TODO: Anonymize (can we cite the broadcast-only editorial regardless?)
TODO: Update title to “identity-centric” rather than “information-centric”?

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

(May 14, 2019 / last commit was 311c3020b7654e285fae0a09b3ac05357f0b543d)

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.

ACM Reference Format:
Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. 2019. Secure Scuttlebutt: An Information-Centric Protocol for Subjective and Decentralized Community Applications (May 14, 2019 / last commit was 311c3020b7654e285fae0a09b3ac05357f0b543d). In Proceedings of ACM XYZ conference (XYZ'19). ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

XYZ'19, Sep 2019, Place, Country

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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" [15] 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 2014. 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 **??**.

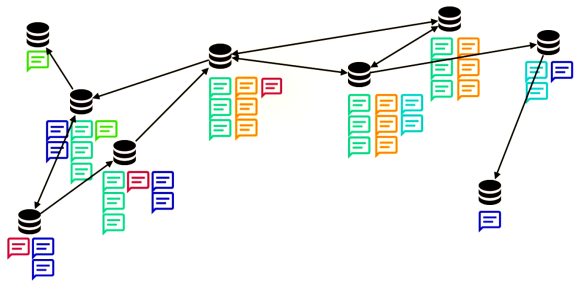

Figure 1: Staltz' "Internet of People" figure, needs to be redrawn such that it can be read in B/W, too.

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 [6, 11, 27–29]

Structure of this paper

…

2 SSB ARCHITECTURE AND PROTOCOL

In SSB, each user is identified by an ed25519 [7] 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 blockchain 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 [22].

The principal function of SSB servers is to connect to other servers and exchange log updates. To do so, they maintain a point-to-point encrypted [5] 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 [19] 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.

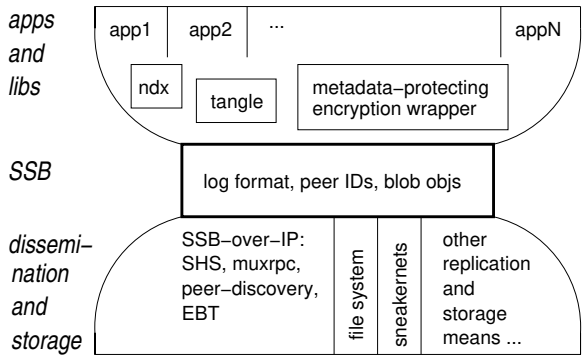

Figure 2: Secure Scuttlebutt's protocol stack.

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.

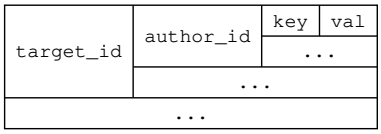

Figure 3: SSB's user directory data structure (after extraction from the logs).

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 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 [18] 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 [10], 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 [24], 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 [21] 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 [13] 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 [12] 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* [16], 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 Running Distributed Applications over Replicated Logs

"Infrastructure-less" distributed application as presented above become possible because central servers can be fully replaced by each peer working on its local set of replicated logs. In this subsection we discuss the particularity of this approach and its constraints.

A common pattern of SSB's applications is that they heavily rely on local database support for organizing the data contained in the logs. Typically a map-reduce strategy is used where the map phase filters the logs and the reduce actions computes the latest application state.

In the user directory application (Sect. **??**), the filtering is done by selecting only `about` entries for a specific target ID

and the reduce action consists in accumulating the latest key-value pairs such that a more recent key-value pair replaces an older one if it was signed by the same author_id. The size of the replicated logs, although locally stored, would lead to very long response times if the map-reduce would be executed at run-time. Instead, almost each application will build indexes and aggressively cache state that was already aggregated. Should ever the indexes become corrupted (e.g. because the user interface app crashed in the middle of a complex indexing step), they can be fully regenerated from scratch, an approach that has also become known as the Kappa architecture [].

An important aspect is whether the reduction step can be done in an incremental fashion by reusing previously computed application state. For example, counting the "likes" that a post receives works fine: incoming log extensions are indexed and if they are of the like type, the counter corresponding to the referenced message in incremented.

Other applications, however, may need a *full* re-evalution of the reduce function each time the underlying index changes. An example for this case is the sequence of post messages: if some peer was added to the set of followed peers, its log gets incrementally replicated, and so are the posts of this peer. For each incoming new post, which may have been written very long ago, one has to insert it at the right place. This problem is shown in Figure **??**.

FIG

A simple solution (adopted in some SSB client software) is to use the timestamp claimed by the author of the post, and in this case one can reuse the existing time-sorted list and insert the new post. However, because an author could lie about the timestamp, the reduce function should do a topological sort based on the causality relationship with other posts and their replies. Insertion into the dependency graph may or may not lead to having to rerun the sort on the whole graph of postings. Clearly, the lack of a central server hosting the reference list of posts and being able to record a post's submissiontime, leads to more complex client software that must prepare for and defend against a broad range of adversarial data found in the logs.

3.4 Synchronization and Eventual Consistency

SSB's basic log replication service synchronizes peers in a consistent way: due to the hash chaining, events (represented by log entries) will be delivered in the order they happened and replica content will be consistent. This does not instantly lead to consistent shared data structures, though, if the corresponding events are spread over multiple logs. Instead, the natural guarantee is that of *eventual consistency* where all peers will see the same reduced application state (if they share the same log set) after sufficient replication progress.

Eventual consistency is the hallmark of Conflict-free Replicated Data Types (CRDTs, see []) which are directly applicable to the SSB setting as they only assume a reliable and (sometimes) in-order delivery of update messages. Potentially, CRDTs permit to implement global data structures featuring eventual consistency without coordination effort (thus are fully scalable). The caveat here is that SSB peers do not necessarily see all involved peers due to their position in the social graph which controls replication and consistency is always modulo that fact. For example, like counts will be eventually consistent with respect to the same set of followed peers but not globally, at least if they are directly counted. Other applications relying on reduction via set union may learn from state that stems from beyond the circle of followed peer. More research is needed to understand the constraints brought by the combination of coordination-less interaction with partial log replication, but SSB's rich set of applications used on a daily basis is a encouraging sign that eventual consistency is a "good enough" basis for real decentralized services.

fo

4 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.

4.1 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.

4.2 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.

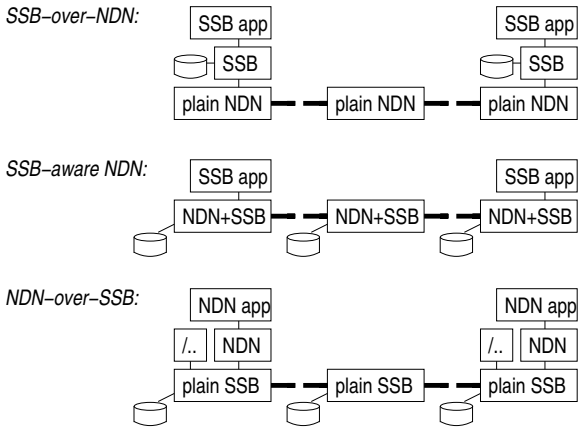

Figure 4: Three different layerings of SSB and NDN, as discussed in this paper

4.3 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. Additionaly, 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.

4.4 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.

4.5 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. 5)? 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.

4.6 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.

4.7 Fixing NDN Routing Stratgies with Replication?

The following scenario highlights a problem in NDN when the same content is provided from multiple sources. The need for such a setting comes either from reliability (cloned repositories) or the concurrent streaming and recording of data streams, as shown in Fig. ??.

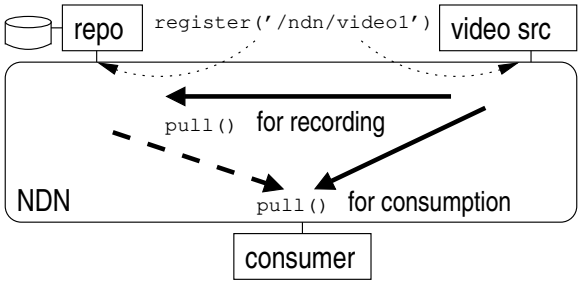

Figure 5: With two NDN sites having registered the same prefix, can the network be smart enough to transparently pull the content from the "appropriate" place?

The figure shows a life video stream that is captured by a repository for archival purposes. The archive registers the same prefix because, in the long run, this is from where the stream can be (re-) fetched. The video source makes its content available under the same name because it permits clients to consume the life stream, without having to wait for the archive to have recorded it. For the client, such a setups should be transparent in the spirit of information-centric networking.

A problem appears when the client temporarily stops streaming. When resuming playout, to which content provider should the network direct the Interest messages? The network would have to know whether the client resumes from where it stoped (then it has to go to the archive site) or whether it wants to reconnect to the life stream. It could also be that the camera goes offline for some time, in which case all Interests should only go to the repo. The problem is that the network has no insights about the different roles (origin, replica) nor the semantics of streams (is a request for frame X to the stream's past, thus only available on the archive side?).

It should be highlighted that this problem does not exist in an SSB setup: content is replicated through push to all interested parties i.e., the archive repo as well as the consumer device, both having declared to the networks their "social" relationship of being interested in the source's content. Consumption, as well as archiving, happens directly from the replicated log without any on-demand protocol in place. In other words, the above network dilemma is an artifact of NDN's pull() model that simply does not exist a push world.

We do not know how to best fix this in NDN. Can a general strategy be engineered for such cases? Or should the three parties engage in a SYNC protocol? Especially the last case would lead to continued probing (using Interest) whether there was a state change, and subsequent pull in fact would inplement the push primitive that NDN tries to avoid. We suggest that this case merits a closer look at a push-only service for NDN.

5 RELATED WORK

The basic ideas behind SSB can be traced back to the nineties: Secure logging [25] and secure relative time-stamping [14]. 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 compactation is what SSB is lacking, but could add (on a per-app basis).
- GIT and DAT
- Kappa and CoucheDB

Also contrast with blockchains? (ssb sidesteps consensus and thus proof-of-whatever)

6 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.

6.1 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 [8] and threaded authentication trees [9] 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.

[20] 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 ommissions 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.

6.2 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.

6.3 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 [23] 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.

6.4 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.

6.5 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.

6.6 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.

7 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.

7.1 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.

7.2 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.

7.3 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.

7.4 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 [17] [30]. 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.

7.5 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 [26] 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?

TODO talk about alternative message addressing (author + seqnum, (a)cyclicity issues)

TODO Naming (!)

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

TODO: This is fairly "close to the metal". An alternative approach could focus on limits of identity-centricity as opposed to icn and point-to-point.

The approach chosen by SSB is not unproblematic. First and foremost, it has strong implications for privacy.

SSB is an inherently pseudonymous system, anonymity is fundamentally incompatile with identity-centric message propagation. Furthermore, the architecture discourages short-lived pseudonyms, favoring the creation of a rather stable network of trust to guide replication. Since all messages

are signed, they are not refutable. Finally, all messages are immutable.

The cocktail of pseudonymity, non-refutability and immutability can be a serious risk to users. Personal details could fuel harassment, political statements could justify persecution, all data could serve as the basis of (future) discrimination. The act of publishing messages is a risky one, reserved to those who hold a lot of privilege. The risks can be reduced by taking care that pseudonyms can not be traced to physical identity, compartmentalizing pseudonyms, using ecryption, and only propagating the messages to trusted parties. Yet the fundamental risk can never be fully mitigated. Consequently, applications must cleary inform users about the peculiarities of the virtual space they participate in.

The pull-only approach of SSB also poses some challenges when onboarding new users. Nobody will pull in their messages, they won't be propagated through the network. The SSB community uses multiple approaches to solve this problem. Pubs can issue *invite codes*. When a user sends such a code to the pub, the pub automatically follows the user, requesting their messages in the process. Invite codes can be distributed out-of-band (commonly through websites, but also on physical pieces of papers). In addition to pubs, the reference server uses lan multicast to discover earby peers. This allows local onboarding where an established user can follow a new user in the same lan. The *manyverse* application uses yet another mechanism, it provides a global DHT to help with onboarding.

The *type* field of SSB messages can be regarded as a global resource without any central coordination regarding its usage. In the worst case, this can lead to multiple applications using the same *type* but in incompatible ways. To mitigate this risk, applications are encouraged to namespace types (comparable to e.g. Java package naming guidelines). An alternative are large random type strings. In particular, a *type* could be the hash of a message that describes the intention behind that *type*.

Non-interoperable *types* are a very tangible symptom of a broader theme, the subjective interpretation of message contents. Taken to an extreme, the community of SSB users could splinter into a multitude of mutually non-understanding fractions that use different kinds of messages or interpretations thereof. Whether the fact that the protocol allows such a divergence is actually a negative property is up for debate — the system has been deliberately designed this way. But this approach certainly runs counter to the philosophy behind more rigid, centralized systems.

A more pragmatic problem is that of scalability: To support replication, full logs have to be stored. While this could be mitigated through mechanisms of partial replication, it is still necessary to store a potentially unbounded number of messages. This boils down to the general fact that all data that is to be propagated needs to be stored *somewhere*. But SSB has stronger expactations about what should be stored than other protocols.

TODO? SLAs (service level agreements, "peer quality") (more of a general p2p problem and not mandated by ssb, thus out of scope here?) While subjectivity and concurrency are great to keep the protocol simple, one could argue that the complexity is merely pushed to the application developer, who is usually the person to *shield* from complexity

TODO These are already covered in the future work section. Mention them anyways?:

- re-keying, forward secrecy - routing and scalability - protocol agility (binary msg format, crypto) - deleting content, gdpr, ephemeral content - multi-device

9 "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)

TODO resistence against sybil attacks (note that the particular mechanism for what to pull in is not hard-coded)

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.

TODO mention the special case of running a dedicated device for backup purposes?

TODO this might be a good place for mentioning encryption?

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. This also leads to sharing of data between applications. For example, the 'about' information (Section **??**) can be reused by all programs, freeing implementors from duplicating work and creating a coherent user experience across apps.

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*!

10 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, stress that there are many settings where SSB is inappropriate due to inherent (?) privacy problems

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] 2015. Designing a Secret Handshake: Key Exchange as a Capability System. http://dominictarr.github.io/secret-handshake-paper/shs.pdf
[6] 2018. https://www.scuttlebutt.nz/
[7] 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.
[8] Ahto Buldas and Peeter Laud. 1998. New linking schemes for digital time-stamping.. In *ICISC*, Vol. 98. 3–14.
[9] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. 2000. Optimally efficient accountable time-stamping. In *International Workshop on Public Key Cryptography*. Springer, 293–305.
[10] Scott Chacon and Ben Straub. 2014. *Pro git (2nd Edition)*. Apress. https://git-scm.com/book/en/v2
[11] Secure Scuttlebutt Consortium. 2012 ?? - 2019. SSB source code. https://github.com/ssbc
[12] Mix Geursen, Piet Irving. 2018. scuttle-poll. https://github.com/ssbc/scuttle-poll
[13] Piet Geursen. 2017. path-gatherings. https://github.com/pietgeursen/patch-gatherings
[14] 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.
[15] Internet Archive. 2018. Decentralized Web Summit 2018, Jul 31 – Aug 2, San Francisco. https://decentralizedweb.net/
[16] Mix Irving. 2018. Scry. https://github.com/ssbc/patchbay-scry/
[17] 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.
[18] Charles Lehner. 2018. Git-SSB: Social Coding on Secure-Scuttlebutt. https://git.scuttlebot.io/%25n92DiQh7ietE%2BR%2BX%2FI403LQoyf2DtR3WQfCkDKlheQU%3D.sha256
[19] 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.
[20] Helger Lipmaa. 1999. *Secure and efficient time-stamping systems*. Citeseer.
[21] Gordon Martin. 2017. ssb-chess. https://github.com/Happy0/ssb-chess

[22] Aljoscha Meyer. 2018. SSB Specification. https://spec.scuttlebutt.nz/feed/messages.html
[23] David Nelson. 2011. *Crypto-Agility Requirements for Remote Authentication Dial-In User Service (RADIUS).* Technical Report.
[24] Noffle. 2016. git-ssb-intro. https://github.com/noffle/git-ssb-intro#push-conflicts
[25] Bruce Schneier and John Kelsey. 1998. Cryptographic support for secure logs on untrusted machines.. In *USENIX Security Symposium*, Vol. 98. 53–62.
[26] Atul Singh et al. 2006. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer.
[27] André Staltz. 2019. SSB Roadmap. https://github.com/staltz/ssb-roadmap
[28] Dominic Tarr. 2018. Notes on a paper for SSB. https://github.com/dominictarr/scalable-secure-scuttlebutt/blob/master/paper.md
[29] ?? Duncan. 2018. Scuttlebutt Protocol Guide. https://ssbc.github.io/scuttlebutt-protocol-guide/
[30] 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).