A Broadcast-Only Communication Model Based on Replicated Append-Only Logs

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

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

This note is about the interplay between a data structure, the append-only log, and a broadcasting communication abstraction that seems to induce it. We identified real-world systems which have started to exploit this configuration and highlight its desirable properties. Networking research should take note of this development and adjust its research agenda accordingly.

CCS CONCEPTS

• **Networks** → **Network design principles**; • **Software and its engineering** → **Publish-subscribe / event-based architectures**; • **Information systems** → *Linked lists*; • **Computer systems organization** → *Peer-to-peer architectures*; *Fault-tolerant network topologies*; • **Theory of computation** → Data structures and algorithms for data management;

KEYWORDS

Global broadcast, append-only logs, distr. systems design, trust.

1 INTRODUCTION

In August 2018 I learned about a project called Secure Scuttlebutt (SSB) – an overlay network created by D. Tarr in 2014 that is tailored for decentralized social applications. This note aims at extracting the gist of SSB, which is that networking with arbitrary data packets could be replaced by networking with coherent data structures. The question is "what are suitable data structures" and what would corresponding networking primitives look like. SSB gives a very radical answer to this, although in retrospect SSB's choice seems natural if one starts from first principles.

In the analog world, propagation of perturbations is the basis of information dissemination: Depending on the physical constraints, a *wavefront* is carrying information omnidirectionally (wireless, water surface) or directionally (in a wire or fiber). However, in computer networks, the unidirectional style has become the dominant and default communication model on which today's network architectures are based, starting from the fact that data packets typically have a source *and* a destination, so do circuits, or we observe wordings like "wireless link" and note that routing algorithms are graph-based, hence link-oriented, without exception.

In a broadcast-centric world, **there is no destination**, just a source; wireless transmission is not a link but naturally implements broadcast; routing is not required because perturbations just propagate infinitely far if not blocked. We should ask ourselves whether broadcast wouldn't be the better *base level* abstraction on which to build other communication services, turning point-to-point communication into a minor (and often inferior) corner case.

Surprisingly, we find a tight relationship between the broadcast model and a data structure, namely append-only logs, that this note wants to highlight. We believe that looking at communication problems from a data structure point of view is an important step towards a better understanding of reliable, synchronized, secure and privacy-preserving operations of distributed applications.

We start by describing the properties of a particular broadcast abstraction which still is somehow close to physical phenomena (potentially making the mapping to the analog realm easier), but is abstract enough such that one can recursively refine and implement this abstraction. A suitable candidate for this abstraction are solitons i.e., information packets that propagate as a solitary wave.

2 LOCAL AND GLOBAL SOLITARY WAVES

Solitons[6] are particle waves or wave packets which travel through space without leaving any disturbance behind them. Such solitary waves have been observed from biology to cosmology and have important applications in communications, for example fiber optics.

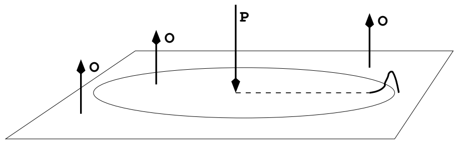

Figure 1: A perturbation triggered by a station P propagates as a solitary wave to all observers O.

The purpose of this note is to take solitons *as the inspiration* for a communication model whose operation is best described in terms of an initial perturbation that propagates in a wave-like form, in all directions. Such a broadcast model shall obey the following three properties:

(1) Each perturbation source has a globally unique identifier that is carried with each perturbation it triggers.
(2) A perturbation and its value eventually reaches all anonymous observers.
(3) All observers sense subsequent perturbations coming from a specific source in the same order.

Using middleware language, the communication primitive having above properties can also be described as reliable, ordered broadcast under an asynchronous communication model. Figure 1 depicts the broadcast-only communication model where one source labeled P triggers a solitary wave that some observers O already have sensed while others still wait for its arrival.

Property (1) can be implemented in the wireless domain by assigning a unique frequency to each station (no need to encode the ID in symbolic form). Similarly, property (3) does not need special implementation effort if all perturbations travel with the same speed and in isolated paths. Fading in wireless systems due to multipath forwarding can be addressed with rake receivers, restoring the soliton-like behavior. How the three properties are implemented is not essential for the model as long as the properties hold.

2.1 Implementing Global Solitary Waves

Leaving temporarily the problem of intermittent connectivity and information loss (due to media interference or node crashes) aside, we look at the state and logic needed for implementing a global solitary wave system based on the forwarding of local solitary waves, first assuming a static network topology and fixed transmission delays.

True to our broadcast-only viewpoint we assume that space is covered with observers called *relays*, capable of picking up solitary waves in one media and propagating them to another media. In a physical implementation these media could be the same where the relay just re-broadcasts an amplified version of the sensed wave. Our goal is to build a global broadcast system relying on the concatenation of many local broadcast domains. The global system should have the same properties that we have defined above and that we have at the local level (see Figure 2). The problem we have to solve is the enforcement of a single "frontier" although a perturbation can reach an observer via different paths or from multiple relays, hence at different points in time.

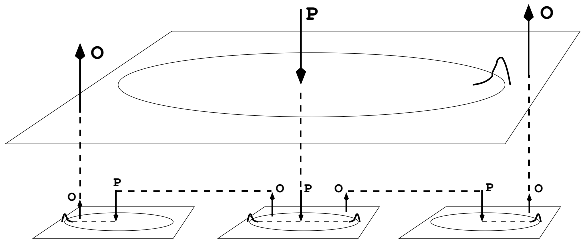

Figure 2: Global solitary waves can be built from local solitary waves forwarded across multiple media, if the solitons carry a suitable reference, permitting to sync on the perturbation front.

We consider a global solitary wave system using flooding: A locally sensed perturbation is (immediately or with some delay) forwarded by every neighbor relay. The common case will be that a relay senses a perturbation several times, but the perturbation should only be processed once for each observer, hence the need for a duplicate suppression mechanism. Such a mechanism is also required for terminating the global wave instead of having it run in circles. Instead of just *assuming* the total order property (3), we add to each perturbation a reference ref that permits to *compute* the order of two perturbations. In a computer system, such references can simply be a per-source logical clock that advances for each triggered perturbation. Alternatively, one can include in a perturbation the hash value computed over the representation of the previous perturbation, thus forming a hash chain. What matters for our purpose is that we can enforce property (3) i.e., total order of perturbations for each source. Having such a reference, and combining it with the perturbation's source ID, we will represent a perturbation as a tuple $\langle src, ref, val \rangle$.

We now sketch the program logic of a relay that bridges two media, assuming a static network and constant propagation times. The relay will keep a `frontier` array of references per source. These frontier variables keep track of the next expected valid perturbation references, as is shown in the following program:

```
Relay_I:      // static network and zero jitter case
  frontier[]  // per source expected next reference

  on_sense(P=<src,ref,val>):
    if frontier[P.src] == P.ref: // filter out duplicates
      forward(P)
      frontier[P.src] = next_ref(P)
      // observer notification of P.val goes here

  next_ref(P):
    return P.ref + 1     // for sequence numbers
    return hash(P)       // for hash chaining
```

With this "filtered forwarding" algorithm (I), a perturbation will be propagated globally and is processed only once and in order for each observer. Note that property (1) enables a flat implementation i.e., we can reuse the local source identifiers at the global level. Each local observer is at the same time an observer at the global level (compare with Figure 2).

Dynamic Network Case

In the relay implementation (I), keeping a per-source reference of the next expected perturbation is sufficient state for building the global broadcast domain on top of local relays, *under the assumption that there is no mobility or change in topology.* Adding a new relay, for example, can instantly lead to a shorter forwarding path such that a perturbation $\langle src, N+2, x \rangle$ is potentially sensed earlier than perturbation $\langle src, N+1, y \rangle$ (e.g., carried over a longer forwarding chain). Also, data mules in settings with intermittent connectivity can lead to old perturbations being delivered after newer perturbations. In these cases, a simple counter or hash reference is not sufficient anymore and relays must pay the "price of asynchronicity" in form of memory, typically bound by the maximum jitter[1]. Our relay algorithm (II) addresses this problem:

```
Relay_II:      // dynamic network and variable delays
  frontier[]   // per source references
  bag          // temporary store for perturbations

  on_sense(P=<src,ref,val>):
    if P.ref >= frontier[P.src] and not P in bag:
      bag.add(P)    // only add new stuff
      while exists Q in bag with frontier[Q.src] == Q.ref:
        forward(Q)
        frontier[Q.src] = next_ref(Q)
        bag.remove(Q)
        // observer notification of Q.val goes here
```

[1]If we knew the maximum delay that a perturbation can obtain, we could limit the size of the bag needed.

The role of the bag is to temporarily store out-of-order perturbations. The while loop will drain the bag for the longest possible "perturbation train" as soon as a valid extension has been sensed for that source's frontier. Note that this algorithm reestablishes order at every relay, forwarding perturbations in strict source order.

Recovering from Loss

Coming back to the problem of lost information, relays must discover that a perturbation was lost and must implement some automatic retransmission mechanism (ARQ), or use forward error correction and rateless coding which we don't consider in this note. If relays do that, we can offer global perturbation propagation also for this general case by arbitrarily stitching together local broadcast domains that conform to the three system properties.

We consider the loss case although one could argue that if the local broadcast media fulfill the three properties, loss should not happen. However, the loss of a perturbation in a media is not the primary concern but the possible crash of a relay which would sink with the stored signal before it could be forwarded.

The following relay algorithm (III) implements a pull mechanism for missed perturbations that complements the (push) broadcast of regular perturbations: relays periodically announce their next expected reference, *asking* all neighbor to (re-) send them any frontier extension ASAP. This is done through broadcasting (this is the only way to communicate in our model) a local AQR perturbation which is then sensed by the relays in vicinity. This request is not propagated: instead, either a neighbor has the requested content (in which case it is returned) or the neighbor waits itself for it and once it receives it, will relay it automatically.

```
Relay_III:   // arbitrary network dynamics, delay, losses
  log[]      // complete perturbation history, per source

  on_sense(P=<src,ref,val>):
    if next_ref(log[P.src].newest) == P.ref:
      forward(P)
      log[P.src].append(P)
      // observer notification of P.val goes here

  on_sense(P=<src,ARQ,ref>):
    // ARQ is a fixed non-reference value
    if exists Q in log[P.src] with Q.ref == P.ref:
      forward(Q)

  on_regular_intervals:
    for all src:
      forward(<src,ARQ,next_ref(log[src].newest)>)
```

There are a few interesting observations for this algorithm (III). Firstly, unlike algorithm (II) where we used a bag to store perturbation that potentially have to be re-broadcast, we use a **log** to keep track of *all* perturbations sensed so far, for every source. This ever-growing data structure is mandated by loss and asynchronicity, i.e. the possibility that some relay can show up which had been cut off for arbitrarily long times. This can also be interpreted with a positive twist by observing that freshly joining relays can "boot" into a running system and catch up by requesting all perturbations that have ever been recorded. Secondly, algorithm (III) is a variation of algorithm (I) where the references kept in `frontier` are now fetched from the logs' newest elements: a perturbation is forwarded only if it extends the frontier for the given *src* (or as a reaction to a *ARQ* request).

So far we have shown how the three desired properties can be maintained for a global broadcast-only system built from an arbitrary mesh of local broadcast systems with the same properties. At the end we are left with relays and observers keeping the full log of perturbations ever emitted by a source. In the next section we show how to use this data structure to implement arbitrary communication patterns at the observer level.

3 INTERACTIVE PROTOCOLS OVER REPLICATED APPEND-ONLY LOGS

Letting two UNIX processes communicate over a pipe can be interpreted as appending to a buffer (file) and reading from the oldest additions. In analogy, interactive protocols can easily be implemented over the replicated append-only logs that are at the core of the relay implementation (III). In this section we first look at the classic `send/recv` primitives but also consider synchronization tasks as well as scalable distributed data types.

Point-to-point Communication

Using the implementation (III) of the previous section, we assign to each communicating party a unique source identifier that is also used as a destination ID. Senders can target specific destinations by generating, for a `send(dest,msg)` call, a $val = \langle dest, msg \rangle$ perturbation which will eventually end up in all observers' log replicas as $\langle src, ref, \langle dest, msg \rangle \rangle$. The observers then watch for new log extensions and filter out the relevant ones (line 9):

```
1  send(src,dest,msg) -> forward( <src,ref,<dest,msg>> )
2
3  ...
4    on_sense(P=<src,ref,val>):
5      if next_ref(log[P.src].newest) == P.ref:
6        forward(P)
7        log[P.src].append(P)
8        if P.val == <dest,msg> and dest == my_ID:
9          recv_upcall(P.src, P.val.dest, P.val.msg)
```

Note that this implementation of `send()` and `recv()` does not suffer from out-of-order messages because of the append-only nature of the (sender's) log.

Multicast, Group Communication and Pub/Sub

The destination selection that is carried out in the point-to-point example above can be used to implement a variety of different communication styles by letting potential destinations filter on other identifiers than their own ID. One can assign unique IDs to different (multi-cast) groups, or different topics of a pub/sub system, and use them as quasi destination. Observers then compare these destination values with their internal set of quasi destinations that encode whether they are a member of a specific group, or have subscribed to some pub/sub channel.

```
pub(topic_id, msg) -> forward( <src,ref,<topic_id,msg>> )

...
  on_sense(P=<src,ref,val>):
    if next_ref(log[P.src].newest) == P.ref:
      forward(P)
      log[P.src].append(P)
      if P.val == <topic,msg> and topic in my_subscriptions:
        consume_upcall(P.src, P.val.topic, P.val.msg)
```

This implementation strategy covers classical social apps like chat rooms in a fully decentralized way: writers publish new chat messages to their own log, tagged with the chat room's ID, and readers watch for such messages in their local log replicas.

Privacy-Preserving Communication

The point-to-point example from above reveals the message to everybody, including sender and receiver ID, but it can be modified to cover fully privacy-preserving communication by encrypting the message such that only the intended receiver(s) can decrypt it (using either a shared secret or asymmetric crypto):

```
private_send(src,dest,msg) ->
             forward( <src,ref,encr(dest_secret,msg)> )
...
  on_sense(P=<src,ref,val>):
    if next_ref(log[P.src].newest) == P.ref:
      forward(P)
      log[P.src].append(P)
      try:
        msg = decrypt(my_credentials, val)
      if success:
        recv_upcall(P.src, my_id, msg)
```

Confidentiality and privacy (for meta-data) is achieved by the replication of the sender's log to *all* observers, leaving an attacker clueless for whom such a message was sent. Real-world implementations may attempt to take shortcuts, as we will discuss in Section 6, hence limit that privacy property. Another limitation is that high-level observations remain possible: e.g. an immediate perturbation event after having received a log extension, although encrypted, has a high chance of being a reply.

Synchronization

The total order of the perturbation logs makes it easy to implement synchronization protocols. A semaphore or lock would be implemented by a process having a source ID to which request messages must be sent. The access token to a specific requester ID is then posted in the semaphore process' log and picked up by the requester after replication. After having worked through the critical section, a release token is put on the requester's log that is then picked up by the semaphore's process.

Letting processes synchronize on an *ensemble* of data elements (e.g., a snapshot whose elements are produced by multiple parties) can be achieved by putting the data elements in each producer's log and collect the references of these items in a manifest (list of references), that is also put in a log. Mutually agreeing on that list (instead of having one peer decide on the snapshot) is more complex, although it is possible (see the next section) if eventual consistency is good enough.

Distributed Data Structures: Conflict-free Replicated Data Types (CRDT) and Trace Algebras

The semaphore or snapshot implementations above have the drawback of introducing single management processes that become a scaling bottleneck if many semaphore or snapshot actions have to be performed in time, as well as being an availability risk and trust hazard. For shared data structures there exists fully decentralized protocols which do not require any central components, hence will scale infinitely, as research in CRDT [12] has shown. This comes with the flavor of (strong) eventual consistency, thus works only with applications that can handle temporary inconsistency.

Append-only logs lend themselves very well for transporting CRDT protocols because the total order of a peer's operations is a direct enabler of so-called *commutative* replicated data types. A simple example is the chat room application that was already mentioned above where posting a message comes with referencing the most recent posts seen. The posted messages, sitting in different logs but referencing each other (and creating global partial order), form a specific directed acyclic graph having only one root node (this data structure is sometimes called a "tangle"): Topological sorting permits to derive a linearized version of the posting sequence from the root to the latest postings that is identical for all participants if the same tie breaking strategy is used and once all log content was eventually delivered to the interested parties – yet no coordination protocol or server was needed.

CRDTs have been adopted in highly-scalable database systems since many years now (e.g. RIAK in 2013, see [14] for a recent historic account). The logs are traces from which application-level data structures are derived: Trace algebras [4] provide the tools to reason about concurrency and event ordering.

Offline and Trustless Communications

Having all logs replicated at every observer permits fully offline and delay-tolerant operations. Application code can tap into the complete set of messages exchanged so far, present derived results to the user and produce followup messages if necessary. New messages are simply appended to that source's log and the log will be replicated, through the global broadcast model, as soon as an observer is online (again). Unlike Delay Tolerant Networking protocols (DTN) that require logic to form special data bundles for requests and replies, what matters in the broadcast-only communication model is that every log extension turns into a perturbation that propagates through the whole network, in an application-agnostic way.

The goal of "decent(tralized)" applications is to eliminate the dependency on central infrastructures and external trust, which can be best characterized by the complete absence of "server not reachable" errors or incidents like "security breach because a hardwired trust assumption was broken" (e.g. root key of DNSSEC was compromised). It suffices that logs are immutable and all entries are signed: from this, applications must compute their own trust, based on what (portion of) logs the end user deems trustworthy. The replicating broadcast network itself could be malicious, e.g. censoring certain perturbation sources in parts of the network. However, because log entries do not care about the way they are transferred (online or sneakernet), it is always possible to ship log entries in the logs of non-blocked sources, in encrypted form. As long as the blocked source's log reaches at least one non-blocked observer, such tunnels can be used to create an "in-lay" network that is not censurable.

4 BROADCAST-ONLY COMMUNICATION IN THE WILD

In this section we briefly review three existing systems that either are implementing the broadcast-only communication model (Secure Scuttlebutt), are structurally equivalent (X.509 PKI), or provide similar services with other techniques (Google's Cloud Pub/Sub service).

4.1 Secure Scuttlebutt (SSB)

Secure Scuttlebutt [16, 18], initiated 4 years ago, is an operational IP overlay peer-to-peer network that implements the relay algorithm (III) of Section 2. In SSB, each source is identified with an ED25519 key pair; Sources manage their append-only log that is cryptographically secured through signed and hash-chained entries; Sources peer with other nodes to replicate the logs of other sources and use sequence numbers to ask for content beyond the local log's height; Applications only consult the locally available log replicas. A dozen such applications, mostly of social networking type, are in use by a growing community of about 10'000 persons at the time of writing.

Because SSB is a running system, some interesting questions have become evident from its operations, which we quickly point out as they also apply to the broadcast-only approach in general. (i) There is no clear understanding how a *Identity Life Cycle* should be organized. One problem is that relays have to learn about new identities and new identities have to learn about relays, both having to also decide whether they should trust each other. In SSB, a social on-boarding mechanism is used to overcome this chicken-and-egg problem. Related to this is the question of garbage collecting inactive IDs and their logs. (ii) SSB has a "social" approach to *Replication Control and Trust*. By following an ID, its log becomes replicated, as well as the logs of that friend's friends. Additional means will be needed to let nodes replicate only portions of the full content which is a tradeoff with privacy. (iii) Forking a log, created by accident or maliciously, leads to network partitioning where only one of the two forks will be visible, per network part. Either the ensemble of peers define a policy to shut out forked logs, or there are mechanisms to let the log owner remediate a fork. (iv) Extracting from (potentially) all local logs the entries that are relevant to some application or data structure is involved and requires considerable indexing effort. Currently, the incremental tracking is part of a monolithic program and internal application-agnostic interfaces still have to emerge.

Despite these open points and shortcomings we consider SSB to be a milestone in communications and distributed system: It has identified a high-level unifying abstraction where networking and distributed applications can meet, with central communications problems already factored out (trust, reliability, synchronization). We resume assessment of SSB in Section 6.

4.2 Public Key Infrastructure (PKI)

The Public Key Infrastructure [3] has become an indispensable pillar of today's Internet. It's main role is trust management through certificates which map real-world entities (some person or other legal entity) to a public key in a way that its correctness can be mechanically traced back to a so-called "Certificate Authority" (CA).

We assert that the PKI has the same logical structure as the Secure Scuttlebutt system. A CA's root key plays the role of a unique source ID in the broadcast-only communication model. Certificates typically carry a serial number: assuming the CAs hand out serial numbers correctly, all certificates signed by a CA form a set of logs (because a CA usually manages multiple root keys).

The PKI does not mandate the replication of all certificates issued by a CA: certificate distribution is linked to the context where they are needed e.g., they are fetched at https connect time. However, root certificates MUST and are replicated in advance, either in web browsers, Java keystores or through operating system distros (CA bundles). While this may lead to a partial replication of all certificate (logs), it was recognized that certain flaws in the PKI system can only be addressed by full replication. Google's "Certificate Transparency" (CT) [11] initiative addresses this problem, namely collecting all certificates in separate logs that can, incidentally like SSB, be replicated through a gossip protocol. Based on these replicated append-only logs, independent third parties (which we call observers in our model) can compute trust properties, i.e. violations of certificate issuing policies [13].

Although not envisaged as an infrastructure to run interactive protocols over it, PKI and CT follow the log replication model in a literal way. In the case of certificates, the total amount of data is small enough to permit worldwide and full replication, at least for some organizations. From a privacy point of view it would be advantageous to replicate the whole data set to every end device (because fetching certificates reveals communication intents), as was pointed out in [13].

4.3 Google Cloud Pub/Sub

The pub/sub concept [5] addresses the need for a wide-area, or even global, event notification system through which distributed applications can coordinate their activities. While it is still a debate (in the Information Centric Networking community) how this could be implemented as a real (low-level) network, pub/sub definitely has been recognized as an essential service at the middleware level. Google implements its *Cloud Pub/Sub* product [8] with very strong guarantees and calls it a secure, durable, highly available and scalable many-to-many messaging system. Cloud Pub/Sub messages are published to "topics" and kept in a persistent message store until they could be delivered to *all* subscribers to that topic (up to a limit of several days). The contract is that once an item has been handed over to Cloud Pub/Sub, nothing can fail.

"Topics" in pub/sub can be seen as "source IDs" of an append-only log, with reliable in-order delivery, plus "log compaction" as an optimization in order to trim the ever-growing logs (as soon as a published item has been delivered to all subscribers). Google's effort to implement this service is considerable, namely collecting all submissions from all publishers to a topic (while SSB is easy: you only have to publish to your own log), then persisting all incoming messages and offering high-availability and durability. This requires consensus-based services like RAFT [15], which internally uses logs, or at least classic write-ahead logs (WAL) as they are used in journaling file systems. In other words: today's advanced communication services like Cloud Pub/Sub rely on logs at various places, but logs have not yet been recognized as a data structure that should be exposed to the communicating parties.

5 RELATED WORK

Abstracting away from events on wires is an old topic in computer science. Agha's Actor model [1] or Gelernter's Linda system [7] introduced mailbox and object store abstractions, which do not refer to hosts anymore, two decades before the advent of Information Centric Networking (ICN). However, the communication model described in this note is closer to networking (than these two systems) and squarely falls into the ICN realm.

Specifically, the naming of items in SSB is literally the naming of DONA [10], which suggested to use a $\langle P : L \rangle$ tuple, P being the principal's cryptographic ID and L a unique label chosen by the principle (SSB uses the ED25519 public key as source ID and globally unique SHA256 values as item references).

Other ICN architectures like NDN [19] have proposed hierarchically organized free-form names and a strict pull model, while the communication model of this note is purely push-based. One could be tempted to say that the local frontier advertisements of algorithm (III) represent a pull request (for new content), but even these are implemented as unsolicited broadcasts, and they are *not* propagated. The consequences of NDN's free-form names is another area of difference where the monotonic sequence of log items automatically provides base-level synchronization. In NDN, considerable synchronization complexity (e.g. VectorSync [17]) comes from the possibility of un-disciplined one-off production and requesting of data with arbitrary names, which is inherited from the philosophy of "independent datagrams", typical for link-based networking.

Relating append-only logs to (transport) protocols is not as alien as it seems at first sight, which we demonstrate with TCP. TCP's sequence number space gives a position to each byte of a (conceptual) log although it is called a stream. Bytes can only be appended in TCP: any segment carrying the wrong frontier label will be discarded by the TCP protocol entity in the same way the relay algorithms (I) to (III) enforce that incoming perturbations are forwarded only when extending the wave frontier. Creating a new stream is brittle because there is no trust anchor (TCP segments are not signed). TCP's solution is to pick a random start sequence number and establish the trustworthiness of that stream anew, for every connection (using TLS and the PKI infrastructure, but see the session resumption in tcpcrypt [2]), while in this note's model the trust relationship persists because of always using the same log. The sliding window behavior is an optimization for trimming the log, as TCP's mindset is delivery-oriented (single bytes are shipped) instead of synchronization-oriented logs (both parties have a copy of the same stream/log content).

Finally we point out that Secure Scuttlebutt is not the only project that has started to build applications on top of *independent* hash chain-like data structures (as one way to implement logs), see e.g. Holochain [9].

6 DISCUSSION

Full replication of all logs may work for small enough communities (10^3) and low source rates but currently seems out of reach as a global network service for communication among 10^9 or more source identifiers and high data volumes. The three limiting factors are bandwidth, memory, and their combination.

Bandwidth-wise, implementing a global broadcast-only model means to stack all source production rates and to deliver the sum of them to every observer. Memory-wise, the every-growing logs must be persisted, on each relay, in order to cope with lost signals and intermittent connectivity. Finally, there is a blatant mismatch in terms of bandwidth and latency between these two technology fields. It would be wrong to construe from all this that the communication model is wrong. On the contrary one should explore how close the broadcast-only model can be approximated and which tradeoffs have to be made to still benefit from the desirable properties.

In SSB, the hope is that by exploiting the social graph (peers typically communicating with a low number of peers only, often living in close geographic neighborhood), relays can limit log propagating to a few regions and within the social neighborhood. Log compaction would have to be introduced in order to curb memory requirements. Both measures have unfortunate negative consequences for privacy (revealing the social graph) as well as security (old entries in the hash chain cannot be fully verified anymore). Google Cloud Pub/Sub shows some possible trade-off: split a source's content into topics, have explicit subscription (and giving up meta-data privacy), purge log entries immediately after complete delivery and limit the retention duration to 7 days, among others, but remains a middleware service.

Our more network-centric suggestion is to explore the spectrum between the (extreme) replicated log approach described in this note and the (extreme) point-to-point approach of today's Internet. In the same way that replication concepts have been added to the Internet (e.g. Content Delivery Networks) one can add link concepts to the broadcast-only model, for example aggregation tunnels with transient source IDs that bundle replication traffic across the atlantic and have a log height of only a few bandwidth-delay products. Moreover, such an approach would fit well the still futuristic light-path switching concept as well as batch- instead of stream-oriented data delivery.

7 CONCLUSIONS

It is too early (and not the point of this note) to tell whether soliton-like technologies are extensible to global scale and could be directly coupled with application-level data structures, or not – this really would be science fiction. But if there is such a "fiber-to-the app" future, replicated append-only logs stick out as an interesting candidate around which network functionality should be organized. Shifting research attention to global broadcast-only networks is one recommendation that this note wants to give as well as raising awareness for a new generation of distributed applications which expect this communication model. Tarr's Secure Scuttelbutt system is just the beginning.

ACKNOWLEDGMENTS

The idea of SSB was developed by Dominic Tarr who also explained to me its rationale at several occasions. I also gratefully acknowledge the many helpful comments by Erick Lavoie, Claudio Marxer, Aljoscha Meyer and Christopher Scherb.

REFERENCES

[1] Gul Agha and Carl Hewitt. 1985. Concurrent Programming Using Actors: Exploiting large-Scale Parallelism. In *Proc. 5th Conference on Foundations of Software Technology and Theoretical Computer Science*. 19–41. https://doi.org/10.1007/3-540-16042-6_2

[2] Andrea Bittau, Daniel Giffin, Mark Handley, David Mazieres, Quinn Slack, and Eric Smith. 2018. *Cryptographic protection of TCP Streams (tcpcrypt)*. Internet-Draft draft-ietf-tcpinc-tcpcrypt-15. IETF Secretariat. https://www.ietf.org/internet-drafts/draft-ietf-tcpinc-tcpcrypt-15.txt

[3] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280. (2008). https://doi.org/10.17487/RFC5280

[4] Jerry R. Burch. 1992. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-92-179.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and Evaluation of a Wide-area Event Notification Service. *ACM Trans. Comput. Syst.* 19, 3 (Aug. 2001), 332–383. https://doi.org/10.1145/380749.380767

[6] Thierry Dauxois and Michel Peyrard. 2010. *Physics of Solitons.* Cambridge University Press.

[7] David Gelernter. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985), 80–112. http://doi.acm.org/10.1145/2363.2433

[8] Google Cloud Pub/Sub 2018. Google Cloud Pub/Sub. (2018). https://cloud.google.com/pubsub/docs/overview.

[9] Holochain – Scalable Distribute Computing 2019. (2019). https://holochain.org/ (retrieved Feb 2019).

[10] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A Data-oriented (and Beyond) Network Architecture. *SIGCOMM Comput. Commun. Rev.* 37, 4 (2007), 181–192. https://doi.org/10.1145/1282427.1282402

[11] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. (2013). https://doi.org/10.17487/RFC6962 See also https://www.certificate-transparency.org/.

[12] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. 2009. CRDTs: Consistency without concurrency control. (2009). http://arxiv.org/abs/0907.0929

[13] Wouter Lueks and Ian Goldberg. 2015. Sublinear Scaling for Multi-Client Private Information Retrieval. In *Proc 19th Financial Cryptography and Data Security Conference.* 168–186. https://doi.org/10.1007/978-3-662-47854-7_10

[14] Christopher Meiklejohn. 2019. Applied Monotonicity: A Brief History of CRDTs in Riak. (2019). http://christophermeiklejohn.com/erlang/lasp/2019/03/08/monotonicity.html (retrieved Apr 2019).

[15] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc USENIX Annual Technical Conference.* 305–319. https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

[16] Secure Scuttlebutt Project 2018. (2018). https://www.scuttlebutt.nz/ (retrieved Feb 2019).

[17] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. 2017. VectorSync: Distributed Dataset Synchronization over Named Data Networking. In *Proc. ACM Conference on Information-Centric Networking (ICN '17).* 192–193. https://doi.org/10.1145/3125719.3132106

[18] Dominic Tarr. 2018. Scalable Secure Scuttlebutt. (2018). https://github.com/dominictarr/scalable-secure-scuttlebutt/blob/master/paper.md (retrieved Feb 2019).

[19] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73. https://doi.org/10.1145/2656877.2656887