Multicast

Node with a piece of information to be communicated to everyone

Distributed Group of “Nodes” = Processes at Internet-based host
Fault-tolerance and Scalability

Needs:
1. Reliability (Atomicity)
   - 100% receipt
2. Speed
Centralized

- Simplest implementation
- Problems?

UDP/TCP PACKETS
Tree-Based

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

UDP/TCP PACKETS
Tree-based Multicast Protocols

- Build a spanning tree among the processes of the multicast group
- Use spanning tree to disseminate multicasts
- Use either acknowledgments (ACKs) or negative acknowledgements (NAKs) to repair multicasts not received
- 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
  - But ACKs only sent to designated receivers, which then re-transmit missing multicasts
- These protocols still cause an O(N) ACK/NAK overhead [Birman99]
A Third Approach

MULTICAST SENDER
A Third Approach

PERIODICALLY, TRANSMIT TO
b RANDOM TARGETS

GOSSIP MESSAGES (UDP)
A Third Approach

OTHER NODES DO SAME
AFTER RECEIVING MULTICAST

GOSSIP MESSAGES (UDP)
“Epidemic” Multicast (or “Gossip”)
Push vs. Pull

- So that was “Push” gossip
  - Once you have a multicast message, you start gossiping about it
  - Multiple messages? Gossip a random subset of them, or recently-received ones, or higher priority ones
- There’s also “Pull” gossip
  - Periodically poll a few randomly selected processes for new multicast messages that you haven’t received
  - Get those messages
- Hybrid variant: Push-Pull
  - As the name suggests
Properties

Claim that the simple Push protocol

- Is lightweight in large groups
- Spreads a multicast quickly
- Is highly fault-tolerant
Analysis

From old mathematical branch of *Epidemiology* [Bailey 75]

- Population of \((n+1)\) individuals mixing homogeneously
- Contact rate between any individual pair is \(\beta\)
- 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 \(x + y = n + 1\)
- Infected–uninfected contact turns latter infected, and it stays infected
Analysis (contd.)

- Continuous time process
- Then

\[ \frac{dx}{dt} = -\beta xy \]  \hspace{1cm} (why?)

with solution:

\[ x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, \quad y = \frac{(n+1)}{1 + n e^{-\beta(n+1)t}} \]

(can you derive it?)
Epidemic Multicast

- **INFECTED**
- **PROTOCOL ROUNDS (LOCAL CLOCK)**
- **6 RANDOM TARGETS PER ROUND**
- **GOSSIP MESSAGE (UDP)**
- **UNINFECTED**
Epidemic Multicast Analysis

\[ \beta = \frac{b}{n} \]  

(why?)

Substituting, at time \( t=c\log(n) \), the number of infected is

\[ y \approx (n + 1) - \frac{1}{n^{cb-2}} \]

(correct? can you derive it?)
Analysis (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 $c \log(n)$ gossip messages [lightweight]
Why is log(N) low?

- log(N) is not constant in theory
- 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
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
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
  • So the analysis we saw in the previous slides is actually behavior with high probability
    [Galey and Dani 98]
• Think: why do rumors spread so fast? why do infectious diseases cascade quickly into epidemics? why does a virus or worm spread rapidly?
Pull Gossip: Analysis

• In all forms of gossip, it takes $O(\log(N))$ rounds before about $N/2$ processes get the gossip
  • Why? Because that’s the fastest you can spread a message – a spanning tree with fanout (degree) of constant degree has $O(\log(N))$ total nodes
• Thereafter, pull gossip is faster than push gossip
• After the $i$th round let $p_i$ be the fraction of non-infected processes. Let each round have $k$ pulls. Then

\[ p_{i+1} = \left( p_i \right)^{k+1} \]

• This is super-exponential
• Second half of pull gossip finishes in time $O(\log(\log(N)))$
Topology-Aware Gossip

• Network topology is hierarchical

• Random gossip target selection => core routers face $O(N)$ load (Why?)

• Fix: In subnet $i$, which contains $n_i$ nodes, pick gossip target in your subnet with probability $(1 - 1/n_i)$

• Router load = $O(1)$

• Dissemination time = $O(\log(N))$
Using: \( \beta = \frac{b}{n} \)

Substituting, at time \( t = c \log(n) \)

\[
y = \frac{n + 1}{1 + n e^{(n+1)c \log(n)}} \approx \frac{n + 1}{1 + \frac{1}{n^{cb-1}}} \\
\approx (n + 1)(1 - \frac{1}{n^{cb-1}}) \\
\approx (n + 1) - \frac{1}{n^{cb-2}}
\]
• Is this all theory and a bunch of equations?
• Or are there implementations yet?
Some implementations

- Clearinghouse and Bayou projects: email and database transactions [PODC ‘87]
- refDBMS system [Usenix ‘94]
- Bimodal Multicast [ACM TOCS ‘99]
- Sensor networks [Li Li et al, Infocom ‘02, and PBBF, ICDCS ‘05]
- AWS EC2 and S3 Cloud (rumored). [‘00s]
- Cassandra key-value store (and others) use gossip for maintaining membership lists
- Usenet NNTP (Network News Transport Protocol) [‘79]
NNTP Inter-server Protocol

1. Each client uploads and downloads news posts from a news server

2. Server retains news posts for a while, transmits them lazily, deletes them after a while.
Summary

- Multicast is an important problem
- Tree-based multicast protocols
- When concerned about scale and fault-tolerance, gossip is an attractive solution
- Also known as epidemics
- Fast, reliable, fault-tolerant, scalable, topology-aware