Gossip-based computing

Anne-Marie Kermarrec

CS-460

Where are we?

- Gossip Protocols
- Consistency protocols
  CAP Theorem
- Distributed/decentralized systems

Data science software stack

Data Processing

Transaction Management

Graph Data
Pregel, GraphLab, X-Stream, Chaos

Structured Data
Spark SQL

Machine Learning

Query Execution

Batch Data
Map Reduce, Dryad, Spark

Streaming Data
Storm, Naiad, Flink, Spark Streaming Google Data Flow

Data Storage

Storage Hierarchies & Layouts

Distributed File Systems
(GFS)

NoSQL DB
Dynamo
Big Table
Cassandra

Distributed Messging systems
Kafka

Ressource Management & Optimization

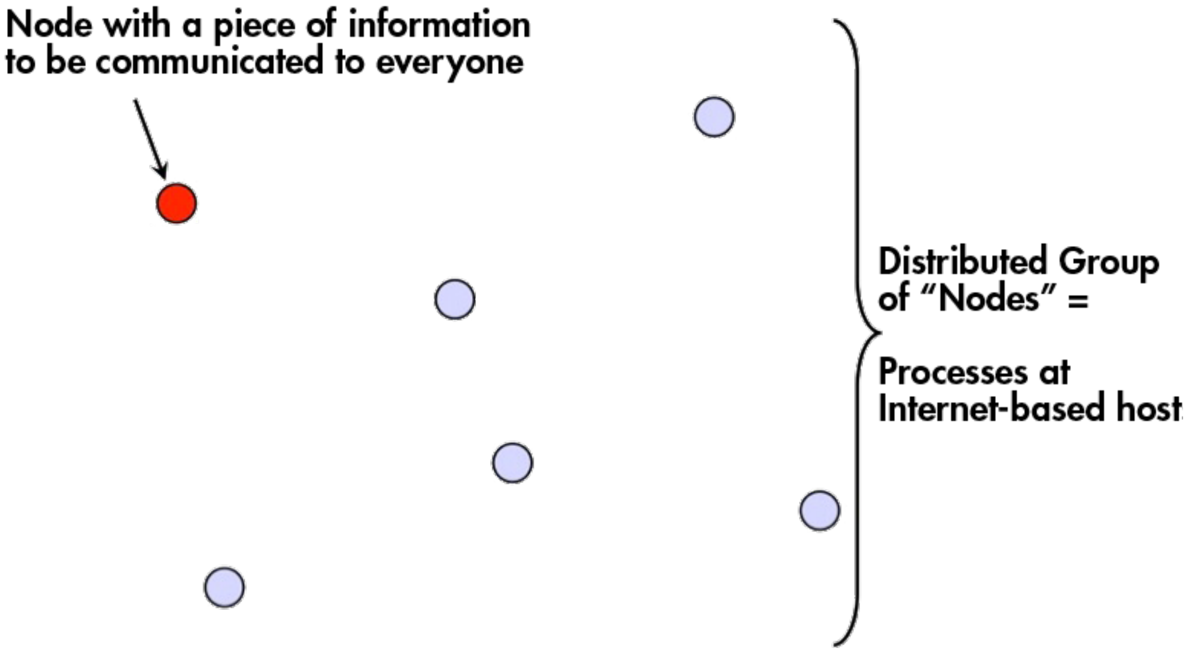
Query optimization

Scheduling (Mesos, YARN)

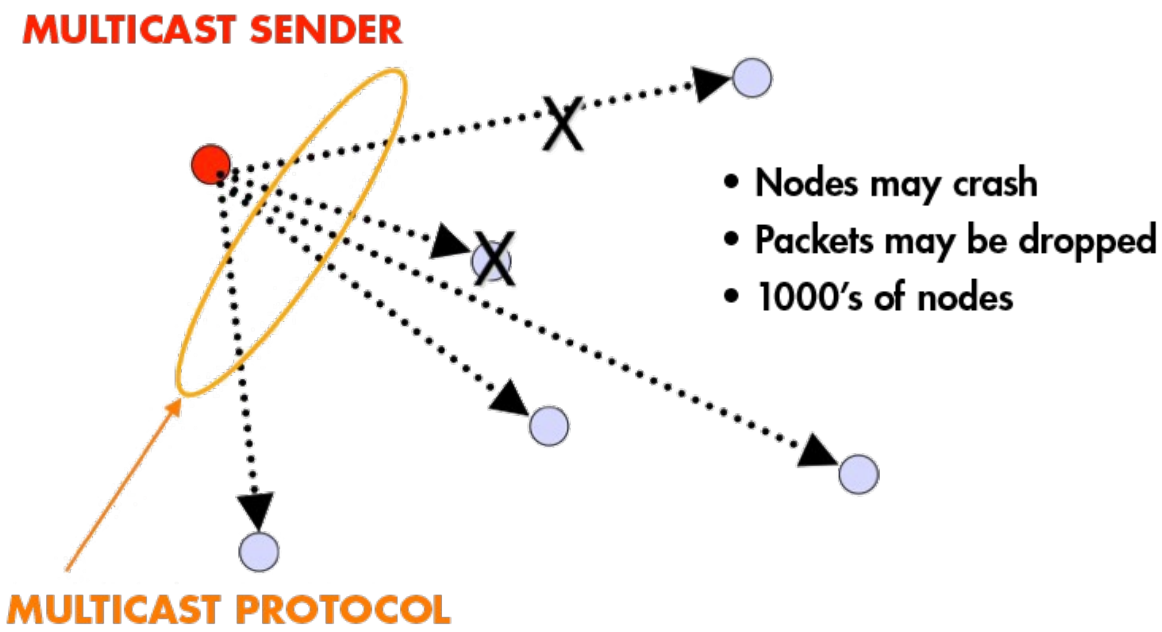
Dissemination - multicast

Consistency protocols
Event dissemination

- Key feature in distributed computing



Fault-tolerant dissemination

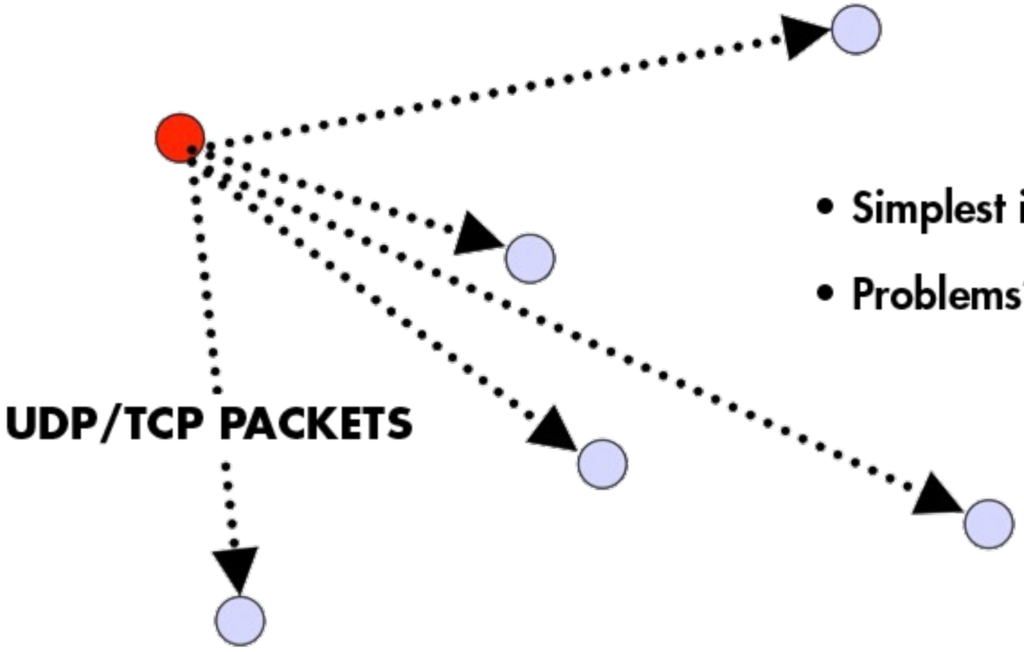


Atomicity: 100% nodes receive the message

Various topologies
- Star
- Chain

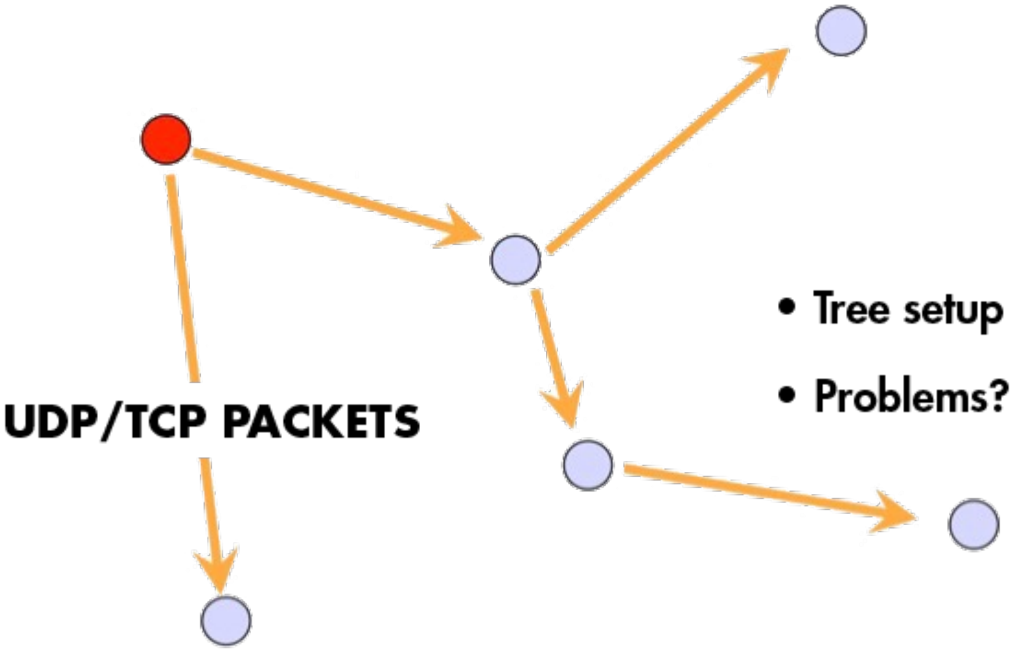
Trade-off: latency/load-balancing/failure resilience

Centralized: Star topology



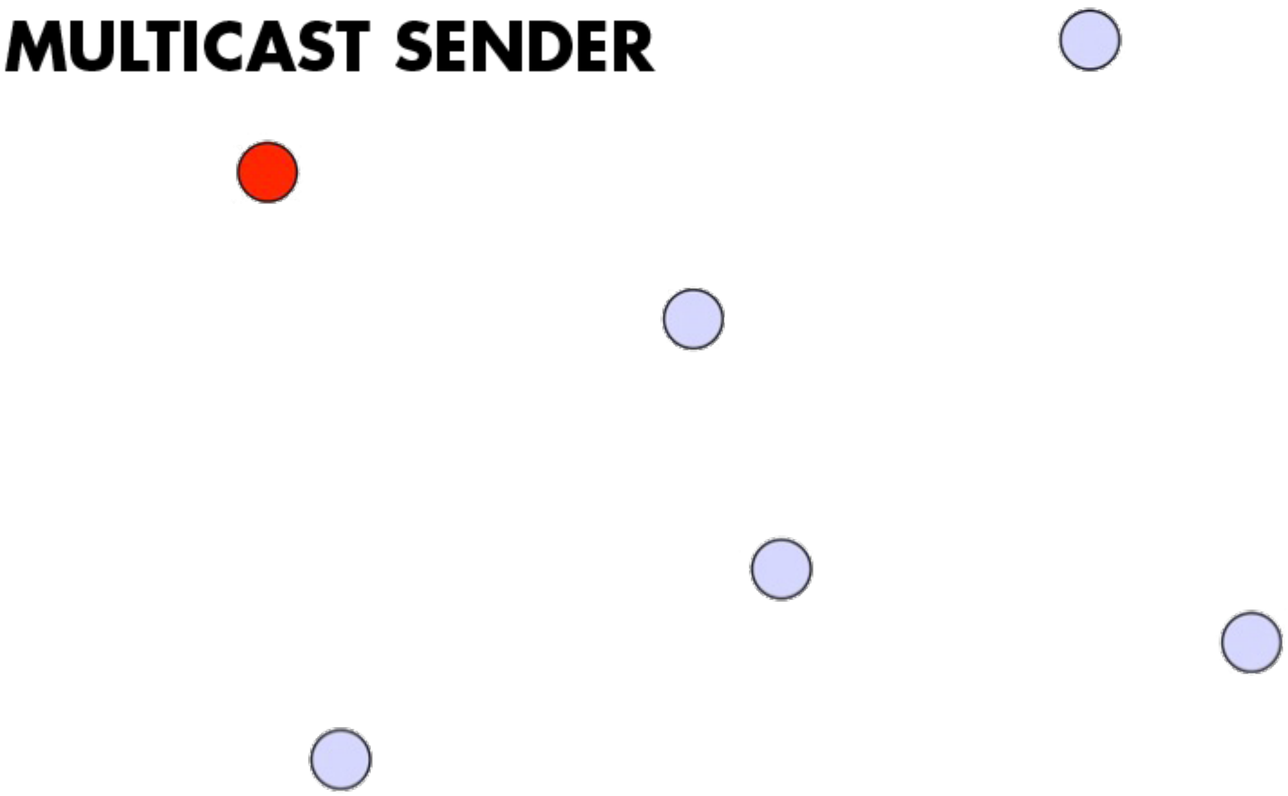
- Simplest implementation
- Problems?

Tree-based multicast

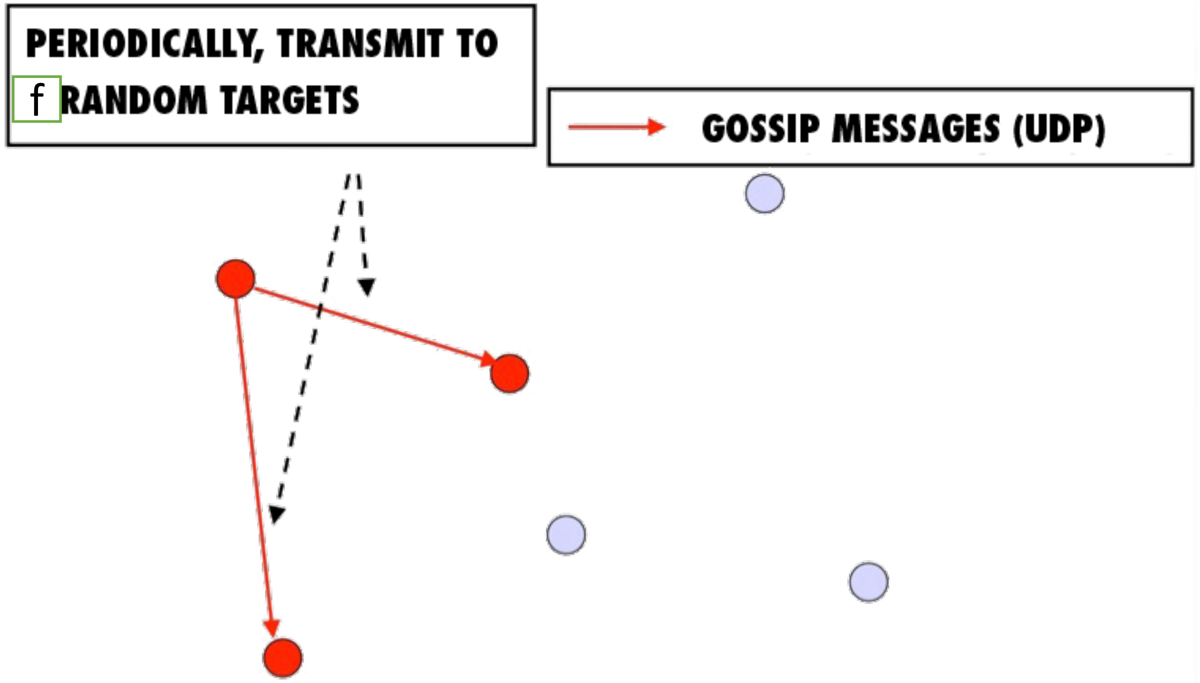


- Tree setup and maintenance
- Problems?

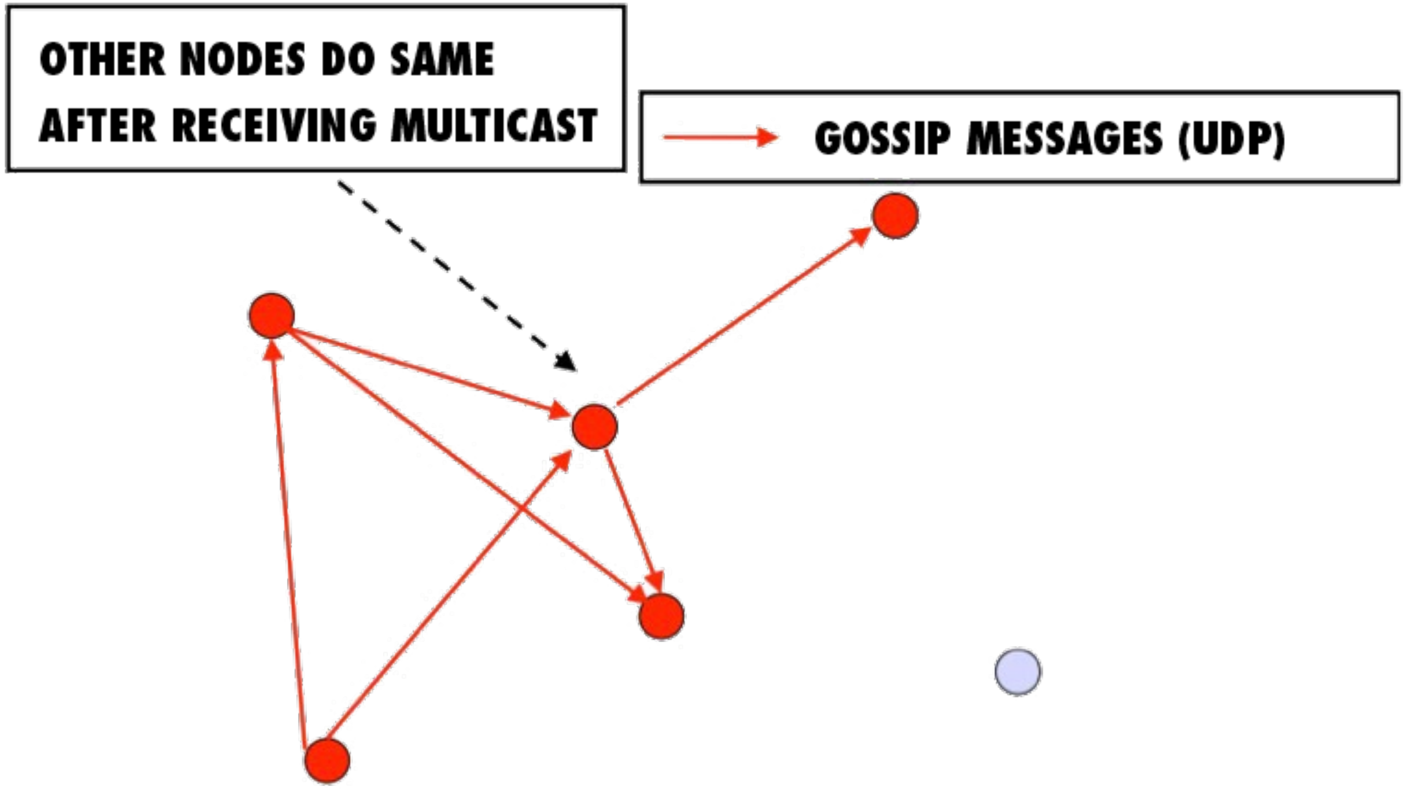
A Third Approach



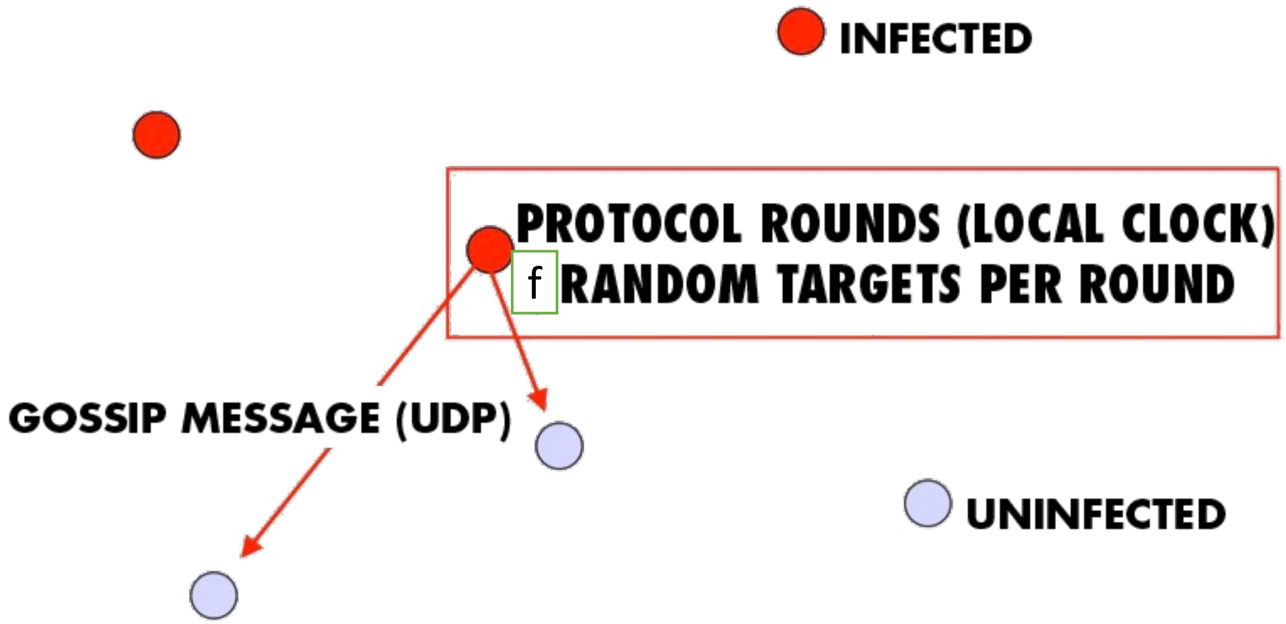
A Third Approach



A Third Approach



Epidemic/gossip-based dissemination



Simple

Reliable

Exponential Spreading

Principle

- Information is spread to allow for local-only decision making
  - Nodes exchange information with their neighbors: Peer to peer communication paradigm
  - Data disseminated efficiently
  - No centralized control
  - Eventual convergence: Probabilistic nature

Epidemic-based dissemination

- **Goal**:
  - Broadcast reliably a message to a large number of peers in a decentralized way
  - Proactive technique to tolerate failures
- **System model**
  - *n* processes
  - Each process forwards the message once to *f* (fanout) other nodes, picked up uniformly at random among the n nodes. Alternatively *f* times to 1 neighbor.
  - SIR model: Susceptible/Infected/Recovered
- **Metrics of the success of an epidemic process**
  - Proportion of infected processes

    $$Y_r = Z_r / n$$

    Z_r is the number of infected processes prior to round r
  - Probability of atomic "infection"

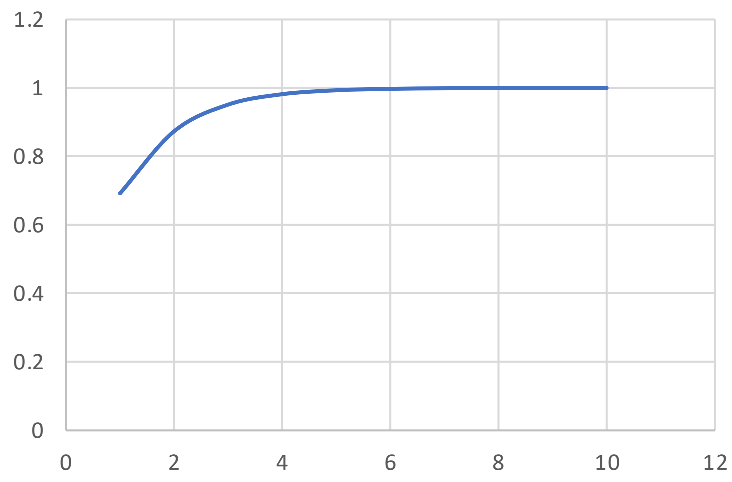
    $$P(Z_r = n)$$

Probability of "atomic" infection

Erdos/Renyi examine final system state, the system is represented as a graph where each node is a process, there is an edge from n_1 to n_2 if n_1 is infected and chooses n_2 .

An epidemic starting at n_0 is successful if there is a path from n_0 to all members. If the fanout is $\log(n) + c$, the probabibility that a random graph is connected is

$$p(connect) = e^{-e^{-c}}$$



The log(n) magic

log(n) is a very slowly growing number
Base 2

log(1000) ~ 10
log(1M) ~ 20
log (1B) ~ 30

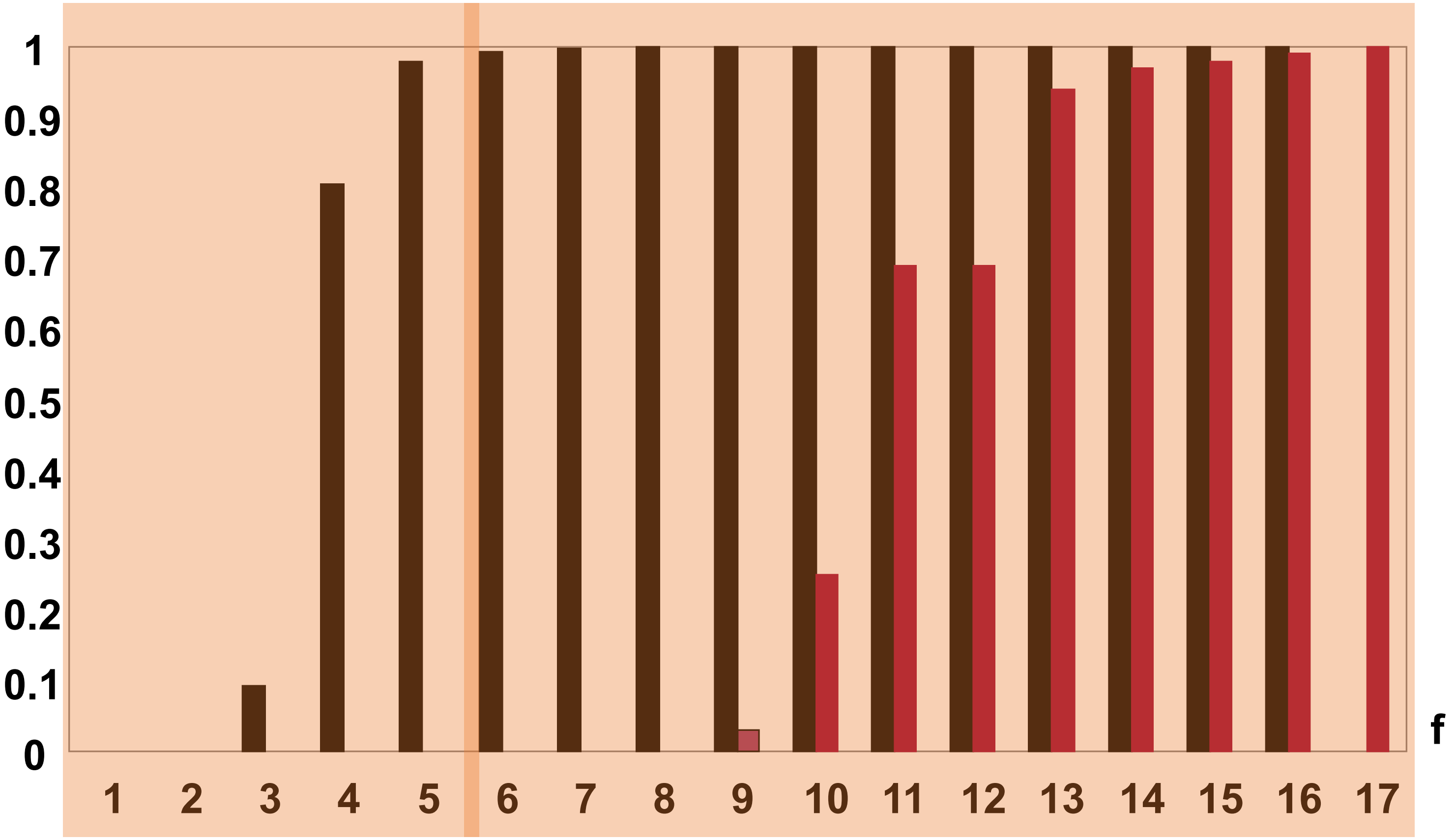
- Simple dissemination algorithm

- Probabilistic guarantees of delivery

- Each node forwards the message to *f* nodes chosen uniformly at random
  - If *f=O(log(n))*, "atomic" broadcast whp in *O(log(n))* hops
  - Result is valid if the fanout for each peer is on average *log(N) + c*, regardless of the degree distribution.

- Relate probability of reliable dissemination and proportion of failure
  - Set parameters

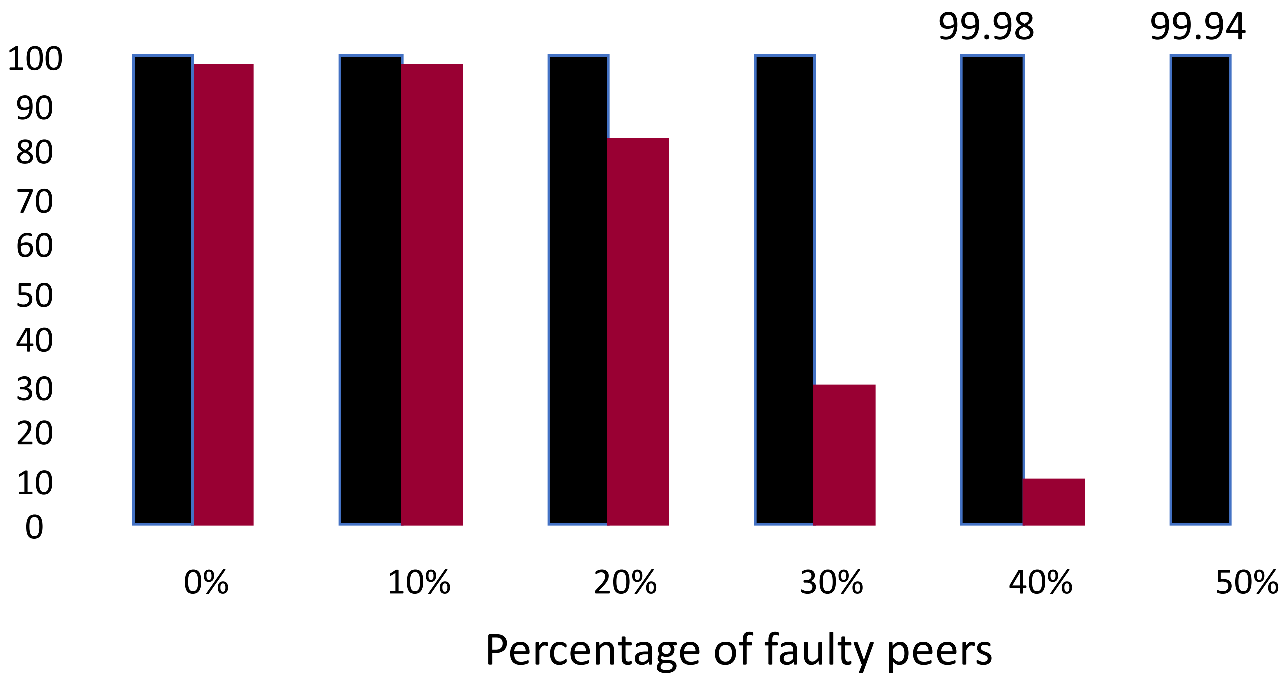
Performance (100,000 peers)



Proportion of "atomic"broadcast

Proportion of connected peers in non "atomic" broadcast

Failure resilience (100,000 peers)



Proportion of "atomic" broadcast

Proportion of connected peers in non "atomic" broadcast

Push versus Pull protocols

- "Push" protocols
  - Once a node receives a multicast message, it starts gossiping about it typically by forwarding it to f nodes
- "Pull" protocols
  - Periodically a node sends a request to f randomly selected processes for new multicast messages that it has not received.
- Hybrid variant: Push-Pull
  - As the name suggests

The relevance of gossip

- Introduces implicit redundancy
- Flexible and simple protocols
- Overhead
  - Small messages
  - Application to maintenance, monitoring, etc…

Differ in the choice of gossip targets and information exchanged

Basic functionnality

- Require a uniform random sample
- How can we do this in a decentralized way?

Achieving random topologies through gossiping

The peer sampling service

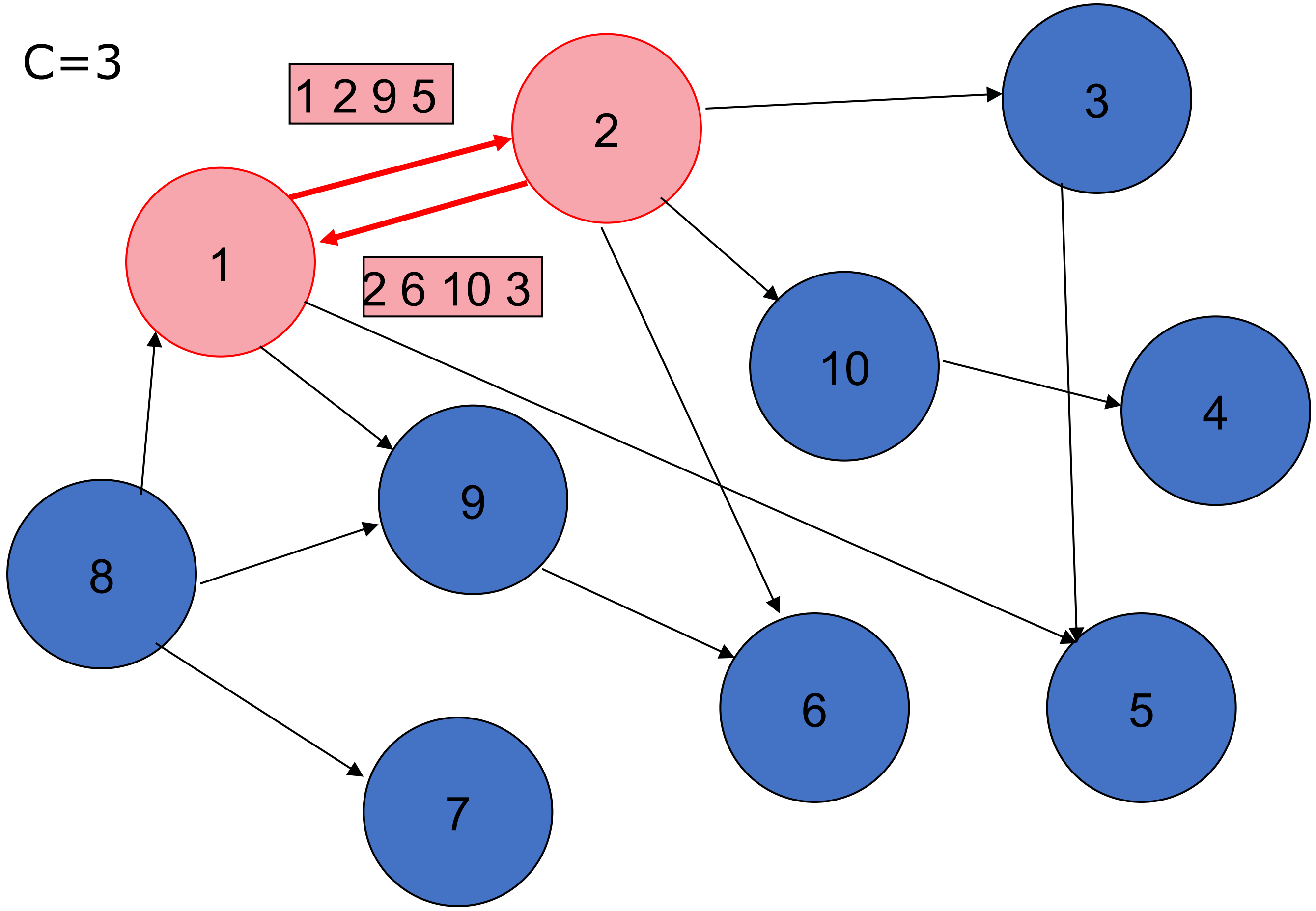
- How to create a graph upon which applying gossip-based dissemination?... **By gossiping**
- **Goal:**
  - Create an overlay network
  - Provide each peer with a **random** sample of the network in a decentralized way
- **Means:** gossip-based protocols
  - What data should be gossiped?
  - To whom?
  - How to process the exchanged data?
- Resulting "who knows who" graphs: overlay
  - Properties (degree, clustering, diameter, etc.)
  - Resilience to network dynamics
  - Closeness to random graphs

Objective

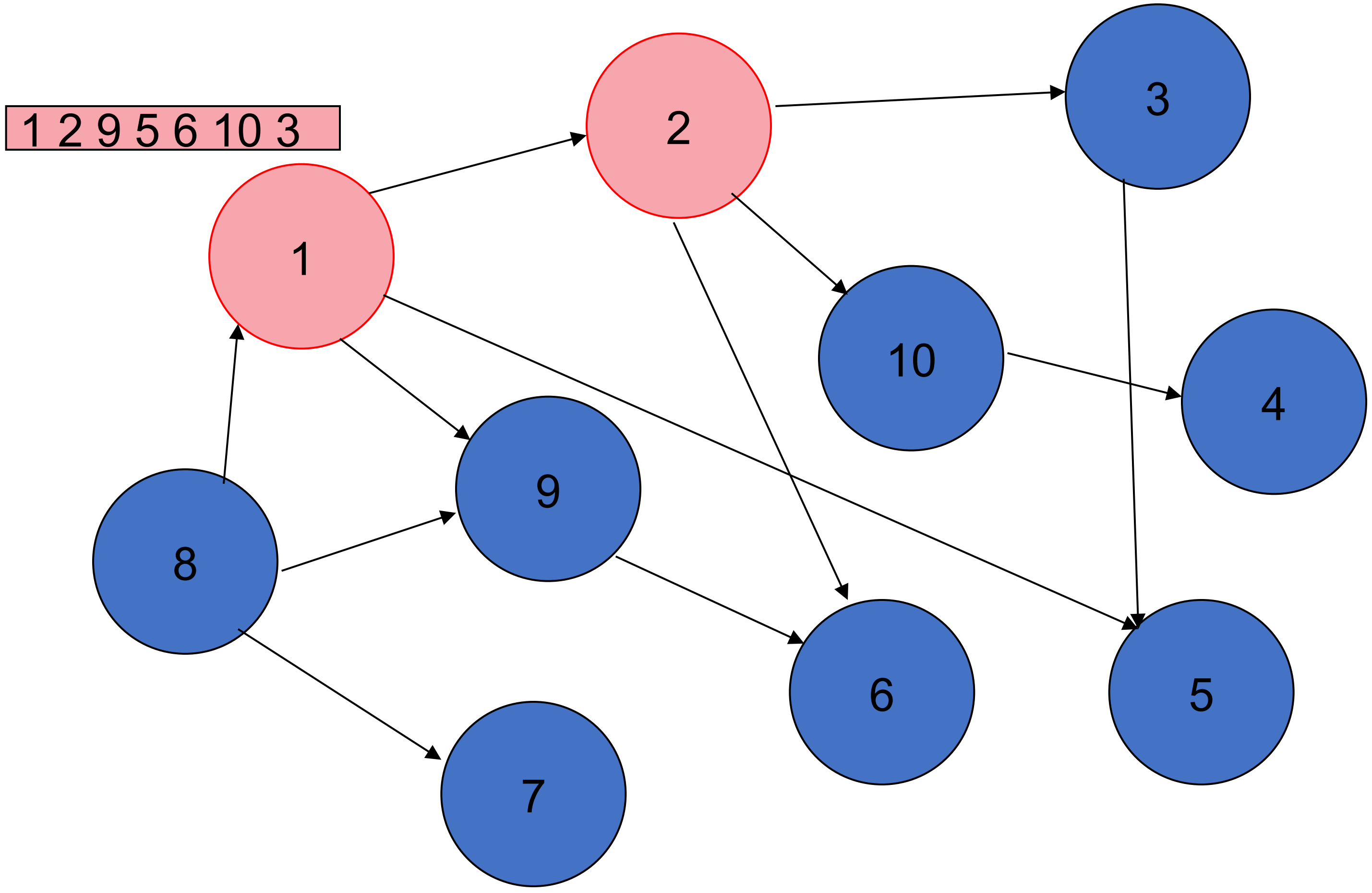
- Provide nodes with a peer drawn uniformly at random from the complete set of nodes
- Sampling is accurate: reflects the current set of nodes
- Independent views
- Scalable service

EPFL

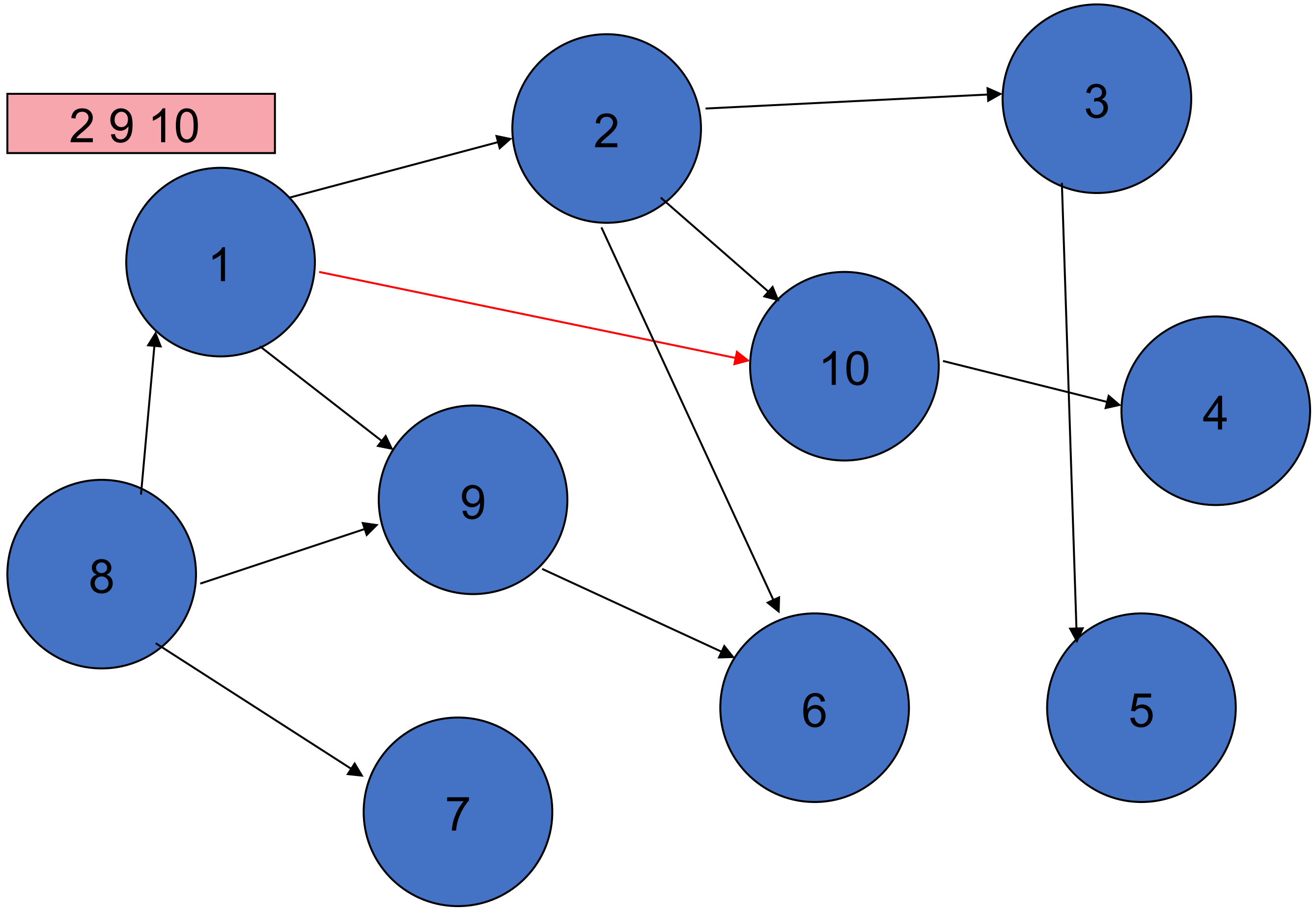
Example: Gossip-based generic protocol



Example: Gossip-based generic protocol



Example: Gossip-based generic protocol



System model

- System of n peers
- Peers join and leave (and fail) the system dynamically and are identified uniquely (IP @)
- Epidemic interaction model:
  - Peers exchange some membership information periodically to update their own membership information
  - Reflect the dynamics of the system
  - Ensures connectivity
- Each peer maintains a local view (membership table) of c entries
  - Network @ (IP@)
  - Age (freshness of the descriptor)
  - Each entry is unique
  - Ordered list
- Active and passive threads on each node

Operations on partial view

selectPeer()	returns an item
permute()	randomly shuffles items
increaseAge()	forall items add 1 to age
append(...)	append a number of items
removeDuplicates()	remove duplicates (on same address), keep youngest
removeOldItems(n)	remove *n* descriptors with highest age
removeHead(n)	remove *n* first descriptors
removeRandom(n)	remove *n* random descriptors

Active Thread

```
Wait (T time units) // T is the cycle length
P <- selectPeer() // Sample a live peer from the current view
if push then // Takes initiative
    myDescriptor <- (my@,0)
    buffer <- merge (view, {myDescriptor}) //temporary list
    view.permute() //shuffle the items in the view
    move oldest h items to end of the view //to get rid of old nodes
    buffer.append(view.head(c/2)) // copy first half of the items
    send buffer to p
else send{} to p //triggers response
if pull then
    receive buffer from p
    view.selectView(c,h,S,buffer)
view.increaseage(view_p)
```

Passive Thread

```
Do foreever

Receive buffer_p from p

if pull then
    myDescriptor <-(my@,0)
    buffer <-merge(view,{myDescriptor})
    view.permute ()
    move oldest h items to end of the view
    buffer.append(view.head(c/2))
    send buffer to p

view.selectView(c,h,S,buffer)
view.increaseage(view_p)
```

Design space

- Periodically each peer initiates communication with another peer
- **Peer selection**

- **Data exchange (View propagation)**
  - How peers exchange their membership information?

- **Data processing (View selection)**: Select (c, buffer)
    - c: size of the resulting view
    - Buffer: information exchanged

Design space: peer selection

selectPeer(): returns a live peer from the current view

- *Rand*: pick a peer uniformly at random
- *Head*: pick the "youngest" peer
- *Tail*: pick the "oldest" peer

Note that *head* leads to correlated views.

View propagation

- push: Node sends descriptors to selected peer
- pull: Node only pulls in descriptors from selected peer
- pushpull: Node and selected peer exchange descriptors

Pulling alone is pretty bad: a node has no opportunity to insert information on itself. Potential loss of all incoming connections.

Design space: data exchange

- **Buffer (h)**
  - initialized with the descriptor of the gossiper
  - contains *c/2* elements
  - ignore *h* “oldest”
- **Communication model**
  - Push: buffer sent
  - Push/Pull: buffers sent both ways
  - (Pull: left out, the gossiper cannot inject information about itself, harms connectivity)

Design space: Data processing

- Select(*c,h,s,buffer*)

1. Buffer appended to view
2. Keep the freshest entry for each node
3. *h* oldest items removed
4. *s* first items removed (the one sent over)
5. Random nodes removed

c: size of the resulting view
h: self-healing parameter
s: shuffle
Buffer: information exchanged

- Merge strategies
  - Blind (*h=0,s=0)*: select a random subset
  - Healer (*h=c/2*): select the “freshest” entries
  - Shuffler (*h=0, s=c/2*): minimize loss

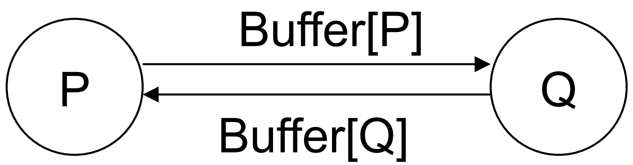
Existing systems

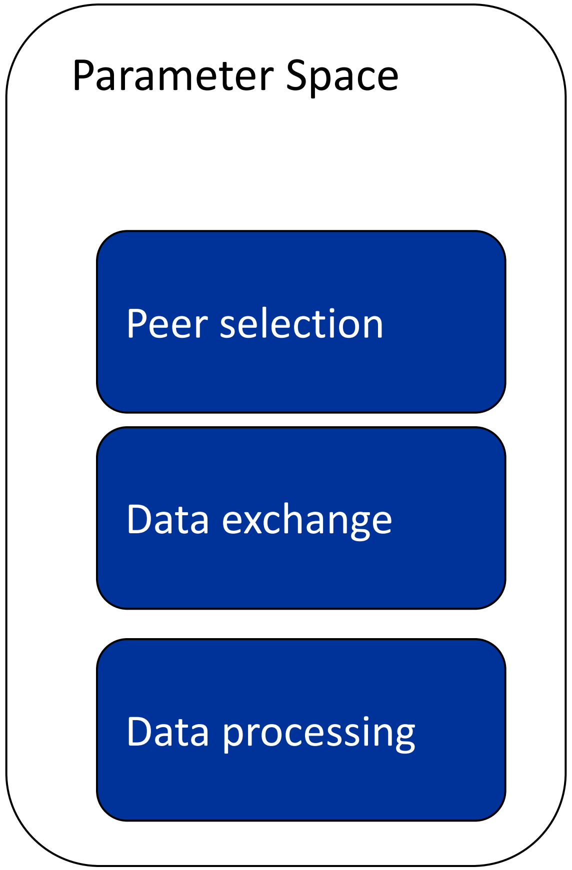
- Lpbcast [Eugster & al, DSN 2001,ACM TOCS 2003]
  - Node selection: random
  - Data exchange: push
  - Data processing: random
- Newscast [Jelasity & van Steen, 2002]
  - Node selection: head
  - Data exchange : pushpull
  - Data processing : head
- Cyclon [Voulgaris & al JNSM 2005]
  - Node selection: random
  - Data exchange : pushpull
  - Data processing : shuffle

A generic gossip-based substrate

Gossip-based generic substrate

- Each node maintains a set of neighbors (*c* entries)
- Periodic peerwise exchange of information
- Each process runs an active and passive threads





A generic gossip-based substrate

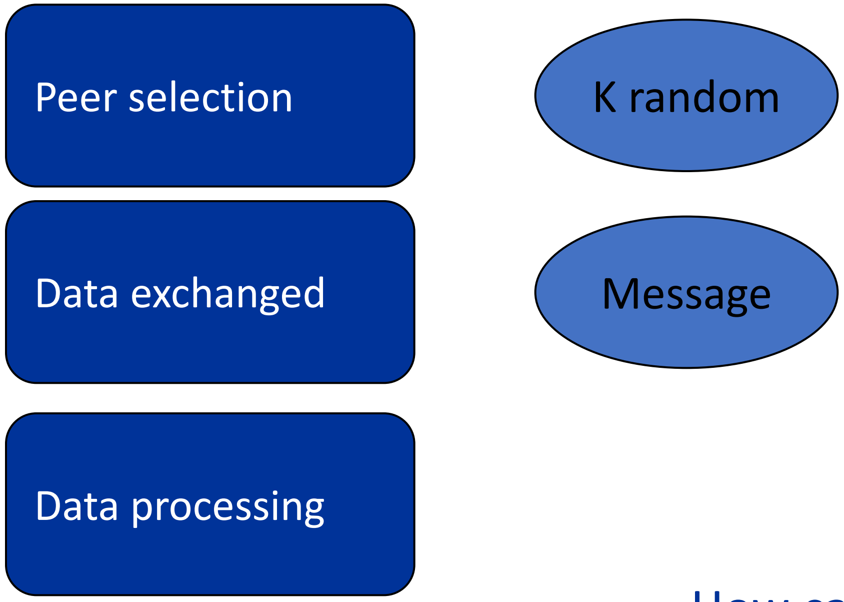
Active thread (peer P)		Passive thread (peer Q)
(1) selectPeer (&Q);		(1)
(2) selectToSend(&bufs);		(2)
(3) sendTo(Q,bufs);	→	(3) receiveFrom(&P,&bufr);
(4) -		(4) selectToSend(&bufs);
(5) receiveFrom(Q,&bufr);	←	(5) sendTo(P,bufs);
(6) selectToKeep(view,bufr);		(6) selectToKeep(view,bufr);
(7) processData(view)		(7) processData(view)

selectPeer: (randomly) select a neighbor

selectToSend: select some entries from local view

selectToKeep: add received entries to local view

Gossip-based dissemination



Dissemination
Data = msg to broadcast

Each process gossips one message once

How can we achieve Random sampling?

Overlay maintenance

	LpbCast [Eugster & al, DSN 2001, ACM TOCS 2003]	Newscast [Jelasity & van Steen, 2002]	Cyclon [Voulgaris & al JNSM 2005]
Peer selection	Random	Head	Oldest
Data exchange **Membership data**	List of Neighbours Push	½ List of Neighbours PushPull	½ List of neighbours
Data processing	Random merging	Age-based Merging (Head)	Shuffle

Decentralized computations

	Aggregation [Jelasity & al., ACM TCS 20025]	System size estimation	Slicing
Peer selection	Random	Random	Random
Data exchange	value	value	Attribute value Random value
Data processing	Aggregation Average	Aggregation	Attribute/random matching

Gossip-based aggregation

- Each node holds a numeric value s
- Aggregation function: average over the set of nodes

do exactly once in each consecutive
δ time units at a randomly picked time
  $q \leftarrow$ GETNEIGHBOR()
  send s_p to q
  $s_q \leftarrow$ receive(q)
  $s_p \leftarrow$ UPDATE(s_p, s_q)

(a) active thread

do forever
  $s_q \leftarrow$ receive(*)
  send s_p to sender(s_q)
  $s_p \leftarrow$ UPDATE(s_p, s_q)

(b) passive thread

Gossip-based aggregation

- Assume getneighbor() returns a uniform random sample
- Update(s_p,s_q) returns $(s_p + s_q)/2$
- Operation does not change the global average but redistributes the variance over the set of all estimates in the system
- Proven that the variance tends to zero
- Exponential convergence

```
// vector w is the input
do N times
  (i, j) = GETPAIR()
  // perform elementary variance reduction step
  w_i = w_j = (w_i + w_j)/2
return w
```

Counting with gossip

- Initialize all nodes with value 0 but the initiator
- Global average = 1/N
- Size of the network can be easily deduced
- Robust implementation
  - Multiple nodes start with their identifier
  - Each concurrent instance led by a node
  - Message and data of an instance tagged with a unique Id

Ordered Slicing

- Create and maintain a partitioning of the network
- Each node belongs to one slice
- Ex: 20% of nodes with the largest bandwidth
- Network of size N
- Each node i has an attribute x_i
- We assume that values (x_1, x_N) can be ordered
- Problem: automatically assign a slice (top 20%) for each node

Where is that used in practice?

- 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) uses gossip for maintaining membership lists
- Bitcoin uses gossip for all communications (pre and post mining)
- Federated and decentralized learning

References

- « The peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementation » M. Jelasity, R. Guerraoui, A.-M. Kermarrec and M. van Steen, Middleware 2004 – ACM TOCS 2007
- « Newscast Computing » M. Jelasity, W. Kowalczyk, M. van Steen. Internal report IR-CS-006, Vrije Universiteit, Department of Computer Science, November 2003
- « Lightweight Probabilistic Broadcast ». P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov ACM Transactions on Computer Systems, 21(4), November 2003.
- « Peer-to-Peer membership management for gossip-based protocols ». A.J. Ganesh, A.-M. Kermarrec, and L. Massoulié IEEE Transactions on Computers, 52(2), February 2003
- "Gossip-based aggregation in large dynamic networks" M. Jelasity, A. Montresor, O. Babaoglu. ACM TCS 23(3), 2005