighbourhoods. Producing recommendations then boils down to querying the trust neighbourhood of a node to find e.g. books which the node has not yet read, but which its neighbours have rated favorably.

Computational trust facets

The trust facets brought up in Section 5.1 are the pillars that enable computational trust. We will expand on the subjectivity, scopedness and measurability of the trust

facets for the sake of clarifying what a system of computational trust would require more concretely.

Implementing a *subjective* trust metric means forgoing any attempts to capture an absolute score describing the trustworthiness for any particular peer. What we instead want to determine is a trust score as seen from the perspective of one particular peer upon another. So for any trust judgement issued by a peer, let us call it the *trust source*, there will be a recipient of the issued trust, the *trust target*.

The judgement each source decrees regarding a trust target is called the *trust weight*. The concept of a trust weight will be present in fundamentally any implementation of computational trust. Either Alice trusts Bob, or she does not, or she trusts him to a certain degree—or she does not know of him at all. This trust weight may be inferred automatically from measured behaviour, or it may be explicitly issued by Alice. This work subscribes to the notion of issuing explicit trust assignments. The reason boils down to the fact that any causal inference is very difficult in the chat domain. The problems of automatic trust inference are explored with a couple of examples in the section *Trust inference* below.

If we are to introduce a notion of computational trust, there should be a component of it that regards the context in which the trust is placed. Many applications may, however, be well suited by a single generalizable domain. Outside of digital systems, we trust each other to varying degree depending on what area of life is being considered. Bob may trust Alice for tips on repairing machines, but not at all for help and advice regarding his garden. This facet is mentioned across the literature as *trust scope* [Jøsang, 2007], *trust purpose* [Jøsang et al., 2003], and *trust categories* [Abdul-Rahman and Hailes, 1998]. As there is no definitive classification for this facet, this work has decided to add another one to the fray: *trust area*. The trust area acts as a grouping for related trust assignments. As will be discussed in *Trust propagation* below, trust is only transitive within a trust area.

The expanded trust facets for computational trust have been summarized in Table 5.1.

Trust facet	**Description**
Trust Source	The issuer of the trust assignment.
Trust Target	The target of the trust assignment; the entity being trusted.
Trust Weight	The amount of trust assigned to trust target.
Trust Area	For example: book recommendations, moderation capabilities

Table 5.1: Computational trust facets

Trust inference Let us consider Alice and Bob again. Let us say they are both participants of a chat system and have not interacted previously. Furthermore, let us assume that we want Alice to automatically assign Bob a trust score, resulting in that Bob is either minimally trusted or not trusted at all.

If Bob mentions Alice and Alice mentions Bob back, that could be taken as a signal that they know each other and are communicating amicably. It can however also be the case that Bob is exhibiting unwanted behaviour and Alice is asking Bob to stop. The situation is ambiguous and no inference can successfully be made. It could be argued that sentiment analysis could be utilized to infer minimal trust, but language use is ever-evolving and ambiguous—especially in informal chat settings, rife with community-specific terminology, jokes and irony (not to mention the abundance of human languages in modern use).

In another example, specific to distributed chat systems, we could look at automatically classifying flooding behaviour to mark flooders as untrusted. Flooding in chat systems is usually defined as a user repeatedly posting, often bogus, information at a high rate. If Alice detects that Bob is posting a lot of messages within a short amount of time, she could automatically mark him as untrusted. In distributed chat systems, however, it is a common-enough use case that peers continue to create good natured posts without synchronizing with any other peers (distributed chat systems are delay independent and continue to work perfectly without connectivity). The flood, then, might just be a peer who has been offline for a period of time and is synchronizing their history after having regained connectivity. This example, too, illustrates the difficulty of automatic trust inference in the chat domain.

Finally, there is nothing to stop domain-specific automated trust assignments from being implemented ontop of a system of explicit assignments—it is simply not the focus of this work.

Trust weights

Trust quantization When considering notions of trust that are understandable for computers, eventually that notion must be translated to a number. In reality, the inverse is more the problem—when constructing computational models of trust the system designers will conceive of trust in various arrangements of numbers, for they are at that stage concerned with how to solve the problem for computers.

The practical issue, then, becomes how to present that quantization to people, such that people's decisions conform to a similarity in range as regards the quantization levels. For example, trusting someone to the degree of 0.85, for the range 0..1, does not meaningfully translate to any real life situation. We must instead conceive of *human-meaningful* labels for *ranges of quantized trust levels* such that they feel natural for people to use, enabling them to interface with the system in a way that causes for less individual distortions. For Alice's semantics regarding a trust assignment of 0.85 might differ from Bob's view of what a trust level of 0.85 means, but they both have notions of *friend* which is more likely to converge than pure floating point representations of trust.

See Table 5.2 for an example on trust quantization using labeled ranges. For each quantization range, the higher value is the value that the range will be quantized to.

Note: the labels that are to be used should ideally be specific to the *trust area*. It

might make less sense to use the label *friend* in a trust area of music recommendations as compared to using the label *great taste*, for instance.

What we lose in granularity and representation in using quantized ranges, we make up for in consistency across the nodes. Ultimately, consistency across the trust graph is what is important when it comes to trust transitivity, the topic of the section *Trust propagation* below.

Human-meaningful label	Trust range
New person	0.00
Acquaintance	0.00 – 0.25
Friend	0.25 – 0.50
Peer	0.50 – 0.75
Partner	1.00

Table 5.2: Human-meaningful labels for quantized trust ranges. The label for a range represents the highest value in that range.

Similarity in judgement Trust weights may also have a specific interpretation regarding chosen values, which is that of *similarity in judgement*. What we mean by this, is that a trust source may issue a high trust weight for an entity whom they are similar to in a given trust area. That is, there is a strong overlap in how they think and act in the trust area.

Jøsang et al formalize this concept as *indirect trust*, which is one their two proposed trust variants (the other being *direct trust*) [Jøsang et al., 2003]. The *indirect trust* variant is essentially a *recommendation edge*, by which we mean a source of trust that is outside an entity's direct reach. The recommendation edge's only purpose is to provide recommendations regarding whom to issue direct trust for in a given trust area, or *trust purpose*, as Jøsang calls it. What this work advocates for is that high trust weights, i.e. high similarities in judgement, naturally and implicitly correspond to highly weighted recommendation edges, or indirect trust. This understanding is greatly influenced by how trust weight is used in Appleseed, the topic of Chapter 6. Finally, our interpretation of trust weight as similarity in judgement echoes the user similarity of [Ziegler and Lausen, 2004b], where trust and user similarity are similarily bound, as well as analyzed in earnest.

Trust propagation

Trust propagation is the term for how trust flows from one entity to another in a trust metric.

> Trust propagation is the principle by which new trust relationships can be derived from pre-existing trust relationship[s]. Trust transitivity is the most explicit form of trust propagation [..] [Jøsang et al., 2006]

Trust transitivity Transitivity is the property that describes the following relation: if *A* is related to *B*, and *B* is related to *C*, *then A* is related to *C*.

Trust transitivity, then, describes relations of trust across entities. An example of a transitive trust relation would be that if Alice trusts Bob, and Bob trusts Carole, then Alice trusts Carole. Trust is not inherently transitive but may be regarded as *conditionally transitive* [Abdul-Rahman and Hailes, 2000], i.e. Alice may trust Carole if certain conditions are met (Alice may also trust Bob, her direct trust relation, more than she trusts Carole). One of the conditions for trust transitivity may be, for example, that trust is only transitive within the same trust area. Another condition may be that trust is only transitive if the trust target is a maximum of three *hops* away from the trust source—*Alice trusts Bob* would constitute one hop, with Alice as the trust source and Bob the trust target.

Small worlds theory Small worlds theory, initially explored by Stanley Milgram [Milgram, 1967], states that social networks can be boiled down to random graphs where one node may be reached by another node given a small amount of edges traversals. Small worlds theory is usually referred to as *six degrees of separation*. This has importance for trust systems as it states that, for a given trust system, and given a small average amount of issued trust statements per node, then a useful trust graph may still be derived, despite any sparseness as viewed from a single node.

The term *useful* in this context is taken to mean that trust statements issued by nodes do not form isolated and unrelated islands, but chains of related neighbourhoods. Essentially, small worlds theory is what makes trust propagation and trust transitivity work in practice.

Distrust

Distrust is a contentious topic with differing interpretations—especially when it comes to trust metrics. In some ways, distrust is defined by what it is *not*. Distrust is not negated trust [Guha et al., 2004]. It is also not a lack of trust, which would be represented by *New person* in table 5.2. This work instead views distrust as a complete discounting of anything to do with a distrusted entity. Statements issued from a distrusted entity are discarded, and trust placed in them by others is equally ignored.

This work adheres to the notion that any information coming from a distrusted entity is unreliable, i.e. the old adage *the enemy of my enemy is my friend* is not applicable. This understanding of distrust is mirrored by the Appleseed authors, Ziegler and Lausen:

> [..] distrust does not allow to make direct inferences of any kind. [Ziegler and Lausen, 2005]

Distrust is not, however, entirely negative, in fact Guha et al. propose that a small measure of distrust is beneficial for any trust system. We reiterate the finding

that was initially presented in Section 5.2:

> [..] one of our findings is that even a small amount of distrust information can provide tangibly better judgements about trust information. [Guha et al., 2004]

Explicitly including distrust in a trust metric helps maintain the semantics of trust actions; trust weights do not have to be overburdened by trying to reflect distrust (such as letting a trust weight of -1 or 0 reflect distrust). Modeling distrust also helps negate potential side-effects of using transitive trust. One example of a negative side-effect would be the following: if Alice trusts Bob, and Bob trusts Carole, then Alice would transitively trust Carole. Alice, however, has had prior bad experiences with Carole and knows not to trust her. If distrust is not explicitly modeled, Alice cannot avoid trusting Carole without impacting her trust for Bob.

It therefore seems wise to include distrust as a component of any trust system. As to how distrust may be included, there are a variety of ways, of which two will now be discussed: *transitive distrust* and *one-step distrust*.

Transitive distrust Transitive distrust essentially builds on the idea of trust transitivity, propagating distrust statements made by one entity regarding another across the trust graph. If Alice distrusts Mallory, and Bob trusts Alice, then, in a system with *transitive distrust*, Bob would also distrust Mallory as a result of his trust in Alice.

One-step distrust One-step distrusts limits the distrust to the issuer. In the example above, Alice would discount anything to do with Mallory but Bob would remain unaffected by Alice's distrust for Mallory. Unlike transitive distrust, one-step distrust does not necessarily need to be known by any other node than the issuer—alleviating any potential social penalities in its use, which could otherwise conceivably impact the effective use of distrust as a system component.

TrustNet, the result of this thesis and presented in Chapter 7, implements one-step distrust. Its simplicity is appealing for many reasons. The primary reason is the fact that the context behind the distrust statement is inherently subjective, both to the trust area and the issuer of distrust. Therefore propagating any distrust is essentially problematic—the criteria for distrust may differ greatly across both entities and trust areas, limiting the usefulness of the transitivity of such a statement.

6

Appleseed

Appleseed [Ziegler and Lausen, 2005] is a trust propagation algorithm and trust metric for local group trust computation. Basically, Appleseed makes it possible to take a group of nodes—which have various trust relations to each other—look at the group from the perspective of a single node, and rank each of the other nodes according to how trusted they are from the perspective of the single node.

The Appleseed algorithm was proposed by Cai-Nicolas Ziegler and Georg Lausen of *Institut für Informatik, Universität Freiburg*, initially in 2004 [Ziegler and Lausen, 2004a].

Although this chapter tries hard to illuminate as many details and create an intuition-based understanding of Appleseed, it may fall short of what some readers require. Therefore, if you need more information than what is presented here, consider reading the original 22 page paper [Ziegler and Lausen, 2005], from which this chapter derives greatly.

6.1 Overview

This section gives a brief overview of the contents of the Appleseed paper [Ziegler and Lausen, 2005]. This is done by first outlining Ziegler and Lausen's proposed classification scheme for trust metrics, then follows a presentation of the most important concepts and properties of Appleseed. The section concludes with a physical metaphor of Appleseed, with the intent to present a mental image of the algorithm, before delving into Appleseed proper in Section 6.2.

Trust Metric Classification

In conjunction with presenting Appleseed, Ziegler and Lausen also propose a classification scheme for trust metrics, which will illuminate labeling Appleseed as a *local group trust* metric, as it was presented in the chapter introduction. The reason the author duo's classification scheme is being brought up in this work is that it provides context and insight into what Appleseed is, as well as how it relates to other trust metrics. The proposed classification is composed of three non-orthogonal axes: the *network perspective* axis, the *computation locus* axis, and the *link evaluation* axis.

Network perspective This axis determines the scope of the described trust metric, classifying metrics as either *global* or *local*. Examples of global trust metrics include EigenTrust and PageRank, which were briefly described in Section 5.2's *Related Work*.

Local trust metrics work instead with subsets of the total trust graph, also known as partial trust graphs—with trust paths from one particular entity to another. The *global-local* split may also be viewed along the lines of *objective-subjective*.

Computation locus The computation locus axis describes, as may be guessed from the name, where the trust is evaluated. This metric has a somewhat unfortunate naming for the two values the axis can take on. A *centralized*, also known as local, computation locus describes metrics where all of the computation is done individually by every node.

A *distributed* computation locus describes metrics where the trust computation is portioned out over the entire network—EigenTrust is a good example of a trust metric that would be classified as distributed along this axis. Since the computational load is distributed across the entire network, distributed metrics are also by definition global—full trust graph information is required.

Link evaluation Link evaluation can take on two values: *scalar* and *group*.

In a *scalar trust metric*, each trust assignment is evaluated independently of any other. That is, the value of one assignment does not depend on the value of another.

For *group trust metrics*, however, trust assignments have inherent dependencies. Each trust assignment has a score dependent on the trust of the issuer, which in turn depends on the issuer's *incoming* trust assignments. Groups of nodes, or neighbour-

hoods, are evaluated simultaneously. Group trust metrics calculate *trust ranks* for each node in the neighbourhood.

Having described the proposed classification scheme we can now take another look at the Appleseed algorithm itself. It is a local, centralized (or local), group trust metric. This means that Appleseed works on *partial* trust graphs, it is a subjective metric, it operates on *neighbourhoods* of nodes, and all of the trust computation is done without relying on any external nodes i.e. participating nodes compute their own trust graphs, and it outputs a *ranking* of the nodes from the partial trust graph.

Appleseed at a glance

It might be difficult, initially, to understand all of Appleseed's details. Thus, it seems useful to give a quick overview of important properties, parameters and ways of understanding Appleseed before delving into the algorithm itself.

Spreading activation Appleseed is inspired by a concept called *spreading activation*. Spreading activation appears prominently in language research, specifically in research which deals with word recall [Anderson et al., 1983]. It has been used as a theory to explain the priming effect, where initial exposure to a starting, or *priming*, phrase causes related words to be more quickly recalled, as a result of the prior exposure. Spreading activation was initially a theory that tried to computerize concepts of human language such as semantics, the meaning of words, and how words are related to each other through their semantics. The original theory treated the underlying meaning of a word, the *concept*, as a node in a directed graph, and the directed edges between concepts represent the relevance one concept has in respect to another [Collins and Loftus, 1975].

Spreading activation has later been repurposed for searching large, unsorted data collections by way of transforming the data into graphs of interconnected terms [Ceglowski et al., 2003].

Spreading activation is based on the idea of iteratively redistributing a *finite amount of energy* along a *weighted graph* structure, starting from a *source* and propagating along its *outward edges*. The energy propagates such that edges with larger weights end up funneling more energy than edges with smaller weights. The energy keeps propagating from node to node along the weighted and directed edges as long as the energy to propagate along the edge exceeds the *activation threshold*. If the activation threshold is not exceeded, the propagation for that edge stops.

The image, presumably scientificially inaccurate, the author of this work has had in mind when learning about spreading activation has been of neurons being activated—bursts of potential causing surging electrical stimulation through axons connecting different neurons, with surges of potential being further propagated if the activation threshold is exceeded in a neuron's dendrites.

Appleseed Properties Ziegler and Lausen outline a few properties of Appleseed in their article which feel relevant to briefly highlight. Appleseed *fully* distributes all of its initial energy, in_0, among its nodes—excluding the trust source, which does

not accrue any trust, and nodes which do not have a path to the trust source. The algorithm is also proven by the authors to converge after a finite (but variable) amount of iterations. Experimentally, the amount of iterations before convergence, using the suggested values for parameters, seems to be around 60 to 100. The produced trust ranking is ordered such that the most trusted node, i.e. highest trust rank, is at the top of the list and the least trusted nodes are at the bottom.

Appleseed takes the following parameters as input: the trust source s, the spreading coefficient d, the convergence threshold T_c, and the amount of initial energy in_0. The algorithm implicitly also takes a set of trust assignments, or a trust graph composed of nodes, V, and weighted, directed edges, E. The range of permissible values for the edge weights is $0.0 - 1.0$.

The input parameters have been summarized in Table 6.1. Where possible, Ziegler and Lausen's suggested parameter values have also been included.

Param.	Description	Default value
s	Trust source	—
d	Spreading coefficient	0.85
T_c	Convergence threshold	0.01
in^0	Initial energy	200
V	Nodes	—
E	Weighted and directed edges	—

Table 6.1: Legend of Appleseed's input parameters and their recommended values.

Pooling Water Analogy

In order to make the idea of Appleseed more graspable, a metaphor of water streaming amongst a group of water holes can be fruitfully employed, see Fig. 6.1.

Imagine you have a tub of water containing 200L. In front of you is a group of interconnected holes in the ground. Holes are connected to other holes by tunnels of varying sizes. All the holes are initially without water. You proceed to pour the tub of water into the hole closest to you. The water poured into the initial hole goes along the tunnels connecting to its closest neighbours, and the water that comes into the neighbours is distibuted along their tunnels to *their* neighbours, and so on.

The water sloshes from neighbour to neighbour, with more water ending up at the holes whose incoming tunnels are wider. Eventually the rushing water calms down, and the water level in each hole stabilizes.

The tub of water in the metaphor is the amount of finite energy that Appleseed redistributes from the trust source along its outgoing direct trust assignments. The width of the tunnels is akin to the trust weights, where wider tunnels are likened to outgoing edges with larger weights. The variable amount of water left in each hole after the slushing has ceased can be likened to the computed trust rank for each

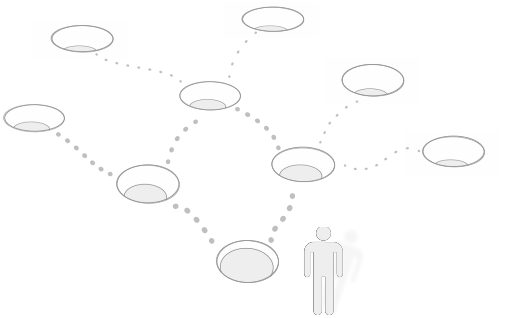

Figure 6.1: A series of interconnected water holes as visual analogy for Appleseed. Each hole is a node in the trust graph, and the water in each hole is the trust a node has after Appleseed converges. Nodes farther away from the trust source ultimately receive lower rankings (less water).

node in Appleseed's trust graph. The more water a hole has, the higher its rank is, and the higher its trust.

The metaphor has flaws in its physicality but serves as a tool to aid in understanding the Appleseed algorithm.

6.2 Algorithm

We now have enough background and context to dive into the algorithm itself, explaining its parameters in greater detail, and how the algorithm itself works.

Algorithm 6.1 contains the full pseudocode for the heuristicless Appleseed trust computation algorithm presented in [Ziegler and Lausen, 2005]. Table 6.2 is a comprehensive legend of the different variables, as well as more complex expressions, that appear in Algorithm 6.1.

Finally, it is important to note that a lot of context and essential details from the paper is not visible when reading the pseudocode listing—one of the aims of the longform descriptions in the *Parameters* section below is to remedy that issue.

Walkthrough

We will now give a guided tour of the code in Algorithm 6.1. Readers are recommended to complement this explanation with the legend in Table 6.2. Line numbers will be referred to according to the convention $\mathscr{L}{:}x$, where x denotes the line being referred to, as seen in Algorithm 6.1's listing.

Algorithm 6.1: Appleseed pseudocode

```
1  function Trust_A (s ∈ V, in^0 ∈ ℝ_0^+, d ∈ [0,1], T_c ∈ ℝ^+) {
2    set in_0(s) ← in^0, trust_0(s) ← 0, i ← 0;
3    set V_0 ← {s};
4    repeat
5      set i ← i + 1;
6      set V_i ← V_{i−1};
7      ∀x ∈ V_{i−1} : set in_i(x) ← 0;
8      for all x ∈ V_{i−1} do
9        set trust_i(x) ← trust_{i−1}(x) + (1 − d) · in_{i−1}(x);
10       for all (x,u) ∈ E do
11         if u ∉ V_i then
12           set V_i ← V_i ∪ {u};
13           set trust_i(u) ← 0, in_i(u) ← 0;
14           add edge(u,s), set W(u,s) ← 1;
15         end if
16         set w ← W(x,u)/Σ_{(x,u')∈E} W(x,u');
17         set in_i(u) ← in_i(u) + d · in_{i−1}(x) · w;
18       end do
19     end do
20     set m = max_{y∈V_i} {trust_i(y) − trust_{i−1}(y)};
21   until (m ≤ T_c)
22   return (trust : {(x, trust_i(x)) | x ∈ V_i});
23 }
```

$\mathscr{L}$:1 The function declaration, $Trust_A$, shows the input parameters being declared, and the conditions that are imposed upon them. The trust source s has to be a node from the trust graph V, the energy to distribute in^0 has to be a positive integer, the spreading factor d in the range 0..1, and the convergence threshold T_c also has to be a positive real number. See the section *Parameters* below for more detail.

$\mathscr{L}$:$2-3$ The initial state is setup. We begin at iteration 0, represented by the variable i, and which explains the subsequent subscripts. The only node we know of at the moment is the trust source s, thus the *discovered nodes*, V_0, is initialised to the trust source.

The trust source's *incoming energy*, $\text{in}_0(s)$, is set to be the energy to distribute over the trust graph, in^0. The incoming energy is what determines the trust rank of a node, which we will explain when we arrive at $\mathscr{L}$:9.

The *current trust ranking* for the trust source, $\text{trust}_0(s)$ is set to 0. It will also stay at 0, as the trust source does not accrue any trust (the trust source is regarded as absolutely trusted, already).

$\mathscr{L}$:$4-7$ We are now looping inside the main body of the algorithm. At the beginning of each loop, the bookkeeping is updated. The current iteration count, i is

incremented, the set of discovered nodes from the previous iteration, V_{i-1}, is carried over to the current iteration, V_i. Finally, the *new iteration*'s incoming energy, $\text{in}_i(x)$, for all nodes that have been discovered, $\forall x \in V_{i-1}$, is initialised to 0. It is important to note that the incoming energy of the *previous* iteration, in_{i-1}, remains intact; the values that were set during the previous iteration are unchanged.

$\mathscr{L}$*:*8 We start to loop over all of the previous iteration's discovered nodes, V_{i-1}; x denotes the node currently under consideration.

$\mathscr{L}$*:*9 This is where the nodes's outgoing trust from the previous iteration is added to the trust rankings of this iteration, for each node that was issued trust. For all of the discovered nodes, $x \in V_{i-1}$, set their *current trust ranking* to be the sum of their previously accrued trust, $\text{trust}_{i-1}(x)$, and the percentage they may keep, $(1-d)$, of the incoming trust, $\text{in}_{i-1}(x)$. The spreading factor d, passed in at $\mathscr{L}$:1 during the function invocation, determines how much of a node's incoming energy will be spread to its trusted nodes. Conversely, it also determines how much energy a node will be allowed to keep. If d is set to 1, nodes pass on all energy, keeping none (the percentage, $1-d$, evaluates to 0).

$\mathscr{L}$*:*10 We start to loop over each edge in the known trust graph. The node x is an already discovered node and present in V_i, while u may be a newly discovered node (it may also, however, be an already discovered node—$\mathscr{L}$:11 handles the undiscovered case).

$\mathscr{L}$*:*$11-15$ The outgoing node was previously undiscovered! It is added to the set of discovered nodes, V_i. The new node's trust ranking, $\text{trust}_i(u)$, is initialised to 0, and so is its incoming energy, $\text{in}_i(u)$. $\mathscr{L}$:14 is special, because it adds the *backpropagating edge*. The backpropagating edge is basically an edge from a node to the trust source s, with its edge weight set to the maximum value. See the section *Backpropagating edges* below for more details.

$\mathscr{L}$*:*16 The proportion of outgoing energy to redistribute to the outgoing node, u, from the currently considered node, x, is determined. x's assigned trust for u is captured by $W(x,u)$, i.e. the edge weight from x to u. The denominator in the fraction, $\Sigma_{(x,u')\in E} W(x,u')$, basically sums up the total outgoing weight from x, where u' exists to keep track of the node under consideration in the summation.

$\mathscr{L}$*:*17 The outgoing node u's incoming energy is set. The incoming energy for u, $\text{in}_i(u)$, denotes the sum of its current incoming energy and u's allocated proportion of x's outgoing energy, $d \cdot \text{in}_{i-1}(x) \cdot w$, where w is what was determined on $\mathscr{L}$:16.

$\mathscr{L}$*:*18 We exit the loop processing all of x's outgoing nodes, initiated on $\mathscr{L}$:10.

$\mathscr{L}$*:*19 We exit the loop processing all of the previous iteration's discovered nodes, initiated on $\mathscr{L}$:8.

$\mathscr{L}$:21 The largest change in the trust rankings is calculated. For each node in the discovered set, $_{y \in V_i}$, determine the largest change in trust rankings between this iteration and the previous iteration, $\text{trust}_i(y) - \text{trust}_{i-1}(y)$, and save that in m for evaluation on $\mathscr{L}$:21.

$\mathscr{L}$:21 The termination condition for the main loop, initiated on $\mathscr{L}$:4. The loop terminates if the largest change in the last iteration, m, falls below the convergence threshold, T_c. If the condition, $m \leq T_c$, evaluates to false the algorithm is regarded as to have converged on the trust rankings for all of the discovered nodes V_i.

$\mathscr{L}$:22 We return the calculated trust rankings for each discovered node x in V_i.

$\mathscr{L}$:23 The function ends.

Param.	Description
s	Trust source
in^0	Energy pool to distribute from s
d	Spreading coefficient. Permissible range $0.0 - 1.0$
m	Largest change in energy in past iteration
T_c	Convergence threshold. Iteration stops if $T_c > m$
V	The set of nodes in the trust graph
E	The set of (weighted and directed) edges in the trust graph
i	Current iteration count
V_i	Current set of discovered nodes
$\text{in}_i(p)$	Incoming energy for node p at iteration i
$\text{trust}_i(x)$	The current amount of accrued trust for node x at iteration i
$W(p,x)$	The value/weight of the edge from node p to node x
$\Sigma_{(x,u') \in E} W(x,u')$	The total value of the outgoing weights for node x

Table 6.2: Legend of variables and expressions appearing in Algorithm 6.1.

Parameters

Spreading coefficient The spreading activation coefficient d determines how much of the incoming energy a node will redistribute. In other words, it controls what portion of the incoming energy the node will *keep*. d is on the interval $[0,1]$, where 1 redistributes *all* of the incoming energy for a particular node, and 0 redistributes none of it—in which case the node keeps all of its incoming energy. The scenario of $d = 0$ is somewhat nonsensical, but a temporary assignment of $d = 1$ is an integral part of the design of Appleseed.

The temporary $d = 1$ value is used for the trust source, which redistributes all of its incoming energy to its direct neighbours—the rationale being that the trust source is regarded as completely trusted, so we don't need to compute any ranking for it.

The expression describing the amount of kept energy can be seen in the psuedocode on line 9 in Algorithm 6.1,

$$(1-d)\cdot \mathrm{in}_{i-1}(x)$$

Likewise, the redistributed energy can be read out on line 17,

$$d\cdot \mathrm{in}_{i-1}(x)\cdot w$$

where w denotes the weight of the outgoing edge being processed.

As seen in Table 6.1, Ziegler and Lausen's suggested value for spreading activation is $d = 0.85$, meaning that 85% of the incoming energy is propagated and 15% is kept by the node redistributing energy.

A final note from the paper on spreading activation:

> Spreading factor d may also be seen as the ratio between *direct* trust in x and trust in the ability of x to *recommend* others [...]
>
> Observe that low values for d favor trust proximity to the source of trust injection, while high values allow trust to also reach nodes which are more distant. [Ziegler and Lausen, 2005]

Convergence threshold Appleseed is an iterative algorithm, meaning it will need several passes to home in on the final result. Iterative algorithms continue looping until a certain criteria has been fulfilled, whereupon the final result is regarded as having been computed. In Appleseed's case, the parameter that determines when to stop iterating is the convergence threshold T_c.

What the convergence threshold says is basically that once the amount of energy that is being redistributed falls below a certain threshold, then the computation has finished—we have our final ranking. More accurately: once the largest amount of energy that has been redistributed in the current iteration, i.e. the assignment of m on line 20, falls *below* the convergence threshold, line 21, *then* we have converged on the final trust ranking for the trust source s, given the nodes V and edges E.

Initial energy The initial energy in^0 defines the energy reservoar which Appleseed distributes from the trust source s via its weighted edges. A nice property of Appleseed is that, after convergence, all of the initial energy in^0 is fully distributed among the nodes in the produced ranking. Another way to view this property is that the total amount of energy in the network, at any point in time, sums up to in^0. That there is no global energy loss simplifies implementation of Appleseed, as it allows implementors to trivially check if the sum of the produced rankings add up to the initial energy in_0—and if that is not the case, then there is an obvious bug that needs to be remedied.

The proposed value of in^0 is 200 and seems to have been derived experimentally with regards to the convergence rate, or, the amount of iterations for the final result—higher values of in^0 cause slightly more iterations.

Trust source The trust source s is something we have come across a few times already. To frame the trust source and ground our understanding of it, let us zoom out a little.

Consider a hypothetical distributed chat system that makes use of trust statements for moderation. In this chat system, there are many different actors, and each actor trusts different people. There may be overlap in whom Alice, one actor, and Bob, another actor, trust—i.e. they may trust the same actor. The chat system taken as a whole, with respect to the actors and their trust statements, makes up a graph with potentially unconnected components (or in the parlance of graph theory, a potentially *disconnected* graph). Appleseed does not operate on this potentially disconnected graph, but instead needs to pick one of the nodes as the root and "lifts" up the graph from that point. This node is the trust source, and its connected graph is a *subjective* view of the potentially disconnected graph we have just been talking about. Viewed in another light, each node that issues trust assignments is, from their point of view, the trust source—and so there exists as many subjective viewpoints of the potentially disconnected graph as there exists nodes.

Appleseed spiders out from the trust source s, along its neighbours and *their* neighbours and so on, to discover nodes eligible for trust rank assignment, and adds them to the set V_i. Ziegler and Lausen's recommmendation for the maximum distance to spider out from the trust source is 3-6 hops, or edge traversals—a recommendation based on Milgram's small world's theory, which was discussed in Section 5.3.

Backpropagating edges

Backpropagating edges, called *backward propagating links* by the Appleseed authors, are virtual edges that are added to every discovered node, with its weight set to 1.0, and the trust source s as its destination. The virtual edges are introduced as a solution to the problem seen in Fig. 6.2, which appears originally in [Ziegler and Lausen, 2005].

In the example, a is the trust source. Let us consider the energy distribution after a has propagated energy along its outgoing weights. Nodes b and d have identical incoming weights, so they will receive the same amount of energy. What happens in the next step is the problem that Ziegler and Lausen identified—which was also previously identified in the EigenTrust paper [Kamvar et al., 2003], but was ultimately left untreated. The problem is that c will end up with *more* energy than any of the children of d—despite c logically being less trusted than e, f, or g (its incoming edge weight at 0.25 is *lower* than any of d's children). The cause is that energy is distributed according to the *relative weight* of a node's outgoing edges; the portion of energy that one node receives is equivalent to its incoming edge weight, divided by the total edge weight of all the outgoing nodes. This is captured in equation 6.1:

$$\text{in}(u) = \text{energy}(x) \cdot \frac{W(x,u)}{\Sigma_{(x,u')\in E} W(x,u')} \tag{6.1}$$

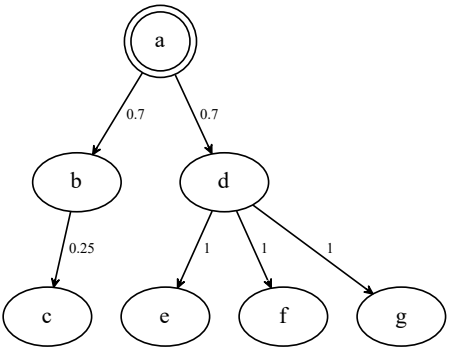

Figure 6.2: A trust graph with a as the trust source.

u is the node we are calculating the incoming energy for, x is the node distributing its energy i.e. energy(x), and u' merely signifies the edge that is being considered in the summation operation of equation 6.1.

Since b only has one edge, the one leading to c with a weight of 0.25, the proportion of energy accorded to c will be:

$$\begin{aligned} \text{in}(c) = &\quad \text{energy}(x) \cdot \frac{W(x,u)}{\Sigma_{(x,u')\in E} W(x,u')} &\iff \\ \text{in}(c) = &\quad \text{energy}(x) \cdot \frac{W(b,c)}{W(b,c)} &\iff \\ \text{in}(c) = &\quad \text{energy}(x) \end{aligned}$$

Hence, c will be accorded all of the outgoing energy of b, despite having a low incoming weight. The nodes $e-g$ all have high weights but will end up with less energy than c, as they have to split d's energy. The described problem is solved by including backpropagating edges.

The inclusion of a virtual edge $E(x,s)$ with weight $W(x,s)=1$ for every node results in c no longer receiving all of b's energy. Instead $0.25/(0.25+1)=0.20$, or one fifth, of the previous energy is accorded. When one of d's children is considered, the accorded energy is now $1/(1+1+1+1)=1/4$, which is higher than the energy accorded to c. As Appleseed is iterative, and the backpropagating edge distributes energy back to the trust source s (which keeps no energy), the reality is that the edges $e-g$ will end up with a lot more energy than c.

The creation of backpropagating edges in Appleseed's trust algorithm can be seen on line 14 in Algorithm 6.1,

$$\text{add edge}(u,s), \text{set } W(u,s) \leftarrow 1;$$

Ziegler and Lausen note a few properties that arise as a result of the backpropagating edges. Appleseed does not need to consider any *dead ends*, dangling nodes which don't pass on any energy—all nodes have at least the outgoing edge leading back to the trust source. Furthermore, they note that another property is an enhanced importance of *trust proximity*. Nodes that are closer to the trust source will receive a higher trust rank—the longer the path from the trust source, the more backpropagating edges have been included and siphoned energy, and the less the amount of energy reaching distant nodes.

Attack Resistance

In order for a trust metric to be considered robust it needs to be attack resistant. What is meant by this is that the metric should not be trivially subverted by a malicious attacker. An example of a trust metric that would be trivially subverted is the following. Let us consider the hypothetical trust metric *FriendPath*.

In FriendPath, an entity is regarded as fully trusted if there exists *any* path from one entity to the target entity. This is trivially attacked by analyzing the trust graph, finding the most trusting participant and causing them to issue a trust statement regarding the attacker.

Ziegler and Lausen mention that Levin's Advogato [Levien, 2003] introduced something called the *bottleneck property* of attack resistance. It is like the bottleneck of a wine bottle—the flow of wine is ultimately limited by the width of the neck. It is the same with the case of the attacker of a metric fulfilling the bottleneck property—the damage is limited by the trust of the attacker. An attacker cannot subvert the trust metric by simply getting the trust of *anyone* in the trust graph; they must instead accrue enough trust that they themselves become highly trusted before they have an impact. In a trust metric that fulfills the bottleneck property, it does not matter that many low trusted entities trust the attacker. Essentially, the bottleneck property says that it is not about the quantity, but the quality of trust.

According to Ziegler and Lausen Appleseed, just like Advogato, fulfills the bottleneck property.

Final notes

We have now described, we hope, everything one needs to know to fruitfully implement Appleseed. The details above were of course derived from the original paper, but a lot of details required the actual implementation effort to unveil.

A few final notes on Appleseed follow, after which we then finish the chapter by enumerating the drawbacks of Appleseed, motivating the existence of TrustNet.

Node judgement Appleseed penalizes nodes that issue significantly larger amounts of trust statements than those that are more reserved. This can be realized from the fact that the energy to distribute is finite, as well as it is distributed in proportion to the amount of outgoing edges. This is a departure from how trust

works in relationships; trust accorded does not detract from previously accorded trust. However, this property does not need to be as negative as it may seem at first glance.

The fact that Appleseed penalizes nodes which issue greater amounts of trust assignments may be viewed as Appleseed slightly promoting nodes with *better judgement*. That is, the idea that nodes who are more selective with whom they trust should be regarded as more credible, and thus promoted over nodes with more lax trust criteria. Using a parallel to everyday life, people who trust a great amount of others may have their recommendations be discounted as compared to recommendations from individuals who are more selective.

Computed trust ranks Appleseed takes the trust assignments that nodes issue each other as input, iterates over them in the manner described in this chapter, and produces a set of *trust rankings*. For completeness sake, we will now briefly elaborate on these computed rankings.

The trust rankings Appleseed produces can be thought of as an ordered list of 2-tuples. Each tuple contains the node identifier, as well as the accorded trust rank. See Table 6.3 for an example.

The computed trust rank is nothing other than the trust energy of the node at the final Appleseed iteration, when the computation has converged. At the top of the list we have the node awarded the most energy—the most trusted entity as seen from s. The second most trusted node comes after it, and so on. Nodes which have been awarded no energy at all do not make an appearance, and nodes which have been awarded the least amount of energy are found at the bottom of the list.

Let us consider the trust graph in Fig. 6.3. Node a is the trust source and the other Appleseed parameters are set to $\text{in}^0 = 200$, $d = 0.85$, $T_c = 0.01$. The produced trust rankings can be seen in Table 6.3. Looking at the table, we see that the trust rankings exclude a, x, and y. a is excluded because it is the trust source; it is the source of truth for the rankings—it does not need to be ranked. Looking at the figure, we see that a has not evaluated any trust for x or y—x and y make up an unconnected component in relation to a's trust graph—this is mirrored in the produced rankings. Finally, we can also see that the computed trust ranks sum up to $\text{in}^0 = 200$, with a loss accounted for by the convergence threshold T_c.

Node	Trust rank
b	84.01307849395832
c	84.01307849395832
d	31.73478305618708

Table 6.3: Table of computed trust ranks for Fig. 6.3 with $s = a$, $\text{in}^0 = 200$, $d = 0.85$, $T_c = 0.01$.

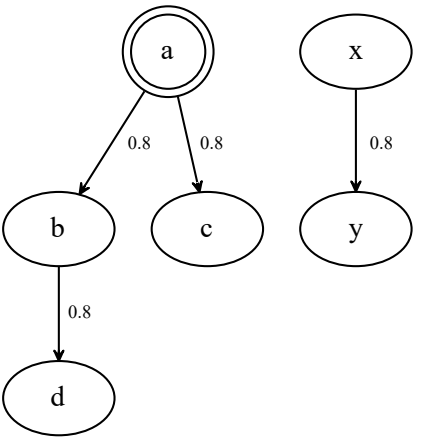

Figure 6.3: A trust graph with trust source a and an unconnected component $x \rightarrow y$.

6.3 Drawbacks

Appleseed accomplishes a lot with its design, and is a comprehensive trust metric. The process of implementing it has, however, revealed a few areas where it is lacking.

Alignment Appleseed does not align the computed trust ranks with the assigned trust weights. As discussed in Section 5.2, trust assignments can be split up into trust ranges with semantics that translate to e.g. a high trust assignment. These semantics are lost in the produced trust ranking—all we have left is a list of numbers with decreasing magnitude. Ziegler and Lausen acknowledge this fault in their paper, and propose a heuristic which is a modification of Algorithm 6.1. The authors state that their proposed heuristic is imperfect but worked well in a range of unpresented cases. The heuristic-aware trust algorithm has not been used in this work due to its unwanted complexity, especially since it does not remedy any of the other drawbacks mentioned in this section.

Distrust Distrust is included in [Ziegler and Lausen, 2005] but only receives a mention in the initial article from 2004 [Ziegler and Lausen, 2004a] as a possible extension. The proposed distrust extension negates some of the merit of the original algorithm, as its inclusion now causes the computed trust ranks to no longer equal the initial energy in^0—i.e. the proposed distrust scheme causes a loss of energy. Fur-

thermore, Ziegler and Lausen's distrust propagates across nodes which goes counter to the subjective notion of distrust which this work adheres to, see the previous discussion on Distrust in Section 5.3.

Practical use The biggest drawback of Appleseed, however, and as far as the author knows this applies to the literature as well, is the lack of consideration for *how* the trust ranks will be used practically. By what is meant, *how* will a list of identifiers and floating point numbers be translated into something of value for end users or application developers? Ziegler and Lausen's treaty of this issue is the following:

> Hereby, the definition of thresholds for trustworthiness is left to the user who can thus tailor relevant parameters to fit different application scenarios. For instance, raising the application-dependent threshold for the selection of trustworthy peers, which may be either an absolute or relative value, allows for enlarging the neighborhood of trusted peers. [Ziegler and Lausen, 2005]

This major drawback is the main reason of the existence for TrustNet.

Lack of trust area Appleseed lacks any notion of trust area, previously discussed in Section 5.3, and the topic is largely left untouched in [Ziegler and Lausen, 2005]. As discussed previously, it is vital for the coherence and consistency of the resulting trust graph that a notion of trust area is captured. This oversight will be discussed and treated in Chapter 7.

TrustNet

Appleseed's drawbacks are mitigated to various extents in the proposed TrustNet trust system, the topic of Chapter 7, and the main contribution of this work.

TrustNet maintains the semantics of issued trust assignments and solves the issue of making Appleseed's final result more practically useful with the same mechanism: ranking strategies. Ranking strategies will be further detailed in Chapter 7. The specific ranking strategy TrustNet implements is one of using k-means clustering on Appleseed's produced rankings to segment the irrelevant bottom rankings from the top ranked nodes.

TrustNet also introduces the notion of trust areas by maintaining one Appleseed trust graph per trust area. A trust area is a semantic cordoning off of trust assignments. Trust assignments issued by nodes in one trust area will not affect the trust graph of another trust area.

Finally, distrust is handled in TrustNet in the manner described in Section 5.3's *one-step distrust*, where a node will ignore trust asssignments pertaining those whom it regards as distrusted.

7

TrustNet

TrustNet is a system for representing, and interacting with, computational trust. The system is comprised of a trust metric, a system for trust propagation, and it takes distrust into account. Most importantly, it provides a simple interface for using and interacting with the results of the trust propagation.

TrustNet is agnostic as to which system it is used with, i.e. the underlying trust assignments provider, and is intended for use with the distributed chat systems of Section 4.2. It may, however, be fruitfully employed in other distributed ledger technologies, or even traditional server-centric models.

The purpose of this work in its entirety, and this chapter in particular, is to motivate and demonstrate the use of TrustNet in terms of facilitating a subjective moderation system, which was the initial impetus for this undertaking. TrustNet has, however, turned out to be more flexible than anticipated and should allow for use cases outside the realm of chat system moderation.

7.1 Overview

The following will be a brief overview of what TrustNet is, as well as how it makes use of and improves upon the Appleseed algorithm from Chapter 6. There is some overlap of the contents of this section and that of the architectural overview in Section 7.2. We will start by outlining what problem TrustNet is explicitly trying to solve, and then detail how it approaches its solution.

Problem statement

The problem that initiated the research into TrustNet was the following:
How to efficiently hide malicious participants in a distributed chat context.

We have previously, in Section 4.1, defined what we mean by *hiding* and *malicious participants*. We have also outlined, Section 4.2, our definition of a *distributed chat context*. Thus it remains to define how to accomplish the above in an *effective* manner, and also to distinguish that from hiding malicious participants in an *ineffective* manner.

Approaches Let us start by detailing the ineffective manner. Assume we have a distributed chat context in which an N active participants are amicably chatting away in an entirely distributed fashion, and without any hierarchical structures—structures which otherwise make up the foundation for the traditional moderation approach detailed in Section 4.1. Let us now say that a malicious participant has entered the chat context and started maliciously flooding links—-the N initial participants have had their chat disrupted. In order to effectively remove the disturbance caused by the malicious participant, they each must individually issue a hide for the flooding participant—i.e. N hides must be issued. This is clearly ineffective, especially given the situation where another malicious participant enters the chat context, requiring an additional N moderation actions, and so on.

Thus the *ineffective*, or *naive*, approach may defined as: each participant in the chat context must individually hide the malicious participant in order to mitigate the disturbance.

An effective strategy would attempt to minimize the amount of actions required to remove the malicious participant. Let us define the *effective* approach as: a *subset* of the chat context must hide the malicious participant in order to mitigate the disturbance for the entire set.

How TrustNet solves the problem

Our proposal for the effective approach is the following. Participants in a chat context issue which other participants they trust, and to which degree. Appleseed is then used to convert the resulting trust graph—from the perspective of a *single* participant—into a ranked list. TrustNet converts the ranked list into an actionable subset of the original list i.e. the most trusted participants, or peers.

The most trusted peers are then used in a moderation capacity by looking at the network hides—see Section 4.3 for the types of hides—issued by the trusted peers. As a result of the network hides, the moderation system automatically issues propagating hides on behalf of the active participant.

In summary: an initial subset of nodes is trusted directly. The nodes are then used to find a network of trusted peers. The most trusted peers are used to collect hide actions, which will be automatically issued by the active participant. There is a high likelihood of the collected hides having been issued in good faith, as their issuers are highly trusted. If a hide action is later discovered to have been unjust in the eyes of the active participant, then the hide issuer may in return be issued a distrust assignment—removing their capacity to issue further actions on behalf of the active participant.

Let us continue with detailing TrustNet more explicitly, describing what low and high trust assignments may look like, how trust areas are incorporated into the design, as well as the simple approach to distrust that is employed.

Trust assignments

TrustNet is built on the assumption that the participants of a chat context will issue trust assignments. An issued trust assignment is essentially a participant saying that they agree with the target participant's view within the particular trust area—the trust area considered in the case of this work, is that of *moderation capabilities*. This assumption feels reasonable as participants are the ones that need to put in the work of assigning trust, and it is also they who will benefit from the assigned trust. Aligning adaptation requirements with benefits has proven to be important for the successful adoption of any new system [Thaler and Aboba, 2008].

The issued trust assignments reflect the facets detailed in *Computational trust facets*, Section 5.3. That is, trust assignment's contain a *trust source*, a *trust target*, a *trust weight*, and a *trust area*. TrustNet further imposes a requirement that trust weights must be in the range of 0..1. This requirement exists as TrustNet makes use of Appleseed to propagate trust and produce trust rankings. We will now detail each of the computational trust facets and how they are put to use in TrustNet.

Trust source The trust source is the entity issuing the trust assignment. The trust source may be confused with the *active participant*, which we have previously referred to in this chapter. When we refer to the active participant, we mean the actual user whose subjective trust graph we are considering. The entity being issued trust by a trust source is known as the *trust target*.

Trust weights Trust weights are restricted to the range 0..1, but what do different values in that range signify? As brought up in the Trust chapter, we can assign different semantics, or meaning, to different ranges of values through the use of human-friendly labels. The labels aid in aligning the numerical value of a trust weight across different participants. But what does a higher trust weight actually

affect? In Appleseed, higher trust weights essentially translate to a greater recommendation power for the trusted entity—the higher the trust weight for a trust target, the greater the influence of the target's trust assignments.

In other words, the magnitude of a trust weight can be viewed in terms of *"how well do I view the trust target's ability to* recommend *others?"*. That is, the magnitude of the trust weight as a recommendation metric or, as previously discussed, as similarity in judgement to oneself. Thus, a trust weight of 1.0 signals an unlimited trust and an absolute belief in the trust target to recommend others, while a trust weight of 0.1 signals a highly limited trust and a very weak belief in the trust target to recommend others.

Trust assignments may be reissued over time. The causes for re-evaluating a trust assignment are many. Perhaps an entity has proven themselves over time, resulting in a previously limited trust being upgraded to a higher level. The reverse may also happen—a lapse in a peer's judgement may result in trust being completely revoked.

Trust areas TrustNet implements support for *trust areas*, one of Chapter 5's computational trust facets. The purpose of a trust area is to contextualize and contain trust—trust in someone's ability to recommend good music does not necessarily correlate to the same person exhibiting good judgment in cases of moderation. By *contain trust*, we mean that one trust area has a distinct set of trust assignments as compared to that of any other trust area.

TrustNet has a simple, but not simplistic, implementation of trust areas: each trust area has its own distinct trust graph, and as such its own set of trusted peers. Thus, trust assignment providers—the underlying chat system—must keep track of which trust assignments pertain to which trust area and pass them on to TrustNet. TrustNet attempts to simplify this process by keeping track of registered trust areas and providing convenience methods for interacting with them.

The number of trust areas a given chat system chooses to implement is advised to be considered well. Too few, and the semantics and behaviour of TrustNet does not map onto user expectations. Too many, and people will become fatigued by the labyrinthine trust taxonomy the well-meaning developers have constructed. Furthermore, it is suggested that trust areas are *not* left to be populated by users, but instead tailored to the particular system and its context. The trust areas may be co-developed with users by starting with a small set of trust areas, such as beginning with a single trust area, and then expanding the number as users discover the need. Too many trust areas will needlessly segment the trust graph and transitivity will falter. That is, trust areas are prone to the same issues as those that plague metadata tag systems. Too many disparate tags, e.g. `rock`, `rock music`, `roc k` (sic.) for the same music genre causes noise and hinders discoverability for tag systems and prevents its effective use. The same is true for trust areas.

Example The example below shows what a trust assignment would look like coming from Cabal, discussed in 4.2:

```
{
  "src": "fcdd825dc7f3085d52d14fc253f0a539640ffc3a5b4cef36664dfbaa539ce43d",
  "dst": "412b4e1c27c8d03f6a59a2a4af4199b149d1d8578a93cf6a4fedceed2f30b279",
  "area": "moderation",
  "weight": 0.8
}
```

The trust source and trust target, or *destination*, are identified by `src` and `dst`, respectively. The actual values of the trust source and destination are base 16 representations of the ed25519 public keys of two participants of a cabal. The trust area is that of *moderation*, and the trust weight is a rather high one.

The example itself is valid *JSON*, an open standard for human-readable data, and also the message encoding currently used by Cabal.

Distrust

Distrust is handled by TrustNet in the one-step distrust manner described in Section 5.3. Each peer has a list of participants they distrust, which is passed to TrustNet by the chat system. Incoming trust assignments are then checked against the distrust list. If either the assignment's trust source or trust target are found in the distrust list, then the entire trust assignment is discarded. This prevents distrusted peers from influencing the trust graph. It is also important to reiterate that a peer which is distrusted within a particular trust area is not necessarily a malicious, it is just that they have an orthogonal view to the active participant within the given trust area.

The distrust list may be openly broadcast to everyone in the chat context, or it may be kept private. There are a few considerations to take into account when determining which approach to choose.

Openly broadcasting the entire list boils down to a social problem, in that people would likely be reluctant to interact with a trust system which publicly broadcasts whom they distrust. This social issue compromises the entirety of TrustNet. Keeping the list private, then, may be done by either encrypting the list such that only the active participant can access it, or if the list is kept entirely local. The disadvantage of keeping a local list is that it may be lost in the event that the local machine is somehow lost.

The problem with the open broadcast-approach, as well as the local-only approach, can be solved by encrypting the contents of the distrust list before publishing it to the distributed context. The key for accessing the encrypted list could be shared across all of the active participant's devices, allowing access to the list even in the face of data loss—anything broadcast to the entire distributed chat context is persistent thanks to the presence of the other peers in the system, mitigating the risk of the local-only approach.

Thus we advocate for the use of a locally persisted distrust list for simplicity's sake—or, if practically feasible, a broadcasted and encrypted distrust list.

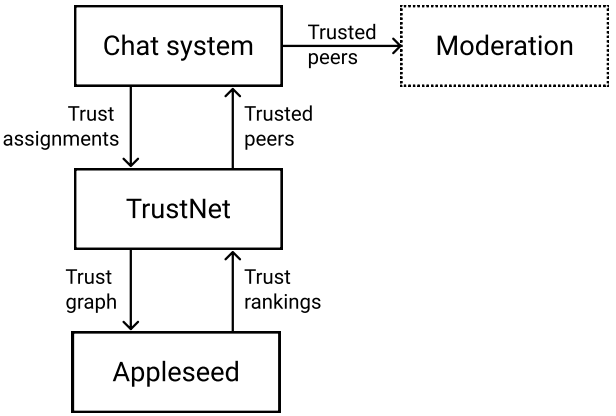

Figure 7.1: Schematic of the components that make up the proposed TrustNet system.

7.2 Architecture

TrustNet is a system composed of many parts. See Fig. 7.1 in order to follow along in the system description below.

TrustNet starts with considering the end users: the chat system, and the participants in one of the chat system's chat contexts. Participants in the chat context interact with each other, and over time trust develops. In a scenario where TrustNet is implemented, the participants ultimately issue trust assignments for each other within different trust areas, such as *moderation*. TrustNet receives these trust assignments from the chat system, processes them and creates a trust graph. The trust graph is then passed to Appleseed. Appleseed processes the trust graph's nodes and edges and produces a list of trust rankings. The trust rankings are passed back to TrustNet, which separates the highly trusted nodes in the rankings from the low-ranked nodes. The trusted nodes, or peers, are then passed back to the chat system, which passes them on to its moderation subsystem. The moderation subsystem looks at the moderation actions issued by the trusted participants and mirrors them on behalf of the active participant.

Ranking strategies

The computed trust ranking returned from Appleseed essentially contains the same nodes as the initial weighted trust graph—the difference is that unconnected components are not present, and that nodes are sorted in order of trust. The computed ranking is a list where the most trusted peers are at the top of the list and the peers

with the least amount of trust, but still trusted, are at the bottom. The problem that arises with such a list is where to make a cut, such that you get one part only containing irrelevant peers—whose rankings are too low to be useful in the given trust area—while the part that remains contains the trusted peers. If we do *not* exclude any peers from the ranking, then that amounts to operating on the entire connected trust graph we started out with, where anyone, regardless of how little trust they accrue or whom trusts them, may make decisions which can impact you, failing the bottleneck property of attack resistance discussed in Section 6.2. To solve this problem, and make the list of peers actionable and more useful, we will implement what we will call *ranking strategies*. A ranking strategy basically operates on a ranked list and produces a subset of it, according to the chosen strategy.

One ranking strategy: from the list of peers, pick the most trusted peer. Or, more generally, pick the N most trusted peers. Another form of strategy would be to use a clustering algorithm to identify clusters of nodes with similar ranks, creating groupings of high trusted peers, while sorting out low trusted and irrelevant peers. This will be further elaborated in the section *Clustering-based strategy* below.

Low trust graphs One caveat with all strategies is that, while the rankings produce the most trusted peers for the given trust graph, we might be operating on a graph based in low amounts of trust. That is, a graph where the trust source has only issued trust assignments with low trust weights (0.1 instead of 0.75, for example). In this case, we would still get a ranking where one peer has the highest amount of trust, but it would be a relatively low trusted peer in terms of the trust source. The reason we end up with a low trust graph has to do with how Appleseed's rankings are computed. The rankings returned from Appleseed are effectively calculated in proportion to the initial set of trust assignments issued by the trust source. Thus, Appleseed's trust ranking will contain more reliable results if the trust source has issued at least one higher trust assignment, as the rest of the trust graph's edges will be placed in relation to the trust source's higher trust assignment. A *meta strategy* could take this into consideration by looking at the trust assignments issued by the trust source, in addition to the main ranking strategy in place. The meta strategy could return an empty list if it discovers that the trust source does not have any outgoing trust assignments of medium or high weights, and otherwise return the result of the previously chosen ranking strategy. TrustNet makes use of this kind of meta strategy.

Since a ranking strategy operates on the computed list of ranked peers, the computed list can be cached. This enables many strategies to be in use at the same time, e.g. a more narrow set of peers might be useful for moderation, while a strategy that gives you a wider range of peers with lesser amounts of trust could be used to allow for a kind of consensus—where you will commit to an action only if the pool of trust from the trusted peers with smaller amounts of trust have also committed to the action. The last strategy could be useful to avoid a kind of *cool kids problem*, where a select few are vastly more popular than any others in the chat context.

Clustering-based strategy The clustering strategy is the main ranking strategy that TrustNet makes use of. Clustering is accomplished with the `k-means` clustering algorithm, described below.

`k-means` operates on ordered data sets and attempts to split the set into k groups, or *clusters*, with the goal that members of a cluster are more closely related to each other than members of another cluster. In each iteration of `k-means`, the members of a cluster are used to compute an average, or *mean*. The k means are then used to define a new centerpoint to use for splitting the dataset into k new clusters. `k-means` terminates when the change in cluster membership between two iterations is regarded as small enough, whereupon the data set has been clustered into its k final clusters. (In actuality, the TrustNet implementation uses *Ckmeans*, which is a dynamic programming-based approach for k-means that is optimal for 1 dimensional datasets [Wang and Song, 2011]—a perfect match for Appleseed's one dimensional rankings.)

In TrustNet's clustering strategy we set $k = 3$. This partitions Appleseed's produced rankings into three related groups, according to their calculated trust ranks. In order to get a reliable set of trusted peers, we discard the cluster containing the *lowest* trust values. If we instead decide to pick only the highest cluster, and discard the other two clusters, the result would on average be equivalent to taking the trust source's direct trust assignments as trusted peers, and discarding the rest of the trust graph—a result which invalidates any kind of trust propagation, as well as the use of Appleseed, in particular. That the highest cluster tends to contain the direct trust assignments has been found experimentally.

7.3 Experiment design

In order to evaluate the effectiveness of using TrustNet to implement a subjective moderation system, we will have to test it against a few scenarios. A scenario simulator, as well as a demonstration framework, have been created in the process of this work to facilitate evaluation.

The scenario simulator pseudo-randomly generates different trust networks according to a set of parameters, while the demonstration framework is a proof-of-concept implementation of TrustNet as a subjective moderation system in Cabal, the peer-to-peer group chat introduced in Section 4.2. The demonstration framework, however, is not implemented directly in any of Cabal's chat applications, but is instead viewable as a webpage that communicates with spawned Cabal instances over WebSocket events, a protocol for opening a two-way communication channel over a TCP connection.

Scenario simulator

The scenario simulator attempts to model the aspects relevant to issuing trust assignments in a social network setting. As such, the simulator has implemented a set

of parameters which can be varied in order to test different types of scenarios.

It has a configurable node count, controlling the total size of the nework. There is a seed parameter, allowing for reproducible experiments in conjunction with the pseudo-random number generator. The trust weights that comprise a trust assignment may be individually configured across the following trust ranges: *none*, *low*, *medium*, *high*, and *absolute*. For example, the trust level *low* can be set to represent a trust weight of 0.25 or 0.30, which allows simulating the difference in the final trust graph.

The skew, or the probability that a trust assignment will be issued for a particular trust level, is also configurable across the five trust levels. The sum of the five skew parameters has to equal a probability mass of 1.0. Thus, if the skew for the trust level *absolute* is 0.01, there is a 1% chance that any given trust assignment will be issued as an *absolute* trust assignment, i.e. corresponding to the trust weight set for the absolute trust level.

Finally, the range on the number of trust assignments nodes will issue is also configurable. This has been represented by two values, the lower and upper ends of the range; how many trust assignments a node will issue is randomly chosen using the two range endpoints.

An example listing of a scenario configuration may be viewed in Listing A.1 in the appendix.

Demonstration framework

With the scenario simulator, we can model a wide array of scenarios and generate their resulting trust graphs. Models are however only as good as their chosen input parameters. Furthermore, as our intent is for TrustNet to be able to be used as the basis for a subjective moderation system, implementing TrustNet into an existing chat system would reveal flaws in the system's implementation as well as any awkwardness in the interface between TrustNet and the host chat system. Thus, the need for a demonstration framework arose. Fig. 7.2 shows a visualisation of the framework.

The demonstration framework uses an adapted version of Cabal's underlying core libraries. Its architecture is as follows. A nodejs-based HTTP server was built to serve the demonstration framework as a website, loadable in modern browsers. The HTTP server was extended with a WebSocket server for relaying requests from the participant—interacting with the framework in the browser—to the underlying Cabal nodes. The same process that serves HTTP requests and relays WebSocket events is also responsible for spawning new processes containing a special kind of Cabal node. The spawned Cabal nodes communicate directly with each other, as per Cabal's normal mode of operation, but they also have the ability to interact with WebSocket events from a defined WebSocket server. See Fig. 7.3 for a schematic of the demonstration framework's architecture.

In this way, we simulate a real Cabal network, while also simplifying setting up trust assignments between nodes, as well as enabling interactive exploration of the

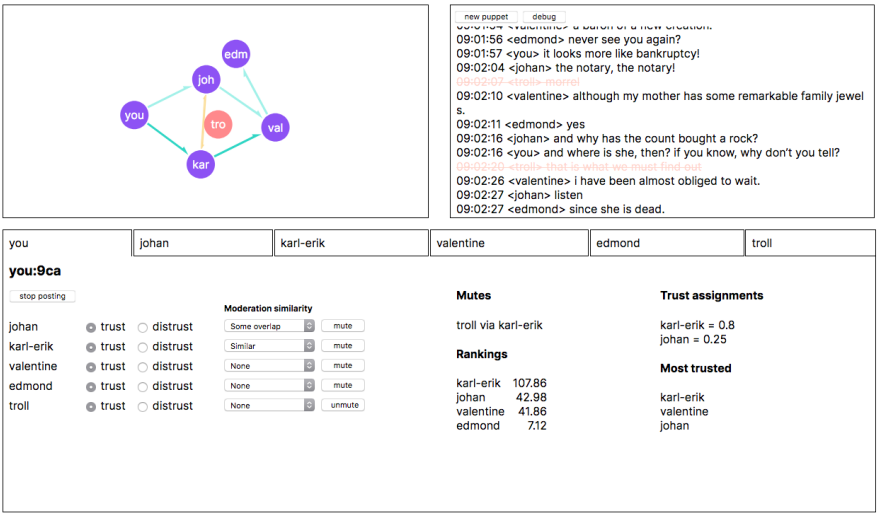

Figure 7.2: A screenshot of the demonstration framework developed to interact with the proof-of-concept implementation of TrustNet into Cabal, a peer-to-peer application for group communication. The trust graph's edges are visualized, as are the rankings returned from Appleseed, and the most trusted participants according to TrustNet. The pink text in the chat window, top right, visualizes content posted by hidden peers. In the image, the peer has been transitively hidden.

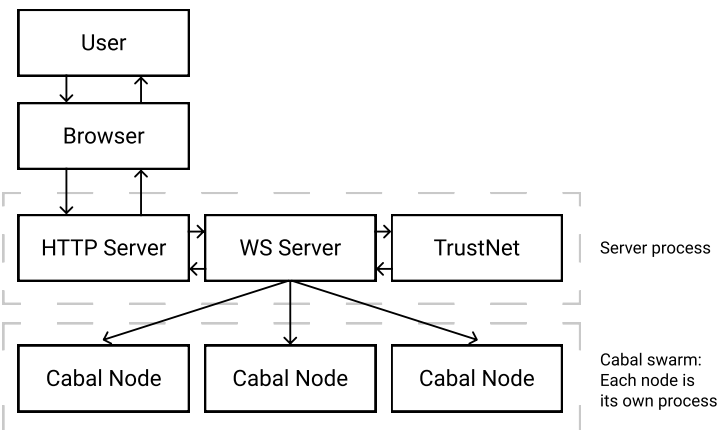

Figure 7.3: Schematic of the architecture and communication paths used in the demonstration framework.

emergent trust network. The demonstration framework in Fig 7.2 also visualizes the trust network as a node graph—see the top left—as viewed from the currently inspected peer. The computed Appleseed ranking is also visible, as well as the trusted peers returned from TrustNet; the section labeled *Most trusted* contains TrustNet's trusted peers.

A component for visualizing how the chat history is affected by hiding nodes in the network is also present. The chat history component also doubles as a debugging view, which can show the underlying events issuing forth over the WebSocket connection.

7.4 TrustNet Example

Let us illustrate what could happen in a scenario of a chat system that makes use of TrustNet to implement a subjective and trust-based moderation system. The trusted peers in the example below were derived by creating trust graphs according to the trust assignments mentoned in the text, and running TrustNet on those trust graphs.

We have the chat participants Alice, Bob, Carole, David, Eve, and Mallory. The participants have issued trust assignments for each other in various ways in the trust area *moderation*. The trust area is used to propagate moderation actions inside the chat system, spreading out the burden of mitigating abuse.

The chat system uses human-meaningful labels for each assignable trust weight:

- **None** (0.0)
- **Some overlap** (0.25)
- **Similar** (0.8)
- **Identical** (1.0)

The default for an unassigned trust weight is the same as assigning *None*. Alice has a similar attitude to moderation as Carole, and assigns her as *Similar*. Alice also assigns Bob as *Some overlap*, as she thinks Bob usually has a good sense of moderation, but she is a bit uncertain as to who Bob may trust in the future.

Carole and David are best friends and similar when it comes to most things in life. Thus, Carole trusts David's judgement, assigning him as *Similar*, and David trusts Carole, assigning her as *Similar*. Carole also trusts Alice, assigning her as *Similar*. The trust graph for the chat system at this point in time can be seen in Fig. 7.4.

At this point in time, the trusted peers for each chat participant are:

- **Alice**: Bob, Carole, David
- **Bob**: None
- **Carole**: Alice, Bob, David

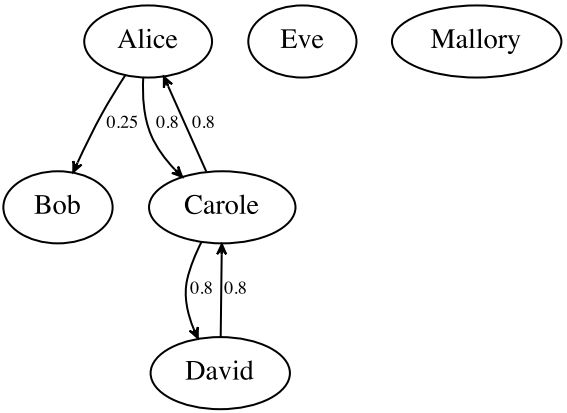

Figure 7.4: Trust graph for the hypothetical chat system after Alice, Carole and David have issued their trust assignments.

- **David**: Alice, Carole
- **Eve**: None
- **Mallory**: None

Eve and Mallory are malicious participants, intent on having fun at the expense of the rest of participants in the chat. Consequently, they try to game the trust system by assigning each other as *Identical*. Let us also assume that Bob, being a bit naive, trusts Eve, assigning her as *Similar*. The updated trust graph for all the participants can be seen in Fig. 7.5.

The most trusted participants in the system, for each participant, are now:

- **Alice**: Bob, Carole, David
- **Bob**: Eve, Mallory
- **Carole**: Alice, Bob, David
- **David**: Alice, Carole
- **Eve**: Mallory
- **Mallory**: Eve

Thus, we can see that Bob's mistaken trust in Eve does not affect those who trust Bob, given that Bob had a relatively low trust to begin with.

Eve and Mallory start posting crude comments about the other participants in the chat. In addition to the tasteless comments, Mallory issues a network hide for Alice—hiding Alice for Eve and Bob, but no one else. As a result of Eve and Mallory's unwelcome comments, Carole issues a network hide for Eve, and Alice issues

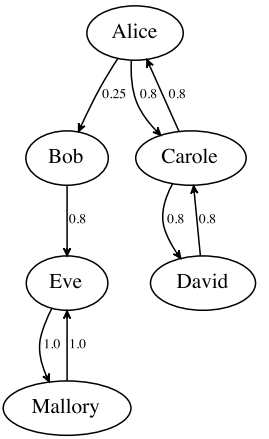

Figure 7.5: Trust graph for the hypothetical chat system after Bob, Eve and Mallory have issued their trust assignments.

a network hide for Mallory. The result is that Eve and Mallory are hidden for Alice, Carole, and David—despite only two moderation actions having been issued by two different people.

Once Eve and Mallory start disrupting the chat, and noticing that Alice has been hidden despite not having done anything, Bob quickly realizes his mistake of trusting Eve. He revokes his trust for Eve and hides both Eve and Mallory. Bob has to hide Eve and Mallory himself as Alice and Carole's moderation actions have no effect for him, since Bob has not assigned trust for either Alice, Carole or David.

The above example is somewhat of a toy example to show the fundamentals of how TrustNet works. In particular, what was demonstrated was how TrustNet could function as part of a chat system. In the next chapter, we will consider larger chat systems in order to analyze the efficacy of TrustNet.

8

Evaluation & Results

In this chapter, we evaluate and present the results from the proof-of-concept implementation of TrustNet. We also outline a baseline *naive* moderation system, which we measure TrustNet's performance against.

8.1 Evaluation

In order to properly evaluate and measure TrustNet's results, we need to have a baseline to measure against, which we will now define. In the baseline moderation system, to successfully hide a malicious participant for the entire network, each node needs to individually hide the malicious participant. We call this approach the *naive* approach.

Evaluating the efficacy of the proposed TrustNet system versus the naive approach was formulated accordingly. We will measure the amount of *blocks*—a moderation action which one peer issues to hide the content authored by another peer—as well as the *total* amount of actions. An *action* is defined as any manual moderation intervention. In our case, an action is regarded as either issuing a trust assignment, or issuing a block. Thus, what we will measure is:

1. How many *blocks* are required to hide a malicious participant for the entire network?
2. How many *actions* are required to hide a malicious participant for the entire network?

Our hypothesis is that the subjective moderation system implemented with TrustNet will significantly reduce the number of blocks and the total actions required as compared to the naive moderation system.

In order to carry out the evaluation, we had to devise some way of optimally deriving the number of blocks required for the subjective moderation system. This was not completely trivial, as the subjective moderation system essentially creates neighbourhood of nodes in which moderation actions are carried forth. This makes it difficult to find which nodes should issue a moderation action for optimal effect. To perform an optimal count of the number of blocks required, we devised the following method.

Influencer accounting First, a trust network is generated using the scenario simulator, described in Section 7.3. This creates a trust network where each node randomly selects other nodes to issue trust assignments. We want to briefly pause the explanation here to point out that this is a kind of worst-case network, as real trust networks will likely have more developed and cohesive structures as regards the trust assignments issued between nodes—i.e some participants will inherently be regarded as more trusted in a real-life chat contexts.

With the generated trust network, we want to issue as few blocks as possible to end up with an optimal number of blocks. This is done by taking each of the network's node and looking at their subjective trust graph, making note of which nodes they end up with as their trusted peers according to TrustNet. We then sort the nodes by the number of other nodes trusting them. We call the nodes with the highest number of other nodes trusting them *influencers*, because their actions influence greater portions of the network. We then iteratively take the top influencer

and issue a block, making note of which other nodes have now blocked the malicious participant as a result of the top influencer's action. This influencer has now exhausted their influence, so they are discarded. The list is re-sorted to find the next top-most influencer, as the number of nodes to influence has changed.

We continue this iterative method until the top-most influencer no longer has any nodes they can influence. The remaining un-influenced nodes are regarded as nodes who do not trust any others, and so they must each individually issue a block. All nodes in the network have now hidden the malicious participant, and we have ended up with an optimal number of blocking actions as regards the generated trust network.

Total number of actions The section above describes how to count the total amount of blocks for the subjective TrustNet-based moderation system. However, it would be misleading to only count the number of blocks required as the trust assignments need to be issued as well for the blocks of influencers to have any effect on the network. In order to account for the total number of actions required, we need to count the number of trust assignments issued alongside the number of blocks issued.

The mean number of trust assignments is calculated as in Equation 8.1, where *low* and *high* are the two endpoints describing the range of the number of trust assignments each node will issue, and N is the total number of nodes in the network.

$$\frac{low + high}{2} \cdot N \tag{8.1}$$

8.2 Results

The parameters in Listing A.1 were used to generate trust networks using the scenario simulator from Section 7.3.

Using the results from 1000 executions, where each execution had a unique seed parameter, a variance of 47 and a mean of 273 was calculated for the number of blocks TrustNet requires to completely hide 1 malicious participant. That is, on average, 273 blocks were required to completely hide a malicious participant for all 1000 nodes. In Fig. 8.3 and Fig. 8.4, a mean of 273 was used.

The number of trust assignments issued per node, also over 1000 executions, had a mean of 4 and a variance of 1. For the following graphs, the amount of trust assignments issued for the 1000 node network was set using the mean of 4.

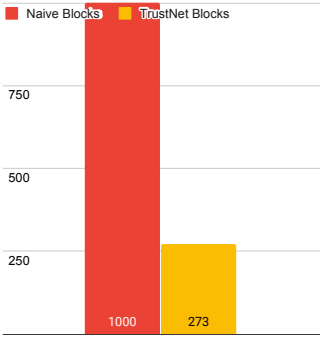

Figure 8.1: Blocks required to hide a single malicious participant for a network of 1000 nodes. Lower staple is better.

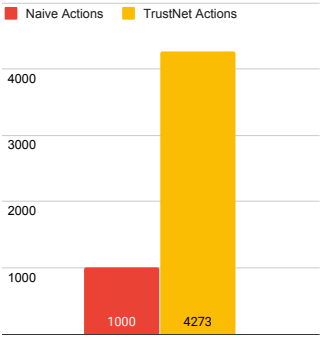

Figure 8.2: Actions (blocks + trust assignments) required to hide a single malicious participant for a network of 1000 nodes. Lower staple is better.

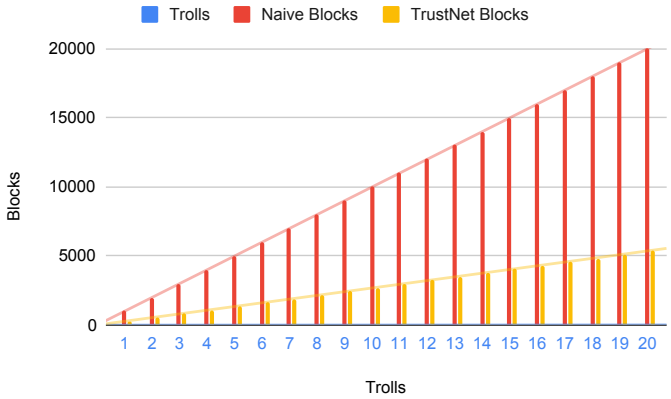

Figure 8.3: Total amount of blocks required to completely hide 1-20 malicious participants (titled *trolls* in the chart) for a chat context of 1000 nodes. Lower is better.

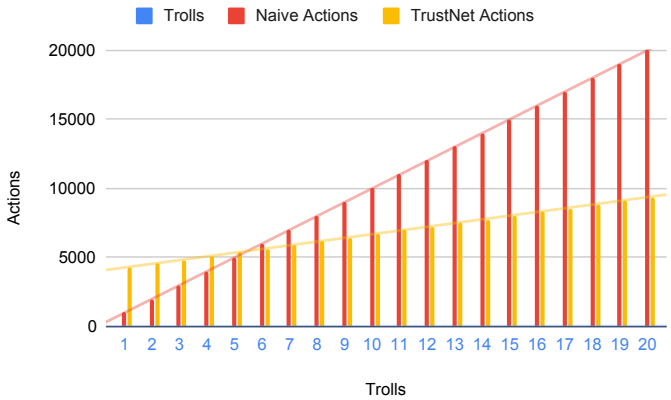

Figure 8.4: Total amount of actions (trust assignments + blocks) required to completely hide 1-20 malicious participants (titled *trolls* in the chart) for a chat context of 1000 nodes. Lower is better.

8.3 Moderation Comparison

The number of blocks required for the TrustNet-based moderation system, as compared to the naive moderation system, for a network of 1000 nodes, was 273 vs 1000 blocks. That is, the number of required blocks was *decreased* by 73%.

As regards the total number of actions to hide 1 malicious participant, we can see in Fig. 8.2 that the TrustNet approach greatly exceeds the number of actions required as compared to the naive approach. This makes sense, as the trust network

between nodes needs to be established. Viewing Fig. 8.4, we can however see that there is a point at which the TrustNet-based system starts to become more effective even when considering the total amount of actions. After 5 malicious participants have been hidden in the chat context, the amount of actions required are less for TrustNet than for the naive approach. After 20 malicious participants, the number of actions required for TrustNet is just under half, at 9460 actions, of the actions the naive approach requires, 20000.

As regards blocks, we can see that TrustNet always outperforms the naive approach, irrespective of the number of malicious participants. At 20 blocked malicious participants, the the number of blocks for TrustNet is 5460 for the entire network of 1000 nodes, while the naive approach remains at 20000 blocks to achieve the same result i.e. hiding all 20 malicious participants.

Thus, TrustNet has an initial cost in the form of the required trust assignments. For a shorter-lived chat context, i.e. one in which the likelihood of any malicious participant entering remains low, it may be less useful to make use of TrustNet—at least as regards a moderation context. If, however, the chat context will experience more than 5 malicious participants in its lifetime, then the initial cost will pay off in terms of the amount of actions. As noted above, TrustNet always pays off in terms of the required blocks, however.

Finally, we would like to finish with the notion that even if TrustNet, in terms of actions and in the shorter term, is less effective as regards the entire network, the neighbourhoods which utilize the trust assignments will have a greatly improved moderation experience than those which do not. If even 10% of a chat context uses the system to share the moderation burden amongst themselves, that, too, is viewed as a positive outcome as the participants of the neighbourhood will have to expend less effort to hide malicious participants. Thus, the system can be incrementally adapted without needing to force its use onto everyone in a chat context.

8.4 Varying the parameters

As it can be useful to see how TrusTnet performs with different types of models, we present below two meaningful variations of the parameters in Listing A.1.

Using a network size of 100

Varying the number of nodes saw similar results as those presented above. Using a network of 100 nodes, the result from 1000 executions yielded a variance of 2 and a mean of 29 for the number of blocks TrustNet requires to completely hide 1 malicious participant for a network of 100 nodes.

The effectiveness using a smaller network size of 100 nodes, instead of 1000, with regard to the number of blocks required as compared to the naive system, was also calculated to be 73%.

The results are presented in Fig. 8.5 and Fig. 8.6.

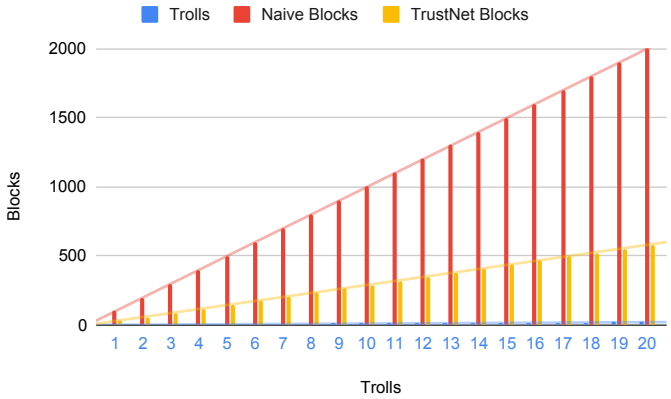

Figure 8.5: Total amount of blocks required to completely hide 1-20 malicious participants (titled *trolls* in the chart) for a chat context of 100 nodes. Lower is better.

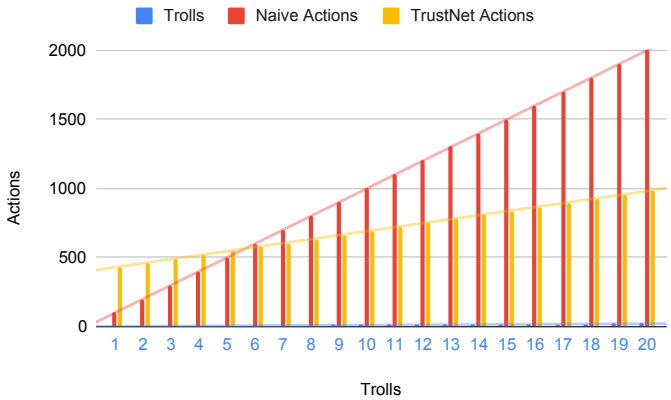

Figure 8.6: Total amount of actions (trust assignments + blocks) required to completely hide 1-20 malicious participants (titled *trolls* in the chart) for a chat context of 100 nodes. Lower is better.

Larger amount of trust assignments

Varying the number of trust assignments, extending the upper range, that is the parameter `high` of Listing A.1, from 5 to 15, while keeping all other parameters the same, yielded the following results.

The average number of trust assignments increased from 4 to 8 with a variance of 0. Using a network of 1000 nodes, the result from 1000 executions yielded a variance of 21 and a mean of 185 for the number of blocks TrustNet requires to

completely hide 1 malicious participant for a network of 1000 nodes. Thus, the effectiveness for the larger range, with regard to the number of blocks required as compared to the naive system, is 81%.

The total number of actions required to hide 20 malicious participants increased from 9460 to 11700, an increase of 24%.

9

Discussion

9.1 On Privacy

TrustNet builds on the transitivity of trust assignments. That if Alice trusts Bob, and Bob trusts Carole, then Alice trust Carole (to a certain degree).

The underlying foundation of transitivity may become problematic, as it exposes delicate relational bounds between the participants of a chat context. The underlying trust graph makes it possible for any participant to easily map out who trusts whom, how much, and in which areas. In one sense, this fact could be potentially beneficial as it allows a new person to orient themselves in the new context—making the perceived issue an unexpected benefit, if the chat is fairly private or secluded.

For a public chat context, it is possible that a system such as TrustNet could further entrench the surveillance capitalist [Zuboff, 2019] status quo—where user data such as predicted personality[Chen et al., 2016], emotional state [Zuboff, 2019], or current physical location, is sold across massive automated ad networks to the highest bidder [Zuboff, 2019]—underlying many of the highly populated services and platforms on today's internet.

The privacy issue is difficult to mitigate technically, and out of the scope of this already extensive work. Perhaps privacy-preserving techniques from other research areas can mitigate this problem—such as techniques inspired by *homomorphic encryption* [Hayes, 2012], which allows participants to operate on data they cannot read.

9.2 On The Difficulty of Simulating Trust

The results of Chapter 8 are based on simulations of randomly-generated trust networks. There are many potential ways in which the model is lacking in its representation of the dynamics of real life people in a chat context.

As such, we believe that TrustNet will perform even better on a real system, as there will exist natural high trust clusters. The clusters, we posit, will be based on factors such as existing friendship cliques, as well as globally highly trusted

participants of the chat system; its earliest members, or participants with repeated instances of keen judgement. We see this pattern with existing *social media platforms* [Facebook, 2020] [Mastodon, 2020], where it is not uncommon for a small number of accounts to have a lot of so called *followers*. Thus, it is not an unreasonable assumption to believe that a similar factor may be at play in subjective systems such as the proposed TrustNet system.

9.3 On Increased Attack Incentives

The advent of Bitcoin [Nakamoto, 2008], a peer-to-peer electronic currency, and other digital currencies, saw the unexpected rise of malicious attacks on previously unattractive targets. Popular code repositories have seen sophisticated infiltration attempts with the intent to surreptitiously install malicious software for stealing digital currencies from unsuspecting targets [Tarr, 2018].

It is not inconceivable, then, that should TrustNet, or another system like it, become popular that it too could risk increasing the incentive for malicious attackers. Attackers may, for instance, try to find ways to attack trust networks in order to manipulate a subset of it for their gain.

9.4 On the Importance of Naming

It would be beneficial for the integration of TrustNet with user interfaces if an alternate phrasing for *trust assignments* could be found. A phrasing which emphasizes that a trust source is not handing out *trust*, but instead identifying people as similar to how they themselves would act in a given trust area. To frame trust in the manner of a *similarity in judgement*, instead of a decree on a person's worth.

Following that strand of thought, it makes sense to assign identities that you control—your desktop identity and your phone identity—with a weight of 1.0 i.e. absolute trust. Because *of course* you have a perfect similarity in judgement to yourself.

The proposed alternate phrasing, by swapping out notions of trust, also prevents social problems from taking root which could otherwise negatively impact social relations but also the use of TrustNet at large. That is, people might be offended when they are assigned a low weight if it is interpreted as *trust*:

> What?! You don't trust me? I thought we were friends...

On the contrary, it is obvious that two friends differ in *judgement*. This becomes exceptionally clear when the trust area is taken into account:

> You gave me a lower weight... Oh, it's in *music recommendations*, yeah that makes sense.

9.5 Other Use Cases of TrustNet

Let us assume you have a network of trusted peers (with varying levels of trust) and running TrustNet on the network gives you an actionable list of trusted peers.

We can use this curated list of trusted peers in multiple ways. This thesis has focused on using it for implementing a moderation system, essentially granting the trusted peers moderator powers to act on your behalf. This effectively creates a dynamic blocklist, with many sources. The peer moderators and their changes can also be revoked if abused (you stop trusting them and unassign the previously assigned trust).

Another use case could regard the trusted peer list as a set of *trusted sources* from which to request pre-computed indexes in a distributed network; speeding up onboarding into the network through bypassing the requirement to index each and every post before the network can become usable.

Yet another use case could be to use the rankings in social media applications, as a suggestion on whom to follow, become friends with, or simply engage more in conversation.

Instead of using the trust network to figure out which moderation actions are reliable, the trust network itself could be used to filter the chat; limiting chat messages to only people from within the trusted network. This kind of mechanism could be implemented as a user-controllable toggle, such that if a participant is feeling overwhelmed they can switch on the toggle to default to a smaller, trusted subset of the chat.

The computed rankings could also be used to inform recommendations within a particular kind of domain, for example the domain of *music taste*. Issued trust weights could then be interpreted as an indication of music taste, higher weights corresponding to a greater alignment in music taste, or a subjectively better taste in music.

These are just a few examples of how the underlying idea proposed in this work could be used outside of the domain of subjective moderation.

9.6 Conclusion: Subjective Moderation & The Future of TrustNet

We have tried to provide an insight into the causes of moderation, as well as the benefits of a subjective, trust-based moderation system. We have proposed TrustNet as one solution that could be used to implement a subjective moderation system.

During the course of working on TrustNet and writing this work, Cabal, the peer-to-peer chat project featured in this thesis, received a grant from Mozilla—one of the current major internet browser developers. The grant was received as part of a successful grant application, written by the author, proposing the implementation of a subjective moderation system for Cabal [Cobleigh, 2020a].

Going forward, the author's current plan is to include TrustNet into Cabal to allow for a variety of trust-based functionality. Among these are automatic downloading of shared documents, enabling custom profile images and only displaying them for the trusted subset of the chat context, as well as making it possible to toggle hiding of private messages sent from untrusted people, and more.

The TrustNet implementation, and the javascript implementation of Appleseed, are planned to be released as open source modules on the author's Github account [Cobleigh, 2020b], after the publication of this thesis.

A

Simulator parameters

Below are the parameters used for running the sceanario simulator, described in Section 7.3.

Listing A.1: Example of parameters for TrustNet's scenario generator.

```
{
"seed": 1,
"nodes": 1000,
"trust": {
    "levels": {
        "none": 0,
        "low": 0.25,
        "medium": 0.50,
        "high": 0.75,
        "absolute": 1.00
    },
    "skew": {
        "none": 0.05,
        "low": 0.35,
        "medium": 0.10,
        "high": 0.49,
        "absolute": 0.01
    },
    "range": {
        "low": 3,
        "high": 5
    }
}
```

Bibliography

Abdul-Rahman, A. and S. Hailes (1998). "A distributed trust model". In: *Proceedings of the 1997 workshop on New security paradigms*, pp. 48–60.

Abdul-Rahman, A. and S. Hailes (2000). "Supporting trust in virtual communities". In: *Proceedings of the 33rd annual Hawaii international conference on system sciences*. IEEE, 9–pp.

Anderson, J. R. et al. (1983). "A spreading activation theory of memory". *Journal of verbal learning and verbal behavior* **22**:3, pp. 261–295.

Aumasson, J.-P., S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein (2013). "Blake2: simpler, smaller, fast as md5". In: *International Conference on Applied Cryptography and Network Security*. Springer, pp. 119–135.

Bernstein, D. J., N. Duif, T. Lange, P. Schwabe, and B.-Y. e. a. Yang (2011). *High-speed high-security signatures*. Electronic Paper. Accessed 2019-04-29. URL: `https://ed25519.cr.yp.to/ed25519-20110926.pdf`.

Bhuiyan, T., A. Josang, and Y. Xu (2010). "Trust and reputation management in web-based social network". *Web Intelligence and Intelligent Agents*, pp. 207–232.

Bizer, C., T. Heath, and T. Berners-Lee (2011). "Linked data: the story so far". In: *Semantic services, interoperability and web applications: emerging concepts*. IGI Global, pp. 205–227.

Brewer, E. A. (2000). "Towards robust distributed systems". In: *PODC*. Vol. 7. Portland, OR.

Brumm, B. (2018). *Sql views and materialized views: the complete guide*. Blog Article. Accessed 2020-04-09. URL: `https://www.databasestar.com/sql-views/`.

Cabal-Club (2020). *Cabal*. Website. Accessed 2020-05-08. URL: `https://web.archive.org/web/20200501104038/https://cabal.chat/`.

Ceglowski, M., A. Coburn, and J. Cuadrado (2003). "Semantic search of unstructured data using contextual network graphs". *National Institute for Technology and Liberal Education* **10**.

Chen, T.-Y., M.-C. Tsai, and Y.-M. Chen (2016). "A user's personality prediction approach by mining network interaction behaviors on facebook". *Online Information Review*.

Cobleigh, A. (2018). *Cabal*. Accessed 2020-05-06. URL: https://web.archive.org/web/20200506130433/https://github.com/new-computers/cabal/issues/1.

Cobleigh, A. (2020a). *Cabal x subjective moderation, mozilla grant*. Website. Accessed 2020-05-20. URL: https://opencollective.com/cabal-club/updates/cabal-x-subjective-moderation-mozilla-grant.

Cobleigh, A. (2020b). *Github*. Website. Accessed 2020-05-20. URL: https://github.com/cblgh.

Collins, A. M. and E. F. Loftus (1975). "A spreading-activation theory of semantic processing." *Psychological review* **82**:6, p. 407.

Diffie, W. and M. Hellman (1976). *New Directions in Cryptography*. IEEE.

Douceur, J. R. (2002). "The sybil attack". In: *International workshop on peer-to-peer systems*. Springer, pp. 251–260.

Facebook (2020). *Groups*. Accessed 2020-04-29. Facebook, Inc. URL: https://web.archive.org/web/20200422191447/https://www.facebook.com/help/1629740080681586?helpref=hc_global_nav.

Felps, W., T. R. Mitchell, and E. Byington (2006). "How, when, and why bad apples spoil the barrel: negative group members and dysfunctional groups". *Research in organizational behavior* **27**, pp. 175–222.

Fidge, C. J. (1987). "Timestamps in message-passing systems that preserve the partial ordering".

Fowler, M. (2005). *Event sourcing*. Blog Article. Accessed 2020-04-08. URL: https://martinfowler.com/eaaDev/EventSourcing.html.

Gilbert, S. and N. Lynch (2002). "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". *SIGACT News* **33**:2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

Guha, R., R. Kumar, P. Raghavan, and A. Tomkins (2004). "Propagation of trust and distrust". In: *Proceedings of the 13th international conference on World Wide Web*, pp. 403–412.

Hayes, B. (2012). "Alice and bob in cipherspace". *American Scientist* **100**:5, pp. 362–367.

Jøsang, A. (2007). "Trust and reputation systems". In: *Foundations of security analysis and design IV*. Springer, pp. 209–245.

Jøsang, A., E. Gray, and M. Kinateder (2003). "Analysing topologies of transitive trust". In: *Proceedings of the First International Workshop on Formal Aspects in Security & Trust (FAST2003)*. Pisa, Italy, pp. 9–22.

Jøsang, A., S. Marsh, and S. Pope (2006). "Exploring different types of trust propagation". In: *International Conference on Trust Management*. Springer, pp. 179–192.

Josefsson, S. (2006). *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. DOI: 10.17487/RFC4648. URL: https://rfc-editor.org/rfc/rfc4648.txt.

Kamvar, S. D., M. T. Schlosser, and H. Garcia-Molina (2003). "The eigentrust algorithm for reputation management in p2p networks". In: *Proceedings of the 12th international conference on World Wide Web*, pp. 640–651.

Keall, D. (2019). *How dat works*. Digital. Accessed 2020-01-15. URL: https://datprotocol.github.io/how-dat-works/.

Kempe, S. (2012). *The nosql movement — what is it?* Blog. Accessed 2020-04-20. URL: https://www.dataversity.net/the-nosql-movement-what-is-it/.

Kleppmann, M. (2015). "A critique of the cap theorem". *arXiv preprint arXiv:1509.05393*.

Kleppmann, M. and J. Kreps (2015). "Kafka, samza and the unix philosophy of distributed data".

Kleppmann, M., A. Wiggins, P. van Hardenberg, and M. McGranaghan (2019). "Local-first software: you own your data, in spite of the cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 154–178.

Kreps, J. (2013). *The log: what every software engineer should know about real-time data's unifying abstraction*. Blog Article. Accessed 2019-11-17. URL: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.

Lamport, L. (1978). "Time, clocks, and the ordering of events in a distributed system". *Commun. ACM* **21**:7, pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

Lamport, L. (1998). *The part-time parliament*. URL: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf.

Levien, R. (2003). "Advogato trust metric [ph. d. dissertation]". *UC Berkeley, USA*.

Mastodon (2020). *Join mastodon*. Accessed 2020-04-29. The Mastodon Project. URL: https://web.archive.org/web/20200425022507/https://joinmastodon.org/.

Maymounkov, P. and D. Mazieres (2002). "Kademlia: a peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer, pp. 53–65.

Merkle, R. C. (1987). "A digital signature based on a conventional encryption function". In: *Pomerance C. (eds) Advances in Cryptology — CRYPTO '87*. Springer.

Milgram, S. (1967). "The small world problem". *Psychology today* **2**:1, pp. 60–67.

Mosberger, D. (1993). "Memory consistency models". *ACM SIGOPS Operating Systems Review* **27**:1, pp. 18–26.

Nakamoto, S. (2008). *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep.

Nakamoto, S. (2009). *Base58.h*. Digital. Accessed 2020-02-20. URL: https://github.com/bitcoin/bitcoin/blob/aaaaad6ac95b402fe18d019d67897ced6b316ee0/src/base58.h.

Nakamoto, S. (2019). *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot.

NIST (2001). *Descriptions of sha-256, sha-384, and sha-512*. Digital. Accessed 2020-01-15. URL: https://web.archive.org/web/20130526224224/http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf.

Oikarinen, J. and D. Reed (1993). *Internet relay chat protocol*. Accessed 2020-04-29. URL: https://tools.ietf.org/html/rfc1459.

Page, L., S. Brin, R. Motwani, and T. Winograd (1999). *The pagerank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab.

Pathirage, M. (2014). *Kappa-architecture.com*. Website. Accessed 2020-04-27. URL: https://milinda.pathirage.org/kappa-architecture.com/.

Schlosser, M. (2019). *Developing the eigentrust algorithm and determining authenticity online*. Digital. Accessed 2020-02-10. URL: https://medium.com/oscar-tech/developing-the-eigentrust-algorithm-and-determining-trustworthiness-online-6c51b2c2938f.

Slack (2020). *What is slack?* Accessed 2020-04-29. Slack Technologies Inc. URL: https://web.archive.org/web/20200429092816/https://slack.com/intl/en-se/resources/slack-101/lesson-1-what-is-slack.

SSBC (2017). *Scuttlebutt protocol guide*. Accessed 2020-05-04. URL: https://ssbc.github.io/scuttlebutt-protocol-guide/.

Stiegler, M. (2005). "An introduction to petname systems". In: *In Advances in Financial Cryptography Volume 2. Ian Grigg*. Citeseer.

Tarr, D. (2015). *Designing a secret handshake: authenticated key exchange as a capability system*. Accessed 2020-05-05. URL: https://dominictarr.github.io/secret-handshake-paper/shs.pdf.

Tarr, D. (2018). *Statement on event-stream compromise*. Website. Accessed 2020-05-18. URL: https://web.archive.org/web/20200418033803/https://gist.github.com/dominictarr/9fd9c1024c94592bc7268d36b8d83b3a.

Tarr, D., E. Lavoie, A. Meyer, and C. Tschudin (2019). "Secure scuttlebutt: an identity-centric protocol for subjective and decentralized applications". In: *Proceedings of the 6th ACM Conference on Information-Centric Networking*. ACM, pp. 1–11.

Thaler, D. and B. Aboba (2008). "What makes for a successful protocol". *Internet Eng. Task Force, Fremont, CA, USA, RFC* **5218**.

Tschudin, C. (2019). "A broadcast-only communication model based on replicated append-only logs". *ACM SIGCOMM Computer Communication Review* **49**:2, pp. 37–43.

Tschudin, C. F. (2018). *The tangle data structure and its use in ssb drive*. Blog Article. Accessed 2020-04-13. URL: `https://github.com/cn-uofbasel/ssbdrv/blob/master/doc/tangle.md`.

Vogels, W. (2009). "Eventually consistent". *Communications of the ACM* **52**:1, pp. 40–44.

Walport, M. et al. (2016). "Distributed ledger technology: beyond blockchain". *UK Government Office for Science* **1**.

Wang, H. and M. Song (2011). "Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming". *The R journal* **3**:2, p. 29.

Ziegler, C.-N. and G. Lausen (2004a). "Spreading activation models for trust propagation". In: *IEEE International Conference on e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004*. IEEE, pp. 83–97.

Ziegler, C.-N. and G. Lausen (2004b). "Analyzing correlation between trust and user similarity in online communities". In: *International Conference on Trust Management*. Springer, pp. 251–265.

Ziegler, C.-N. and G. Lausen (2005). "Propagation models for trust and distrust in social networks". *Information Systems Frontiers* **7**:4-5, pp. 337–358.

Zuboff, S. (2019). *The age of surveillance capitalism: The fight for a human future at the new frontier of power*. Profile Books.