Epidemics

Context

“This course is designed to cover some of the key ideas that have proved useful or are expected to be useful for designing and building tomorrow's distributed systems. The course focuses on fundamentals.”

- Some fundamentals
  - Atomic Commit
  - Reliable Broadcast
  - Consensus
- Isis Toolkit

[Birman, van Renesse et al.]

Y. Amir, D. Dolev, S. Kramer, and D. Malki. *Transis: A communication sub-system for high availability*. In Proc. 22nd Annual International Symposium on Fault-Tolerant Computing, pages 76--84, July 1992.

Scalability

- Long message delays
- Unreliable communication
- Network partitions

A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, BC, August 1987, pp. 1-12.

Setup

- Database replicated at thousands of sites across the nation
- Unreliable point-to-point links
- 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

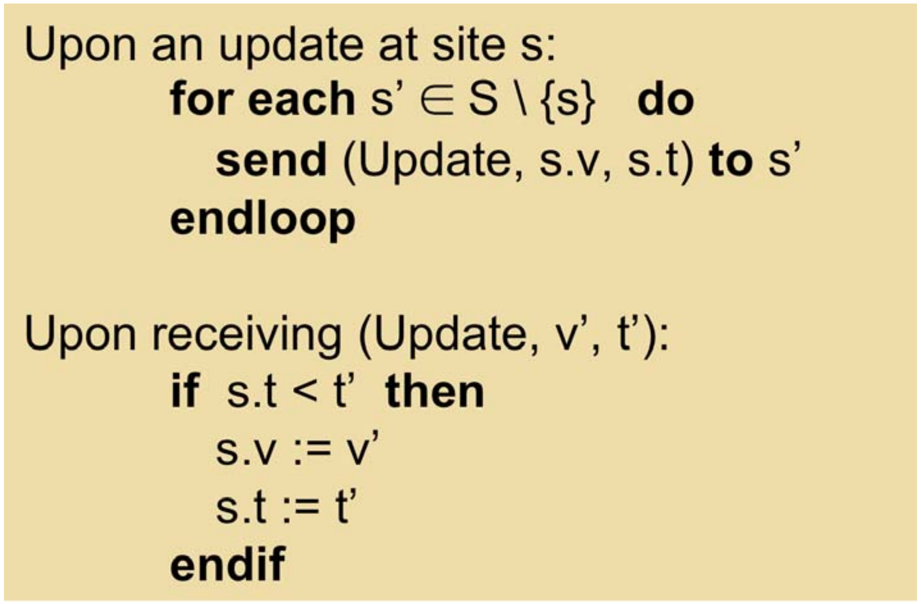
Notation

- *S* is a set of n sites (replicas)
- Pretend there is only one database entry
  - *v* is the value of the database entry
  - *t* is the timestamp associated with v
- Timestamps are totally ordered
- All sites are consistent iff

$$\forall\, s, s' \in S : s.v = s'.v$$

Broadcast

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' ∈ S \ {s}   do
           send (Update, s.v, s.t) to s'
        endloop

Upon receiving (Update, v', t'):
        if  s.t < t'  then
           s.v := v'
           s.t := t'
        endif
```

Weakness: send is unreliable
what if crashes occur

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:
        for some s' ∈ S \ {s}  do
            ResolveDifference(s,s')
        endloop
```

```
Push:
  ResolveDifference(s,s')  {
     if   s.t  > s'.t   then
            s'.v := s.v
            s'.t = s.t
     endif
```

```
Pull:
  ResolveDifference(s,s')  {
     if   s.t  < s'.t   then
            s.v := s'.v
            s.t := s'.t
     endif
```

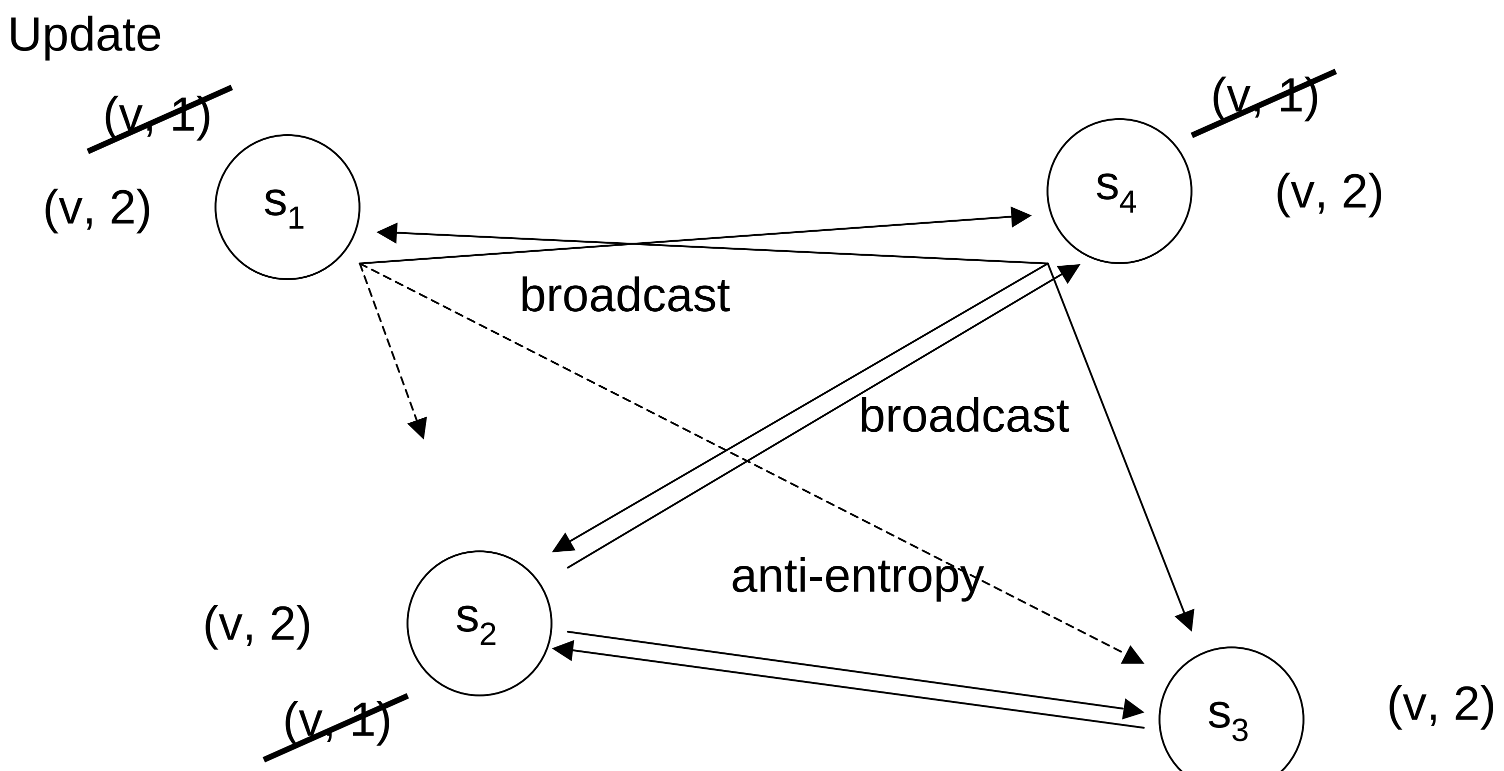
Push vs. pull analysis

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



Anti-entropy facts

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

Epidemics

- Resilient against 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 <u>susceptible</u>
- When a site *s* receives a new update, it becomes <u>infective</u>
- *s* periodically chooses another site *s'*
- If *s'* does not know the rumor, then it receives the update and also becomes <u>infective</u>
- If *s'* already knows the rumor, then *s* becomes <u>removed</u> with some probability

Rumor mongering protocol

```
For a site $s$:
    let $L$ be a list of (initially empty) infective updates

    periodically:
        for some $s' \in S \setminus \{s\}$ do
            for each update $u \in L$
                send u to s'
                if s' already knows about u then
                    remove u from L with probability 1/k
            end loop
        end loop

    upon receiving new update $u$:
        insert $u$ into $L$
```

Analysis of rumor mongering

i = fraction of infective sites
s = fraction of susceptible sites
r = fraction of removed sites

$$\frac{ds}{dt} = -si$$

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

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

$$\int di = \int\left(-\frac{k+1}{k} + \frac{1}{ks}\right) ds$$

$$i(s) = -\frac{k+1}{k}s + \frac{\ln s}{k} + c$$

$$c = \frac{k+1}{k}$$

$$s = e^{-(k+1)(1-s)}$$

Rumor mongering facts

- Expected fraction of susceptible sites

  $$s = e^{-(k+1)(1-s)}$$

- Back up mongering with anti-entropy
- Redistribution of update
  - Rumor mongering vs. broadcast
  - 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 – all servers not in R discard death certificate after time $t + t_1$
    - t_2 – all servers discard the certificate after $t + t_2$
- Interaction with anti-entropy?

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

K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*. 17(2): 41-88. May 1999

Class I – Strong reliability

- Properties: agreement, validity, termination, integrity
- Expensive and limited scalability
- Unpredictable performance under congestion
- Degraded throughput under transient failures
  - Why?
  - 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

Bimodal multicast claims

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

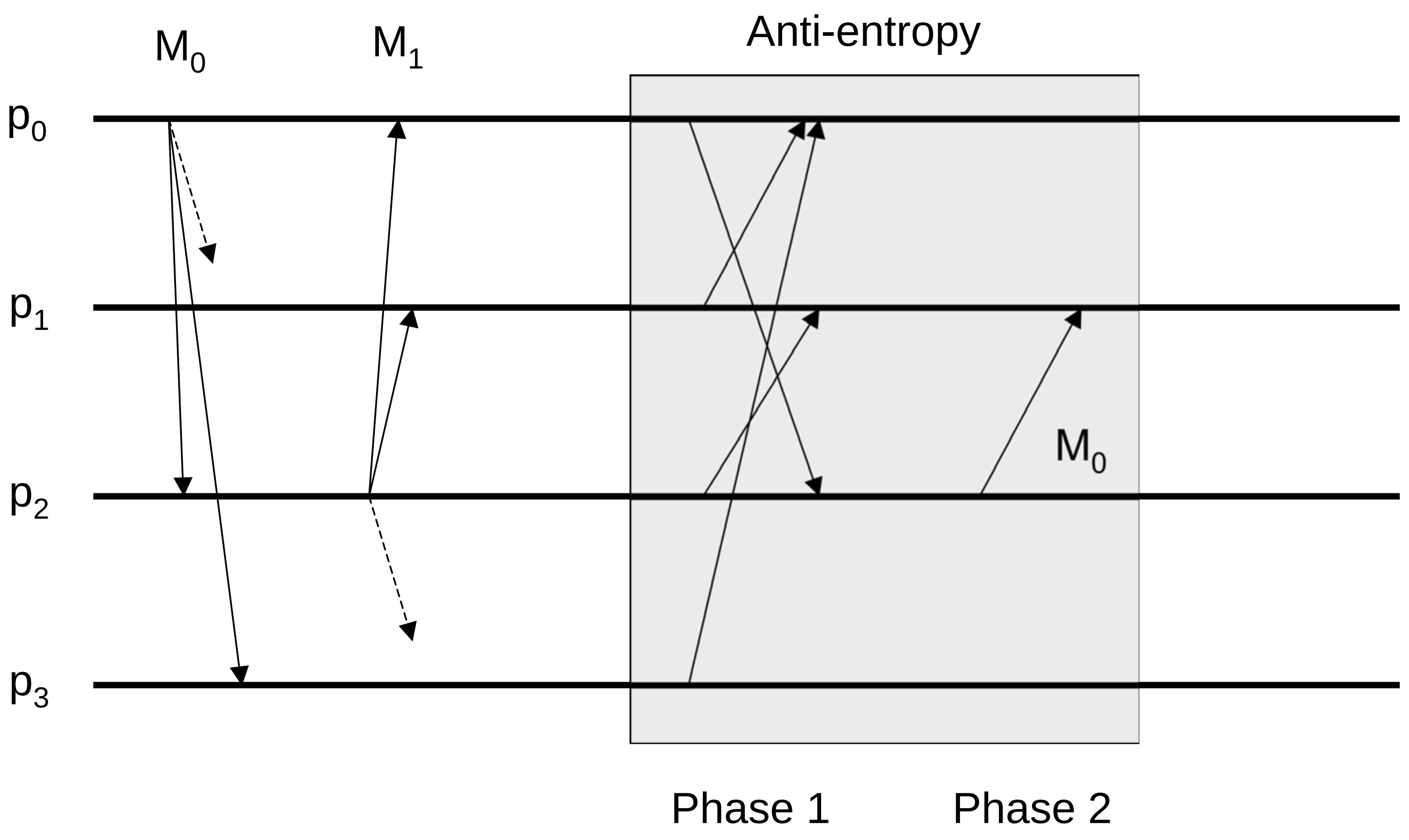
System assumptions

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

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



Optimizations

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

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~~ suffix
- If a process cannot recover a message, it gives up and notifies application

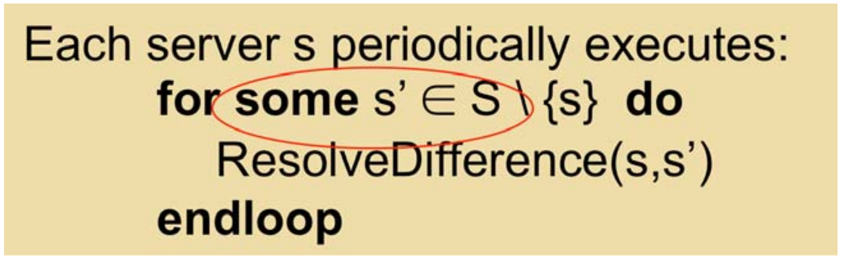
A problem to our solution

- Applications that need high throughput (frequent updates) and can tolerate small inconsistencies
- Examples: flight telemetry, stock trading, video conferencing

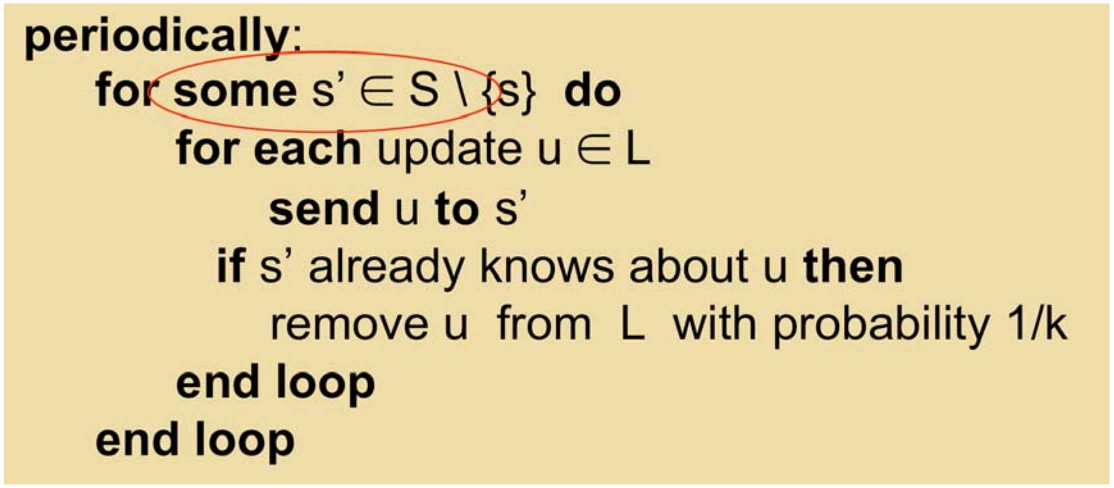
Recovery from delivery failures

- 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
  - Maintain a few very large buffers
  - Employ a state transfer technique

Back to scalability



```
Each server s periodically executes:
    for some s' ∈ S \ {s}  do
        ResolveDifference(s,s')
    endloop
```



```
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
        end loop
    end loop
```

P. Th. Eugster, R. Guerraoui, S.B. Handurukande, P. Kouznetsov, A.-M. Kermarrec. Lightweight Probabilistic Broadcast. ACM Transactions on Computer Systems (TOCS) 21(4):341-374, November 2003