Gossip-based protocols

Where we were

- Programmers face problems in building distributed applications
- Fundamental problems
  - Consensus
  - Atomic Broadcast / Multicast
  - Group membership
- Isis Toolkit [Birman, van Renesse et al.]

Where we are

Scalability

- Database replicated at thousands of sites
- Network is slightly unreliable
- Point-to-Point communication abstraction
- Crash failure model

Setup

Setup

- Database replicated at thousands of sites
- Network is slightly unreliable
- Point-to-Point communication abstraction
- Crash failure model
- Updates injected at a single site
- Updates must propagate to all other sites*
- Want contents of all replicas to be identical if updates stop and system left alone

Notation

- $S$ is a set of $n$ sites (replicas)
- $K$ is a set of keys
- $V$ is a set of values
- $T$ is a set of timestamps (totally ordered)
- For any site $s$ and key $k$,
  
  $s.\text{ValueOf} : K \rightarrow (V \times T)$

More notation

- Pretend there is only one key
  
  $s.\text{ValueOf} \in (V \times T)$

- Consistency definition
  
  $\forall s, s' \in S : s.\text{ValueOf} = s'.\text{ValueOf}$

- To update the database with value $v$ at time $t$
  
  $s.\text{ValueOf} := (v, t)$

Direct mail

Idea: If an update is injected at site $s$, then $s$ mails the update to every other site in $S$

```
Upon an update at site $s$:
  for each $s' \in S \setminus \{s\}$ do
    send (Update, s.\text{ValueOf}) to $s'$
  endloop

Upon receiving (Update, (v,t)):
  if $s.\text{ValueOf}.t < t$ then
    $s.\text{ValueOf} := (v, t)$
  endif
```

Weakness: send is not reliable what if site crashes?
**Anti-entropy**

Idea: Every site regularly chooses another site at random and exchanges database contents with it to resolve differences.

Each server $s$ periodically executes:

```plaintext
for some $s' \in S \setminus \{s\}$ do
  ResolveDifference(s, s')
endloop
```

---

**Push vs. pull analysis**

Let $p_i$ be the probability that a site still has not been updated by the $i^{th}$ try at anti-entropy.

For large values of $n$:

- Push: $p_{i+1} = p_i e^{-1}$
- Pull: $p_{i+1} = (p_i)^2$ Converges much faster for small $p_i$

---

**Example using pull mechanism**

- Guaranteed to eventually propagate update to everyone with probability 1
- Anti-entropy infects everyone in $O(\log n)$ for uniformly chosen sites
- Backup mechanism for direct mail
- Weakness: must go through entire database
Epidemic terminology

- Resilient to unreliable communication
- Anti-entropy is a simple epidemic
- Complex epidemics
  - Sites can become “cured”
  - Terminology: susceptible, infective, removed
  - Strengths: sites do not mail everyone and do not have to enumerate entire database
  - Weakness: some may be left susceptible

Rumor mongering (informal)

- All sites start out susceptible
- When a site s receives a new update, it becomes infective
- s periodically chooses another site s’
- If s’ does not know the rumor, then it receives the update and also becomes infective
- If s’ already knows the rumor, then s becomes removed with some probability

Rumor mongering protocol

For a site s:
let L be a list of (initially empty) infective updates
periodically:
  for some s ∈ S \ {s} do
    for each update u ∈ L
      send u to s'
      if s’ already knows about u then
        remove u from L with probability 1/k
    endloop
upon receiving new update u:
  insert u into L
Analysis of rumor mongering

\[ \frac{ds}{dt} = -si \]

\[ \frac{di}{dt} = +si - \frac{1}{k}(1-s)i \]

\[ \frac{di}{ds} = - \frac{k+1}{k} + \frac{1}{ks} \]

\[ i(s) = \frac{k k+1}{k k} \ln \frac{1}{s} \]

\[ c = \frac{k+1}{k} \]

\[ s = e^{-(k+1)(1-s)} \]

Rumor mongering facts

- Expected fraction of susceptible sites
  \[ s = e^{-\frac{(k+1)}{k}(1-s)} \]
- Back up mongering with anti-entropy
- Mongering vs. direct mail
  - Redistribution
  - Consider case when half of sites receive update
  - Old rumors die fast

Death and its consequences

- Replace deleted item with a death certificate = (NIL, t_{now})
- Provided no further updates, a death certificate eventually “deletes” all copies of an item…but when?
- Problem: what if a single site is down?

Death certificates

- Death certificate contains two values
  - t = time of deletion
  - t_{1} = threshold value, all servers discard death certificate after time t + t_{1}
Dormant death certificates

- Death certificate contains four values
  - $R$ – set of sites that keep a dormant death certificate after $t + t_1$
  - $t$ – time of deletion
  - $t_1$ – threshold value, all servers not in $R$ discard death certificate after time $t + t_1$
  - $t_2$ – all servers discard the certificate after $t + t_2$

---

Dormant death certificates

- Death certificate contains five values
  - $R$ – set of sites that keep a dormant death certificate after $t_a + t_1$
  - $t$ – time of deletion
  - $t_a$ – time of activation
  - $t_1$ – all servers not in $R$ discard certificate after $t_a + t_1$
  - $t_2$ – all servers discard the certificate after $t_a + t_2$
Bimodal multicast


Class I – Strong reliability

- Properties: Agreement, validity, termination, integrity
- Costly protocols
- Limited scalability
- Unpredictable performance under congestion
- Degraded throughput under transient failures (full buffers and flow control)

Class II – Best effort reliability

- “If a participating process discovers a failure, a reasonable effort is made to overcome it.”
- Better scalability than Class I protocols
- Difficult to reason about systems without concrete guarantees

War games
Bimodal multicast claims

- Scales well
- Provides predictable reliability and steady throughput under highly perturbed conditions
- Very small probability a few processes deliver
- High probability almost everyone delivers
- “Vanishingly small probability” in between

A problem to our solution

- Applications that need high throughput (frequent updates) and can tolerate small inconsistencies
- Examples: health care, stock trading, streaming data

System assumptions

- At least 75% of healthy processes will respond to incoming messages within a known bound
  - 75% of messages will get through the network
  - Crash failures

Protocol details

- Consists of two subprotocols
- Unreliable multicast (i.e. – IP multicast)
- Anti-entropy that operates in rounds
  - Each round contains two phases
  - Phase 1: randomly choose another process and send message history to it
  - Phase 2: upon receiving a message history, solicit any messages you may be missing
Bimodal multicast example

What’s new about this?

- To save space, keep a message for anti-entropy only for a fixed number of rounds
- Processes try to achieve a common prefix
- If a process cannot recover a message, it gives up and notifies application

Optimizations

- Reducing unnecessary communication
  - Service only recent solicitations
  - Retransmission limit
  - Most recent first transmission
- Random graphs for scalability
- Multicast some retransmissions

Recovery from delivery failures

- In previous protocols, a lagging process could drag the system down
- In bimodal multicast, a lagging process is effectively partitioned from the rest of the system
  - Do nothing
  - Maintain a few very large buffers
  - Employ a state transfer technique
Throughput results

- Eight processes running on an SP2
- Data rate = 75.7 KB multicasts per second
- Two cases
  - Sleep a process for 100 ms with .05 probability
  - Sleep a process for 100 ms with .45 probability
Lightweight Probabilistic Broadcast


Bimodal Multicast

- Scalability addressed with respect to reliability and throughput
- Processes knew entire membership set

Probabilistic Membership

- Each process has a view of \( l \) processes it believes are members
- Each buffer \( b \) has at most \(|b|_m\) elements
  i.e. \(|\text{view}|_m = l\)
- Piggyback membership updates on each gossip message

Setup

- Set of processes \( \{p_1, p_2, \ldots\} \) with distinct identifiers
- Unreliable point-to-point network
- Processes join and leave dynamically
- Two kinds of messages
  - Broadcast messages (events)
  - Gossip messages
    (events, membership updates)
Broadcast message

- **id** = uniquely identifies each message as well as the sender
- **event**

Gossip message

1. **events** = Set of all events received for the first time since the last outgoing gossip message
2. **eventIDs** = Set of all eventIDs for messages received by this process
3. **subs** = Set of processes “currently” joining
4. **unsubs** = Set of processes “currently” leaving

Process variables

Each process maintains 6 variables

1. **events** : set of events received for first time since last gossip
2. **eventIDs** : set of eventIDs received
3. **subs** : set of processes “currently” joining
4. **unsubs** : set of processes “currently” leaving
5. **view** : set of “current” members
6. **retrieveBuf** : set of eventIDs to retrieve

Broadcast reception

Upon receipt of broadcast (id, event)

- **events** := events ∪ {event}
- **eventIDs** := eventIDs ∪ {id}

Gossip transmission

**periodically**

- **let** gossip be a new gossip message
- gossip.events := events
- gossip.eventIDs := eventIDs
- gossip.subs := subs ∪ {p_i}
- gossip.unsubs := unsubs
- choose F random members t_1, t_2, ..., t_F ∈ view
- for all j ∈ {1..F} do
  - send gossip to t_j
- **events** := ∅
Upon reception of gossip

(phase 1: update unsubscriptions)
for all unsub ∈ gossip.unsubs do
  view := view \ {unsub}
  subs := subs \ {unsub}
while |unsubs| > |unsubs|m do
  view := view \ {unsub}
  subs := subs \ {unsub}
unsubs := unsubs

(phase 2: update subscriptions)
for all newsub ∈ gossip.subs \ {p} do
  if newsub ∈ view then
    view := view \ {newsub}
    subs := subs \ {newsub}
while |view| > i do
  target := random element in view
  view := view \ {target}
  subs := subs \ {target}
while |subs| > |subs|m do
  remove random element from subs

(phase 3: update events)
for all e ∈ gossip.events do
  if e.id ∈ eventIDs then
    events := events \ {e}
    DELIVER(e)
    eventIDs := eventIDs \ {e.id}
  for all id ∈ gossip.eventIDs do
    if id ∈ eventIDs then
      retrieveBuf := retrieveBuf \ {id}
while |eventIDs| > |eventIDs|m do
  remove oldest element from eventIDs
while |events| > |events|m do
  remove random element from events

Subscribing & Unsubscribing

- To subscribe, a process \( p_i \) must know a process \( p_j \) already in the membership set and send \((\emptyset, \emptyset, \emptyset, \{p_j\})\) to \( p_j \)
- To unsubscribe, a process \( p_i \) can inject its own unsubscription with a timestamp -or- just leave

Analytical evaluation

Assumptions

- \( n \) processes \( \{p_1, p_2, \ldots, p_n\} \)
- Gossip protocol runs in synchronized rounds
- Independent uniformly distributed views

Probability that a given process belongs to \( p_i \)'s view

\[
\begin{align*}
\text{# of possible views for } p_i \text{ containing our given process} & = \frac{(n-2)}{(n-1)} \\
\text{# of possible views for } p_i \text{ containing our given process} & = \frac{(n-2)}{(n-1)} \frac{(n-1)}{(n-2)} = \frac{len}{n-1}
\end{align*}
\]

\# of possible views for \( p_i \)

\( len = |\text{view}| = i \)
Event propagation analysis

Consider an event $e$

Let $s_r$ be the number of processes infected with $e$ at round $r$

$s_0 = 1$

Goal: define a lower bound on probability that a given susceptible process $p_1$ is infected by a given gossip message from a given process $p_2$

$$\text{(prob. that $p$ is in $p'$'s view)} \times \text{(prob. that $p$ chooses $p'$ to gossip with)} \times \text{(prob. that $p'$ does not crash and message is not lost)}$$

$$p = \left( \frac{\text{len}}{n-1} \right) \times \left( \frac{F}{\text{len}} \right) \times k = \frac{F}{n-1} k$$

Calculating distribution for $s_r$

Let $s_r$ be the number of processes infected with $e$ at round $r$

$$P(s_{r+1} = j \mid s_r = i) = \begin{cases} 
\frac{(n-i)}{j-i} \left( 1 - q \right)^{j-i} \left( q \right)^{i-j}, & j \geq i \\
0, & j < i
\end{cases}, \quad q = 1 - \frac{F}{n-1} k$$

$$P(s_0 = j) = \begin{cases} 
1, & j = 1 \\
0, & j = 0
\end{cases}$$

$$P(s_{r+1} = j) = \sum_{i=0}^{n} P(s_r = i) P(s_{r+1} = j \mid s_r = i)$$

Event propagation analysis

Let $s_r$ be the number of processes infected with $e$ at round $r$

$$p = \frac{F}{n-1} k$$

Probability that a given susceptible process $p_1$ is infected by a given gossip message from a given process $p_2$

$$q = 1 - p$$

Probability that a given susceptible process $p_1$ is not infected by a given gossip message from a given process $p_2$

# of combinations of susceptible processes to infect

$$P(s_{r+1} = j \mid s_r = i) = \begin{cases} 
\frac{(n-i)}{j-i} \left( 1 - q \right)^{j-i} \left( q \right)^{i-j}, & j \geq i \\
0, & j < i
\end{cases}$$

Analysis?

- $|\text{view}|_m$ has no expected effect on latency or reliability
- Mathematical guarantees only hold for independent uniformly distributed views
- Results show that their algorithm is “close” to perfect views, but also show that their reliability does depend on $l$
Events buffer optimization

- Age-based message purging
  - Idea
    - Estimate number of rounds event has been in system
    - If necessary, purge buffer of “older” events

Why can’t you use age-based for subs buffer, too?

Subs buffer optimization

- Frequency-based message purging
  - Idea
    - Tag each subscription with number of times it has been gossiped
    - If necessary, purge subscriptions that have been gossiped more

Why does this not work for events buffer?
Directional Gossip


Probabilistic broadcasts
- Initial unreliable multicast followed by subsequent gossip rounds
- Achieves high reliability
- Assumes an underlying point-to-point communication mechanism

Example WAN

Flooding
- Upon receiving a new message, a process forwards it to all neighbors the process believes have not received it yet
- Easy to implement
- High overhead in LAN
**Marrying gossip and flooding**

- Uniform gossip:
  - Less overhead in LAN than flooding
  - Only sends to random subset of processes

- Flooding
  - Less overhead on inter-network routers than uniform gossip
  - Only sends to neighbors

---

**First marriage**

Idea: Forward messages to processes with less connectivity.
Gossip to processes with more connectivity

```plaintext
when p receives a new message m
while p believes not enough neighbors have m {
  q = a neighbor process of p
  send m to q
}
```

---

**Setting**

Each gossip server maintains a set of adjacent routers.
Two servers are neighbors if their adjacent routers sets have a common element.

How to measure connectivity?

---

**Link cut sets**

- Given a connected graph $G = \langle V, E \rangle$, the link cut set is a set of edges $E_{\text{LCS}}$, such that $G' = \langle V, E \setminus E_{\text{LCS}} \rangle$ is disconnected.
- The link cut set with respect to nodes $p$ and $q$ is a set of edges $E_{pq}^\prime$ such that removing all edges in $E_{pq}^\prime$ will disconnect $p$ and $q$.
Weights and link cut sets

- \( p \) assigns a weight to \( q \) equal to the size of the smallest link cut set for \( p \) and \( q \)
- Menger’s Theorem:
  
  For any two nodes of a graph, the maximum number of link-disjoint paths between them equals the size of the minimum link cut set between them.

Inter-network router notation

- A pair of servers (in different LANs) that are neighbors identifies an inter-network router
- A path of \( k \) servers \( \langle p_1, p_2, ..., p_k \rangle \) identifies a trajectory of \( k-1 \) inter-network routers
  
  \[ \text{INR}(\langle p_1, p_2, ..., p_k \rangle) = \langle r_1, r_2, ..., r_{k-1} \rangle \]

Paths in a gossip message \( m \)

- \( m.\text{path} \) is a path of servers \( \langle p_1, p_2, ..., p_k \rangle \) such that \( p_1 \) originated the message and sent it to neighbor \( p_2 \) who then sent it to \( p_3 \), etc.
- A path of servers \( \langle p_1, p_2, ..., p_k \rangle \) implicitly contains \( n-1 \) timestamps
  
  - \( p_2.\text{timestamp} \) is the time that process \( p_2 \) received the message
  - Given \( \text{INR}(\langle p_1, p_2, ..., p_k \rangle) = \langle r_1, r_2, ..., r_{k-1} \rangle \),
    
    \( r_i.\text{timestamp} = p_{i+1}.\text{timestamp} \)

Initiating a gossip message

for process \( p \):
  
  let \( m \) be a new gossip message
  
  \( m.\text{path} = \langle p \rangle \)
  
  for each \( q \in \text{Neighbors}_p \)
    
    send \( m \) to \( q \)

\[ \begin{align*}
  & p \quad m.\text{path} = \langle p \rangle \\
  \quad \quad \quad \rightarrow \quad \quad \quad \rightarrow \quad \quad \quad \rightarrow \\
  & q_1 \quad m.\text{path} = \langle p, q_1 \rangle \\
  \quad \quad \quad \rightarrow \quad \quad \quad \rightarrow \quad \quad \quad \rightarrow \\
  & q_2
\]
Initially: \( V q \in \text{Neighbors}_p : \text{Trajectories}(q) = \{ \text{INR}(p, q) \} \)

when \( p \) receives a gossip message \( m \) for the first time 

\[
\text{int} \; \text{sent} := 0 \\
\text{for each} \; q \in \text{Neighbors}_p \\
\quad \text{if} \; (q \in m.\text{path}) \; \text{then} \\
\quad \quad \text{UpdateTrajectories}( \text{Trajectories}(q), \text{INR}(\text{Trim}(m.\text{path}, p)) ) \\
\]

void UpdateTrajectories( reference to set of trajectories \( T \), trajectory \( R \) ) {
    \text{if} \; (\text{all trajectories in } T \text{ are disjoint with } R) \; \text{then} \\
    \quad T := T \cup \{ R \} \\
    \text{else} \; \text{for each} \; f \in T \\
    \quad \text{for each router } r_1 \in f \; \text{and router } r_2 \in R \\
    \quad \quad \text{if} \; r_1 = r_2 \; \text{then} \\
    \quad \quad \quad r.\text{timestamp} := \max(r.\text{timestamp}, r_2.\text{timestamp}) \\
}

Spatial Gossip [KKD01]
- Embed processes in \( D \) dimensional space
- let \( d_{u,v} \) be distance between \( u \) and \( v \)
- \( u \) sends a gossip message to \( v \) with probability proportional to \( \frac{1}{d_{u,v}^p}, \; p \in [1,2] \)

Expected time for message to reach nodes at distance \( d = O(\log^{1+\varepsilon} d) \)

DoS attacks [BKS04]?

System model
- Network is fully connected
- Asynchronous communication
- Insecure channels
- Loss rate on communication links is bounded and uniform
- Adversary can generate and insert messages into channels and snoop on messages
Drum protocol (informal)

- Use public key cryptography
- Pull mechanism
  - p sends \(\text{history, port}_{\text{rand}}\) to q on well-known port
  - q sends \(\text{msg}_{\text{miss}}\) to p on port\_rand
- Push mechanism
  - p sends \(\text{push-offer, port}_{\text{rand}}\) to q on well-known port
  - q sends \(\text{history, port'}_{\text{rand}}\) to p on port\_rand
  - p sends \(\text{msg}_{\text{miss}}\) to q on port\_rand
- Bound number of messages processed per port
- Discard all messages in buffers after a round