Amazon S3 incident July 20th, 2008

S3 service: Provides a simple web services interface to store and retrieve any amount of data.

- Intent: highly scalable, reliable, fast, inexpensive storage infrastructure…
- Scale:
  - Large number of customers. Amazon itself uses S3.
  - Lots of objects stored
    - 4billion Q4’06 □ 40B Q4’08 □ 100B Q2’10 □ **2 trillion Q2’13**
- SLA guarantees 99.9% monthly uptime
  - Less than 43 minutes of downtime per month

Amazon S3 incident on Sunday, July 20th, 2008

8:40am PDT: error rates began to quickly climb

10 min: error rates significantly elevated and very few requests complete successfully

15 min: Multiple engineers investigating the issue. Alarms pointed at problems within the systems and across multiple data centers.

- Trying to restore system health by reducing system load in several stages. No impact.

Amazon S3 incident on Sunday, July 20th, 2008

1h01min: engineers detect that servers within Amazon S3 service have problems communicating with each other

- Amazon S3 uses **an epidemic protocol** to spread servers’ state info in order to quickly route around failed or unreachable servers
- Engineers determine that *a large number of servers were spending almost all of their time* in the epidemic protocol

1h52min: unable to determine and solve the problem, they decide to shut down **ALL** components, clear the system's state, and then reactivate the request processing components.

Restart the system!

Amazon S3 incident on Sunday, July 20th, 2008

2h29min: the system's state cleared

5h49min: internal communication restored and began reactivating request processing components in the US and EU.

7h37min: EU was up correctly, and US location began to process requests successfully.

8h33min: Request rates and error rates had returned to normal in US.

Post-event investigation

Message corruption was the cause of the server-to-server communication problems

A few messages on Sunday morning had a **single bit corrupted**

MD5 checksums were generally used in the system, but Amazon did **not** apply them to detect errors in this particular internal state

The **corruption spread** wrong state throughout the system, and led to **increased** system load

Amazon's the report for the incident http://status.aws.amazon.com/s3-20080720.html

Current status for Amazon services http://status.aws.amazon.com/

Lessons: Preventing a similar incident

- Verify message and state correctness – all kind of corruption errors may occur
  - Add checksums to detect corruption of system state and messages where possible
  - Verify invariants before processing state
- Engineer protocols to control the amount of traffic they generate.
  - Add rate limiters.
  - Put additional monitoring and alarming for gossip rates
- An emergency procedure to restore clear state in your system may be the solution of last resort. Make it work *quickly*.

You get a big hammer … use it wisely!

Design stories

- Context1 – generic: RPC / network file system (NFS) stories
  - Takeaway: not 100% ‘transparent’. Pay attention to deviations in semantics

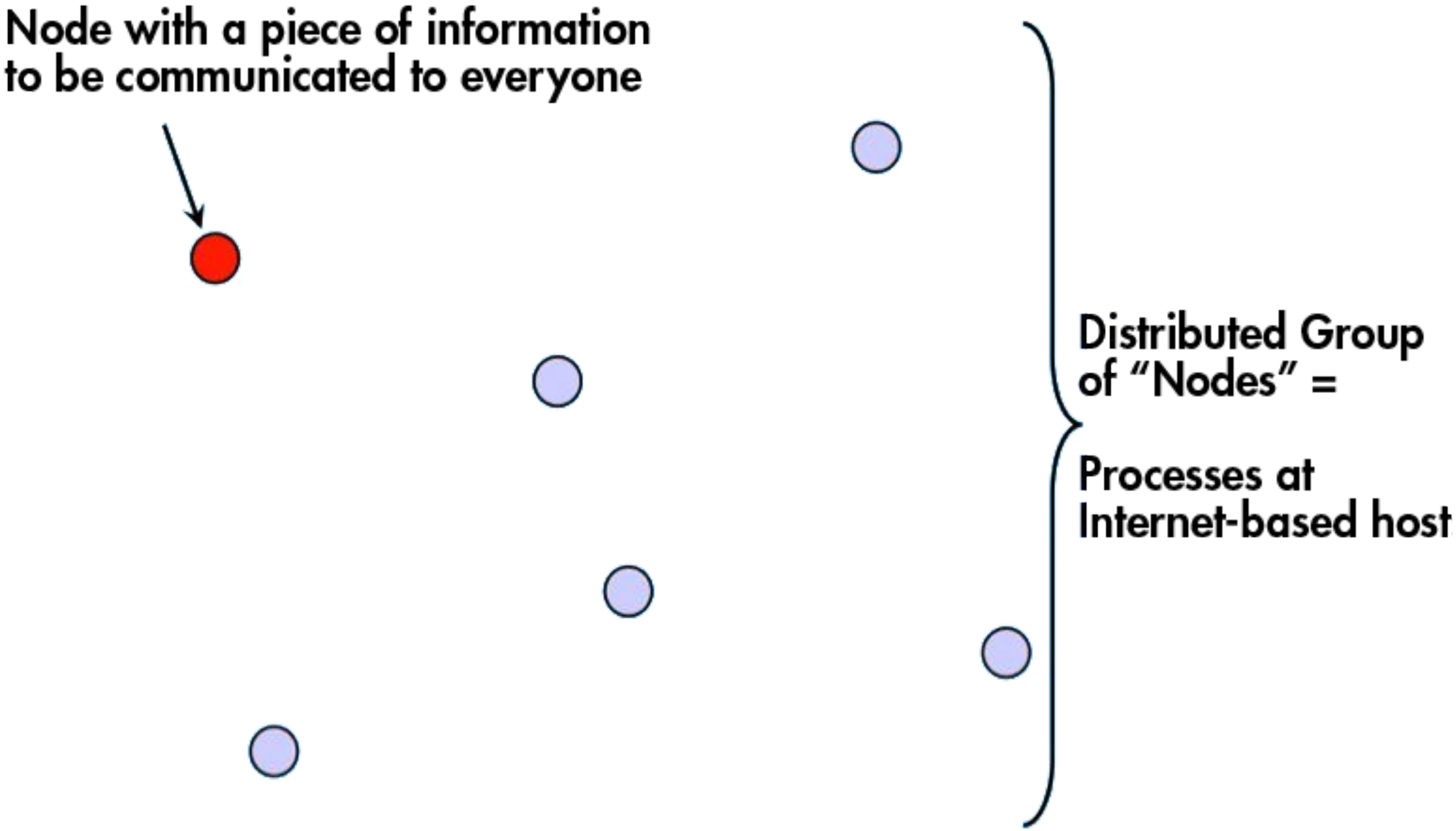
- Context2 – specialization: distributed garbage collection
  - Keep track of active [remote] references to an object

  (reference lising style)
  - Take away: Light-weight solutions are possible in certain cases.

- Context3: one-to-many communication

Environment: Unreliable communication, nodes may crash

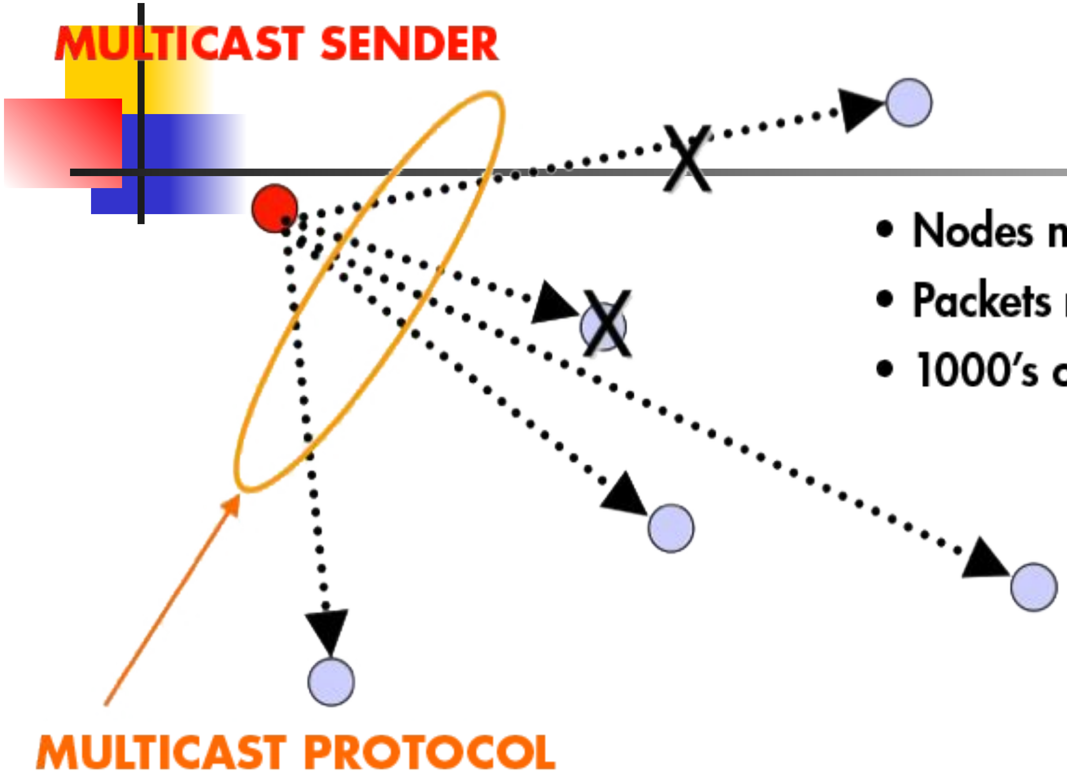
ACK: some of these slides contain material developed by Indranil Gupta (UIUC).



Node with a piece of information to be communicated to everyone

Distributed Group of "Nodes" =

Processes at Internet-based host



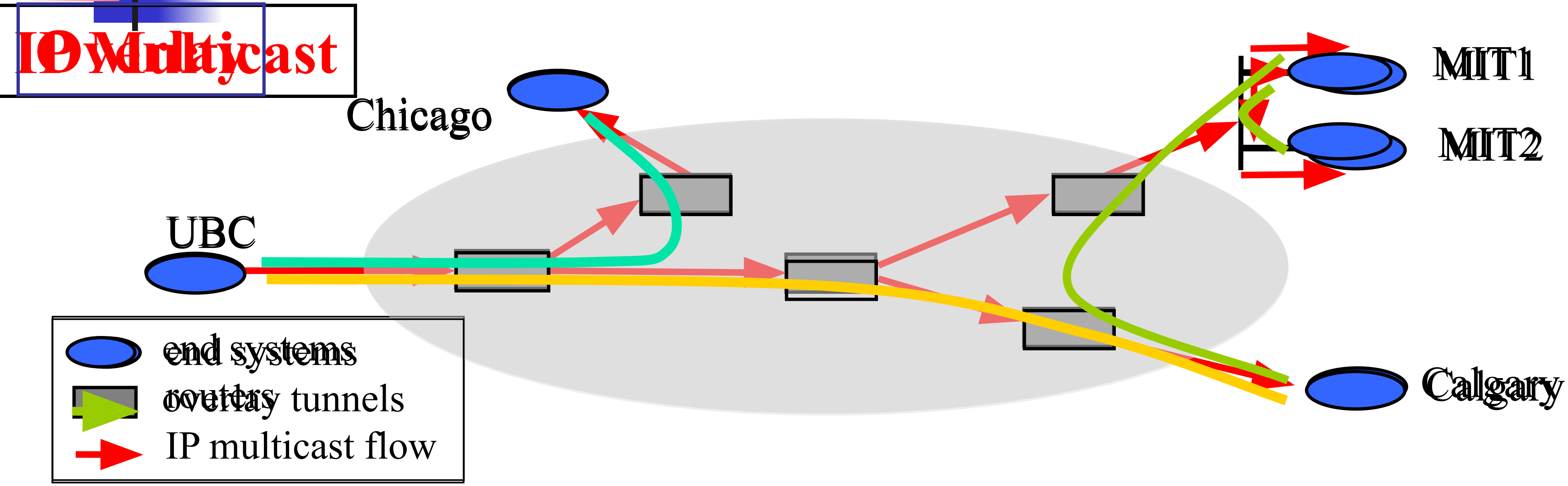
- Nodes may crash
- Packets may be dropped
- 1000's of nodes

What's success?

1. Reliability

   (100% delivery ideally)
2. Latency
3. Overheads
4. ….

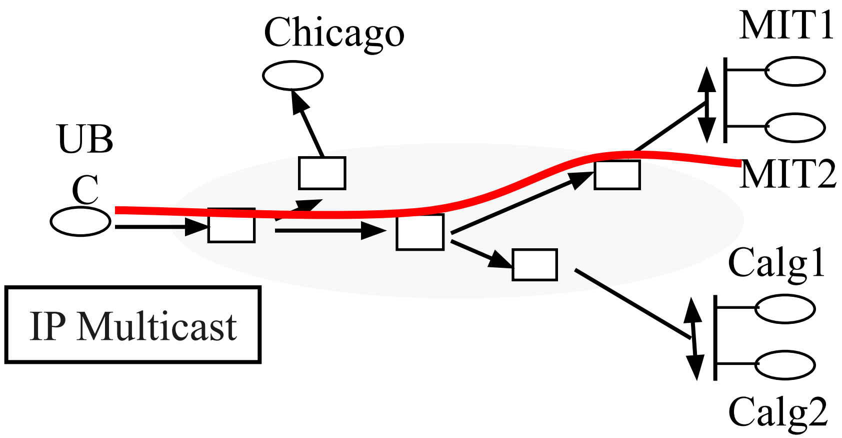
Multicast: IP-based vs. overlay

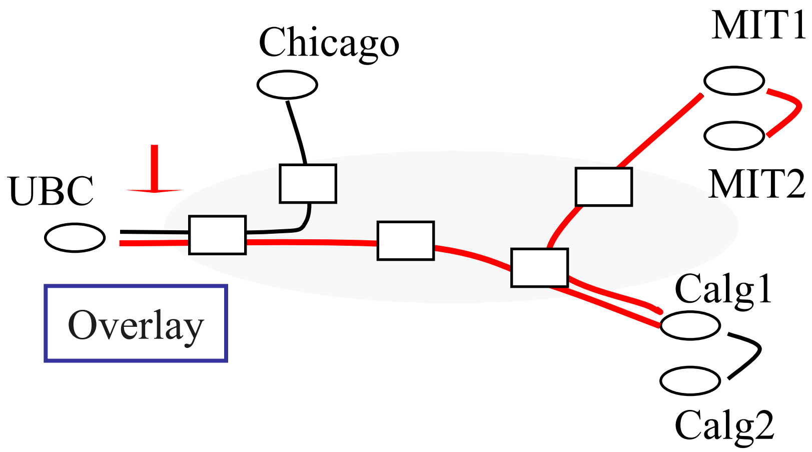


- Two categories of solutions:
  - Based on support from the network: IP-multicast
    - Issues: limited WAN-level deployment, scaling, dealing with message loss,
  - Without IP-layer support: application-layer
    - Issues: Overheads, dealing with dynamicity (nodes leaving / joining)
    - Depending on structure: Tree-based, Epidemic

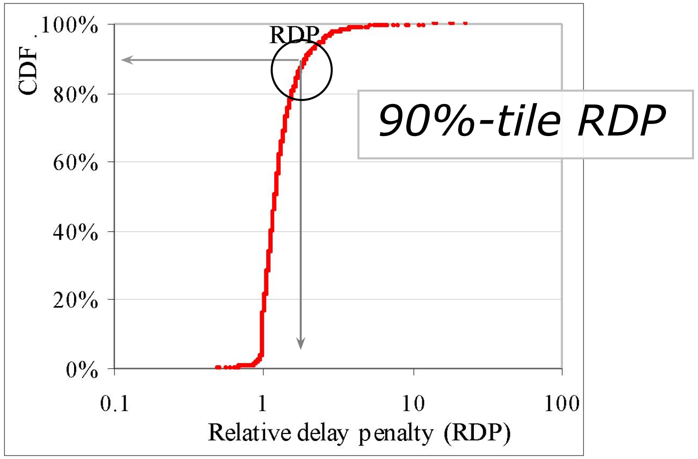
Application-level multicast success metrics: Relative Delay Penalty and Link Stress

- Relative Delay Penalty (RDP): Overlay-delay vs. IP-delay
- Link Stress: number of duplicate packets on each physical link



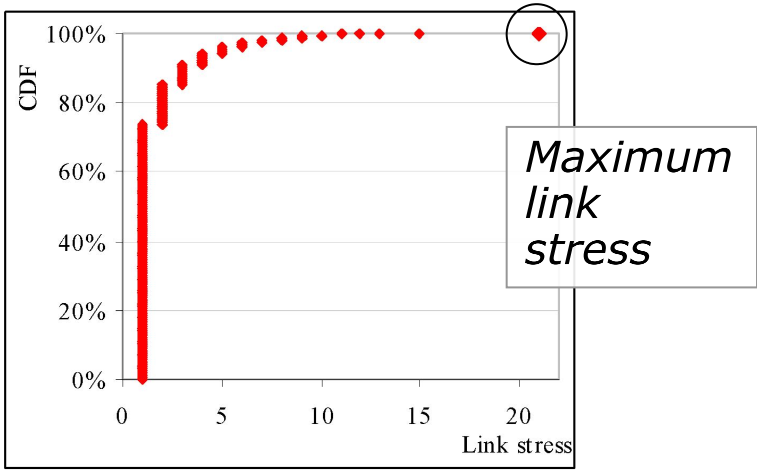


Relative delay penalty distribution



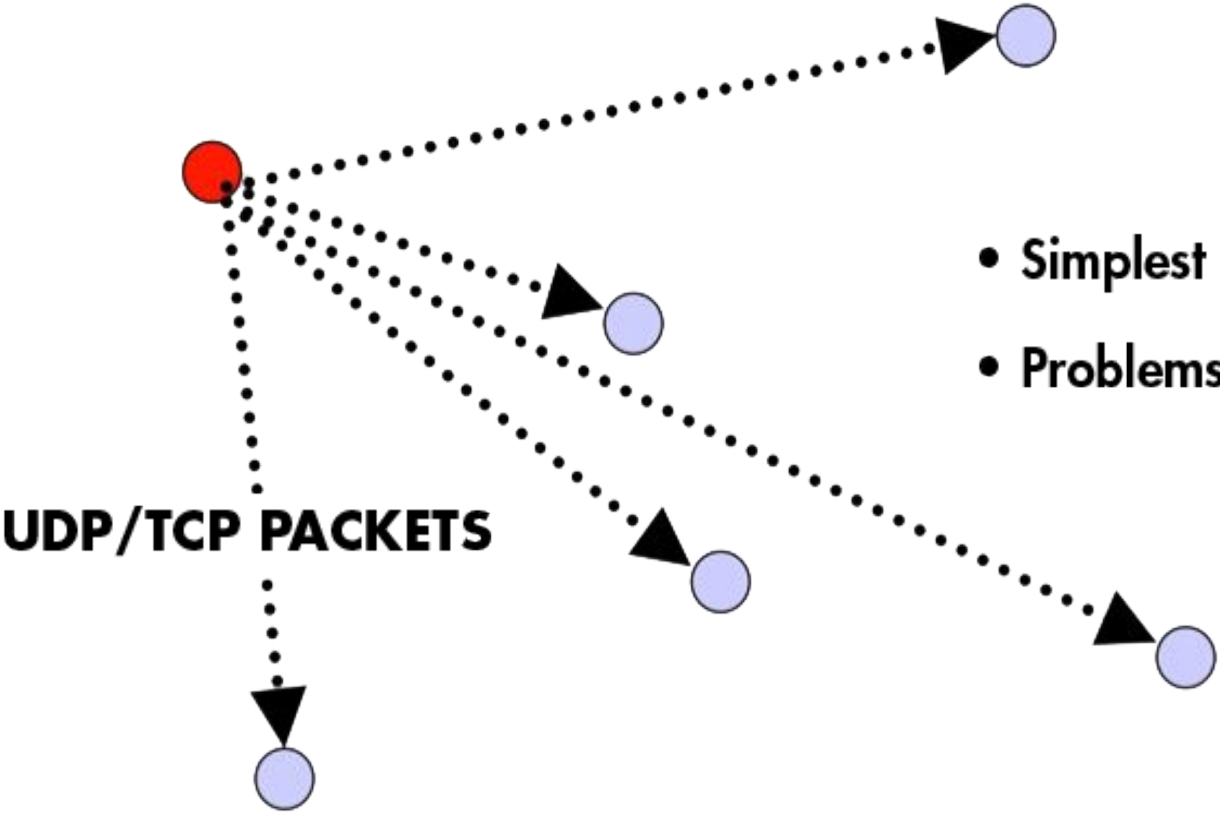
90%-tile RDP

Link stress distribution



Maximum link stress

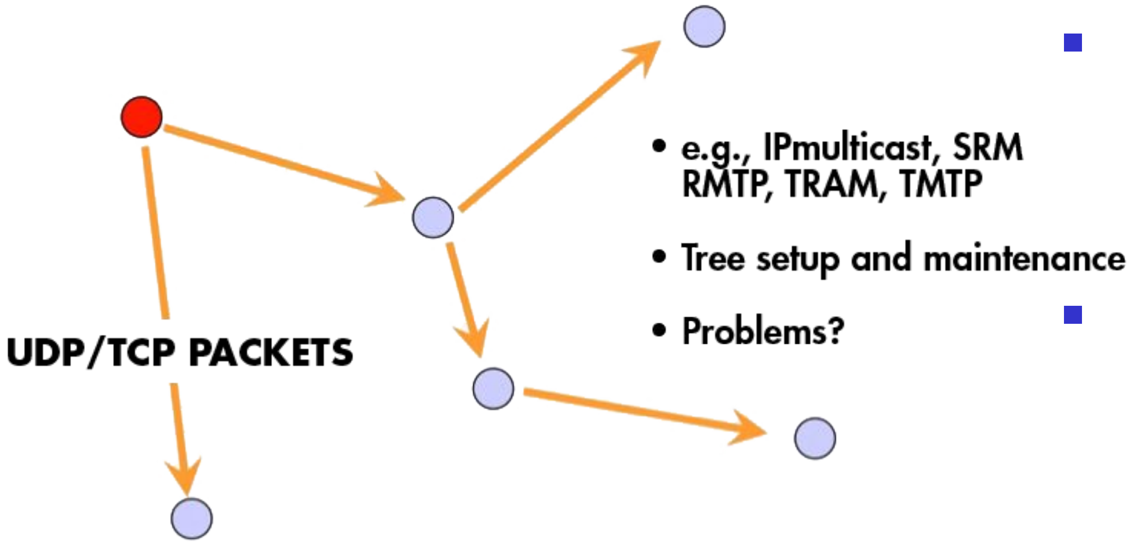
Possible solution: Centralized



- Simplest implementation
- Problems?

Possible solutions: Tree-Based

- Build a spanning tree among processes in the multicast group
  - Use it to disseminate packets
- Use ACKs to repair multicasts not received
  - or negative ACKs (NAKs)
- SRM (Scalable Reliable Multicast)
  - Uses NAKs, but adds random delays, and uses exponential backoff to avoid NAK storms
- RMTP (Reliable Multicast Transport Protocol)
  - Uses ACKs, sent to designated receivers, which then re-transmit missing multicasts
- These protocols may still cause an O(N) ACK/NAK overhead [Birman99]



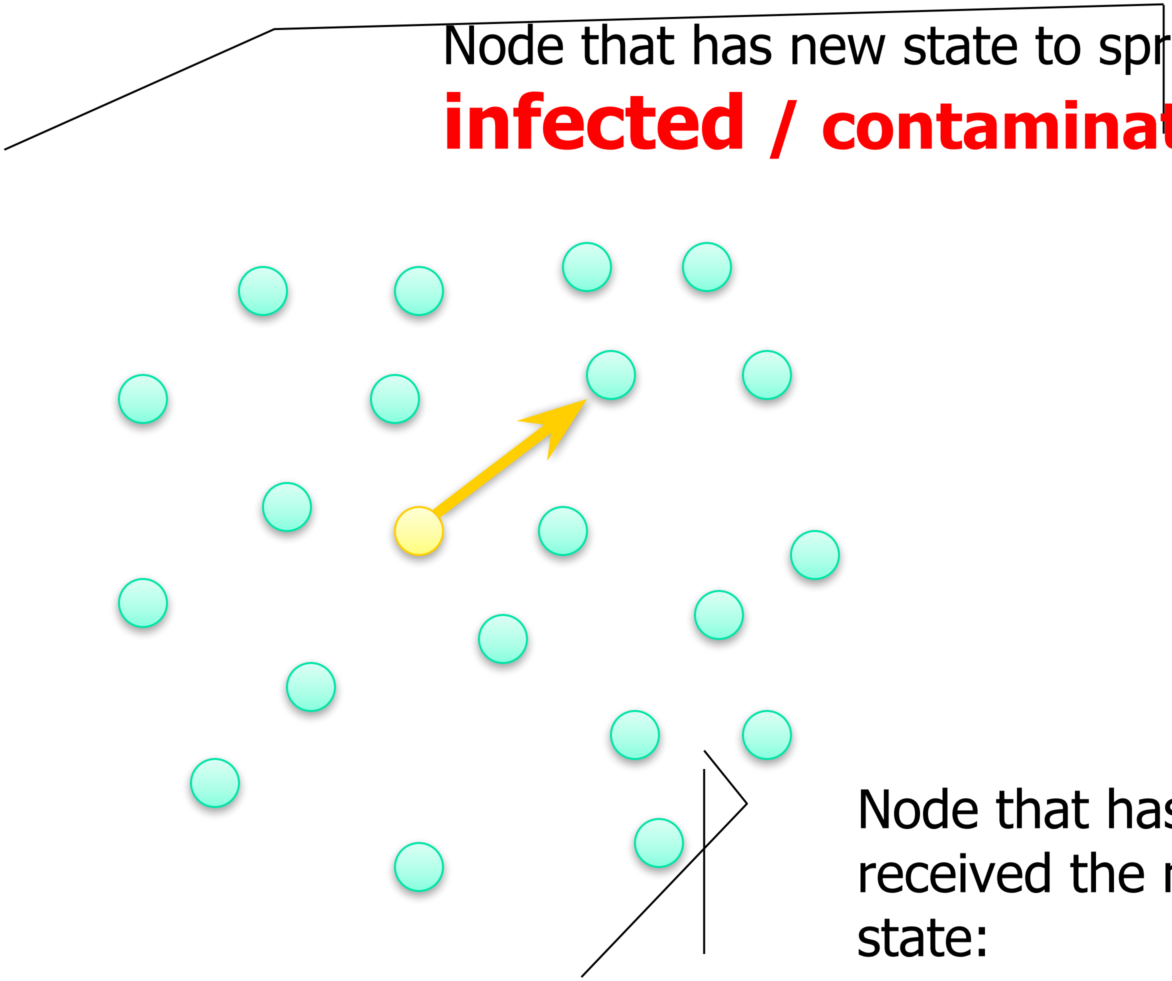
- e.g., IPmulticast, SRM RMTP, TRAM, TMTP
- Tree setup and maintenance
- Problems?

Possible solutions: Epidemic

- Goal:
  - Propagate data from one initial data source (node) to all participants
    [Assumption: if multiple sources, there are no write–write conflicts]

- Context: makes sense to use this solution if
  - (relatively) frequent node failures and/or message loss
    - thus no coordinator is feasible, fixed tree structure costly
  - message propagation can be ‘lazy’,
    - i.e., solution does not require ‘immediate’ propagation

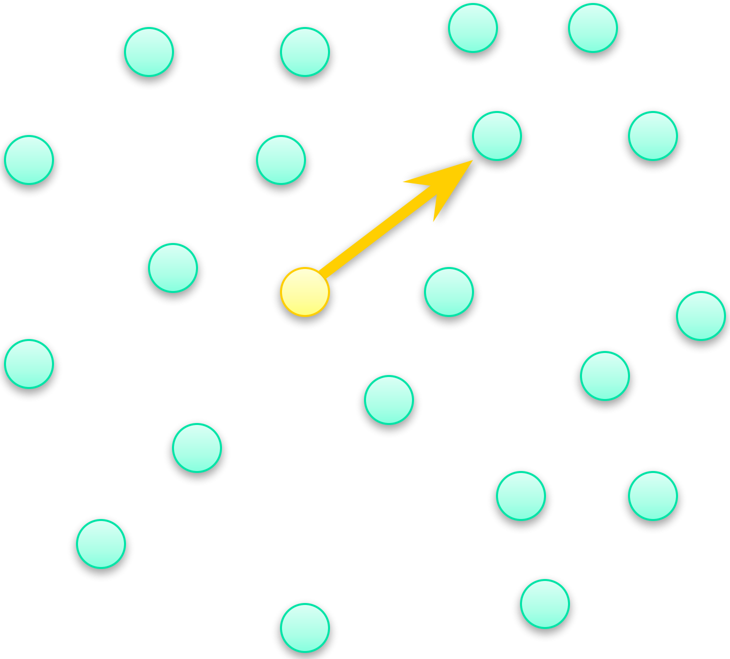
An epidemic protocol

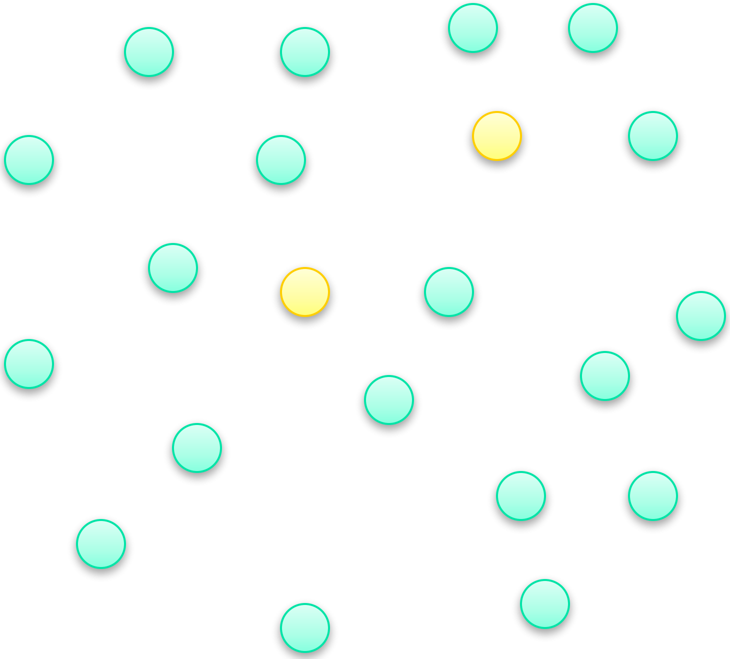


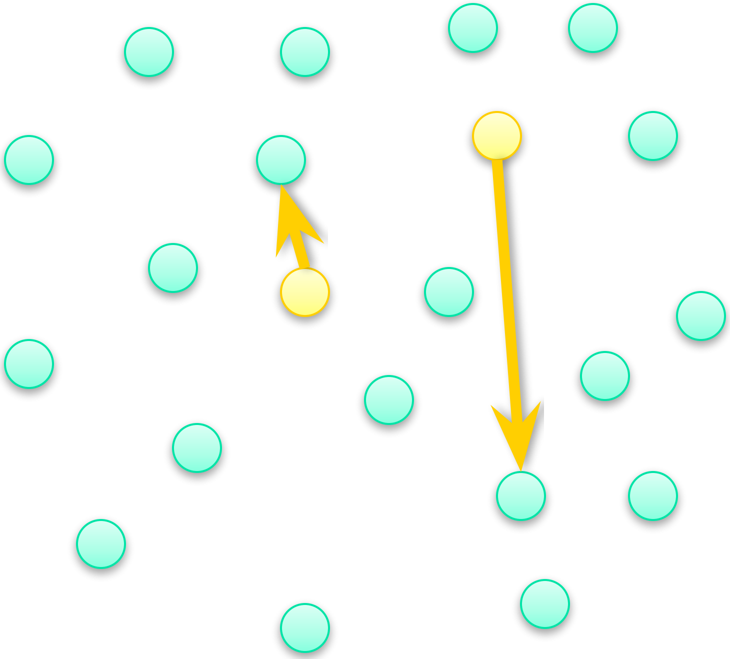
Node that has new state to spread: **infected / contaminated**

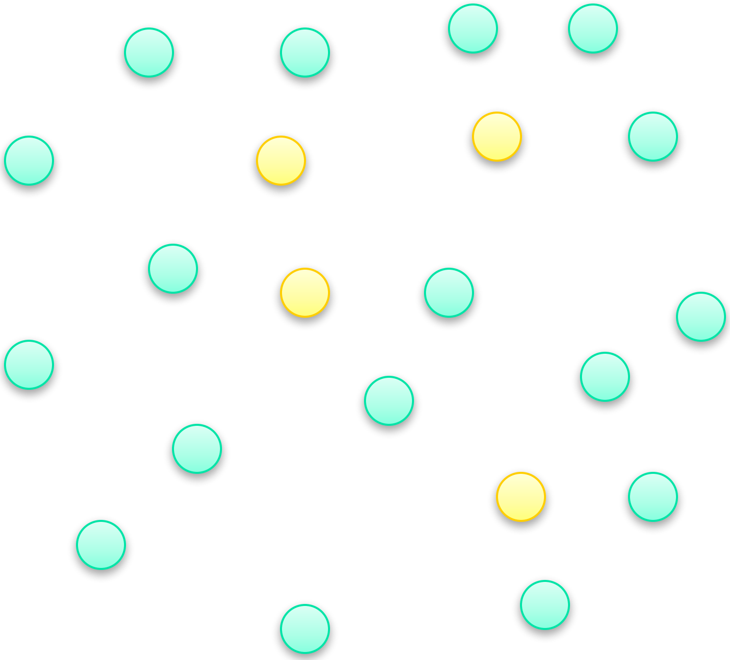
Node that has not yet received the new state: **susceptible / not contaminated**

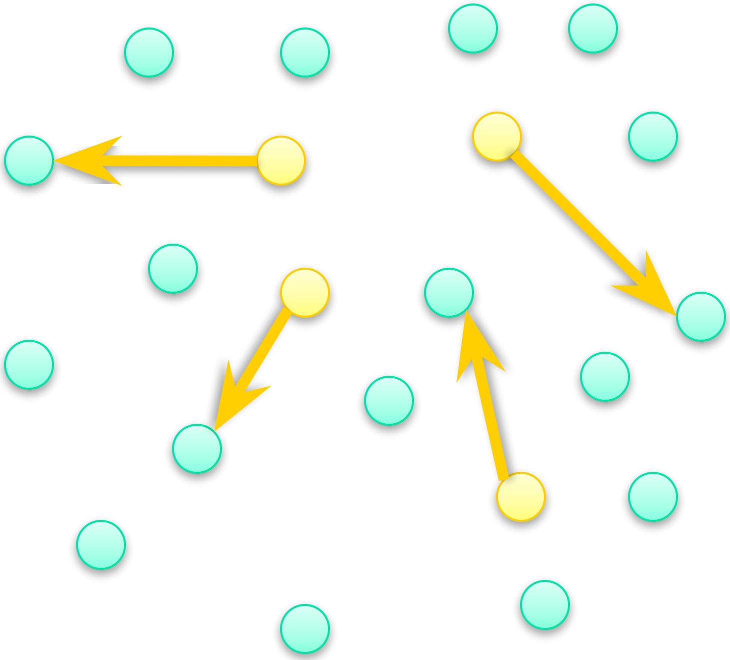
Language used: borrowed from epidemiology

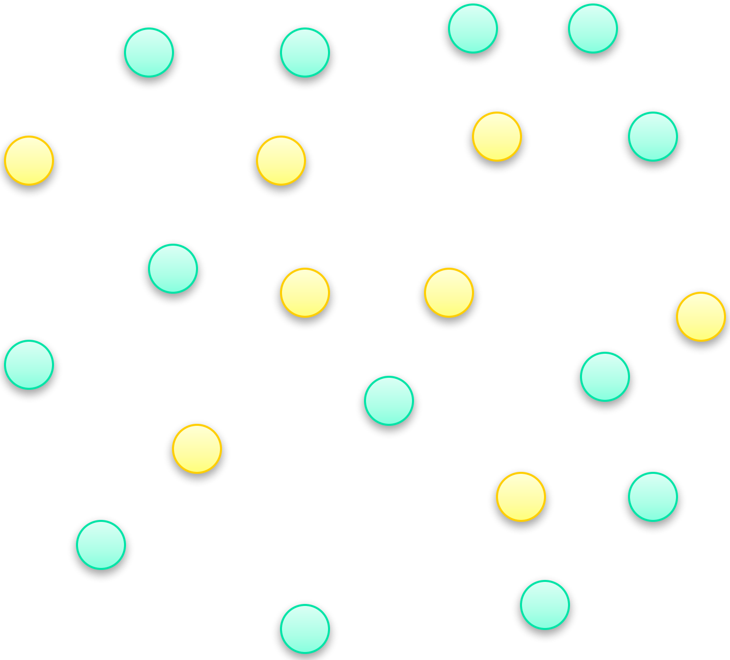


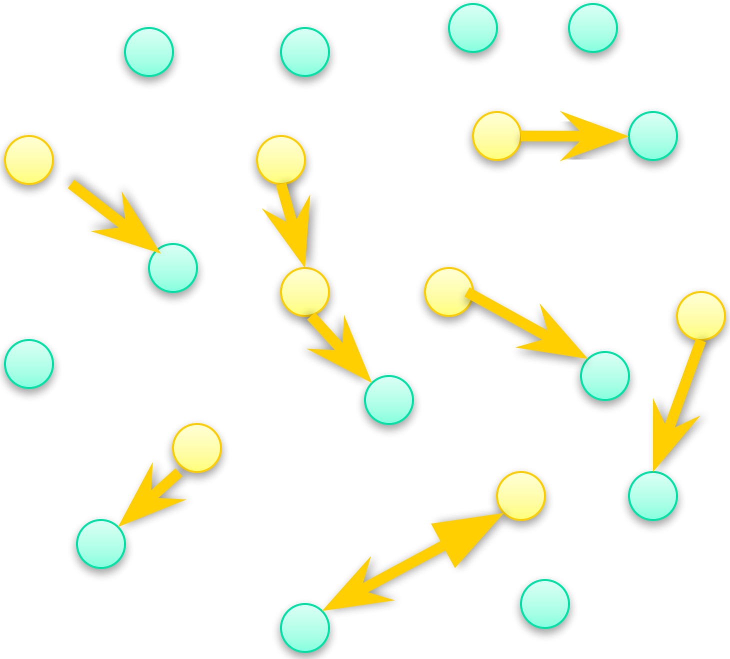


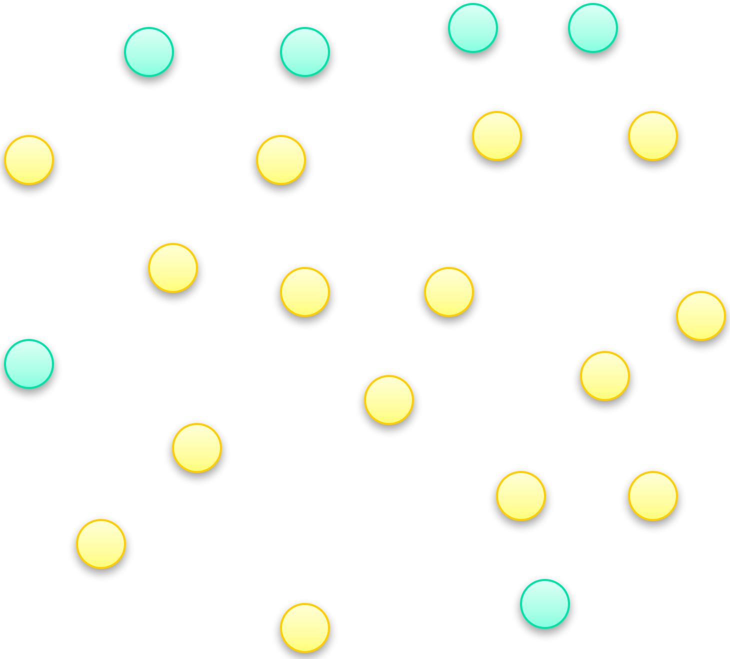


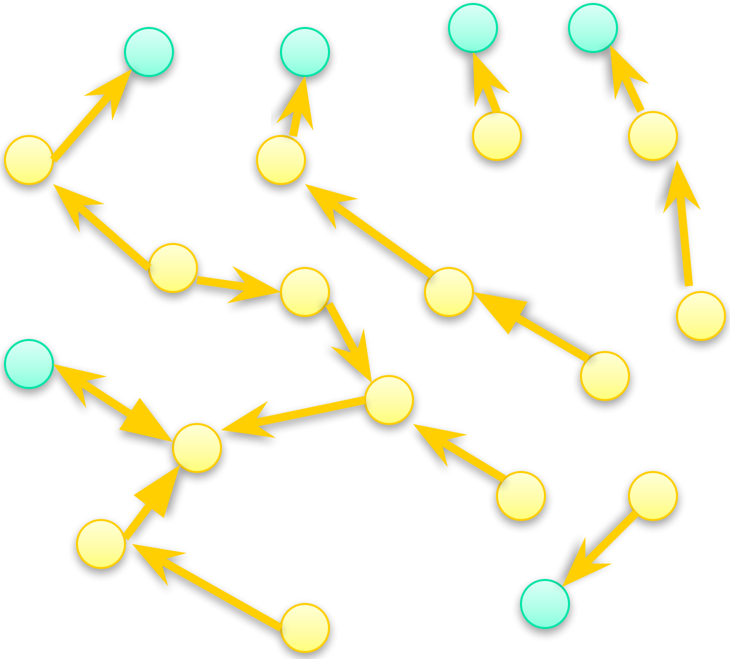


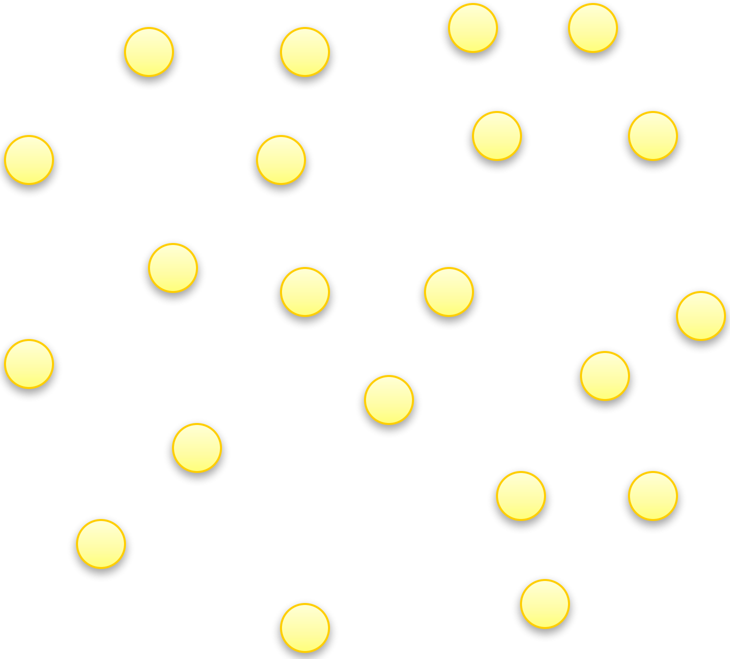












Advantages of epidemic techniques

- *Asynchronous communication pattern*.

  Operate in a 'fire-and -forget' mode, where, even if the initial sender fails, surviving nodes will receive the update.

- *Robust with respect to message loss & node failures.*

  Once a message has been received by a few other nodes it is almost impossible to prevent the spread of the information through the system.

- *Autonomous actions*.

  Nodes take actions based on the data received without the need for additional communication to reach agreement with partners; nodes can take decisions autonomously.

- *Probabilistic model yet rigorous mathematical underpinnings.*

  Good framework for reasoning about the spread of information through a system over time.

- *No central coordinator!*

Cautionary tale

Summary so far

- Supporting: 1-to-N communication
  - Limited deployment for IP-level support
  - Can be implemented adequately at the application level

- Tree-based multicast protocols

- When concerned about scale *and* fault-tolerance, epidemic-style protocols are an attractive solution
  - Fast, reliable, fault-tolerant, scalable

Key issues:

- Who participates (initiates) in each round? (contaminated nodes / non-contaminated / both?)
- When to stop?

Anti-entropy epidemic protocols:

- Algorithm works in ‘phases’
  - Each node regularly chooses another node (or small subset) at random
    - state gets updated (passed single direction or both directions)
- Termination: #phases

[Variation] Gossiping:

- No ‘phases’. A node which has just been updated (has been contaminated), tells a number of other nodes about its update (contaminating them as well)
- Termination: each node decides independently

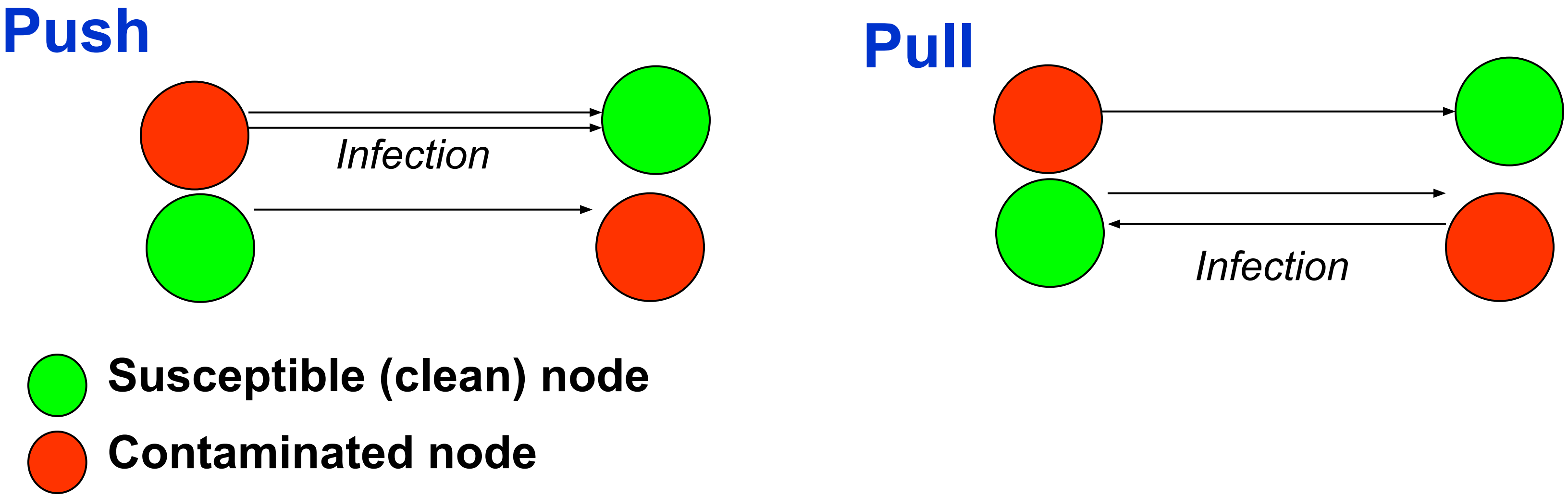
Anti-entropy Protocols

Directionality: Push / Pull / Hybrid

- “Push” protocol: ‘contaminated’ nodes initiate exchange
  - IF ‘contaminated’ (e.g., have a new message) THEN start passing ‘infection’
- “Pull” protocol: ‘susceptible’ nodes initate
  - IF NOT ‘contaminated’ THEN

    contact other node at random

    IF contacted node is ‘infected’ THEN epidemic is transmitted
- Hybrid variant: Push-Pull: everyone



Performance and reliability analysis

Claim: the simple PUSH protocol

- Is lightweight in large groups
- Is fast: spreads a multicast quickly
- Is highly fault-tolerant

Analysis

- Population of (*n+1*) individuals mixing homogeneously
- Contact rate between any individual pair: β

At any time, each individual is either uninfected (numbering x) or infected (numbering y)

- Then, $x_0 = n, y_0 = 1$

  and at all times $x + y = n + 1$

Dynamic:

- Infected–uninfected contact turns the latter infected (and they stay infected)
- Other contacts have no consequence.

old mathematical branch of *Epidemiology* [Bailey 75]

Analysis (contd.)

Continuous time process

$$\frac{dy}{dt} = \beta xy$$ **(why?)**

with solution:

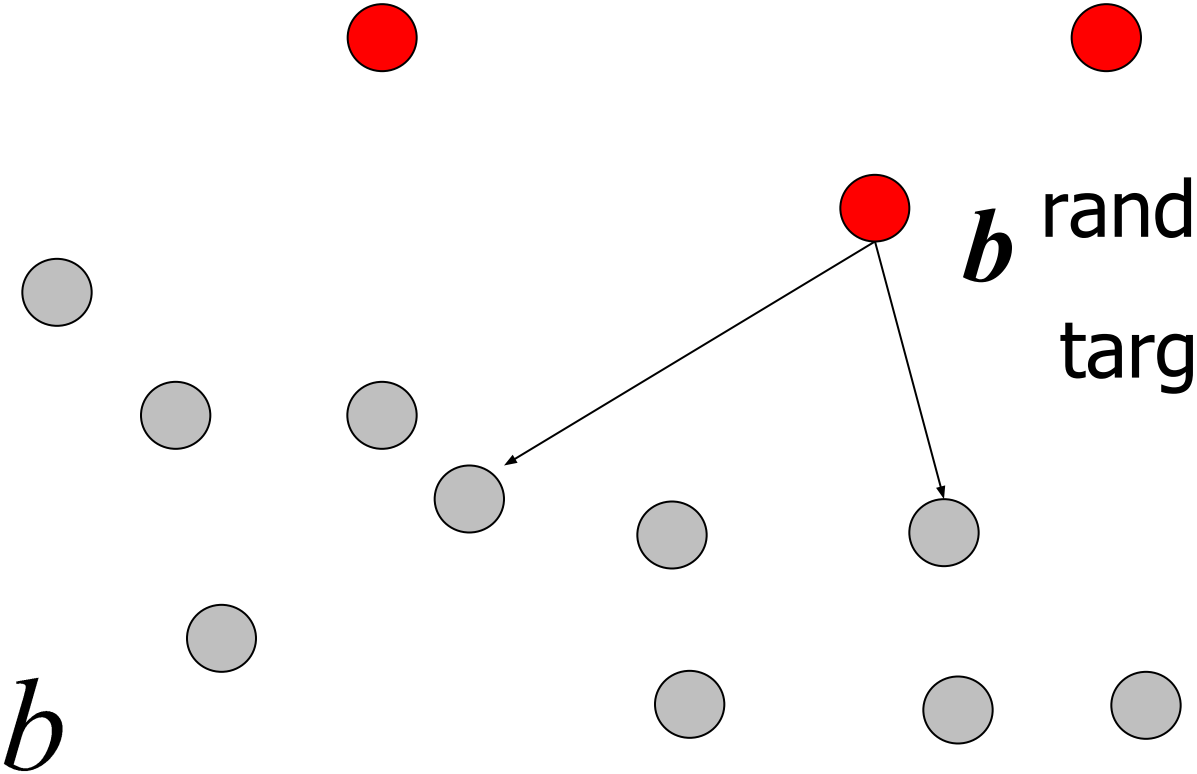
$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}},$$ **Not infected**

$$y = \frac{(n+1)}{1 + ne^{-\beta(n+1)t}}$$ **Infected**

(can you derive it?)

Epidemic Multicast

Protocol operates in rounds



INFECTED

b random

targets / round

SUSCEPTIBLE
(NOT INFECTED)

$$\beta = \frac{b}{n}$$

(reasonable approximation.)

Epidemic Multicast Analysis

Substituting, $\beta = \frac{b}{n}$

at time $\boldsymbol{t = c * log(n)}$ the number of infected nodes is

$$y \approx (n+1) - \frac{1}{n^{cb-2}}$$ **(infected)**

Answer – Push Analysis (contd.)

Using: $$\beta = \frac{b}{n}$$

Substituting, at time *t=clog(n)*

$$y = \frac{n+1}{1+ne^{-\frac{b}{n}(n+1)c\log(n)}} \approx \frac{n+1}{1+\frac{1}{n^{cb-1}}}$$

$$\approx (n+1)(1-\frac{1}{n^{cb-1}})$$

$$\approx (n+1)-\frac{1}{n^{cb-2}}$$

Analysis of push protocol (contd.)

Set *c*, *b* to be small numbers independent of *n*

Within *c* * *log(n)* rounds, **[low latency]**

- all but $\frac{1}{n^{cb-2}}$ number of nodes receive the multicast

[reliability]

- each node has transmitted no more than b * *c* * *log(n)* messages **[lightweight]**

Why is number of rounds [log(N)] low?

- log(N) is not constant
  - but pragmatically, it is a very slowly growing number
- Base 2
  - log(1000) ~ 10
  - log(1M) ~ 20
  - log (1B) ~ 30
  - log(all IPv4 address) = 32

Within *c* * *log(n)* rounds **[low latency]**

- all but $1/n^{cb-2}$ number of nodes receive the multicast **[reliability]**
- each node has transmitted no more than b * *c* * *log(n)* messages **[lightweight]**

Fault-tolerance

- Packet loss
  - 50% packet loss: analyze with *b* replaced with *b/2*
  - To achieve same reliability as 0% packet loss, takes twice as many rounds
- Node failure
  - 50% of nodes fail: analyze with *n* replaced with *n/2* and *b* replaced with *b/2*
  - Same as above

Within *c* * *log(n)* rounds **[low latency]**

- all but $1/n^{cb-2}$ number of nodes receive the multicast **[reliability]**
- each node has transmitted no more than b * *c* * *log(n)* messages **[lightweight]**

Fault-tolerance: With failures, is it possible that the epidemic might die out quickly?

- Possible, but improbable:
  - Once a few nodes are infected, with high probability, the epidemic will not die out
  - … the analysis we saw in the previous slides is actually behavior *with high probability* [Galey and Dani 98]

Intuition: Most rumors do spread fast. Infectious diseases do cascade quickly into epidemics. A computer virus or worm does spread rapidly.

PULL Protocol Analysis

- In all forms of push protocol, it takes O(log(N)) rounds before about N/2 processes get “infected”
  - This is an upper bound. Why? Because that’s the fastest you can spread a message: a spanning tree with constant fanout (degree) has O(log(N)) nodes

- Thereafter, pull is faster than push
- Let p_i be the fraction of non-infected processes after the i^{th} round
- Let each round have k pulls.
  - Then, for each process, the probability to stay non-infected in next round is $p_i^{\,k}$.
  - This makes p_i decrease faster than exponential
- Second half of pull gossip finishes in time O(log(log(N))

- Is this all theory and a bunch of equations?

  … or are there implementations and real use

- Two big categories:
  - Updating state
  - Computing aggregates

Example applications (Updating state)

- Updating replicas – distributing updates:
  - E.g., disconnected replicated address book maintenance – Demers et al., *Epidemic algorithms for replicated database maintenance*. SOSP'87
  - NNTP protocol

- Membership protocols:
  - Multiple-key value stores
    - e.g., Amazon Dynamo service: *Dynamo: Amazon's Highly Available Key-value Store*, SOSP'07
    - Cassandra,
    - Amazon S3, Uber
  - Various p2p networks (e.g., Tribler)

Example applications: aggregation functions

Sensor networks

The problem: compute the max value for a large set of sensors

The Solution

- Let every node *i* maintain a variable x_i. When two nodes engage in the protocol, they each reset their variable to

$$x_i, x_k \leftarrow \max(x_i, x_k)$$

- Repeat for log(N) rounds
- Result: in the end each node will have the max

Data aggregation: avg;

The problem: compute the average value for a large set of sensors

How would one use an epidemic style protocol to *calculate the number of sensors* in a sensor network.

- Recap of the idea
  - Update operations are initially performed at one node
  - Node passes its updated state to a limited number of ‘peers’ (*often chosen randomly*) …
  - … which, in-turn, pass the update to other peers
  - Eventually, each update will reach every node

- In what contexts is this kind of design is useful?
  - Frequent node failures
  - Update propagation can be lazy, i.e., not immediate
  - There are no write–write conflicts

- Questions: What if I drop each of these assumptions: why an epidemic solution may not be adequate?

Backup

Termination alternative

Anti-entropy protocols work in rounds:

- A round: each node takes the initiative to start one (or fixed number of) exchange(s)
- **Termination:** for push-pull it takes O(*log*(*N*)) rounds to disseminate updates to all *N* nodes
- But: Need to have some approximation of *N*

[Alternative] Gossiping (aka rumor mongering protocols)

- Individual node decisions

Gossiping

Basic model: A ‘contaminated’ node starts propagating the infection (contacts other randomly chosen nodes)

- All newly contaminated nodes proceed similarly

Key question: When does everyone stop?

- Intuition: the state of a randomly chosen node contains (probabilistic) information about the spread of the infection

Termination decision (individual):

```
while (true) {
  if the contacted node is already infected
     then stop with probability $^1/_k$
}
```

// (k allows tradeoffs between effort and likelihood to cover all nodes)

Gossiping: Termination decision

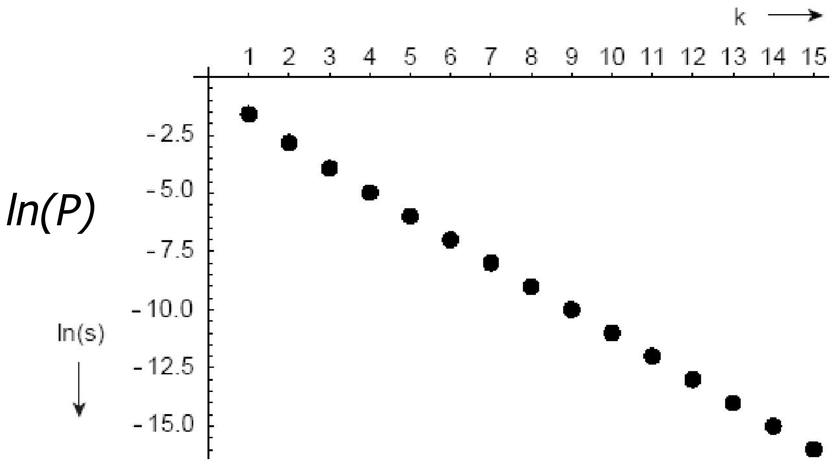
Termination decision:

```
While (true):
    if the contacted node already is already
       then stop with probability $^{1}/_{k}$.
```

P the share of nodes that have **not** been reached

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

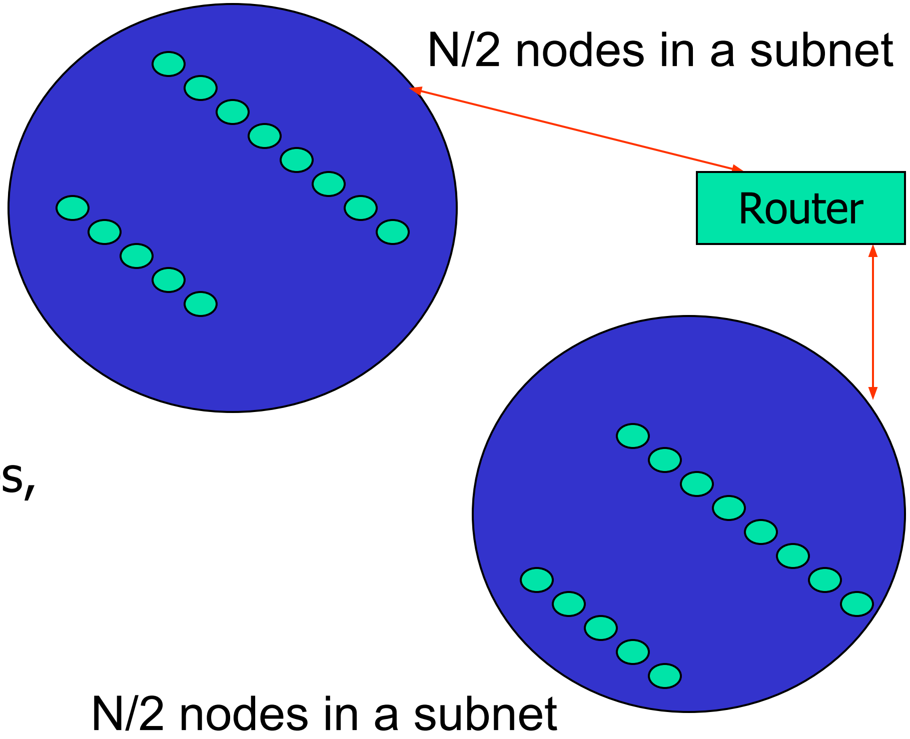
K	P
1	20.0%
2	6.0%
4	0.7%



Topology-Aware Epidemic Protocol

- Network topology is hierarchical
- Random push/pull target selection

- Fix:
  - In subnet *i*, which contains n_i nodes, pick target in your subnet with probability $(1-1/n_i)$
  - Router load=O(1)
  - Dissemination time=O(log(N))



Quiz-like questions

Discuss the tradeoffs between a multicast overlay and an epidemic protocol.

- Give motivated examples where one or the other solution may be more appropriate

Quiz-like questions

A distributed application operates on a large cluster. The application has one component running on each node.

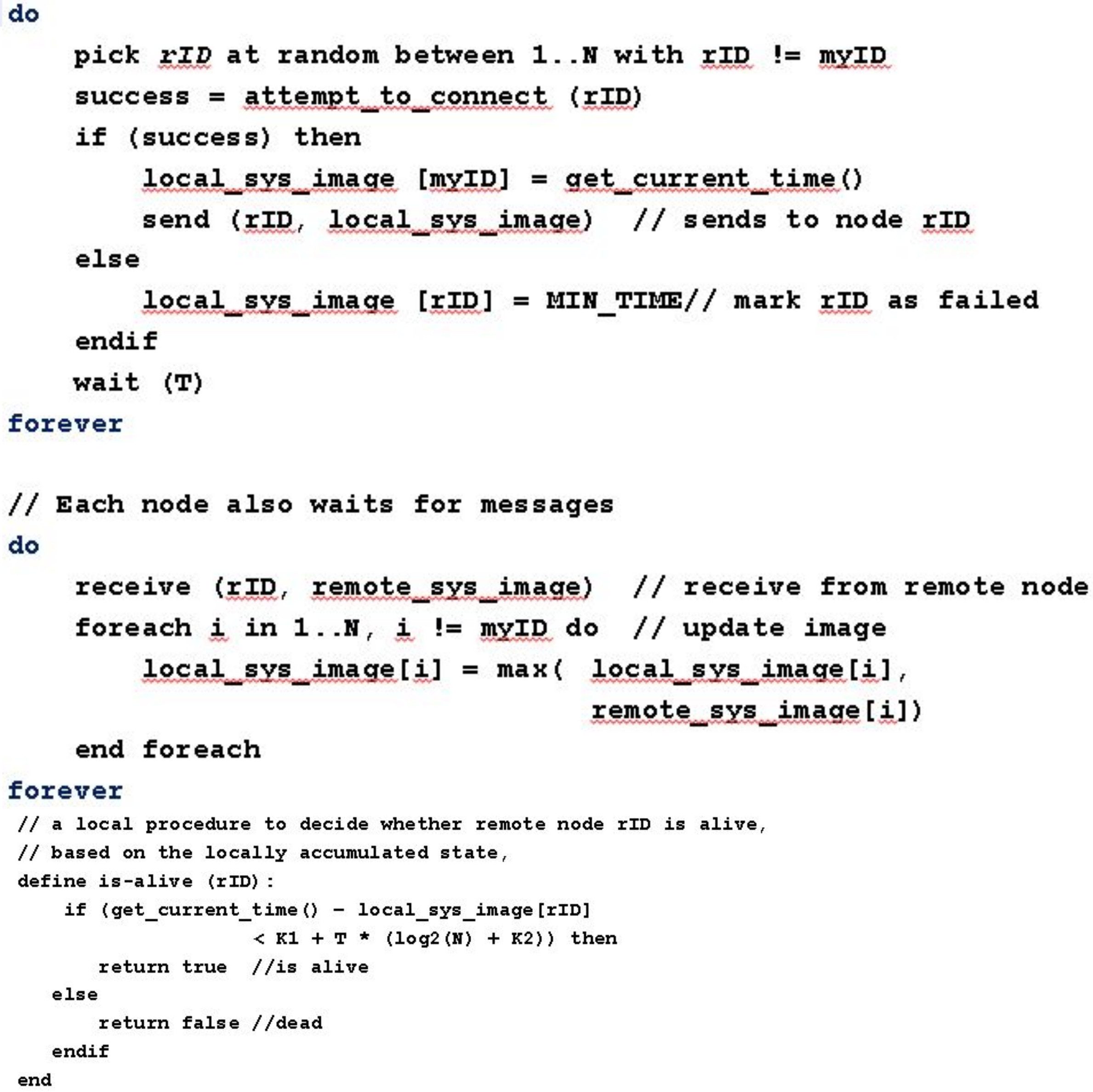
Cluster nodes might be shut down for maintenance, they might simply fail, or (new) nodes might come (back) online.

To function correctly each application component needs an accurate list of all other nodes/components that are active.

TO DO: Design a (*distributed*) mechanism that provides this list

- Describe the mechanism in natural language
- Provide the pseudocode.
- Evaluate overheads

Is this buggy? (Piazza: W4-Q1-2020)

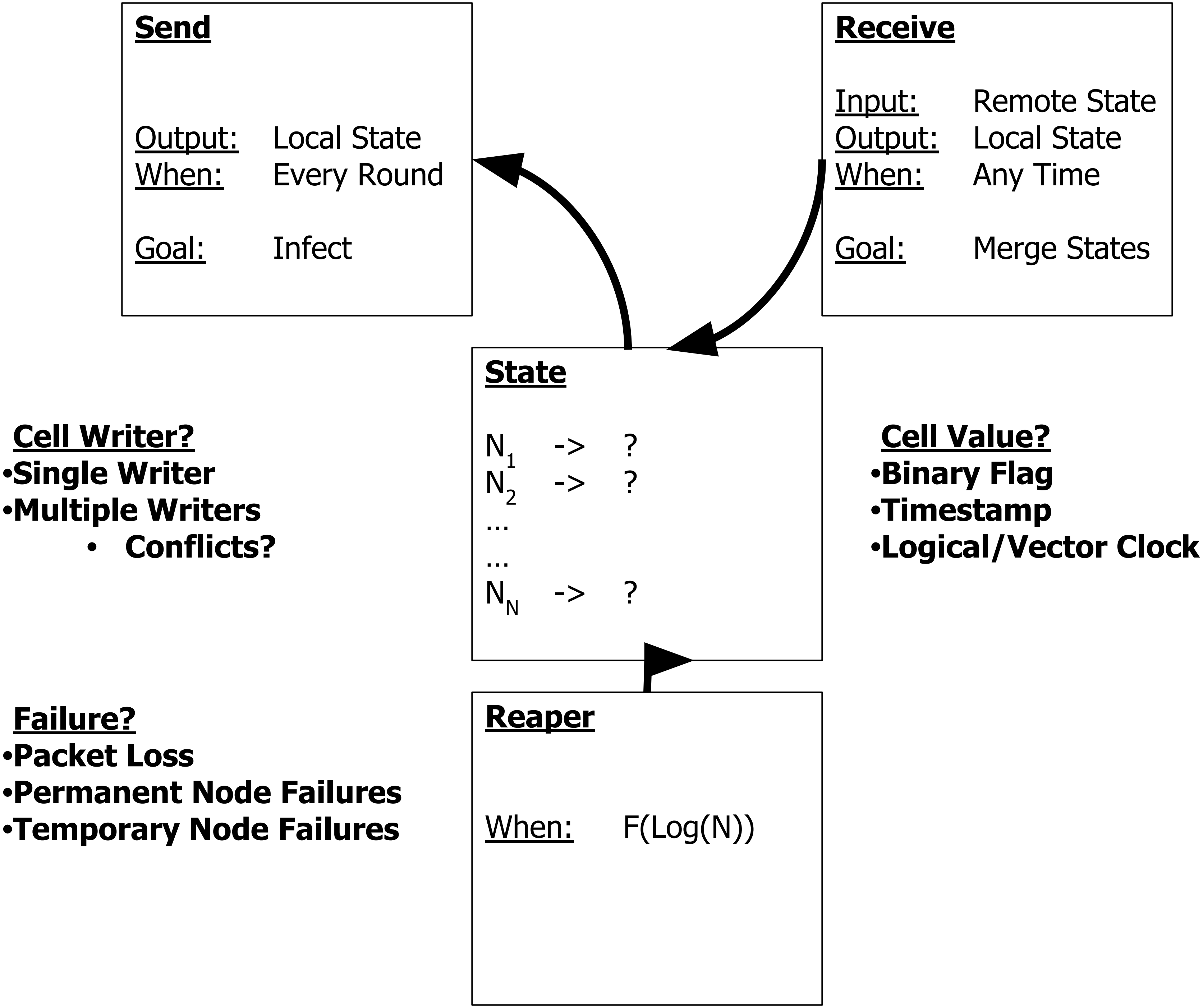


```
do
    pick rID at random between 1..N with rID != myID
    success = attempt_to_connect (rID)
    if (success) then
        local_sys_image [myID] = get_current_time()
        send (rID, local_sys_image)   // sends to node rID
    else
        local_sys_image [rID] = MIN_TIME// mark rID as failed
    endif
    wait (T)
forever

// Each node also waits for messages
do
    receive (rID, remote_sys_image)   // receive from remote node
    foreach i in 1..N, i != myID do   // update image
        local_sys_image[i] = max( local_sys_image[i],
                                  remote_sys_image[i])
    end foreach
forever
```

```
// a local procedure to decide whether remote node rID is alive,
// based on the locally accumulated state,
define is-alive (rID):
    if (get_current_time() - local_sys_image[rID]
                    < K1 + T * (log2(N) + K2)) then
        return true  //is alive
    else
        return false //dead
    endif
end
```

```
do
    pick rID at random between 1..N with rID != myID
    success = attempt_to_connect (rID)
    if (success) then
        local_sys_image [myID] = get_current_time()
        send (rID, local_sys_image)  // sends to node rID
    else
        local_sys_image [rID] = MIN_TIME// mark rID as failed
    endif
    wait (T)
forever

// Each node also waits for messages
do
    receive (rID, remote_sys_image)  // receive from remote node
    foreach i in 1..N, i != myID do  // update image
        local_sys_image[i] = max( local_sys_image[i],
                  remote_sys_image[i])
    end foreach
forever
```

```
// a local procedure to decide whether remote node rID is alive,
// based on the locally accumulated state,
define is-alive (rID):
    if (get_current_time() - local_sys_image[rID]
          < K1 + T * (log2(N) + K2)) then
       return true  //is alive
   else
       return false //dead
   endif
end
```



Send

Output: Local State
When: Every Round

Goal: Infect

Receive

Input: Remote State
Output: Local State
When: Any Time

Goal: Merge States

State

N_1 -> ?
N_2 -> ?
…
…
N_N -> ?

Cell Writer?

- **Single Writer**
- **Multiple Writers**
  - **Conflicts?**

Cell Value?

- **Binary Flag**
- **Timestamp**
- **Logical/Vector Clock**

Reaper

When: F(Log(N))

Failure?

- **Packet Loss**
- **Permanent Node Failures**
- **Temporary Node Failures**