Second Assignment

Antonio Bucchiarone

Fondazione Bruno Kessler, Trento – Italy
bucchiarone@fbk.eu

02 December 2020

Context

- Interplay between a *data structure*, the append-only log, and *the broadcasting communication abstraction.*

- **Context:** *Secure Scuttlebutt (SSB) project*
  - *https://ssbc.github.io/scuttlebutt-protocol-guide/*
  - *https://handbook.scuttlebutt.nz/*
  - *https://github.com/ssbc/ssb-server*

- An overlay network that is tailored for decentralized social applicat*ions*

- Networking with arbitrary data packets could be replaced by networking with coherent data structures.
- *What are suitable data structures?*
- *What would corresponding networking primitives look like?*

Middleware for Social Applications

- The infrastructure is provided by participants.
- They are developed by open-source communities.
- Scuttlebutt has pioneered the use of replicated *authenticated single-writer append-only logs:*
  - Chains of ordered immutable events specific to each participant.
  - Replicated by gossip algorithms that are driven by social signals.
- Two **gossip models** that can be used for replication:
  - ***Open model*** that works best in small and trusted groups.
  - ***Transitive-interest model*** that scales to thousands of participants.

Middlewares for Social Applications

- Minimal infrastructure requirements beyond participants' devices.
- *Hypercore: to share large scientific datasets between research teams.*
- *Secure-Scuttlebut (SSB) is actively used for social applications*
  - *Blogging and code development.*
  - *Active open-source mobile clients.*
- *Both projects pioneered the use of replicated authenticated single-writer append-only logs.*
  - Chains of ordered immutable events, similar to blockchains.
  - Specific to each participant and without the need for global consensus.
- *SSB replicates logs using a gossip protocol*
  - By only transmitting the latest missing events between pairs of replicas.

SSB vs other Gossip Algorithms

- SSB has less parameters to tune
  - **No temporary** buffer size;
  - **Logs are persisted** because storage is abundant and cheap;
  - **No retransmission**, events are exchanged over reliable channels when found missing from meta-data;
  - **No fan-out number of neighbours** to contact

- *The result is a simple yet practical messaging middleware.*

SSB Concepts

- SSB organises events in logs, that are replicated between stores.
- **Events** may represent a user action or the result of processing operations.
- **Logs** organise events in a data structure for efficient replication and integrity.
- **Stores** hold many replicated logs either in the storage of an active process, or passively on a storage device.
- Stores are connected over reliable communication channels
  - E.g., TCP connections over the Internet or USB connections on a local machine.
- Many data formats and communications protocols can be used.
  - We can abstract from this decision.

Event

- ***id**: is the publicKey of the creator*
- ***previous:*** is the hash of the previous event in the log including the signature. Null if none
- **index:** *(sequence number) is the position of the event in the log*
- **content***:* is defined by applications
- **signature**: is the cryptographic signature of id, previous, index, and content, obtained with the privateKey that corresponds to the publicKey.

- All fields except content represent the meta-data about events.
- They enable efficient replication and integrity of the event chain

Log

- A **log** is an identifier id associated to a sequence of events,
  - An empty log is *id: [ ]* and a log with two events is
  - *id: [(id,null,0,..,..),(id,hash0,1,..,..)] with the following operations.*

Table 1. Authenticated single-writer log operations

$log \leftarrow create(publicKey)$	create a log from a public key
$log \leftarrow log.append(content, privateKey)$	extend the log with a new event created locally (as owner) from *content*
$log \leftarrow log.update(events)$	extend the log with the subset of compatible *events* previously created remotely
$events \leftarrow log.get(start, end)$	get the set of events with index included between *start* and *end* (can be the same)
$index \leftarrow log.last$	get the index of the last event stored locally
$id \leftarrow log.id$	get the id (publicKey) of the log

- All operations are implemented to ensure the log is,
  - *Secure*
  - *Monotonic*
  - *Linear (total order)*
  - *Single Writer*
  - *Connected*

Store and Frontier

- A **Store** is a set {log1,…} that represents the logs stored locally.

- Adding or removing logs
  - Direct user actions, or
  - Indirectly the result of operations triggered by new events added to logs.

- A **frontier** is a set {(log_i.id, log_j.last),..} that represents the latest known indexes about logs in a store.

- The *difference between two frontiers* represents the new events in one store that are not yet in the other.

Store and Frontier

- Two frontiers may differ in the logs they contain because two stores may contain only partially overlapping sets of logs

- Stores and Frontiers are linked to events and logs by the following operations

$store \leftarrow store.add(log)$	add a log to the store
$store \leftarrow store.remove(id)$	remove the log with *id* from the store
$log \leftarrow store.get(id)$	get the log with *id* from the store (if present)
$ids \leftarrow store.ids$	get the set of ids of the logs in the store
$frontier \leftarrow store.frontier(ids)$	get the current *frontier* of the store only for *ids*
$events \leftarrow store.since(frontier)$	get the set of *events* that happened after *frontier*
$store \leftarrow store.update(events)$	update the logs in *store* with *events*

Updating

- Two stores in different locations may diverge because new events have been added locally.

- Updating is the process of propagating the new events in one store not present in the other.

Algorithm 1 *Update(store, store′)*: update *store* and *store′* with missing new events present in the other.

$frontier \leftarrow store.frontier(store.ids)$
$frontier' \leftarrow store'.frontier(store.ids)$
$news \leftarrow store.since(frontier')$
$news' \leftarrow store'.since(frontier)$
$store \leftarrow store.update(news')$
$store' \leftarrow store'.update(news)$

Private Content

- *SSB uses Ed25519 key pairs both for signature and encryption.*

- *Access to content is restricted by encrypting it using the* publicKey of recipient(s) using a private-box, a scheme that hides the recipients, their number, and the content.

Open Gossip

- *The goal of this model is to maximize the diffusion of events by replicating them in all participants.*

Algorithm 2 Open Gossip

```
loop
    Randomly pick store and store' from Participants
    for id' in store'.ids − store.ids do store.add(create(id'))
    for id in store.ids − store'.ids do store'.add(create(id))
    Update(store, store')
end loop
```

Transitive-Interest Gossip

- Conversations between many participants, are spread over multiple logs.
- To ensure participants receive all messages, the logs in the transitive graph of interests should therefore be replicated.
- Additionally, this may serve to discover new interesting people to follow
  - *Ex: Friend may introduce a friend to another one.*
- The logs participating in the gossip algorithm propagate the interest/disinterest information.

Algorithm 3 Transitive-Interest Gossip

▹ *Participants* abbreviated *P* and *store* abbreviated *st*.

loop
    Randomly pick *(id,st)* and *(id,st')* from *P*
    for *f* **in** *followed(st,id)* – *st.ids* **do** *st.add(create(f))*
    for *b* **in** *blocked(st,id)* ∩ *st.ids* **do** *st.remove(b)*
    ... ▹ Same for store *st'*
    Update(st, st')
end loop

Second Assignment

- **Deadline:** Monday 4, January 2021 - 6pm
- **Reference paper available on my website**

1) **PDF document** reporting all the computational analysis of the implemented algorithm using the Simulator and a small description of the model designed (i.e., a tutorial to execute the model)
2) **A GitHub Repo** containing :
   i. a README file that describes the members of the project and a summary of the implemented algorithm.
   ii. Source code of the simulation.
   iii. A JAR file of the model installer.