Secure Scuttlebutt: An Information-Centric Protocol for Subjective and Decentralized Community Applications

(May 14, 2019 / last commit was 90be21fd94db39bd8704b99fc41a3595b1ce2fdb)

Dominic Tarr
ssb:@EMovhfIrFk4NihAKnRNhrf
RaqIhBv1Wj8pTxJNgvCCY=.ed25519

Erick Lavoie
McGill University, Montreal, Canada
erick.lavoie@mail.mcgill.ca

Aljoscha Meyer
TU Berlin, Germany
aljoscha.t.meyer@campus.tu-berlin.de

Christian Tschudin
University of Basel, Switzerland
christian.tschudin@unibas.ch

ABSTRACT

Secure Scuttlebutt (SSB) is a novel peer-to-peer information-centric event-sharing protocol and architecture for social apps. In this paper we describe SSB's features, its operations as well as the rationale behind the design. We also provide a comparison with traditional information-centric networking protocols and discuss SSB's limitations and evolution opportunities.

At the transport level, SSB is a replication protocol for append-only logs. applications communicate indirectly by writing to the local node's log and by reading from the locally available replicated logs. From these, each app instance constructs its own intepretation of the shared app state (an approach called "subjective reader"). Scaling is achieved through in-network caching of the immutable log updates and by routing the content along the social graph.

SSB's current set of applications include classical social media apps (chat, with end-to-end encryption and meta-data privacy), game and lifestyle apps (chess, book reviews) as well as technical applications like a p2p git, a shared file system, or a social backup app for crypto keys using a secret sharing protocol.

1 INTRODUCTION

A simple conceptual architecture for community applications consists of a global data pool to which every person can contribute and where every person can tap into the shared data – data sharing being the purpose of such applications. This model still is valid if one adds access control to the picture, either tied to the data (encryption giving access to content only to entitled holders of the decryption keys) or encrypting data in transit (login and TLS). Facebook and other centrally organized social app service providers fit well under this global data pool model but have been strongly criticized for abusing their central provisioning position. The "decentralized web movement" [14] is the most visible technical response to this critique, pointing out implementation alternatives.

One of these alternatives is a project called Secure Scuttlebutt (SSB) that started in 2012. After several iterations of protocol design and implementation, SSB has become a stable service for over 10,000 users offering them rich media community applications with strong cryptographic protection (end-to-end encryption and metadata privacy) and running in pure peer-to-peer mode.

Selective Complete Log Replication

SSB's spin on the above conceptual model is that all participants *replicate the global data pool*, which enables off-line operations, avoids redundant data transfers and has become feasible – at least in principle – because storage nowadays is a cheap resource, just that the sheer volume of social app data prohibits a full replication. However, because a participant is mostly interested in content from its peers, which is a very small number compared to all participants, only portions of the global data pool need be replicated. This observation is leveraged in SSB's transport layer which is tasked to *selectively replicate* the global data pool *along the edges of the social graph*, as we will explain in Section **??**.
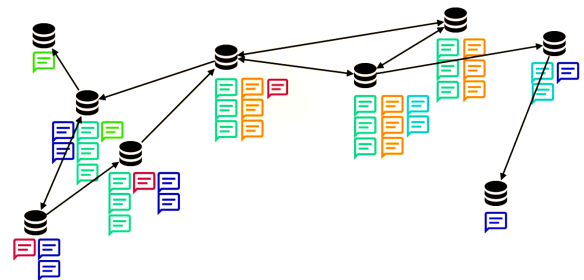

Figure 1: Staltz' "Internet of People" figure, needs to be redrawn such that it can be read in B/W, too.

A second spin of SSB is that replication is done *pro-actively* and at the granularity of the *complete input by a single peer* to the global data pool. This novel way of implementing the global data pool model is in stark contrast to current solutions where

central server repositories hosting this pool are accessed by client software in an on-demand fashion, assuming a storage-less end-device. Today's data dissemination mode is reactive (client-server protocol to fetch data) and piece meal (only the instantly needed data items are requested, possibly multiple times, and bundled with fresh ads, as the client does not necessarily cache all requested data). In SSB however, the central objects are the *complete* collection of a specific peer's input to the pool, organized as an append-only log. In Sections **??** and **??** we will explain the advantages of this approach (trust, off-line operations and efficient real-time notifications), and discuss its drawbacks (index building, lack of delete operation, storage bloat, re-keying).

Subjective Reader

Because replication in SSB is selective and is driven by a peer's social graph, different end devices will have access to different sets of log replicas, leading to different views of the world which is also called a "subjective reader" approach. SSB's viewpoint is that this is not a deficit but a desirable property: Each peer is free to consider data sources of its own choosing instead of having to feed from a centrally provisioned or otherwise converged view. While it is possible to implement consensus protocols over SSB, or to designate central data aggregators from which many peers consume the consolidated outputs, the SSB network itself deliberately doesn't offer consensus services nor central content (directories etc). In Section **??** we will show the implication of this technological choice on the structure of distributed applications that can only read from and write to local logs.

Novelty

Putting complete replication of individual append-only logs at the core of SSB's protocol avoids several hard problems in distributed systems. **First**, it is a radically decentralized approach requiring only minimal specification-level coordination among the participants but no run-time checks or configuration management. Typically, active/active multi-master replication in databases, which comes closest to SSB's approach, requires tight configuration control while in SSB, existing peers can go offline and new peers can come and go at any time, at the price of weak consistency guarantees, though. **Second**, although append-only data structures are well known for their benefits and are at the core of crypto currencies' consensus finding, SSB uses logs without any consensus properties. Quite on the contrary, SSB completely sidesteps consensus but provides the hooks for implementing service abstractions like tangles for implementing CRDTs on top of SSB i.e., eventual consistency. **Third**, crafting a cryptographic ID system and maintaining a social graph that informs routing creates a very narrow filter: it implements a receiver-driven approach (similar to e.g. Named Data Network, but at a higher object granularity) where data only flows where it is needed and without imposing any fine-grained request/reply protocol. Instead, any dissemination technology is adequate, including broadcast and push mode, because network elements can verify data validity (due to the logs' signed entries) and monotonicity of the updates without additional key and certificate material, guaranteeing that only conformant traffic is propagated at any forwarding step. **Fourth**, it makes every peer a publisher by

design. This property goes beyond the decentralized approach like DAT [] or IPFS [] which assume that there exist replication servers but keep the separation between a data transport network and a server layer. In SSB, bidirectional communications is only possible if both parties *are* repositories. **Last but not least**, log replication leads to a distributed system with inherent high resilience as any communicating element MUST replicate the others' logs. In traditional distributed systems, coordinating the data persistence as a basis for resilience is often an add-on task, or requires at least a special recovery service.

Scuttlebutt = Information-Centric Gossip

… why this name … it applies to the app-level, and can apply to the network layer, but doesn't have to: any dissemination method works for gossip. … some words about the SSB community, available documentation, past evolution, subjective roadmap [5, 10, 26–28]

Structure of this paper

…

2 SSB ARCHITECTURE AND PROTOCOL

In SSB, each user is identified by an ed25519 [6] keypair. Since anybody can generate a random keypair with very low probability of multiple peers generating the same keypair, no central authority is necessary for introducing users to the system.

The single-writer append-only logs of SSB consist of *messages* that include a *backlink*: The cryptographic hash of the previous message (or a special indicator for the first message of a log). The most distinguishing feature from a regular blogchain is that each user maintains their own log and cryptographically signs all their messages. Messages whose backlink points to a message in a different log (i.e. from a different author) are considered invalid.

These constraints still allow creation of arbitrary trees rather than logs. To enforce log structure, each SSB server checks that every message has exactly zero or one incoming backlinks. If it has more than one, the feed is considered *forked*. All messages from the point of the fork onwards are ignored, the log can not be appended to anymore.

Concretely, each message contains the following pieces of data:

- the *backlink* to the previous message, or a null value
- the public key of the message's *author*
- the *sequence number* of the message, which must be one more than the sequence number of the previous message, or exactly one if it is the first message of the log
- a claimed *timestamp* of when the message was created
- a *hash* indicator that specifies the concrete hash function that was used for the backlink
- the *content* of the message
- the author's *signature* over all the previous data

The *content* is a json object that must contain a message *type* string that serves as a hint for how the content should be interpreted. SSB enforces that the content is valid json. Alternatively, the content can be a byte string of encrypted data, together with a tag that signifies which encryption algorithm was used.

SSB defines a format for encoding the public keys of identities and the hashes of messages and blobs (see below) as strings. This

allows applications to scan the content of messages for such references, e.g. in order to create database indices.

The precise, byte-for-byte definition of the json-based message encoding is given in [21].

The principal function of SSB servers is to connect to other servers and exchange log updates. To do so, they maintain an end-to-end encrypted overlay network over which they run a gossip protocol. When two servers start gossiping, they exchange the current sequence numbers of all logs they are interested in. If a peer receives a lower sequence number for a feed than it sent, it transmits the log messages that the peer is lacking. As an optimization, this *eager* gossip is only performed over the edges of a spanning tree, maintained via the plumtree [18] protocol.

In addition to this primary replication mechanism, SSB provides two other ways of exchanging information. *Blobs* are content-addressed pieces of data that are not part of any log. They are not widely disseminated automatically, but rather fetched on demand via a simple request-flooding protocol. *Out-of-order messages* are a similar mechanism to address and fetch log entries on demand via their hashes.

Beyond replicating logs and checking their validity, an SSB server offers an API to *SSB clients*. These can be arbitrary programs that issue RPCs to the server over an IPC mechanism. The exposed functionality includes writing to the author's log, reading from arbitrary logs, specifying which logs a server should replicate, and fetching blobs and out-of-order messages. SSB thus becomes a platform for building applications, encapsulating the complexity of data replication.

The reference implementation of SSB also includes a mechanism for loading *plugins* into the server to extend its functionality. There are a few default plugins, these can be thought of as client programs that are always running. Of particular importance are those that guide the replication process. The *friends* plugin scans the server's log for specific messages that indicate which other authors the identity *follows*. The plugin then instructs the server to fetch and replicate these logs. These other logs might of course also contain some of these messages. The friends plugin transitively replicates these friends-of-a-friend logs as well, up to a configurable maximum distance in the friends graph. Transitive replication can always be overridden via special *block* messages.

This example showcases a crucial property of SSB: Decisions about whom to replicate can be guided by the content of the very data that is replicated. By storing the relevant information inside the author's log (as opposed to a local database), other peers can also use this informtion to guide their decisions.

Information stored in the log is also used to solve the problem of joining the overlay network: Authors can publish the static ip addresses of highly-available servers (called *pubs*) to their log. When a server needs to connect to the overlay, the responsible plugin can scan any available log for this information.

It is worth noticing that this architecture spans four independent layers of protocols. The most fundamental protocol is the message format. All peers need to agree on what consitutes identities, valid messages, and how to compute hashes to address messages and blobs. This is the "thin waist" of SSB (see figure 2). Next is the specific mechanism by which servers exchange data. The default

mechanism is one option, but alternative mechanisms such as distribution via a sneakernet could also be deployed. Different peers that do not share a common replication mechanism could still interact indirectly, as long as there are some servers that understand multiple replication protocols. The protocol by which a server serves its clients is independent from the previous procotols. And finally, the way that different clients might publish and interpret messages is again a separate affair.
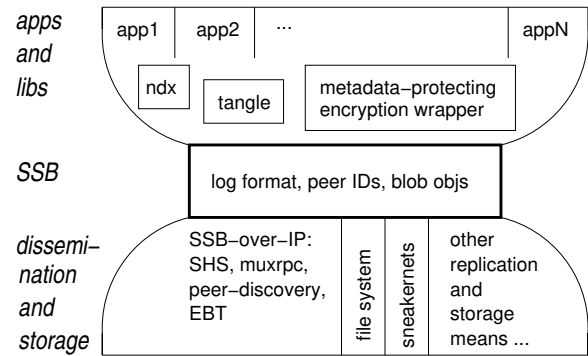

Figure 2: Secure Scuttlebutt's protocol stack.

3 DISTRIBUTED APPS AND DATA STRUCTURES OVER SSB

Wordsmithing needed: The goal of this section: show how SSB's distributed apps can reconstruct the app's state by parsing the logs, and operate on the app state by writing to the peer's log. We give more details for SSB's user directory and also list and quickly characterize other existing SSB apps. A crucial (performance) aspect is incremental indexing which we expand on in the subsection about the Kappa approach. More node-local support in form of libraries and conventions at the level of log entries is discussed in the subsection on tangles.

3.1 Example: SSB's user directory

'about' is SSB's user database i.e., an application that associates cryptographic IDs with (typically) human-readable attributes. A single log entry format has been defined to this end:

```
'content': {
  'type'  : 'about',
  'about' : target_id,
  attr_name : attr_value   // multiple times
}
```

The about app scans all logs for all entries of type 'about' and constructs a database as shown in Figure 3, retaining the most recent attribute assignment found. To each target user ID we associate a directory where key/value pairs are collected on a per author basis (which is extracted from the log entry's envelope).

Currently the name, description and image attributes are understood by most SSB user interfaces and are used to substitute or decorate the cryptographic ID. If target_id and author_id are identical, the attributes are self-chosen, and otherwise given.

In terms of CRUD actions, creation happens once a new SSB peer adds its own about entry to its log; reading the user database is

target_id	author_id	key	val
		...	
		...	
	...		

Figure 3: SSB's user directory data structure (after extraction from the logs).

performed on the above data structure; updates are expressed by adding an about entry –regardless whether it relates to the peer itself or to another peer– to one's own log and all peers updating their extracted database; deleting a user entry is not possible, at least not directly (one would have to block that user ID as well as all IDs which wrote an update for that user).

There can never be confusion about the sequence or scope of attribute assignments because they are orderd by the log (and thus in time) and kept separate, per author ID. Note also the presence of the "subjective reader" property: The content of a peer's user database is dependent on its position in the social graph. The "subjective" mindset is also visible by letting every user assign attributes to anybody, leaving it to the user interface (and human viewer) to select which of the self-chosen or given display names and images is most suitable for a given ID.

3.2 Profiles of other selected SSB apps

Multiple applications have been written by contributors and are used daily by the SSB community. We briefly present selected examples because they represent alternatives to well-known services and they illustrate both opportunities and challenges of communication through replicated append-only logs.

Git-ssb [17] is an alternative to GitHub [3] that replicates git-based version-controlled code repositories through contributors logs. It provides an encoding of repositories in SSB logs, a bridge to interoperate with git repositories, and a web-based viewer to browse repositories. The object model of Git [9], based on immutable hash-referenced objects organized in a chain of commits, has a similar structure to SSB's logs, making the mapping natural. Hash objects are blobs and commits are messages in individual append-only logs. Other git operations, such as creating a repository, creating merge commits, creating a branch, or requesting the merging of an alternative branch (*pull-request*) to a core maintainer, are all SSB messages. This model provides automatic distribution of code repositories and their updates through the replication of SSB logs. Many developers can also update and perform operations on the *same* repository, as defined by its creation message, independently. Consensus on the "official" master branch and its latest commit is enforced through social coordination because the developers of the community know and trust each other. Nonetheless, in case of concurrent updates to the same branch in the same repository [23], the git bridge will create multiple branches when a local git repository is updated. A user can then resolve the fork by merging the diverging branches and updating the repository. Referencing both concurrent updates in a later merge commit in effect resolves the ambiguity through a tangle extension (Section 3.4).

Ssb-chess [20] is a correspondence chess application in which players can invite one another to play, alternatively share their next move until the game ends, and external observers can comment on the game. The core data structure that represent a game is a linked list with nodes representing chess moves alternating between the two participants' logs. The detection of invalid moves is performed by the user interface using a validation library, because participants are trusted to only publish valid moves on their feed. This latter assumption was made because games are played between friends in a non-competitive setting. In effect, the validity of the latest move is implicitly confirmed by the next player choosing to follow with another move because if they were to disagree on the validity, the conflict can be handled outside the game and the game can be abandoned. Moreover, a chess application is easy to encode in append-only logs because the rules of chess preclude concurrency, i.e. at any time there is always only one of the two participants that is permitted to modify the state of the chess board by making a move. The game state also cannot be corrupted by external parties because only the participants, explicitly mentioned in the original invitation, are allowed to modify the state of the game. Any non-participant extending the game with their own move is simply ignored.

Gatherings [12] are alternatives to Meetup [1] that enables participants to signal their intention to attend or not attend to physical events. Gatherings can be public, in which case anyone that replicates a log will see them, or private, in which case only explicitly-invited people will be notified of the event and may signal their intention. A gathering is defined by its creation message but otherwise has no fixed properties. Anyone that has a reference to the creation message may change its properties, such as location, start and end dates, description, and image, by publishing an update message. The value of those properties are the most recent set by anyone. Initially, recency was determined by the time of creation, as reported by the user's client implementation (*self-stated creation time*). While this required trust in other users, practice has shown that the hypothesis was reasonable. To be more robust to potential invalid timestamps however, some client implementations have started using the time at which message updates are *received*, then disambiguate using the self-stated creation time.

Scuttle-poll [11] is an implementation of the polling model of Loomio [4], an online group decision-making platform. In addition, it serves as the basis for *Scry* [15], an alternative to Doodle [2], itself an online calendar tool for organizing meeting. A poll is a request for opinions, which for example may express preference for one choice among many possibilities or provide a list of time availabilities. A poll is created by publishing a poll message in a log with a number of possible options and a deadline. Participants then publish their *position* among the choices available. The poll creator finally publishes a *resolution* based on the other participants' positions. Concurrency in polls is limited because the creation and resolution of the poll are done by the same user, and therefore works quite well with the SSB communication model.

These applications show how the SSB communication model greatly simplifies the infrastructure required to build social applications since none of the previous examples have to deal with issues of distribution of messages. The fact that a small community with a dozen or so of core developers, which are self-funded and

working mostly voluntarily, could produce alternative applications that work well enough to be used daily suggests the SSB communication model does make the implementation of common social applications simpler.

3.3 Kappa architecture

(Heavy) indexing effort, necessary for slicing logs on a per-app basis and computing the state of data structures by "summing up" updates through all logs (see for example the above "about" app).

Overview: https://milinda.pathirage.org/kappa-architecture.com/ Read above related work and provide a discussion of similarities and differences with SSB

3.4 Tangles, consensus, synchronization

Tangles intro: https://github.com/cn-uofbasel/ssbdrv/blob/master/doc/tangle.md

Eventual consensus?

For all application examples presented in Section 3.2, the design decisions in most cases assumed a really high-level of shared trust between users/developers, which simplified the implementation and enabled a faster bootstrap of useful applications with limited development resources, as most of the work has been done voluntarily or self-funded so far. The community fully understands that causal ordering and well-defined data structures based on Tangles (Section 3.4) would make the implementations more robust in contexts with lower levels of trust and is currently in the process of adopting them to make the applications more robust.

4 COMPARING SSB WITH NAMED DATA NETWORKING (NDN)

4.1 Naming (restriction)

4.2 Publisher by default

4.3 Social graph induced routing

4.4 Push vs Pull, subscription granularity

4.5 Architectural Incompatibility with NDN

– NDN cannot mimic the social-graph-informed routing?
– emulate SSB as an inter-repo protocol?
– flow-balance approach too narrow: replication would need notification, or polling?
– can we use NDN's CA trust model on a by-peer basis, mimic SSB's IDs?

5 FUTURE WORK: LOG FORMAT

TODO better introduction for this (and also the next?) section

TODO explicitly refer to these as potential areas of academic interest?

This section outlines a few dimensions in which the SSB log format — SSB's "thin waist" — could be modified to provide additional useful properties.

5.1 Partial Replication

By using a linked list of messages as the underlying datastructure, a message can only be verified in time linear to its sequence number. Since all previous messages need to be available for verification, this also implies linear storage overhead. A more granular notion of this problem is to look at the complexity requirements for verifying the hash chain between any pair of messages. For SSB, this complexity is linear in the difference of their sequence numbers.

A natural start for improving on SSB is to look for data structures that allow messages to be verified by only traversing a sublinear number of messages. Such data structures have been studied in the context of secure timestamping, in particular both anti-monotone binary graphs [7] and threaded authentication trees [8] provide verification using a logarithmic number of messages, while only adding a single additional hash to each message.

An authenticated append-only log built on these data structure would allow *partial replication* of logs while still being able to verify all messages that it replicates. Only a logarithmic number of additional messages would need to be stored. Such a system would solve the scaling problems of SSB's all-or-nothing approach.

As an extreme case of partial replication, a feed could fetch a single specific message, without having to request its full feed. SSB's out-of-order messages also allow this, but they forfeit the ability to verify the received message. With partial replication, only a logarithmic number of additional messages would have to be fetched to verify the specific message.

[19] defines *head* and *tail* sets for each entry in an anti-monotone binary graph or a threaded authentication tree, such that the union of the heads and tails of any two messages includes all messages necessary to verify the newer message against the older one. These sets are of size logarithmic in the sequence number. As long nodes request and store these additional sets, trasitivity of replication is maintained even when partially replicating logs.

An interesting problem in this context is how peers would indicate the subsets they want to replicate. While listing sequence numbers works fine, applications would benefit from utilizing semantic criteria instead, for example subscribing to only messages of certain types. Finding a general framework for specifying partial subscriptions based on semantic frameworks is an open problem.

One particular problem in this setting is that malicious peers could deliberately withhold a message even though it fits a particular semantic criterium. Unlike in sequence number based replication, there is no immediate way to detect such omissions. This could be mitigated by adding one additional sequence number per criterion to each message. This approach of however greatly limits the expressivity of the criteria specification mechanism. The example of subscribing to messages of a certain type would work fine: Each message would include a number indicating how many preceding messages of the same type are in the feed. But e.g. subscriptions based on arbitrary prefixes of types would require one sequence number *per character* of the message type, which could already be too costly. If there were multiple criteria and peers were able to select any number of them, the number of additional sequence numbers would be equal to the product of the number of sequence numbers of each criterium. Since this approach does not scale, it would be beneficial to find alternate mechanisms for detecting malicious ommisions of partially replicated feeds.

Due to the above problems, it might often make sense to define an application-specific partially-replicable log format rather than opting for a general-purpose application framework like SSB. When designing such a framework, it is important to keep in mind that

partial replication and subscriptions increase the complexity of the API offered to the applications.

5.2 Local Deletion

SSB includes the full content data in its message signatures. Accordingly, the content needs to be available in order to verify the message. This poses a problem: What happens if a single message in a log has objectable content? If a server locally deletes that content, then it can not replicate the log to its peers beyond the point of that message, since the peers could not verify beyond that point.

This situation could be improved by only including a *hash* of the content in the signature, rather than the content itself. This way, content could be deleted from a local log replica, while keeping the hash, so that the whole log could still be verified and thus replicated.

This change does slightly increase the complexity of log replication and the API between server and clients: Missing message content introduces a new case, whereas in current SSB a message is either missing or completely avaiable.

It should be noted that in current SSB, blobs support both local deletion and targeted replication. In a protocol with both partial replication and local deletion, there would be no need to rely on blobs for these properties. Messages essentially become blobs where authorship and relative ordering can be verified.

5.3 Cryptographic Agility

TODO help, I don't speak scientific crypto lingo

Since SSB relies on multiple cryptographic primitives (signatures and hashes for the log format, encryption for the replication protocol), cryptographic agility [22] is an essential concern. All hashes and signatures in the logs include an indicator of the cryptographic primitive that has been used. This means that the protocol can introduce the use of new primitives as old ones become broken.

An open problem is how old log entries can be "saved" once their primitives become insecure. The naive approach of republishing old messages with a new key changes the hashes of all those messages, thus breaking inter-message references. Alternate approaches could be based on publishing new messages that assert facts about older messages, or determining the trustworthyness of old messages based on how they are being referenced.

5.4 Log Management

In SSB, there is no mechanism for terminating a log. But it would be straightforward to add a mechanism that declares that the log will not be extended in the future (and any future extensions should thus be discarded). By allowing this mechanism to carry some payload data, key rotation could be supported: The log termination record would include the public key of a new log that should serve as the extension of the terminated one.

5.5 Encoding Simplifications

The json-based encoding turned out to be a major source of incidental complexity when implementing SSB in languages other than javascript. A principled redesign should probably use a simple binary encoding instead.

Additionally, messages should more clearly separate integrity metadata (backlink, sequence number and signature), content metadata (type and timestamp), and the actual content.

5.6 Fundamental Changes

The previously mentioned changes to the log format would all result in protocols that would function in basically the same way as SSB. But it is also possible to envision some little changes that would result in systems with fundamentally different behavior. An incomplete list of interesting ideas:

- introducing a mechanism to "merge" forked feeds rather than discarding them
- using different data structures than linked lists as feeds, e.g. sets, maps, trees or directed-acyclic graphs
- making the protocol aware of application-level semantics to support lossy but semantics-preserving log compaction

The design space for systems of identity-centric data replication is vast and not well explored. SSB only occupies a tiny fraction of it.

6 FUTURE WORK: USERSPACE

Beyond protocol-level evolution, there are a couple of higher-level issues that often arise when designing applications on top of SSB. This section gives an overview over the most common ones and sketches potential approaches.

6.1 Multi-Device Support

If two different devices used the same SSB identity to publish messages concurrently, this would result in a forked feed. It is thus recommended to create a distinct identity for each device to eliminate this risk. This however leads to problems such as being unable to decrypt encrypted messages on some device because they have been encrypted to the public key used by a different device. The user also needs to follow or block identities once per device.

One approach could be to develop schemes that allow sharing the same private key across multiple devices to allow read-access, while enforcing mutual exclusion on writes.

A different angle is to write applications in a way that anticipates that there might be a one-to-many mapping from users to SSB identities. Since the messages in a single feed are totally ordered but messages across multiple feeds might only be partially ordered, it is not sufficient to naively treat a set of feeds as a compound feed. Instead, the application needs to be designed from the ground up to deal with partially ordered sets of messages.

Orthogonal to the issue of *using* data from aggregated, partially ordered feeds is the issue of determining which feeds to aggregate in the first place. Settling on a common scheme for signaling compound feeds will be necessary for SSB to successfully improve on the multi-device situation.

6.2 Access Control

Currently, an ssb server will hand out data to anyone who asks. The messages that control replication (*follows* and *blocks*) only specify where data is wanted, but they can't express bounds on how far data should be spread. All data is conceptually visible globally.

To improve privacy, some servers choose to only forward messages of some feed F to identities that are followed by the feed F. This is a rather ad-hoc solution, assigning meaning to *follow* messages that might not have been intended. Work is underway for specifying dedicated messages that place bounds on how far data should be propagated through the social graph. These bounds can not be enforced however, anyone who has the data can obviously choose to ignore these requests. This is just as true as in any distributed setting.

In addition to publishing policies on who should get access to messages, encryption can be used to ensure that only the intended recipients can access the data. Currently, the only supported mechanism is encrypting a message to a small set of recipients. More sophisticated approaches such as encrypted groups could be adapted for ssb.

6.3 Content Moderation

The social nature of most applications buit on top of SSB necessitates some form of moderation features, allowing users to shape and regulate interactions such that they can feel safe in their virtual space. Because no user has a global view of the system, and because the subjective reader property allows even users with identical views to interpret those views in different ways, traditional centralized approaches for moderation can not be applied directly to ssb. In particular, it is not possible to globally "ban" an identity.

Users (and applications on their behalf) have two fundamental options when dealing with unwanted content. They can stop replicating a feed and delete it from their local database. Less drastically, applications can choose to ignore specific (sets of) messages.

While these are powerful primitives that give users full control over their environment, they place the burden on the affected individual. Going forward, it will be important to find mechanisms that allow to share the task of moderation and shift it to users who are privileged enough to be able to invest the necessary energy and time. A simple example could be to automatically adopt the *block* messages of trusted peers. Since human dynamics are very nuanced and every human has their specific needs, we expect a lot of experimentations and different groups of users settling on different tools.

6.4 Causal Ordering

Whenever a message m_1 contains the hash of another message m_2, this implies that m_2 must have already existed at the point where m_1 was created. Taking the transitive closure of this irreflexive relation yields a strict partial order. For any pair of messages m_1, m_2, m_2 is guaranteed to be older than m_2 if (m_1, m_2) is an element of this order. Equivalently this can be interpreted as the existence of a (directed) path from m_2 to m_1 on the graph with the set of all messages as vertices and edges corresponding to the hashes inside those messages.

Storing this transitive closure in a space-efficient way such that it can be quickly queried and efficiently updated is a difficult but well-studied problem in the database literature [16] [29]. It should be possible to leverage some properties of SSB to get higher-quality results. The immutability of messages reduces the number of cases when dynamically maintaining the index structure, and the fact that

each feed forms a total order can be utilized to efficiently encode the relation. Designing and implementing a general framework for performing causal ordering queries on SSB messages would be both an interesting research topic and a powerful tool for building applications.

6.5 Replication Improvements

The currently used default replication protocol does not protect against adversarial nodes, for example it would be fairly easy to launch an eclipse attack [25] against the overlay network. But whereas it is difficult to defend against these attacks in general, SSB can make use of data such as the friend graph to protect against them. A *follow* message can be interpreted as an expression of trust. Assuming that trusted peers are unlikely to perform an attack, keeping a certain number of trusted peers in the views of the peer sampling service would protect against eclipse attacks.

Another area where the replication protocol could be improved is by using private set intersection when determining the set of feeds that both parties are interested in. That way, untrusted peers would not be able to learn about new ids purely from the replication layer. Combined with an access control mechanism that only forwards data to authorized identities, this would provide resilience against bots "spidering" the network.

TODO mention alternative models (indexing-in-the-cloud, lite clients) here?

7 "SSB SUCCESS STORIES"/BENEFITS

TODO rename the section. Merge this and the pain-points into a single section?

TODO introduce the section

TODO touch on encryption (as usual, Alj is uncomfortable writing about crypto)

An often stated mantra in SSB development is "no global singletons". SSB is thus a fully decentralized system, or more accurately a collection of decentralized systems that overlap to varying degrees. It is consequently highly resilient to (targeted) failures, users do not need to depend on any single, priviledged central authority, including cloud-based service providers. Going beyond the common mentality of a grand unified social platform, there are deliberately isolated networks, for example limited to a family, or a specific local area network.

Since SSB applications only interact with the local replicas of logs, complete offline operation is automatically built in. Offline operation is simply a special case of a network partition. Because this case occurs so often, the protocol is geared towards handling network partitions gracefully, further contributing to the resilience of SSB. In particular, all operations are delay tolerant.

Continuing with the theme of resilience, data loss is highly unlikely, since full log replicas are stored throughout the network. Indeed it is common practice when migrating between devices or during development to delete the whole local database, keeping nothing but the keypair and optionally a list of servers with a well-known ip address. Upon starting a server again, all data gets retrieved from the network.

By leveraging trust instead of trying to eliminate it, there is no need for proof-of-work and similar wasteful techniques. The

delay tolerance allows routing layers that can optimize for efficient usage of resources rather than going for minimal latency at any cost. This mentality could go a long way towards systems that are more sustainable than current alternatives. And there are strong arguments that favoring systems that try to eradicate inter-human trust might not lead to a particularly desirable future.

The freedom of applications to interpret data in whatever way they see fit reduces the ability of centralized powers to coerce users. Furthermore, the usage of json as a common, self-describing data format prevents information being "closed in". Rather than depending on a proprietary API, interoperability and the ability to create alternative frontends is the default.

Due to the "subjective reader" approach, all ssb servers can operate concurrently. There is no need for costly synchronization across nodes. Applications can fully embrace this, for example by using CRDTs. The monotonically growing logs are well-suited for implementing CRDTs and similar techniques.

As another consequence of the subjective interpretation of data, there is no need for central coordination to introduce or evolve features. Applications can simply start producing new kinds of messages, and interoperability works out with all users who share the same interpretation of those messages. Especially when evolving functionality, it is often sufficient to add more entries to a json object. Other applications simply ignore them but will continue working.

This results in an organic evolution of features, without "the system" ever shutting down. Like any uncoordinated evolutionary process, the end result might not be as streamlined as a shiny, pre-designed solution. But unlike many such proposed, streamlined systems, the continuous and incremental improvement of the SSB ecosystem has resulted in a thriving ecosystem that serves the needs of thousands of users. And it will continue to evolve.

TODO is this *too* political/opinionated? Or not political *enough*? We really need that "sociological computer science" journal... Feel free to tone this section down. But damn, writing those words *felt good*!

TODO another benefit: Data can automatically be reused across apps - frees devs from reimplementing identity management etc. again and again, leads to a very coherent user experience

8 "SSB PAIN POINTS", LIMITATIONS (ANALYSIS / CRITICAL REVIEW)

8.1 Onboarding

8.2 Crypto

re-keying peers, forward secrecy

8.3 Trusting Pubs

Pubs will see your complete social graph, you have to trust them. If pub is operated in the cloud, then its private key is difficult to protect (= potentially accessible to the cloud provider),

8.4 Routing and Scalability

8.5 Protocol Agility

binary msg format, crypto

8.6 Resource attacks

e.g. easy to follow a clique of peers that has been setup for DoS reasons (with humongous logs), victims will include them in the 2-hop neighborhood, difficult to automatically block (detecting the shape of the clique, clique members added faster than victims can remove them).

8.7 Deleting content, GDPR, ephemeral content

8.8 SLA for Blobs, privacy of blob transfer

8.9 Scaling

Pubs may experience scaling problems: unlike routers, an SSB pub can not rely on queues (and taildrop) to postpone work: has to store all logs of all 2hop neighbors. Social graph of a pub may grow too big (X times 100 new nodes per new user, on average?). Currently no policy to randomly drop logs? If too many pubs drop the log of a specific peer, this peer may "loose SSB-connectivity", fall out of the social graph, technically?

8.10 KawaiiPunk's critique

8.11 Problem of multi-device peers

8.12 Global (thus central) coordination needed of 'type' field name space, applies also to other fields at the SSB waist (e.g. new crypto algos)

8.13 Unfriending pubs

…

9 COMPARING SSB WITH NAMED DATA NETWORKING (NDN)

After a brief introduction to NDN we compare and relate SSB to NDN in three different ways: layering SSB on top of NDN, layering NDN on top of SSB, and a hybrid mode NDN forwarders are made SSB-aware. An interesting case is made of a NDN problem that has no equivalence in SSB, thus seems to be due to NDN's architectural choices. But some of the other problems we encounter point to questions that apply to NDN as well SSB.

9.1 Named Data Networking

NDN is a receiver-driven datagram access protocol: The receiver has to request content by name –in a so called `Interest` packet– and at most one matching content is returned in a corresponding `Data` packet. The Data packet includes the content's name and is signed such that a forwarding nodes can verify the correctness of the name-to-content binding. Checking the validity of a signature requires additional certification data which a forwarding node can fetch using the standard `Interest/Data` packet pattern. Validated data packets are typically cached such that subsequent requests for the same name can be served from in-network memory.

```
--> I('/ndn/some/item')
<-- D('/ndn/some/item', data, signature)
```

In order to facilitate routing of Interest packets towards a data source or a data repository, NDN uses a hierarchical name space.

Routing rules are based on name prefixes, typically aggregating all data items made available by a publisher. In a forwarder node, incoming Interest packets are matched against these prefixes on a longest-prefix matching basis, yielding the interface(s) to where an Interest has to be forwarded. Interests for the same name that arrive close in time are deduplicated using a PIT (pending interest table). On the return path, a data packet is copied to all interfaces from where a corresponding interest came in, and the PIT entry is deleted.

In such a pull-only communication model, streaming protocols –as well as fetching content that is larger than the 4KB datagram size– use name prediction (e.g., sequence numbers as part of the name): Several Interests are sent ahead without waiting for the first Interest to be answered. If name prediction is not available, manifests [?] can be used i.e., a datagram is returned in lieu of the content which contains a sequence of names pointing to single Data packets or further sub-manifests.

9.2 Layering (or Merging) of SSB and NDN

In this paper we examin three different configurations: Whether NDN can be used as a transport network for SSB, whether a "SSB-aware" NDN layer would work better, and whether SSB could be used as a support for NDN networking. Figure [?] shows the three different scenarios which we expand in the following subsections. The purpose is to shed light on the sometimes implicit assumptions behind SSB and NDN.
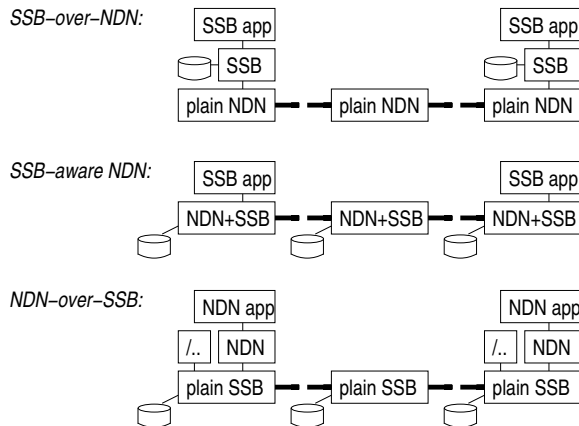
Figure 4: Three different layerings of SSB and NDN, as discussed in this paper

9.3 SSB over NDN

SSB's log entries, also limited to 4KB, map well to NDN if one abstracts away from the wire format: A SSB log entry for $\langle id : seqno \rangle$ would become

    D('/ssb/logs/<id>/<seqno>', content, signature)

where content includes the name of the previous log entry. In order to support access to a log entry via it hash, all content would also be published under

    D('/ssb/sha256/<hashval>', content, signature)

This double-publishing is necessary because NDN's implicit name component mechanics requires a *full* name, hence cannot do a longest prefix match on "/ssb/ImplicitSha256DigestComponent =<hashval>". Moreover, the digest algorithm of SSB may change in the future and diverge from NDN's, hence requires a name subtree of its own (sha256 in this case).

A SSB client wanting to replicate a log from scratch would issue Interest packets starting at I('/ssb/logs/<id>/0') and increase the sequence number until no more log entries can be fetched.

The ease with which this mapping could be introduced is misleading, as there are several problems with such an approach. First and foremost, implementing SSB's push model leads to continuous polling at the level of pull-oriented NDN. Second, there is no prefix aggregation possible because of SSB's flat ID space.

The latter concern could be addressed in two ways, both having undesirable consequences. The first would be to introduce a NDN routing strategy that mimics SSB's forwarding along the receiver's social graph. Such a modified NDN forwarder would have to either parse all logs or to receive the graph information from somewhere else. Additionally, Interests would have to carry the ID of the receiver (otherwise the forwarder does not know which graph to use), which is contrarion to interest aggregation, which would have to be changed. Such a special "SSB routing strategy" would have to be deployed globally, essentially converting NDN into a SSB core. We will come back to a SSB-aware NDN layer in the next subsection.

A second approach would be to use NDN's LINK objects in order to redirect Interests towards the location of the storage provider for a given $\langle id \rangle$. This requires a NDN name server [?] and a permanent storage provider to which the SSB devices would have to upload new content. The change in this case is on SSB's side, whose community would have to accept a logically centralized name service component. The question then is whether a NDN name service could handle a global database for the $\langle id \rangle$-to-storage mapping, or whether also here we would have to introduce SSB's social graph for scaling reasons. In fact, SSB's use case is a case of producer mobility where it is questionable that NDN could handle a world consisting of mobile terminals only.

SSB-over-NDN would either require changes to NDN or put tremendous stress on either on the NDN routing system or a name service, also potentially forcing SSB users to work with "centrish" content repositories and mapping services.

9.4 SSB-aware NDN

Assuming that SSB's replication approach (limiting content propagation to a device's social graph works) leads to scalable solutions, NDN could be made SSB-aware. This would require that NDN inspects the logs in order to learn about the social graph. In fact, it would promote NDN forwarders to become SSB nodes that replicate SSB content (instead of a name server infrastructure and dedicated repo providers). A new, corresponding forwarding strategy doesn't requires routing entries as it simply floods incoming interests that cannot be satisfied from the local replicas to all SSB+NDN neighbors along the social graph.

However, using such a strategy would be a travesty as it turns NDN into a push network for interest packets (due to the polling, by every node, whether new content is available or not) – only sporadic

data packets would be returned during the replication process. An optimization would be to use long-lived interests, turning NDN into a pub/sub system. Finally we point out that some variant of NDN's SYNC protocols could be suitable to capture SSB's goal, namely to sync the replicas across (a part of) the network. Such a variant would be able to benefit from SSB's strict log extension rule. But looking at the way Sync is implemented in NDN, this comes back at a push-style communication where interests are used to poll neighbors about any state change. We think that bringing a SSB mindset into NDN would change NDN beyond recognition, especially flow-balance would be lost during this transition.

9.5 NDN over SSB

An interesting thought experiment is to reverse the layering, stress-testing SSB in this case. Is SSB universal enough so that one can "emulate" NDN over SSB (see the third subfigure of Fig. 4)? SSB would have to implement three distinct NDN features: the hierarchical name space, and the pull-model and NDN's trust system.

Again assuming rough equivalence of NDN data packets and SSB log entries (i.e., a triple $\langle name, content, signature \rangle$), the pull-model part is easy to answer: Either some item is already in one of the eagerly replicated local logs, or it is not available yet (because SSB is push-based).

The major problem is NDN's hierarchical namespace which is a globally shared construct with the service level agreement that any (existing) item referenced through this tree can be fetched. Even if a delegation model is used, this global resource introduces a central authority, or at least a consensus algorithm, to allocate prefixes. This entity would have a "well-known" SSB id and a log from where the rest of the SSB world would inform itself about its decisions. Once these prefixes are handled (by a special SSB app for supporting NDN's namespace, as shown in the third subfigure), repo IDs can be introduced such that an end device can address them and request content from them. However, in SSB's worldview, a repo would have to follow all potential customers i.e., to learn about their IDs, otherwise these customers cannot express interest in some content (which could be delivered through log replication of transient SSB IDs, for example). When looking at NDN we realize that NDN has a social contract along the interest path: a NDN forwarder accepts any downstream node as a friend, accepts its interest packet (= pushed replica of the request), and then relies on a similar contract with its upstream node. Following this insight, our NDN emulation would have to introduce "NDN forwarding providers" at SSB level. Once these "NFPs" are in place, we would also let them implement NDN's trust model by validating content through NDN's certificate authorities.

While it doesn't seem strictly impossible to continue that emulation argument, it is already obvious from the above discussion that one would not benefit from SSB's social graph replication mindset. We try to explain this result in the following subsection.

9.6 Information-Centric vs ID-Centric ?

At first sight, the stark difference of PULL vs PUSH is the main differentiator between NDN and SSB. However, the above layering discussion shows that there is a second, equally strong discrepancy which we pin to the way IDs are handled.

NDN has no notion of receiver ID, by design, which has the benefit of easy Interest aggregation but also is the basis of the social contract with the forwarder (to accept interests from anybody). SSB, however, is deeply ID-centric: Only by following (= declaring interest in) some ID, *all* its content will be pulled towards the interested party. Selective data replication, which requires a bidirectional Interest/Data protocol, only would work in SSB if both sides follow each other - hence the difficulty to implement NDN's global fetch mechanics over SSB. NDN, on the other hand, introduces repositories (somehow linked to routing prefixes) as quasi-IDs: NDN's service model is to interconnect (many) ID-less clients with (few) identified repos. As such, NDN has a centralization bias which obviously clashes with SSB's decentralization stance.

One can also draw the following picture: NDN works with repo IDs (prefixes) on top if which we have IDs for content (= content names extending a repo ID). In NDN, IDs have no role for the receiver or in the replication process except that forwarding validates the origin of data items. On the other hand, these repo IDs must be global routable through some unspecified routing protocol outside the NDN specs. SSB also has producer-side IDs, but it is mandatory that clients also have an ID because otherwise they could not publish their replication needs (towards SSB's routing logic).

Finally, IDs have different weights in NDN and SSB when it comes to replicas and packet loss. NDN only validates that a data item's signature can be traced back (via publisher IDs) to some trust anchor, while ARQ (automatic retransmission request) must be used to recover from lost packets. In SSB, the whole log associated with an ID is validated (assert strict ordering and completness), which factors out difficult tasks to the benefit of the applications (no retransmission logic, no need to SYNC).

We conclude that so far we failed at finding ways how NDN could be made suitable for SSB, or be enhanced by SSB properties without turning NDN into SSB, and how to port NDN to SSB without deviating from SSB's decentralization agenda. However, there are good technical, not political reasons, to continue exploring such mappings because of an undesirable artifact of NDN, described below, that only shows up in NDN but not in a SSB context.

9.7 Fixing NDN Routing Stratgies with Replication?

...

10 RELATED WORK

The basic ideas behind SSB can be traced back to the nineties: Secure logging [24] and secure relative time-stamping [13]. The major innovation of SSB is to use these techniques for disseminating data through a gossip protocol in a network of untrusted peers.

- Linda tuple space (global data pool): apps working with wr() and rd() only.
- Haggle (pocket-switched networks), opportunistic networking, DTN
- ICN in general
- ...
- WAL is anchor of persistence in distributed systems. Discuss scenarios with node crashing before/after contacting a peer. Rare chance of forking a log in case of network partition.

- RAFT, resilient log-based consensus. Log compaction is what SSB is lacking, but could add (on a per-app basis).
- GIT and DAT
- Kappa and CoucheDB

Also contrast with the decentralization provided by blockchains?

11 CONCLUSIONS

We presented secure scuttlebutt, a fully decentralized, peer-to-peer event-sharing protocol. The core novelty is that data replication occurs at the granularity of complete append-only logs of messages by a particular author. This approach leads to a simple, yet efficient replication protocol that lends itself well to a large class of applications. By embracing *eventual* delivery and subjective interpretation of data, SSB gets to sidestep common sources of complexity. A community of multiple thousand users interacting through a variety of different applications confirms the viability of the approach.

TODO explicitly mention privacy challenges

TODO bottom line: identity-centric replication as a new paradigm introduces a unique set of trade-offs and is well worth exploring further

…

REFERENCES

[1] 2002 - 2019. meetup.com. https://www.meetup.com/
[2] 2007 - 2019. doodle.com. https://www.doodle.com/
[3] 2008 - 2019. Github. https://github.com
[4] 2014 - 2019. Loomio. https://www.loomio.org/
[5] 2018. https://www.scuttlebutt.nz/
[6] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (2012), 77–89.
[7] Ahto Buldas and Peeter Laud. 1998. New linking schemes for digital time-stamping.. In *ICISC*, Vol. 98. 3–14.
[8] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. 2000. Optimally efficient accountable time-stamping. In *International Workshop on Public Key Cryptography*. Springer, 293–305.
[9] Scott Chacon and Ben Straub. 2014. *Pro git (2nd Edition)*. Apress. https://git-scm.com/book/en/v2
[10] Secure Scuttlebutt Consortium. 2012 **??** - 2019. SSB source code. https://github.com/ssbc
[11] Mix Geursen, Piet Irving. 2018. scuttle-poll. https://github.com/ssbc/scuttle-poll
[12] Piet Geursen. 2017. path-gatherings. https://github.com/pietgeursen/patch-gatherings
[13] Stuart Haber and W Scott Stornetta. 1990. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*. Springer, 437–455.
[14] Internet Archive. 2018. Decentralized Web Summit 2018, Jul 31 – Aug 2, San Francisco. https://decentralizedweb.net/
[15] Mix Irving. 2018. Scry. https://github.com/ssbc/patchbay-scry/
[16] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. 2012. SCARAB: scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 169–180.
[17] Charles Lehner. 2018. Git-SSB: Social Coding on Secure-Scuttlebutt. https://git.scuttlebot.io/%25n92DiQh7ietE%2BR%2BX%2FI403LQoyf2DtR3WQfCkDKlheQU%3D.sha256
[18] Joao Leitao, Jose Pereira, and Luis Rodrigues. 2007. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 301–310.
[19] Helger Lipmaa. 1999. *Secure and efficient time-stamping systems*. Citeseer.
[20] Gordon Martin. 2017. ssb-chess. https://github.com/Happy0/ssb-chess
[21] Aljoscha Meyer. 2018. SSB Specification. https://spec.scuttlebutt.nz/feed/messages.html
[22] David Nelson. 2011. *Crypto-Agility Requirements for Remote Authentication Dial-In User Service (RADIUS)*. Technical Report.
[23] Noffle. 2016. git-ssb-intro. https://github.com/noffle/git-ssb-intro#push-conflicts
[24] Bruce Schneier and John Kelsey. 1998. Cryptographic support for secure logs on untrusted machines.. In *USENIX Security Symposium*, Vol. 98. 53–62.
[25] Atul Singh et al. 2006. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer.
[26] André Staltz. 2019. SSB Roadmap. https://github.com/staltz/ssb-roadmap
[27] Dominic Tarr. 2018. Notes on a paper for SSB. https://github.com/dominictarr/scalable-secure-scuttlebutt/blob/master/paper.md
[28] **??** Duncan. 2018. Scuttlebutt Protocol Guide. https://ssbc.github.io/scuttlebutt-protocol-guide/
[29] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. 2013. Dagger: A scalable index for reachability queries in large dynamic graphs. *arXiv preprint arXiv:1301.0977* (2013).

A DOMINIC TARR: SCALABLE SECURE SCUTTLEBUTT

This is notes on a paper for ssb.

A.1 assumptions about social networks

TODO: find refrences to justify this assumption.

- *power law assumption*: we expect activity to follow a power law. (a small proportion of users are update very frequently, a large number, only infrequently)
- *tendency for local connections*: if you are "friends" with a peer it's likely that another friend is also a friend.
- *hub connections assumption*: short paths to hubs: some users are "hubs", being connected with a large number of other peers. Paths to even distant users are short due connections via hubs. It's highly likely that someone you know follows any given celebrity.

For simplicity, we model the following replication protocol designs in the context of connected random swarms. In practice, we do not want a design that replicates all messages in the entire network (especially because we intend it to scale to millions of users). For social media applications, users view the feeds of they have explicitly followed/friended. However, due to the strongly connected nature of the social graph (the primary way people meet is being introduced to friends of friends) the chance of having a considerable overlap in your follow graph with any friend is quite high. Thus, the simpler to model random network is a reasonable approximation to friend replication in a real social network - how good this approximation is is discussed in
TODO: write this section.

A.2 data models

We use a simple data model that well describes any social media application. The main resource is a *feed*, which is an append-only log of *messages*. Each feed has strictly a single author. Each peer is the publisher of their own feed, and the subscriber to zero or more other feeds.

Each feed is an append-only log of messages, and each message contains the id of the feed, an always incrementing sequence number, and some content. (Also, the hash of the previous message and a signature, but this paper focuses on the performance of our design, not the security, so we can leave that out for now.)

```
Feed f = [{ id: f.id, sequence, content }, ...]
```

A peer is usually the author of at least one feed, but may be a "lurker" who does not post. In this paper we can assume that each peer is the author of one feed and that the peer's id is also the id of that feed.

Each peer is also a subscriber to their own, and zero or more other feeds.

```
Peer p = { id: p.id, feeds: { <id>: [msg,...] } }
```

A.3 comparison of replication algorithms.

Starting with the simplest, develop models of data replication.

I basically just made up the O() notations... maybe this should be based on simulations instead? especially since some of my arguments depend on a certain factor being limited somewhat (by the nature of networks).

A.3.1 polled scan: (RSS) Really Simple Syndication.

A publisher ('pub', of type 'Peer') hosts content, and a subscriber ('sub', also of type 'Peer') connect and the publisher sends their content.

At each request, the publisher sends the entire feed.

(footnote: In practice, RSS truncates the feed and may not send older messages, so isn't provably eventually consistent, we've analyzed a simplified version, which has provable eventual consistency.)

This is extremely simple to implement at the server end (RSS provides an XML file over HTTP) and slightly more complex at the client end, as clients append only the new values. It's assumed that messages are fairly small, text only, and large files are referenced as some sort of link.

When a subscriber connects, the publisher replies `received = pub.feeds[pub.id]` (which means sending 'pub.feeds[pub.id].length' messages) the subscriber then appends any new messages to their copy of that feed. 'sub.feeds[pub.id].append(received[sub.feeds[pub.id].length...])' such that both copies of the feed are the same, that is, contain copies of the same messages. 'sub.feed[pub.id] == pub.feed[pub.id]'.

New messages are published over time, and so the subscriber periodically makes a request to each publisher.

```
interval(sub.pollFrequency, () => sub.feeds.each(id => sub.connect(
```

So, every 'sub.pollFrequency' all publishers are connected to and all messages from them are downloaded, old messages end up being sent many times unnecessarily, so the amount of bandwidth needed scales very badly.

Bandwith needed for a subscriber can be calculated as the following:

(footnote: Assume that 'pollFrequency' is number of polls within the given timeframe that we are calculating resource usage for. The important thing is how many polls are made. If we were to calculate usage per day, and there was one poll per day, pollFrequency is 1. In any case, we are more interested in exploring the relationship between the various design factors and resources used, so the important thing to observe here is that the total resource used is multiplied by 'pollFrequency', doubling 'pollFrequency' doubles the number of messages sent)

```
total\_messages = sum(map(sub.feeds, id => sub.feeds[id].length))
sub.pollFrequency * total\_messages
```

Each interval, the subscriber polls every publisher, and receives all messages. Hence the total set of messages is redownloaded every interval.

Bandwith needed for the publisher can be calculated as the following:

```
subscribers = sum(peers, peer => peer.feeds[pub.id] ? 1 : 0 ))
avg\_poll\_frequency = sum(peers, peer => peer.feeds[pub.id] ? peer.
subscribers * avg\_poll\_frequency * pub.feed[pub.id].length
```

Clients have a tradeoff between bandwidth and latency. Either they use lots of bandwidth or wait a long time for new messages. So this design is not suitable for real-time communication.

For publishers, this design also suffers from uncontrollable expenses. If there are suddenly many subscribers, or they set their 'pollFrequency' very high, this increases costs for the publisher, which in practice will lead to outages. Thus the most popular content is the most likely to be unavailable, which is the opposite of what is needed.

Also, this model uses a network connection per poll, is likely to be a limiting factor for publishers with large numbers of subscriptions.

The total number of network connections over some time period is for the subscriber:

'avg_poll_frequency * sub.feeds.length'

and the publisher:

'poll_frequency * subscriptions'

A.3.2 append-only poll.

Messages in a feed have a total order defined by an always increasing value such as a sequence, such that any message's sequence is strictly greater than any preceding message. If the sequence number of the first message is 1, then the number of messages in the feed ('feed.length') is also the sequence number of the last item.

(footnote: By sending messages in order, if a transmission fails part-way, the requester's copy of the feed is still a valid append-only log with no gaps - but their latest message is just not the true latest message. Next time they connect they will receive the missing messages.)

Instead of sending all messages per poll, the subscriber requests all messages greater than the sequence number of the latest message they currently have. This requires sending on a tiny header (the sequence number) and the publisher only sends each message to each subscriber once.

The publisher expects a sequence number, and returns any messages greater than that.

```
pub.serve(n => pub.feeds[pub.id][n...])
```

The subscriber connects to a pub, and appends the messages the pub returns to their copy,

```
received = sub.connect(pub.id, sub.feeds[pub.id].length)
sub.feeds[pub.id].append(received)
```

now the subscriber is consistent with the publisher.

(footnote: The publisher sends the messages in order, so if a connection fails part-way through, the subscriber's copy still has sequential messages)

The cost for the subscriber is as follows

```
sub.pollFrequency * sub.feeds.length + total\_messages
```

This is a significant improvement over polled scan because each message is only downloaded once. However, the subscriber must still send their current sequence number to each publisher, on each poll. Although we can resonably assume that the sequence number is significantly smaller than a message, if the 'pollFrequency' or 'sub.feeds.length' is high this can become significant.

The number of connections needed are the same as polled scan.

For a suddenly popular publisher, many incoming requests can still lead to availability problems, as the simple number of requests becomes overwhelming, although because the entire feed of messages does not need to be sent the practical limit is much higher.

A.3.3 append-only gossip (scuttlebutt).

In a gossip protocol, instead of subscribers polling publishers, "peers" which can be both publisher and subscriber, connect to each other randomly. On each connection, instead of requesting a single feed, peers send a "vector clock". Instead of representing a global sequence, a vector clock just includes the sequence on each peer that contributed to the state. A peer's current vector clock is just a map of the lastest sequence of each feed:

```
vector\_clock = map(peer.feeds, id => peer.feeds[id].length)
```

When a peer receives the remote vector clock, they can simply calculate whether there are any messages they need to send and send them.

```
peer.serve(clock => mapValues(clock, (id, sequence) => peer.feeds[id
```

A client just connects to a random peer, sends their clock, and appends messages they receive

```
each(
  peer.connect(random\_peer.id, vector\_clock),
  msg => peer.feeds[msg.id].append(msg)
)
```

Since a connection now sends the list of subscriptions, but only needs to connect to a single peer each poll interval, more bandwidth is used per connection, but less connections are used. The overall bandwidth used by a peer is the same as with append-only poll, but the number of connections is now only 'O(poll_frequency)'.

Because messages are no longer passed directly from the publisher to each subscriber, describing the time needed to disseminate a new message is more complicated. In the first poll interval, the publisher will be connected to at least 1 other peer. (The publisher makes 1 outgoing connection, but may receive any number of incoming connections.) If it gets passed to only a single peer, but in the second poll interval, there are now two peers able to disseminate the message. If they do not connect again, in the 3rd interval there will be 4 peers, and so on in powers of 2. However, as the number of peers with a given message increases the chance that any two connecting peers already both have the message increases too, and the rate of dissemination decreases. Thus overall rate of dissemination resembles an S curve. Since calculating the actual rate of dissemination is more complicated, and is affected by practical matters such as the probability that multiple peers connect a particular peer at once, instead of calculating the time, we take measurements from a simple simulation.

The pattern of dissemination of a single message is the same as flooding gossip. For a random network with 10,000 peers and each peer creating a connection to one other peer randomly each interval (so a given peer may receive zero or more incoming connections, but makes only one outgoing connection), the total number of intervals needed to diseminate a single message is very small compared to the number of peers.

```
round, dR, dT
1, 9, 10
2, 51, 61
3, 293, 354
4, 1195, 1549
```

```
5, 3903, 5452
6, 3875, 9327
7, 666, 9993
8, 7, 10000
```

In Amazon Dynamo, this protocol design is used to replicate membership information within a cluster of Dynamo nodes. The peers run inside a trusted enviroment, and all peers replicate all other peers. To add a peer to the network, that peer just needs to know any other peer. It's not necessary to inform any master node, and the cluster is highly resilient.

This design has a significant advantage with availability. If a peer that originated a message goes offline, if they have disseminated a message to at least one other peer that message will continue to flood the network. If a publisher suddenly becomes very popular, it will not cost them extra resources, because it's the other peers which will provide the dissemination.

A.4 update frequency, overlap, and peer selection

In Amazon Dynamo, scuttlebutt replication is used as a subcomponent of the whole system - to keep track of the membership in the database cluster, and what range of the database each node is responsible for. When database requests come to a node, that information is used to route the request to nodes which can handle it. Each node therefore needs to replicate *all information* about membership in the cluster, and also, that information must be kept continually up to date. Each node emits a regular heartbeat and this is gossiped across the cluster, and the other nodes use this information to calculate the probability that a given node is still active - thus wether requests should be routed to it.

Other applications using are likely to differ in terms of whether peers need to replicate the entire dataset, or the regularity with which they broadcast updates, or both. For example, a chat application combines messages from everyone one in the "room", so each peer replicates the entire dataset, but each peer only writes messages to the chat as frequently or infrequently as they choose to. It's quite likely that a few peers write very frequently and others read but do not write, or write very little.

Indeed, in most real world applications, not all updates are created on a regular basis. There may be a small number of people you communicate with frequently - the closest family and friends, but then a broad range of aquaintances that you speak with occasionally. This pattern, known as a power-law distribution, is frequently found in both natural and artificial phenomena. Books and Movies are dominated by a small amount of best sellers, but also a large number of cult classics, flops, or break-evens. Most companies fail in the first few years, but a small number become so successful that it offsets venture investments in all the companies that fail. Likewise, it's reasonable to expect that most applications, if they do not have an explicitly regular update pattern, such as a enviromental sensor, will probably have activity following a power law, in the distribution of updates. However, if many peers have only infrequent update, it's likely that any two peers will exchange vector clocks with mostly the same values, and this is wasted bandwidth.

The other question, what portion of the dataset should be replicated to each node? Another question, is what data should be distributed to each node. In Dynamo, or the chat room, the replicated data is replicated to all nodes, but in most other applications, it's not really diserable for all peers to have all data. For example, in email, the only peers that really need a particular message are the sender and the receiver (mail forwarding agents are a necessary evil)

Email is probably not suited to a replication pattern, as only the recipient and sender are intended to have a use for a given message, and email has enough problems with spam that replicating 3rd party messages seems strange. On the other hand, social media, seems extremely well-suited to a replication design: firstly, content is already log-oriented. typically, users explicitly "follow" or "friend" each other, and the main user interface element is viewing a combined feed of all follow's messages. "shares" are broadcast, usually intended to be read by all followers/friends. Each peer may want to only replicate their friend's data, but since the main way of meeting new friends is by meeting your friend's friends, there is a good chance that any friend also holds messages you wish to replicate.

If less than the entire dataset is to be replicated to each peer, this also raises the question of *which peers to connect to?* in email, this is not an easy question to answer, as any one knowing your email address can send you messages. On the other hand, social media applications present an elegant answer to this question: The peers you follow are the peers you should connect to, you are likely to share mutual friends with them, and thus they are likely to have the feeds you are looking for, and want the feeds you have.

A social media application provides good an simple ways to both choose a partial dataset to replicate and choose who to replicate it with, and because of the high degree of connectivity of the social graph, it seems extremely likely that such an application built on top of an efficient gossip replication protocol could easily scale to an unlimited number of users. Provided the implementation can scale to the needs of most individual users, each user's data overlaps with their friends, and thus the network could easily cover the entire globe.

The design we come up with here could be used in any application that needs to replicate data with a few thousand peers, wether the dataset be shared fully, or having a well defined overlap. We present a social media application only as a relatively flexible base-architecture.

A.5 append-only gossip with request-skipping

In practice, activity in most datasets follows a power law: some authors are highly prolific, but most only publish rarely. Thus, it is likely that when two peers exchange a vector clock in append-only gossip, the majority of feeds mentioned have not changed.

(footnote: Indeed, this became a practical problem in secure-scuttlebutt, on each connection, each peer sending over half a megabyte of requests, yet not actually needing to send any messages.)

The chance that no new messages are sent during a connection increases with 'poll_frequency'.

request-skipping is an optimization to avoid making feed requests if it seems unlikely that a feed has changed, it requires storing the

received clock from remote peers, but saves sending many headers after the first connection.

On the first connection between two peers, the entire clock is sent, but on subsequent connections, the current clock is compared with the stored copy of the remote clock, and only the feeds that differ are sent.

```
// first connection
local\_clock = map(peer.feeds, id => peer.feeds[id].length)
// take the stored remote clock, or an empty clock if this is the first connection.
remote\_clock = peer.clocks[remote.id] || {}
conn = peer.connect(remote.id)

conn.send(filter(local\_clock, (id, seq) => remote\_clock[id] != IGNORE \&& remote\_clock[id] != seq))

remote\_clock2 = conn.recv()
remote\_clock = peer.clocks[remote.id] = merge(remote\_clock, remote\_clock2)

// if they have requested feeds we did not send, send our current seq for those feeds.
conn.send(map(
  filter(remote\_clock2, (id, seq) => local\_clock[id] != seq),
  id => local\_clock[id] || IGNORE
))

// finally, send any needed messages
conn.send(mapValues(remote\_clock, (id, seq) => if local\_clock[id] > seq \&& seq != IGNORE then peer.feeds[id][seq ... ]))
each(conn.recv(), msg => peer.feeds[msg.author].append(msg))
```

'IGNORE' is a special value used to indicate that the remote has requested a feed that we choose not to replicate. It is necessary to make a definite response in this case, because this enables the remote to remember we are not interested in this feed, and so they will avoid requesting this feed next time they respond.

Once we receive the remote's clock and have compared it to the stored copy, we can calculate everything that needs to be send or received. In practice, long-lived connections are used, and we allow new clocks to be sent at any time, but for simplicity of describing the algorithm we represent it here as having 5 phases: *send initial clock, receive remote clock, send response clock, send messages, receive messages*.

*(footnote: It is essential that we only update our record of the remote clock with data they have explicitly sent us, and **not** based on the messages we have sent them. It is possible that a connection fails before our peer receives a message, but if they send us something we know they meant it.)*

If peers A and B are consistent with respect to feed X, neither will mention X the next time they connect. However, if either peer receives a new message in X, one of them will mention it and the other will respond, and the first will send the message. If both receive the new message before they next reconnect, they'll both mention it, but see they are at the same message and not send it.

If peer A requests a feed id X that B has not chosen to replicate, B receives 'X: <seq>' from A, and will reply with 'X: IGNORE'. A will store 'A.clocks[B.id][X] = IGNORE', and B will store 'B.clocks[A.id][X] = <seq>'. 'IGNORE' is never sent in the initial clock, only in the response. If B later chooses to replicate X, the next time they connect to A, they'll check their current sequence (which will be 0 at the time they choose to replicate X), against the stored clock for B. They'll see that it's different and send 'X: 0' in the initial clock. A will then see that B is no longer ignoring X, and will respond with their sequence for X. If B doesn't change their mind about X, A will never mention it again.

(footnote: In the case that B decides to replicate X, but somehow ends up with the same sequence that A has for X, then they won't mention it, however, sooner or later, they will receive a new message in X from someone else, and after this will mention it to A)

The worst case, for two given peers exchanging a single feed, is when the poll frequency is greater or equal to the frequency that new messages are added. This means that each peer sends a vector clock element for every message added to that feed, so the maximum number of vector clock elements is the same as the number of messages sent. If the poll frequency is lower than the message frequency, efficiency increases as each vector clock element will correspond to potentially many messages. Since this at worst a constant factor of the number of messages, it's within acceptable bounds and poll frequency can be selected for maximum availability without trading off bandwidth usage.

It is expected that in practice, message frequency differs greatly by feed. Request skipping saves sending vector clocks elements for infrequently updating feeds, so a great deal less vector clock elements need be sent than in append-only gossip, especially when using high poll frequencies.

$$messages * (peers\_connected\_to * peer.feeds[id].seq * (peer).pollFr$$

There is now only one multiplicative factor in the bandwidth complexity. We must send the entire vector clock to each peer that we will connect to, the first time we connect to them. However, luckily, to get provable eventual consistency, we do not actually need to connect to every peer. As messages are relayed, we only need the eventual connections to form a connected graph, *not* for each peer to eventually connect. Consequently, a value for 'peers_connected_to' can be somewhat smaller than the whole swarm.

Simulating random networks with varying numbers of random connections, the measured probability that the graph is fully connected rapidly approaches 1 as the average number of connected peers passes 2. As the number of edges continues to rise, the distance across the graph (and thus dissemination rate) drops.

```
edges, P(connected), average, stdev
1, 0.05, 57.26, 19.385365614297818
1.1, 0.46, 23.33, 2.549725475418886
1.2, 0.69, 18.1, 1.6763054614240047
1.3, 0.7, 15.08, 1.188949115816149
1.4, 0.8, 13.52, 1.2765578717786399
1.5, 0.91, 12.33, 0.8130805618141443
1.6, 0.9, 11.45, 0.82915619758885
1.7, 0.96, 10.59, 0.8011866199581761
1.8, 0.97, 9.83, 0.6333245613427602
1.9, 0.99, 9.29, 0.4958830507287036
2, 1, 8.72, 0.5306599664568481
3, 1, 6.91, 0.2861817604250792
5, 1, 5.39, 0.48774993593029137
10, 1, 4.59, 0.4918333050943186
20, 1, 4, 0
```

I would suggest using a fixed number of connections per peer in the range 5-10, would effectively gaurantee a fully connected network, and small dissemination rate, without scaling the number of full vector clocks to be sent by very much.

Also note, this design requires storage of vector clocks, so reducing the number of peers connected to also keeps that within acceptable bounds.

A.6 overlapping replication sets

So far, we have analyzed the problem space as if all peers under consideration are replicating the same set of publishers. In some application designs it may make sense for all peers to replicate the same set of feeds, for example, in a task tracking system within a medium sized company or other organization. On the other hand, the really interesting use-cases are ones that scale to millions of users, and so it might not feasible to replicate all their data on the average device, even if you did want to. In secure-scuttlebutt, the target application is a social network. This provides an interesting middle ground, with both a fair amount of overlap and a reasonable expectation of it, since one of primary ways that people meet new friends is by meeting friends of friends. These encounters might be more or less formal, but nevertheless, the chance that any two friends have a number of mutual friends in common is fairly high.

In the most conservative design, it might be desired to replicate only the direct friends "followed" by the user. If the follow graph is known, a set of replication peers can be carefully selected to ensure coverage of all follows. For each feed a remote peer follows that the local peer does not, an feed id and 'IGNORE' will be sent, but after that, subsequent requests for that feed will be skipped.

In the current secure-scuttlebutt design, by default peers replicate their friends, and the friends of their friends. Sampling the actual ssb data, choosing 5 random peers to replicate, and replicating feeds two hops out on the follow graph (friends, and friends of friends), in all samples, all the direct friends of the user were within 2 hop range of the 5 random peers, also on average 75% (TODO: GRAPHS THESE) of friends of friends were replicated by at least one peer. In ssb, since this could be more carefully optimized by selecting peers carefully to maximize coverage, and since request-skipping means we'll only send headers for unreplicated feeds one time, we can just connect to more random feeds and still get acceptable efficiency.

A.7 real-time broadcast

It is obviously desirable that a communication network would carry messages quickly. For human to human text communication, latency within a few seconds is usually sufficient. However, most of the above replication strategies would be unviable with 'poll_frequency' of a few seconds, not to mention, establishing a TCP connection has overhead, and several extra messages must be passed to make that an encrypted TCP connection. So, instead of simple polling, we should have connections with a longer lifespan - when a new connection is formed we exchange clocks and receive any old messages we are mssing, via the above polling algorithms, but then we "stay on the line", and if our peer receives any additional messages they send those too. Thus, we our model becomes *sync then broadcast*.

In the non-gossip models, we must eventually connect to every peer we subscribe to. It would be unviable to hold long-lived connections to every peer, as they may number in the thousands, and the overhead of a each connection would be too much for most user devices. But with gossip, we can connect to just a small number of peers at a time and still receive messages from many peers.

A.8 random connected network

N peers are randomly connected with average K outgoing connections per peer. (outgoing, because each peer randomly chooses to connect to K other peers) as discussed in the previous section, the chance that the network is fully connected rapidly approaches 1 when as K approaches 2, and then the average shortest path between nodes shortens as redundant connections increase. For the network to broadcast a message, the originating peer sends it to all neighbouring peers, and when a peer receives a *new* message, they send it to all their connected peers except the peer they received the message from. Consider a network with 3 peers and 2 connections each. A creates a new message and transmits a message to B and C, and B and C then transmit the message to each other. Thus the message is sent twice by A and once each by B and C. The total bandwidth used by the network is 4. Since A creates the message and there are only two other peers, only the transmissions to B and C are necessary, but B and C don't know that the other already has the message.

Simulating a broadcast in a random network with up to 20 connections per peer, and measuring hops, average hops, messages transferred:

```
|K|peers|hops|avg|msgs|inefficiency|
|-|-----|----|---|----|------------|
|1|1000|14|6.657|999|1|
|2|1000|7|3.657|2981|2.984|
|3|1000|6|2.944|4947|4.952|
|4|1000|5|2.842|6913|6.92|
|5|1000|5|2.605|8861|8.87|
|6|1000|5|2.515|10803|10.814|
|7|1000|4|2.388|12731|12.744|
|8|1000|4|2.361|14671|14.686|
|9|1000|4|2.306|16605|16.622|
|10|1000|4|2.193|18487|18.506|
|11|1000|4|2.201|20357|20.377|
|12|1000|4|2.136|22237|22.259|
|13|1000|4|2.118|24163|24.187|
|14|1000|4|2.118|25993|26.019|
|15|1000|4|2.027|27877|27.905|
|16|1000|4|2.008|29709|29.739|
|17|1000|4|2.046|31567|31.599|
|18|1000|4|1.994|33393|33.426|
|19|1000|4|1.94|35281|35.316|
|20|1000|4|1.933|37135|37.172|
```

(footnote: With 1000 peers and one connection we only need to send 999 messages because the first peer is the author of the message and did not need to send it.)

Note, with more than one connection, number of hops (which is the time taken for the last message to arrive) decreases slowly, but the average case, time for 50% of the network to receive the
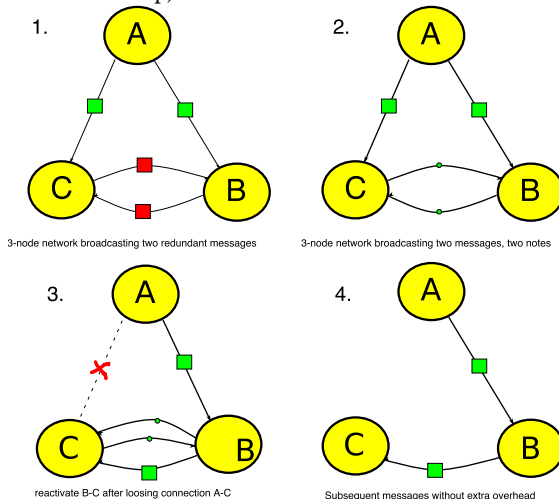
message, decreases much quicker and the (bandwidth) inefficiency increases fastest. With K=2, nearly 3 times as many messages as necessary are sent. and with K=5, nearly 9 times too many messages are sent!

So with a simple flooding design, we pay a lot in bandwidth for reducing latency.

If we were to prune the redundant connections, we could get low latency without bandwidth overhead. However, since a pure spanning tree has no redundancy it's also very fragile. If one connection close to the root of the tree (the originator of a message) fails, all downstream peers will be cut off.

A.9   spanning trees

Epidemic broadcast trees (EBT) is an algorithim to form a spanning tree from a random network, but instead of completely removing redundant connections, they are just moved into a *lazy* or *pull* state. When in the lazy state, only headers (equivalent to vector clock elements) are sent. Which connections are redundant can be detected by each peer observing the order in which they first receive a message, and thereafter observing latency. For example, in the 3 node network discussed in the previous section, A transmits a message to B and C, neither of them have received this message before, so they know that their connection to A is not redundant. Then, they each receive a second copy of the message from B,C so they both know that for messages from A, the connection between B-C is redundant. So, B and C exchange short messages each requesting the other to disable that connection (for messages from A). When A broadcasts another message, B and C receive it directly from A again, but since the redundant connections are disabled, they do not transmit it again. Instead, they only send a short message, equivalent to a vector clock element, to indicate they know this message exists. If later, the connection between A and C breaks, and A broadcasts another message. It will only be received by B. B then sends the short lazy check to C, who then realizes that this is the first they have heard about this message - therefore, B must now be closer to the source than they are. C then sends a message to re-request active transmission of messages from A, and B sends the message to C. (note, re-establishing an active connection takes just one round-trip)

1. 3-node network broadcasting two redundant messages
2. 3-node network broadcasting two messages, two notes
3. reactivate B-C after loosing connection A-C
4. Subsequent messages without extra overhead

EBT still sends redundant data, but the notes sent along the redundant connections are significantly smaller than the messages. Also, if a delay is introduced, it is not necessary to send a note for every message, but just the latest message. If several are received in quick succession, only one note needs to be sent. Also, if a random factor, somewhat greater than round trip time is added, then 50% of the time the same note is received before it is sent.

For example, B and C receive the message from A at approximately the same time, if B decides to wait one second, and C waits two seconds, and the note from B to C arrives in 0.1 seconds, C knows that B already knows about that message, and now does not need to send a note back.

A.10   singleton hub

(footnote: To make the strongest arguement for the performance of EBT + request-skipping, compare it to a fully centralized model.

To this point, most social networks have been implemented along a star shaped network. Essentially one peer that distributes all messages to all peers. If this was designed around a replication protocol, a client would use something like the append-only poll, except the server would remember each client's vector clock at each timestamp, all their subscriptions, and the client would only send the time they last synced. The server would then send all new messages on any of their subscriptions.

On each connection, the client needs to send their last connection time, and the server still has to send each message. If a client polls at low rate, the client sends one header and receives many messages. If the client polls at a high rate, maybe they make one request per message. (Long-lived connections would also help here.)

They would request the sequence number representing their own read feed, on each connection they'd request any messages that have occured since the last connection, but the central server still has to send the messages.

'O(poll_frequency + messages)'

the central server of course, must pay for a lot of resources, bandwidth:

'O(network_clients * poll_frequency + peers * messages)' and connections:

'O(network_peers * poll_frequency)'

If a network is successful, 'network_clients' can easily get very very large: millions or billions of clients.

A.11   conclusion

An idealized centralized network is presented as the best possible in efficiency, yet it only beats our design by a constant factor. Between EBT with a fixed number of peers and request-skipping, we can manage the bandwidth performance, but the main difference is only in vector clock elements, which are very small compared to messages.

In the current secure-scuttlebutt implementation, which uses base64 encoded strings to encode 256 bit public keys plus a base 10 integer, vector clock elements are about 60 bytes, and the average message is 660 bytes (although maximum message is 8kb) so the average message is 11 times bigger than a single vector clock element.

I would expect, that for a typical peer, most messages would be replicated after being offline for a while, so one vector clock element brings in many messages. For messages replicated in real-time, the extra bandwidth used is managed by limiting the number of connections.

The performance of our design is close enough to the optimal centralized system to realistically argue that it's viable at massive scale. In practice, we believe that small difference will easily be made up by all the other advantages by adopting a decentralized system. For example, the significant costs associated with running such a system are now spread around the network participants evenly. With a fully decentralized gossip protocol, peers can join in any topology. If two peers are offline, but nearby each other, it is possible for them to share data directly over bluetooth, wifi, or by directly exchanging physical media. This means secure-scuttlebutt is potentially able to service remote areas of the earth that have not yet received modern infrastructure, as well as areas where that infrastructure is disrupted by warfare or other disasters.