Emanuele Carlini
emanuele.carlini@isti.cnr.it
These slides took inspiration from slides of Laura Ricci, Alberto Montresor and Amir Payberah. Also, the chapter “Gossip” from Mark Jelasity has been taken as inspiration.
Introduction
Gossip plays and played a key role in human society
Gossip plays and played a key role in human society

“language in fact evolved in response to our need to keep up to date with friends and family”

Grooming, Gossip, and the Evolution of Language
Robin Dunbar, 1996
Human Gossip

• Human gossip is a large area of research

• At its core, gossip is an highly effective solution when it comes to spread information
  • fast, information spread very quickly
  • resistant, it is very difficult to stop the spreading (no central control)

• Bio-inspired process: information disseminates like a virus
It went viral!

- Viral marketing is one of the most examples of gossip-like information diffusion in social network
- also: memes, (fake) news, conspiracy theories
It went viral!

• Viral marketing is one of the most examples of gossip-like information diffusion in social networks.
• Also: memes, (fake) news, conspiracy theories.
Small effort, big impact

- That kitten video got 70M+ visualisation thanks to the power of gossip in social networks

- a small effort (sharing on SN) from many persons achieve a fast and wide dissemination of information

The ultimate feature behind the power of gossip is its simplicity
From Human Gossip to Distributed Protocol
Gossip in distributed applications

• A gossip protocol is a way to distribute information in a network of nodes based on message exchange.

• A gossip protocol mimics the way information spread in the real world, and applies it to a network:
  • Nodes infect each other (transmit information) through contact (messages).

• The idea of the protocol is rather simple: send data to some nodes, which in turn send it to some other nodes, which in turn send it to some other nodes, which in turn send it to some other nodes, which in turn send it to some other nodes.
Why is good

• Gossip is a very attractive way to handle information dissemination in distributed network
  • Simple: to model and to program
  • Scalable: effort is distributed among nodes
  • Fast (relatively): nodes work in parallel
  • Robust (relatively): to node failures, loss of messages, network partitioning
When to not use Gossip

- When there are strong requirements on the latency of information updates
  - real time systems
  - live video streaming
- When there are strong requirements on consistency
- **Eventual consistency**: If no new information updates are injected, after some time $t$ eventually all nodes will have the same copy of the information
  - problem is: $t$ can be very long
Gossip vs Tree vs Flooding

- Gossip trades speed to achieve scalability and robustness
Design assumptions

- Typical environment of a P2P application running on top of the Internet

- **Nodes are part of a network**: any node can reach every node if it knows an address (e.g. the Internet)

- **No shared memory**: nodes cannot access to the other nodes’ memory directly

- **Asynchrony**: no bound on execution times, or requirement on messages latency

- **Failures**: nodes can fail, message can be lost
Network of Nodes

- Each node has a list of neighbours $N$

- Each node has an internal state $S$

---

**Diagram:**

- Node 1: $N = \{2, 4\}$, $S = \{v1\}$
- Node 2: $N = \{1, 3, 4, 6\}$, $S = \{v2\}$
- Node 3: $N = \{2, 5\}$, $S = \{v3\}$
- Node 4: $N = \{2, 1\}$, $S = \{v4\}$
- Node 5: $N = \{3, 6\}$, $S = \{v5\}$
- Node 6: $N = \{2, 5\}$, $S = \{v6\}$
Simple gossip template

- Two behaviours: **active** (send messages) and **passive** (process messages)

<table>
<thead>
<tr>
<th>active behaviour()</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = select_node</td>
</tr>
<tr>
<td>data = send_state(p)</td>
</tr>
<tr>
<td>update_state(data)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>passive behaviour()</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = receive_from(q)</td>
</tr>
<tr>
<td>update_state(data)</td>
</tr>
<tr>
<td>send_state(q)</td>
</tr>
</tbody>
</table>
Defining Operations

**active behaviour()**

```
p = select_node
data = send_state(p)
update_state(data)
```

**passive behaviour()**

```
data = receive_from(q)
update_state(data)
send_state(q)
```

**select_node()**  choose a neighbour to communicate with

**update_state(data)**  update the internal node state

**send_state(p)**  send the internal state to a node
Possible applications

• Gossip enables efficient and robust dissemination of information

• Application-level information dissemination are important building blocks for today’s distributed applications

  • Datacenter monitoring

  • File sharing

  • Sensor networks
Currently, used in:

- **Amazon S3** (Simple Storage System): information diffusion
  - gossip to detect the available servers and the node state
- **Amazon Dynamo**: failure detection and membership service
- **Cassandra**: distributed database (exploited in the first version of Facebook)
- **BitTorrent**: information diffusion within a swarm
Information Dissemination (broadcast)
Information dissemination

A source node wants to deliver a piece of information to all the nodes in the network.
Origins of the problem

- In 1986 Xerox Clearinghouse Servers had to sync all their 300 DNS databases all over the world
  - All of them producing updates, and replica needs to be the same at each site
  - Their solution: at night, each server sends emails with the updates. For a domain stored at 300 sites 90,000 mail messages might he introduced each night
  - At the time there was no Internet, lines were slow, few bandwidth, and high message lost rate
  - The solution didn't scale, at the end of the night the databases were in an inconsistent state (manual interventions to restore consistency)
First Solution

• 1987: Alan Demers et al., proposed (the first?) epidemic protocol to disseminate database entry updates in the networks

  • Anti Entropy + Rumor mongering

• Solved the problem: scales with the number of servers, copes with the network unreliability
Information like a virus

- Gossip models the information diffusion like a virus diffusion within a population of persons
  - Virus transmits when two persons meet (information is the virus, the network is the population)
- **Susceptible**: A person not yet infected, but may become infected if comes in contact with an infected person;
- **Infective**: A person that has contracted the virus;
- **Recovered**: A person that has recovered and acquired a permanent immunity. They can no longer be infected nor pass the virus.
- These states are relative to one specific piece of information
  - If there are several concurrent updates, one node can be infected with one update, susceptible to another update, etc... (real case)
Contagion Models

• We’ll see two models and relative protocols:

  • **Susceptible Infective** *(SI)* = Anti Entropy protocol
  
  • **Susceptible Infective Removed** *(SIR)* = Rumour Mongering protocol
Anti-Entropy
The SI contagion

- **Susceptible-Infective (SI):** An entity is initially *Susceptible*, then becomes and remains *Infective*. The state Infective is an absorbing state.

![Susceptible to Infective State Transition Diagram]

- The SI contagion models the **Anti-Entropy** gossip protocol.
- Nodes can just have the information (Infective) or not (Susceptible).
Anti-Entropy

- Basic version: a single node has a piece of data that has to be spread to all other nodes
  - Realistic version: all nodes have different, independent piece of data (just like in the Xerox case)
  - Pieces of information are independent
- The protocol tries to increase the “order” and decreases the “variance”
  - In Anti-Entropy each node (let’s say q) periodically contacts a random node (p) selected from the population
    - (Selecting a random node is a problem by itself — we’ll see more about it later)
- Three strategy to spread the contagion: Push, Pull, PushPull.
Anti-Entropy: Push

- At the beginning, a few nodes are infected and the probability to select a Susceptible node is high.

- Afterwards, the probability to select a Susceptible node becomes lower.

active behaviour()

if (Infected)
    p = select_random_node
    send_message(p)

passive behaviour()

receive_from(q)
set_state(Infected)
Active behaviour

```
if (Infected)
    p = select_random_node
    send_message(p)
```

Passive behaviour

```
receive_from(q)
set_state(Infected)
```

- At the beginning, a few nodes are infected and the probability to select a Susceptible node is high.
- Afterwards, the probability to select a Susceptible node becomes lower.

Nice, but how to be sure that gossip works?
Anti-Entropy: Push

active behaviour()
if (Infective)
    p = select_random_node
    send_message(p)

passive behaviour()
receive_from(q)
set_state(Infective)

- At the beginning, a few nodes are infected and the probability to select a Susceptible node is high
- Afterwards, the probability to select a Susceptible node becomes lower

Nice, but how to be sure that gossip works? with math!!
Anti-entropy: Push probability to be susceptible

- Probability to be susceptible at the iteration $t+1$

$$E(s_{t+1}) = s_t \left(1 - \frac{1}{N}\right)^{N(1-s_t)} \approx s_t e^{-(1-s_t)}$$

$S_t =$ susceptible nodes at the end of the $t$-th cycle

$$1 - \frac{1}{N} = \text{probability of an infected node to not infect a susceptible node}$$

$N(1-s_t) =$ infected nodes at the end of the $t$-th cycle
Anti-entropy: Push number of cycles

- First cycle where all nodes have the information

\[ S_N = \log_2 N + \log N + O(1) \]

- Proof is rather technical, intuition on three phases:
  1. first phase, infected nodes roughly double
  2. second phase, constant number of cycle
  3. third phase, takes \( \log N \)
Anti-entropy: Push number of messages

- Number of messages to infect all nodes

\[ O(N \log N) \]

optimal would be \( O(N) \) which is achieved in the first phase

then there is a \( O(\log N) \) phase when most of the messages are sent to already infected nodes
At the beginning the probability to be infected is low

No guarantee that the diffusion of the information starts at the first cycle

The whole population may be infected at the first cycle, if all node are lucky enough to contact the infected node
Anti-entropy: Pull probability to be susceptible

\[ E(s_{t+1}) = s_t \cdot s_t = s_t^2 \]

\( s_t = \) susceptible nodes at the end of the t-th cycle

When \( s_t \) is large, push is better
When \( s_t \) is small, pull is better

Can we have best from the two approaches?
Anti-Entropy: PushPull

active behaviour()
  p = select_random_node
  data = send_message(p)
  if (p is Infective)
    set_state(Infective)

passive behaviour()
  receive_from(q)
  send_message(q)
  if (q is Infective)
    set_state(Infective)

- At the beginning, the push dominates, because the probability that an Infective node contacts a Susceptible node is high at the start, but decreases at each cycle.

- Later, the pull dominates, because the probability that a Susceptible node contacts an Infective node grows at each cycle.
Anti-Entropy: PushPull performances

\[ S_N = O(\log N) \quad \text{number of cycles} \]

Since \( O(\log N) \) cycles are required, and each node will send at least one request per cycle:

\[ O(N \log N) \quad \text{message complexity} \]

If we consider only messages with state (larger than just request messages) we can achieve:

\[ O(N \log \log N) \quad \text{update complexity} \]
Termination

- Rumour Mongering considers the problem of termination, i.e. when to stop sending messages?
  - Anti-Entropy push cannot terminate
  - Anti-Entropy pull could, but needs to know the list of all updates (only possible in trivial cases)
  - The idea is to introduce another state (recovered) that defines nodes that do not spread the information
    - When all nodes are recovered, the protocol stops
The SIR contagion

- **Susceptible-Infected-Removed (SIR, Kermack and McKendrick 1927):** Individuals recover from the disease and gain immunity from it.

- The SI contagion models the **rumour mongering gossip protocol**.

- Recovered nodes no longer transmit the information.
Rumour mongering

- Just like anti-entropy, but.. nodes may “lose interest” in the information diffusion

- When?
  - **Counter**: lose interest after $k$ contacts
  - **Coin**: lose interest with probability $1/k$.

- Why?
  - **Feedback**: lose interest only if the recipient knows the rumour. Requires the knowledge of the state of the infected node: already infected?
  - **Blind**: lose interest regardless of the recipient. No information on the contacted node.
Rumour mongering /2

• Let’s considering PushPull with (coin, blind)

• The probability that an Infective node contacts a Susceptible one is equal to the number of Susceptible nodes in the system

active behaviour()
\[
p = \text{select\_random\_node} \\
data = \text{send\_message}(p) \\
\text{if (p is Infective)} \\
\quad \text{set\_state(Infective)} \\
\text{if (prob > 1/k)} \\
\quad \text{set\_state(Recovered)}
\]

passive behaviour()
\[
\text{receive\_from}(q) \\
\text{send\_message}(q) \\
\text{if (q is Infective)} \\
\quad \text{set\_state(Infective)}
\]
Rumour mongering
The model

\[
\begin{align*}
S_{t+1} &= S_t - \beta s_t I_t \\
I_{t+1} &= I_t + \beta s_t I_t - \kappa I_t = I_t(1 + \beta s_t - \kappa) \\
R_{t+1} &= R_t + \kappa I_t
\end{align*}
\]

- \( \beta \) = probability to pass the contagion (usually 1)
- \( I_t \) = number of infected nodes at time \( t \)
- \( \kappa \) = probability that an infected node becomes removed
- \( R_t \) = number of recovered nodes at time \( t \)
Rumour mongering /3

- Initially, a single infected entity
- Susceptible becomes infective
- Infective become removed
Rumour mongering /3

- Initially, a single infected entity
- Susceptible becomes infective
- Infective become removed

In rumour mongering not all nodes are guaranteed to receive the information
Rumour Mongering + Anti Entropy

• Fast update spreading with Push Rumour Mongering
  • low traffic
  • some nodes may not receive information

• Not-so-fast Push-Pull Anti-Entropy
  • Executed periodically
  • “repair mechanism” for updates not secured by the rumour mongering
  • eventually nodes get the update with probability 1
Information Aggregation
from local to global knowledge
Information aggregation

A node wants to have the knowledge about a network-dependent information
Information aggregation

- **Aggregation** protocols allow to estimate global properties only by disseminate local information. Pieces of information are dependent!

  - Emphasis goes also on processing, in addition to dissemination

- Upon receiving a message, a node updates its own local state by executing a **function**

  - The **average** load of nodes in a distributed computation system
  - The **maximum** storage space available on any node
  - The **count** of nodes in the system
Aggregation Example: Push-Pull Average

- Problem: estimate the average of local values of the nodes in the network
- Every node begins with estimate (\textit{est}) equal to the local value
- When two nodes \textit{p} and \textit{q} communicate (PushPull), they update their \textbf{estimation} of the average
- Variance of estimation is reduced at each communication, and nodes eventuality converge to the same value

\textbf{active behaviour()}

\begin{verbatim}
p = select_random_node
value = send(p, est)
est = est + data / 2
\end{verbatim}

\textbf{passive behaviour()}

\begin{verbatim}
value = receive(q)
send(q, est)
est = est + data / 2
\end{verbatim}
Aggregation Example: Average

Average = 12
Variance = 126
Aggregation Example: Average

Average = 12
Variance = 126
Aggregation Example: Average

Average = 12
Variance = 53.1
Aggregation Example: Average

Average = 12
Variance = 53.1
Aggregation Example:

Average

Average = 12
Variance = 41
Aggregation Example: Average

Average = 12
Variance = 41
Average Convergence

- First observation is about **mass conservation**: sum of estimations remains constant for any $t$

- difference between minimal and maximal estimation can only decrease with time

- the difference converges to zero, with values equal to the average

\[
\sum_{i=1}^{N} x_i(t) = N \bar{y}
\]

$x_i$ avg estimation at node $i$

$\bar{y}$ value average

So average will converge, but how **fast**?
Average Convergence

Fig. 1.1 Illustration of the averaging protocol. Pixels correspond to nodes (100x100 pixels = 10,000 nodes) and pixel color to the local approximation of the average.

\[ \sigma^2_N = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{y})^2, \]  

which describes how accurate the set of current estimates is. It is shown (see [11, 9]) that

\[ \mathbb{E}(\sigma^2_N(t+1)) \leq \frac{1}{2} \sigma^2_N(t), \]

where the time index \( t \) indicates a cycle. That is, variance decreases by a constant factor in each cycle. In practice, 10-20 cycles of the protocol already provide an extremely accurate estimation: the protocol not only converges, but it converges very quickly as well.

1.3.1.1 Asynchrony

In the case of information dissemination, allowing for unpredictable and unbounded message delays (a key component of the asynchronous model) has no effect on the correctness of the protocol, it only has an (in practice, marginal) effect on spreading speed. For Algorithm 3 however, correctness is no longer guaranteed in the presence of message delays.

To see why, imagine that node \( j \) receives a \texttt{PUSHUPDATE} message from node \( i \) and as a result it modifies its own estimate and sends its own previous estimate back to \( i \). But at that point, the message conservation property of the network will

Average Convergence

For a more rigorous proof we study the variance, which tells how accurate the whole network is about the estimation

\[ \sigma^2_N = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{y})^2 \]

It is shown* the following, which means that such variance decreases by a constant factor each cycle

\[ E(\sigma^2_N(t + 1)) \leq \frac{1}{2} \sigma^2_N(t) \]

Practically, few cycles are enough to converge!!

* For the nerds:
Network size

• By using aggregation is possible to count the number of nodes in the network
Network size

- By using aggregation is possible to count the number of nodes in the network

Any idea?
Network size

• By using aggregation is possible to count the number of nodes in the network
  
  • All nodes starts their estimation at 0, except the **initiator** that starts from 1
  
  • Nodes gossip just like in the average
  
  • Eventually all nodes converge to $y = 1/N$
  
  • So network size is simply $N = 1/y$
Going real

- Problems:
  - Asynchrony (message delay, not sync protocols)
  - Message lost
  - Node failures
  - Dissemination is affected only in terms of speed, correctness is ok
  - Aggregation is also affected in terms of correctness
    - in short: mass conservation is not valid anymore
    - for message lost and node failures the more practical way to resolve the problem is to restart the protocol
Lesson Take Away

- Gossip is a very efficient way to disseminate information, both for human and computer networks.

- Gossip protocols are based on periodic information exchange, and supports the implementation of application as information dissemination, aggregation, (peer sampling, topology management..)

- Gossip protocols trade off deterministic guarantees of success (but with probabilistic ones) for a significant increase in scalability and simplicity.
Lesson Take Away \2

• Push-pull information dissemination can reach a time complexity of $O(\log N)$ and a message complexity of $O(N \log \log N)$

• Push-pull average computation requires very few cycles to converge

• In case of failures or dynamism of network it is probably better to restart the protocol instead to fix it in a smart way