Gossip for Data Dissemination

References:

- Any serious recent distributed systems/P2P book 😄
- Slides partially based on http://disi.unitn.it/~montreso/ds/handouts/05-epidemic.pdf

XEROX in 1987

- Database replicated at thousands of nodes heterogeneous, unreliable network
- Independent updates to single elements of the DB are injected at multiple nodes
- Updates must propagate to all nodes or be supplanted by later updates of the same element
- Replicas become consistent after no more new updates
- Assuming a reasonable update rate, most information at any given replica is “current”

AMAZON in 2015

- Amazon uses a gossip protocol to quickly spread information throughout the S3 system
- Amazon's Dynamo uses a gossip-based failure detection service
- The basic information exchange in BitTorrent is based on gossip

Basic Assumptions

- System is **asynchronous**
  - No bounds on messages and process execution delays
- **Processes** fail by **crashing**
  - Stop executing actions after the crash
  - We do not consider Byzantine failures
- **Communication** is subject to **benign failures**
  - Message omission
  - No message corruption, creation, duplication

Data Model

- We consider a database that is replicated at a set of n nodes $\Pi = \{p_1, \ldots, p_n\}$
- The copy of the database at node p_i can be represented by a time-varying partial function:

$$value_i : K \rightarrow V \cdot T$$

where:

- K is the set of keys
- V is the set of values
- T is the set of timestamps

- The update operation is formalized as:

$$value_i(k) \leftarrow (v, now())$$

where *now*() returns a globally unique timestamp

Goal

- The goal of the *update distribution* process is to drive the system *towards consistency*.
- Definition (**Eventual consistency**)

  If no new updates are injected after some time *t*, eventually all correct nodes will obtain the same copy of the database:

$$\forall p_i, p_j \in \Pi, \forall k \in K : value_i(k) = value_j(k)$$

- For simplicity, we will assume a single key, i.e., $value_i(k) = value_i$

Best Effort

- What happens if the sender fail "in between"
- What happens if messages are lost?
- What is the load of the sender?

Best effort protocol executed by process p_i

---

upon $value_i \leftarrow (v, \textit{now}())$ **do**
  foreach $p_j \in \Pi$ **do**
    send $\langle \text{UPDATE}, value_i \rangle$ **to** p_j

upon receive $\langle \text{UPDATE}, (v, t) \rangle$ **do**
  if $value_i.time < t$ **then**
    $value_i \leftarrow (v, t)$

Anti-Entropy

- Every node regularly chooses another node at random and exchanges contents, resolving differences.

Anti-entropy protocol executed by process p_i

---

repeat *every* Δ *time units*
  Process $p_j \leftarrow \mathsf{random}(\Pi)$
  `/* exchange messages to resolve differences */`

Push

Anti-entropy PUSH protocol executed by process p_i

repeat *every* Δ *time units*

  Process $p_j \leftarrow \mathsf{random}(\Pi)$

  send $\langle \text{PUSH}, value_i \rangle$ **to** p_j

upon receive $\langle \text{PUSH}, (v, t) \rangle$ **do**

  if $value_i.time < t$ **then**

    $value_i \leftarrow (v, t)$

Pull

Anti-entropy PULL protocol executed by process p_i

repeat *every* Δ *time units*

- Process $p_j \leftarrow \mathsf{random}(\Pi)$;
- **send** $\langle \text{PULL}, p_i, value_i.time \rangle$ **to** p_j

upon receive $\langle \text{PULL}, p_j, t \rangle$ **do**

- **if** $value_i.time > t$ **then**
  - **send** $\langle \text{REPLY}, value_i \rangle$ **to** p_j

upon receive $\langle \text{REPLY}, (v, t) \rangle$ **do**

- **if** $value_i.time < t$ **then**
  - $value_i \leftarrow (v, t)$

Push-Pull

Anti-entropy PUSH-PULL protocol executed by process p_i

repeat *every* Δ *time units*

- Process $p_j \leftarrow \mathsf{random}(\Pi)$;
- **send** $\langle \text{PUSH-PULL}, p_i, value_i \rangle$ **to** p_j

upon receive $\langle \text{PUSH-PULL}, p_j, (v, t) \rangle$ **do**

- **if** $value_i.time < t$ **then**
  - $value_i \leftarrow (v, t)$
- **else if** $value_i.time > t$ **then**
  - **send** $\langle \text{REPLY}, value_i \rangle$ **to** p_j

upon receive $\langle \text{REPLY}, (v, t) \rangle$ **do**

- **if** $value_i.time < t$ **then**
  - $value_i \leftarrow (v, t)$

What if…

- Nodes join/leave?
- Nodes crash suddenly?
- *Up to now*:
  - Nodes have **full view** of the network
  - Each node periodically “gossips” with a random node, **out of the whole set**
- *From now on*:
  - Nodes have a **partial view** of the network
  - The partial view is dynamic, reflecting nodes joining/leaving
  - Each node periodically “gossips” with a random node, **out of its partial view**
- Idea:
  - Nodes gossip with their neighbors about... other neighbors!
  - Old nodes are removed / new nodes are inserted
  - Random shuffling of views

Dynamic Gossiping

- Each node has a view containing ***C* neighbors**
- Each node **periodically** contacts a neighbor
- They **exchange** their views
- The **neighbor descriptor** of node *p* contains
    - The address needed to communicate with *p*
    - *Timestamp* information about the age of the descriptor
    - *Additional information* that may be needed by upper layers
- How to deal with **failed neighbors**?

Dynamic Gossiping

Generic anti-entropy protocol executed by process p

upon initialization do
- $view \leftarrow$ descriptor(s) of known nodes

repeat *every* Δ *time units*
- Process $q \leftarrow \mathsf{select}(view)$
- Message $m \leftarrow \mathsf{request}(view, q)$
- **send** $\langle \text{REQUEST}, m \rangle$ **to** q

upon receive $\langle \text{REQUEST}, m \rangle$ **do**
- Message $m' \leftarrow \mathsf{reply}(m.view, m.q)$
- **send** $\langle \text{REPLY}, m' \rangle$ **to** q
- $view \leftarrow \mathsf{merge}(view, m.view, m.q)$

upon receive $\langle \text{REPLY}, m \rangle$ **do**
- $view \leftarrow \mathsf{merge}(view, m.view, m.q)$