# *Cloudy Weather for P2P, with a chance of gossip*

Now available in 3D!

Alberto Montresor, Luca Abeni

Università di Trento, Italy


SBRC 2010
28th Brazilian Symposium on Computer Networks and Distributed Systems

# The Clash of the (buzzword) Titans: P2P versus Cloud computing

- **Cloud computing represents an important enabling technology to compete in the ICT market**
  - Provides the illusion of *infinite availability of computing and storage resources*
  - Eliminates *upfront commitment* of resources
  - Enable *small- and medium-size enterprises* to play the same game as web behemoth like Google, Amazon, Microsoft, etc.
- **Wait... wasn't this what P2P is exactly about?**
- **Well... no.**

# The Clash of the (buzzword) Titans: P2P versus Cloud computing

- **P2P pros and cons**
  - P2P resources are for free, but..
  - churn may compromise availability
  - P2P is good for very large groups (*scale up*),
    but does not perform so well for small groups (*scale down*)
- **The verdict:**
  - you cannot beat P2P from an economic point of view, but
  - the superior availability of the cloud (5-9s) makes it a more believable environment if we want to create a novel web business and cannot afford to lose clients due to the best-effort philosophy of P2P.

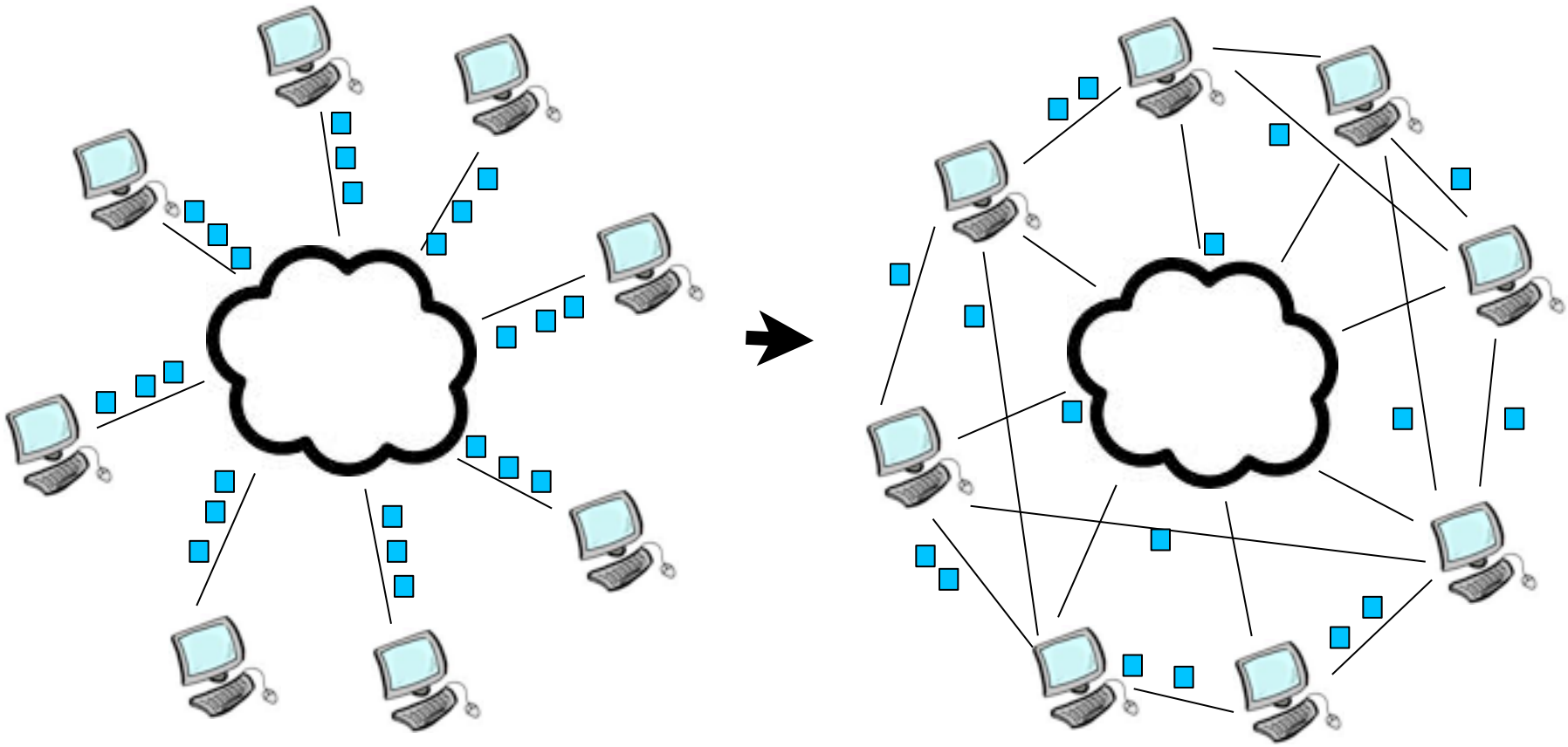
# Nobody likes cloudy weather

- **If your research is in the P2P field**
  - You may feel "threatened" by the cloud
  - One possible reaction is to create "bad press" against your enemy



# If you can't beat them, join them

- **The idea**
  - P2P as a "support group" for the cloud
  - Reduce the number of (costly) cloud interactions as much as possible



## If you can't beat them, join them

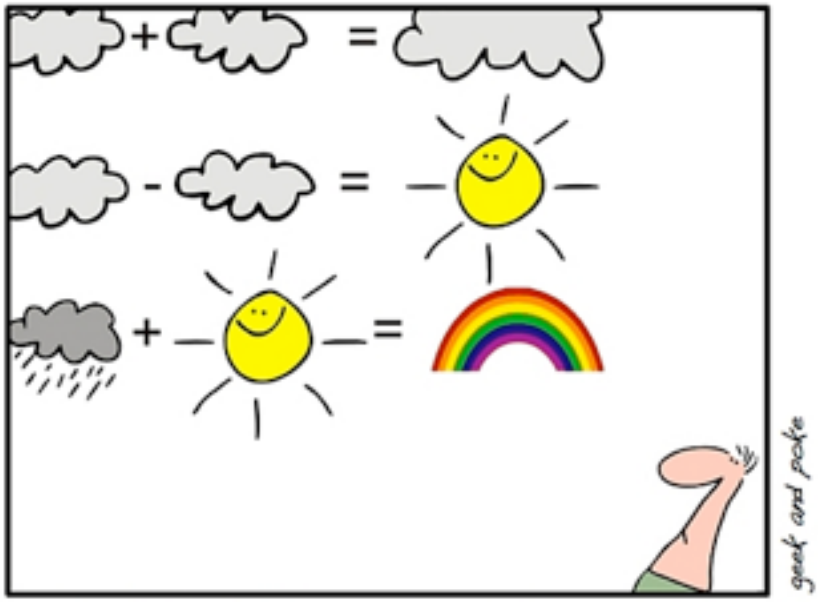
- **Do not confuse with the practice of adding stable super-peers to P2P networks:**
  - Example: Skype
- **That approach**
  - either requires investment in hardware + continuous connectivity
  - or require to rent cloud computing nodes for extended periods of time
  - Both cases - costly!
- **We are pursuing the other way around:**
  - We add peers to the cloud, to reduce storage, bandwidth and computational costs
  - We will not use stable super-peers

## The position

- **I call for research in the field of "P2P-supported cloud computing"**
- **The question: given an higly-available service based on the cloud,**
  - how much load you can relieve from the cloud without compromising the availability of the service?
  - how much cost reduction (in term of money) you can gain from this approach?
- **During the keynote**
  - I will show you a very simple idea
  - but a lot of questions remain open

# Keynote workplan

- **Setting the stage ✓**
- **A specific problem to be solved**
  - News feed
- **(Brief) introduction to cloud computing**
  - Elastic Computing Cloud (EC2)
  - Amazon Simple Storage Service (S3)
- **The algorithm**
  - Cloud-enabled peer sampling
  - Information dissemination
- **Evaluation**
  - With real monetary costs!




SIMPLY EXPLAINED – PART 17:
CLOUD COMPUTING

# The problem

- **News feed**
  - A service capable to provide news to a large user base
- **dilbert.com**
  - 60.000 unique visitors / day
  - 200.000 pages / day
  - Each page 5 strips
  - Each strip is around 50KB
  - Over a year: ~ 18 Terabyte


- **Ashton Kutcher's twitter account**
  - ~5.000.000 followers
  - 8 tweets / day, 140 bytes each
  - Over a year: ~ 2 Terabyte

# Amazon Web Services

### Compute

- Amazon Elastic Compute Cloud (EC2)
- Amazon Elastic MapReduce
- Auto Scaling

### Content Delivery

- Amazon CloudFront

### Database

- Amazon SimpleDB
- Amazon Relational Database Service (RDS)

### E-Commerce

- Amazon Fulfillment Web Service (FWS)

### Messaging

- Amazon Simple Queue Service (SQS)
- Amazon Simple Notification Service (SNS)

### Monitoring

- Amazon CloudWatch

### Networking

- Amazon Virtual Private Cloud (VPC)
- Elastic Load Balancing

### Payments & Billing

- Amazon Flexible Payments Service (FPS)
- Amazon DevPay

### Storage

- Amazon Simple Storage Service (S3)
- Amazon Elastic Block Storage (EBS)
- AWS Import/Export

# Elastic Computing Cloud (EC2)

- **EC2 provides resizable compute capacity in the cloud**
  - You get virtual machines on the cloud, accessible through `ssh`
  - You can add / remove instances through web interfaces (scale up/down)
  - You pay by the hour
  - One small instance for one year : $365 \cdot 24 \cdot 0.085\$ = 744\$$

US – N. Virginia | US – N. California | EU – Ireland | APAC – Singapore

| Standard On-Demand Instances | Linux/UNIX Usage | Windows Usage |
|---|---|---|
| Small (Default) | $0.085 per hour | $0.12 per hour |
| Large | $0.34 per hour | $0.48 per hour |
| Extra Large | $0.68 per hour | $0.96 per hour |
| **High-Memory On-Demand Instances** | | |
| Extra Large | $0.50 per hour | $0.62 per hour |
| Double Extra Large | $1.20 per hour | $1.44 per hour |
| Quadruple Extra Large | $2.40 per hour | $2.88 per hour |
| **High-CPU On-Demand Instances** | | |
| Medium | $0.17 per hour | $0.29 per hour |
| Extra Large | $0.68 per hour | $1.16 per hour |

# Simple Storage Service (S3)

- **Features**
  - Write, read, and delete *objects* containing from 1 byte to 5 gigabytes of data each.
  - Objects are stored in *buckets* and retrieved via unique, developer-assigned keys
  - Objects can be made *private* or *public*, and *rights* can be granted to specific users.
  - Uses standards-based *REST* and *SOAP* interfaces
  - "*Owner-pays-buckets*" vs "*Requester-pays-buckets*"
- **How much our testcases would cost on a cloud?**
  - Dilbert : 2700$
  - Kutcher : 300$
  - Note: without overhead, just transfer-out

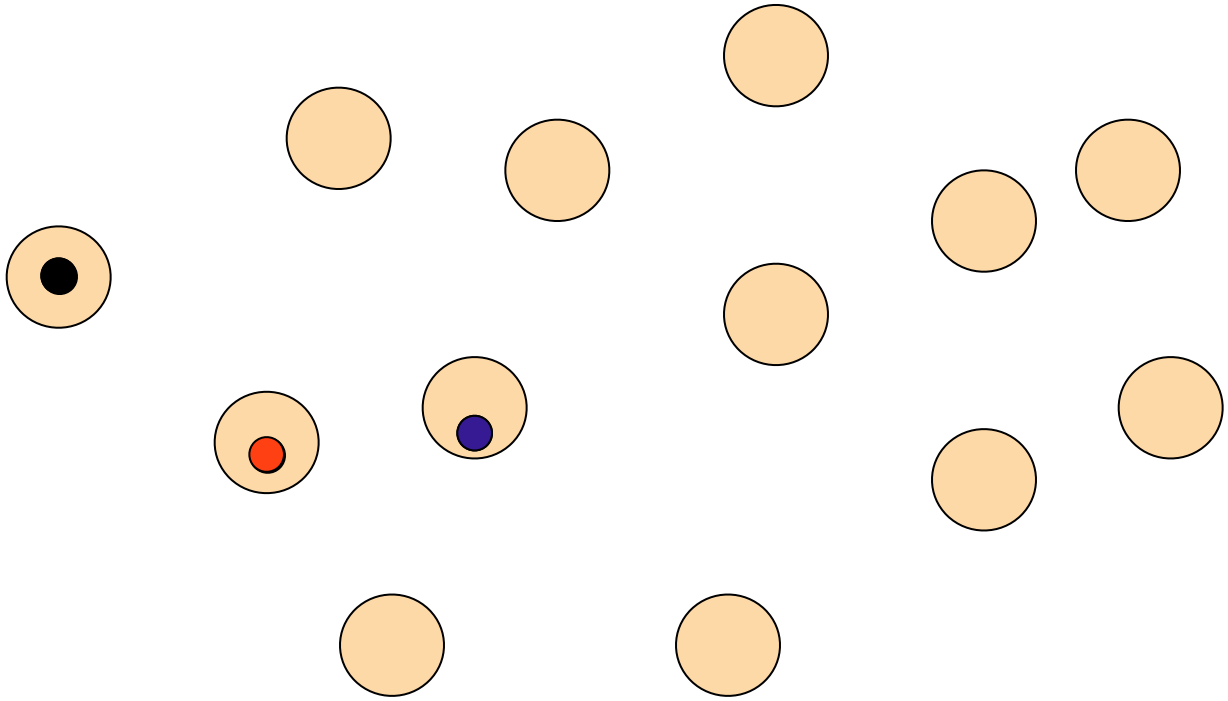
# Amazon S3 Pricing

US – Standard | US – N. California | **EU – Ireland** | APAC – Singapore

| Storage (Designed for 99.999999999% Durability) | | Reduced Redundancy Storage (Designed for 99.99% Durability) | | Data Transfer* | | Requests | |
|---|---|---|---|---|---|---|---|
| **Tier** | **Pricing** | **Tier** | **Pricing** | **Tier** | **Pricing** | **Type** | **Pricing** |
| First 50 TB / Month of Storage Used | $0.150 per GB | First 50 TB / Month of Storage Used | $0.100 per GB | All Data Transfer In | Free until June 30th, 2010** | PUT, COPY, POST, or LIST | $0.01 per 1,000 Requests |
| Next 50 TB / Month of Storage Used | $0.140 per GB | Next 50 TB / Month of Storage Used | $0.093 per GB | First 1 GB / month data transfer out | $0.000 per GB | GET and All Other Requests*** | $0.01 per 10,000 Requests |
| Next 400 TB / Month of Storage Used | $0.130 per GB | Next 400 TB / Month of Storage Used | $0.087 per GB | Up to 10 TB / month data transfer out | $0.150 per GB | | |
| Next 500 TB / Month of Storage Used | $0.105 per GB | Next 500 TB / Month of Storage Used | $0.070 per GB | Next 40 TB / month data transfer out | $0.110 per GB | | |
| Next 4000 TB / Month of Storage Used | $0.080 per GB | Next 4000 TB / Month of Storage Used | $0.053 per GB | Next 100 TB / month data transfer out | $0.090 per GB | | |
| Storage Used / Month Over 5000 TB | $0.055 per GB | Storage Used / Month Over 5000 TB | $0.037 per GB | Greater than 150 TB / month data transfer out | $0.080 per GB | | |

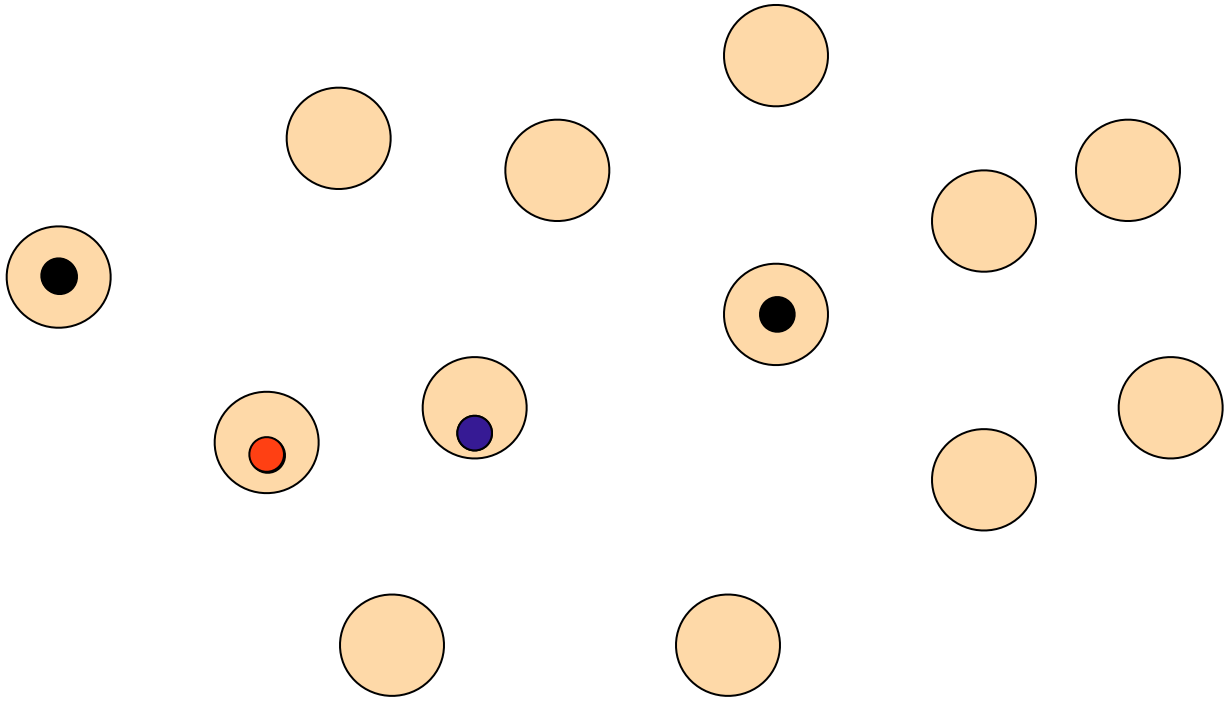
# Solution - Introduction

- **A classical solution for the problem - epidemic protocols**
- **Nodes "gossip" about news (rumors) in random fashion**
  - *Rumor-mongering* - to quickly push the rumors toward on-line nodes
  - *Anti-entropy* - to make sure that everybody gets all the news



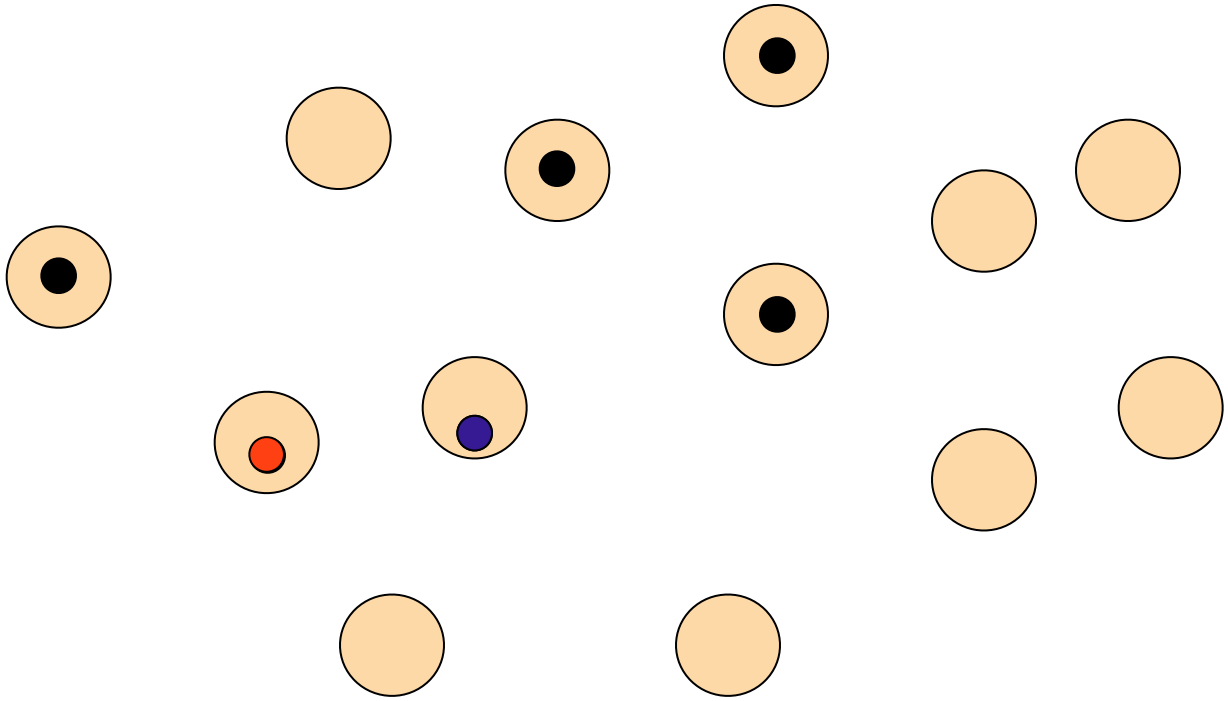
# Solution - Introduction

- **A classical solution for the problem - epidemic protocols**
- **Nodes "gossip" about news (rumors) in random fashion**
  - *Rumor-mongering* - to quickly push the rumors toward on-line nodes
  - *Anti-entropy* - to make sure that everybody gets all the news



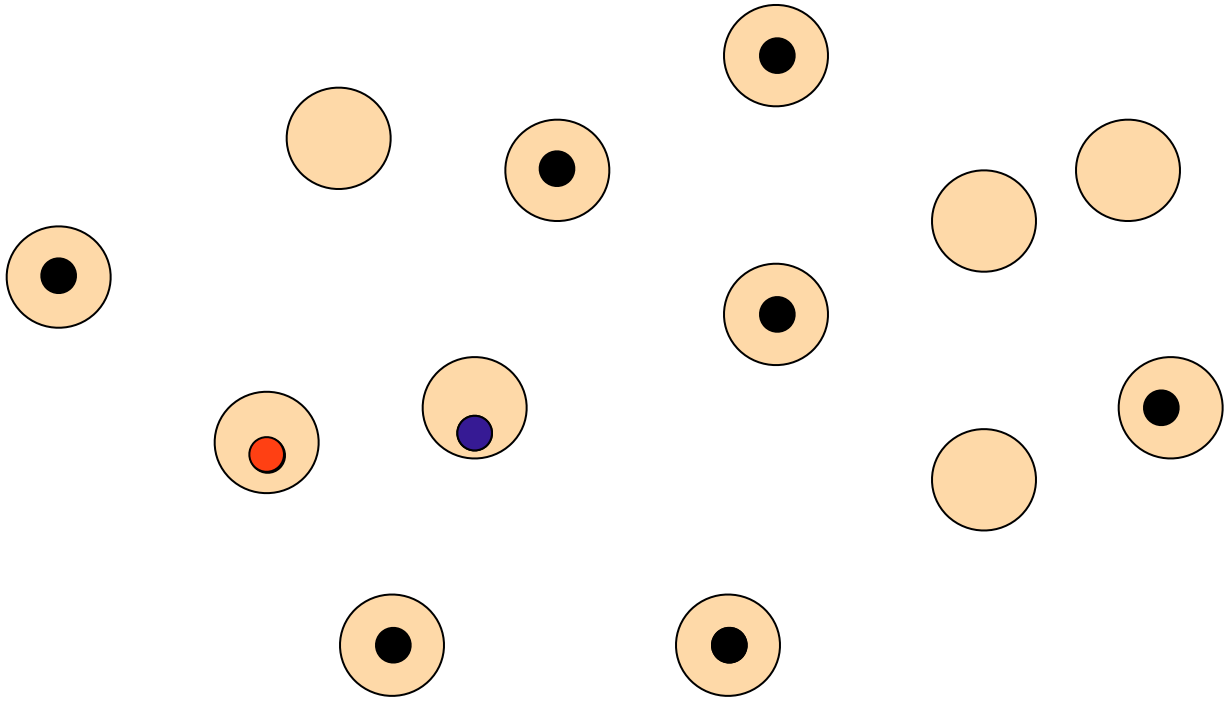
# Solution - Introduction

- **A classical solution for the problem - epidemic protocols**
- **Nodes "gossip" about news (rumors) in random fashion**
  - *Rumor-mongering* - to quickly push the rumors toward on-line nodes
  - *Anti-entropy* - to make sure that everybody gets all the news



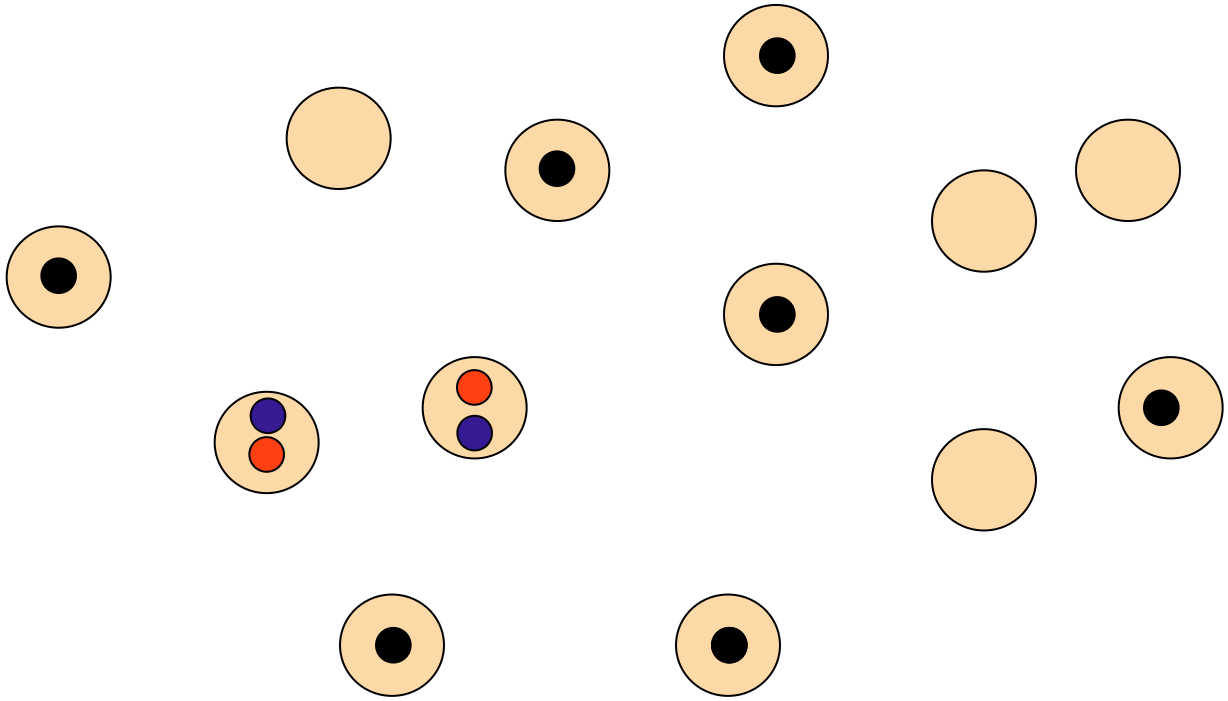
# Solution - Introduction

- **A classical solution for the problem - epidemic protocols**
- **Nodes "gossip" about news (rumors) in random fashion**
  - *Rumor-mongering* - to quickly push the rumors toward on-line nodes
  - *Anti-entropy* - to make sure that everybody gets all the news



# Solution - Introduction

- **A classical solution for the problem - epidemic protocols**
- **Nodes "gossip" about news (rumors) in random fashion**
  - *Rumor-mongering* - to quickly push the rumors toward on-line nodes
  - *Anti-entropy* - to make sure that everybody gets all the news

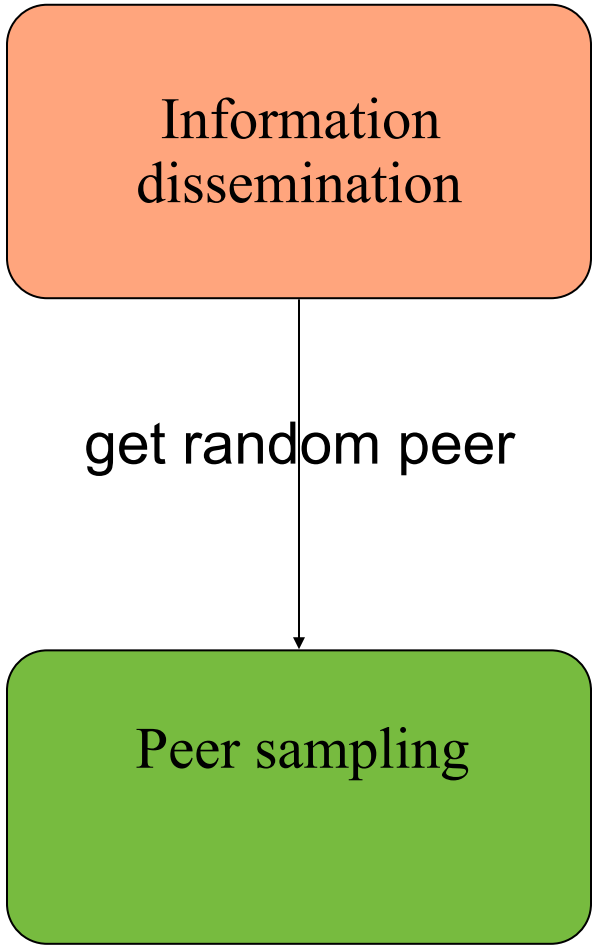


# Issues

- **Epidemic protocols require a list of nodes forming the network**
- **Where do we get it from?**
  - The list could be quite large
    - Store 5.000.000 followers in the cloud?
  - System is dynamic
    - The list must be frequently updated
  - The list could be (temporarily) small
    - Nodes may have no access to other nodes
    - This problem could be exhacerbated by NAT/firewalls

# The idea

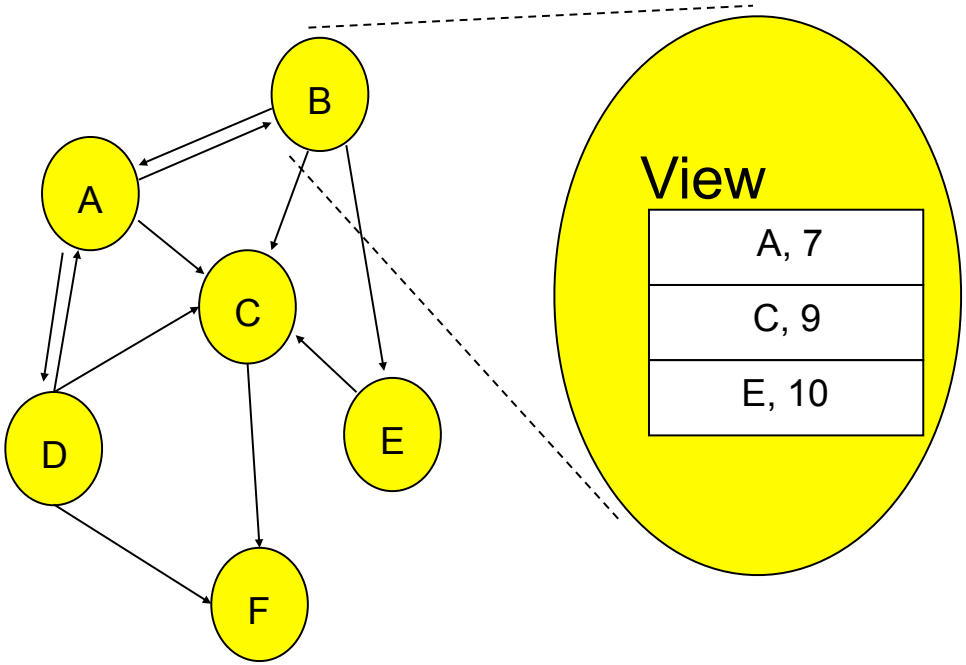
- **About the need of storing a list**
  - Instead of storing the complete list, store *partial views* at each node, containing random samples of the entire list
  - *Peer sampling* - the generation of random samples
  - We will discuss the CYCLON protocol
- **About the "down-scale" problem**
  - The cloud join the group - so there will be always one member




S. Voulgaris, D. Gavidia, and M. Van Steen. *Cyclon: Inexpensive membership management for unstructured p2p overlays*. Journal of Network and Systems Management, 13(2):197–217, 2005

# Overlay networks

- **State of each node:**
  - A *partial view* containing *c* neighbor *descriptors* (*c* = view size)
- **Descriptors of node *p* contains**
  - The *address* needed to communicate with *p*
  - A *timestamp* / age counter
- **Nodes + neighbors = overlay network**



## A generic gossip protocol - executed by process *p*

Init: initialize my local *state*

**Active thread**

**do once every δ time units**

$q$ = getPeer($state$)

$s_p$ = prepareMsg($state$, $q$)

**send** (REQ, $s_p$) **to** $q$

A "cycle" of length δ

**Passive thread**

**do forever**

**receive** $(t, s_q)$ **from** *

**if** ( $t$ = REQ ) **then**

$s_p$ = prepareMsg($state$, $q$)

**send** (REP, $s_p$) **to** $q$

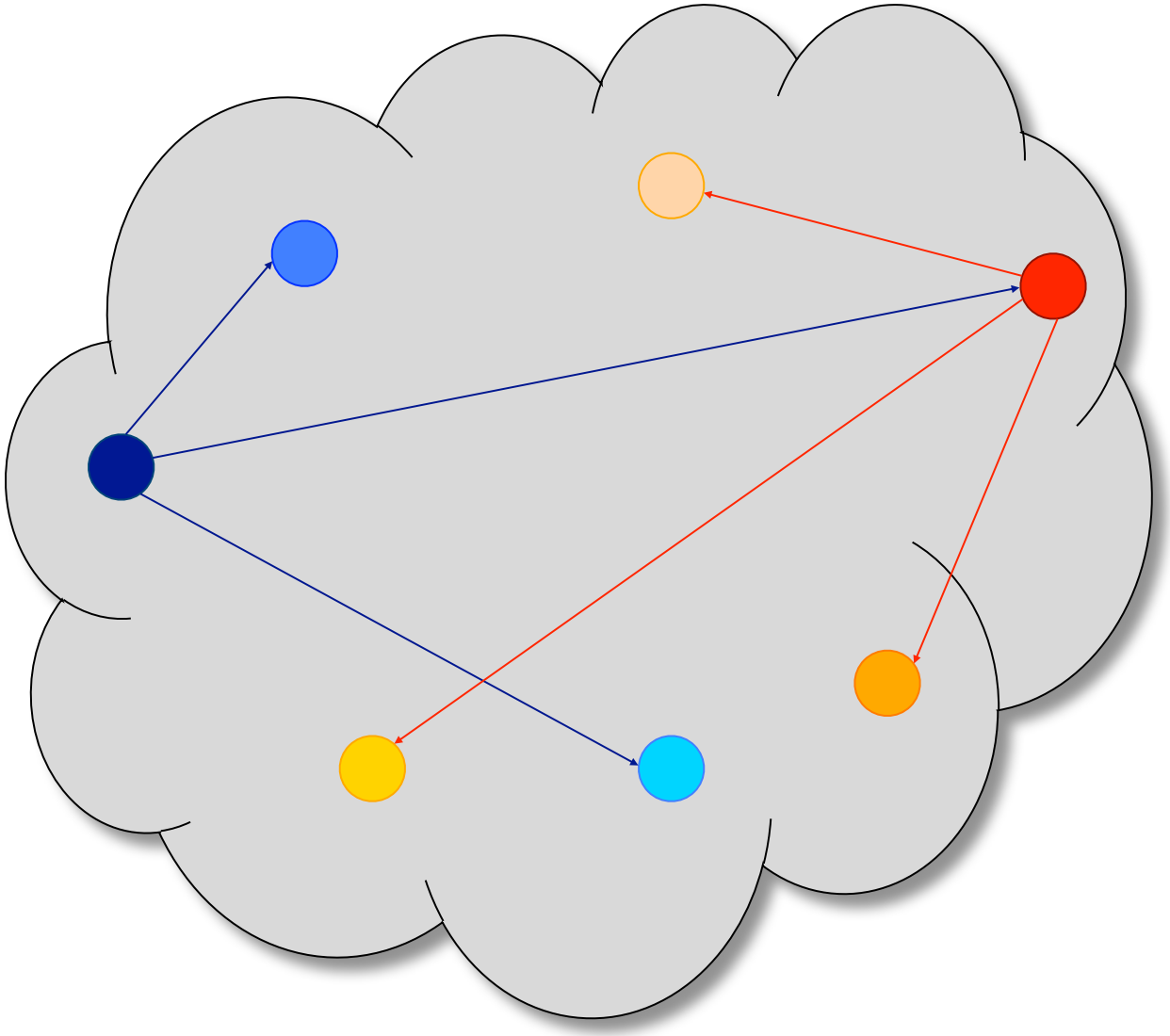
$state$ = update($state$, $s_q$)

# CYCLON

- **Descriptor: address + timestamp**
- ***getPeer*()**
  - select the oldest descriptor in the view
  - remove it from the view
- ***prepareMsg*(*view*, *q*)**
  - In active thread:
    - returns a subset of *t-1* random nodes, plus a fresh local identifier of itself
  - In passive thread:
    - returns a subset of *t* random nodes
- ***update*(*view*, $msg_q$)**
  - discard entries in $msg_q$: *p*, nodes already know
  - add $msg_q$ , removing entries sent to *q*

# CYCLON

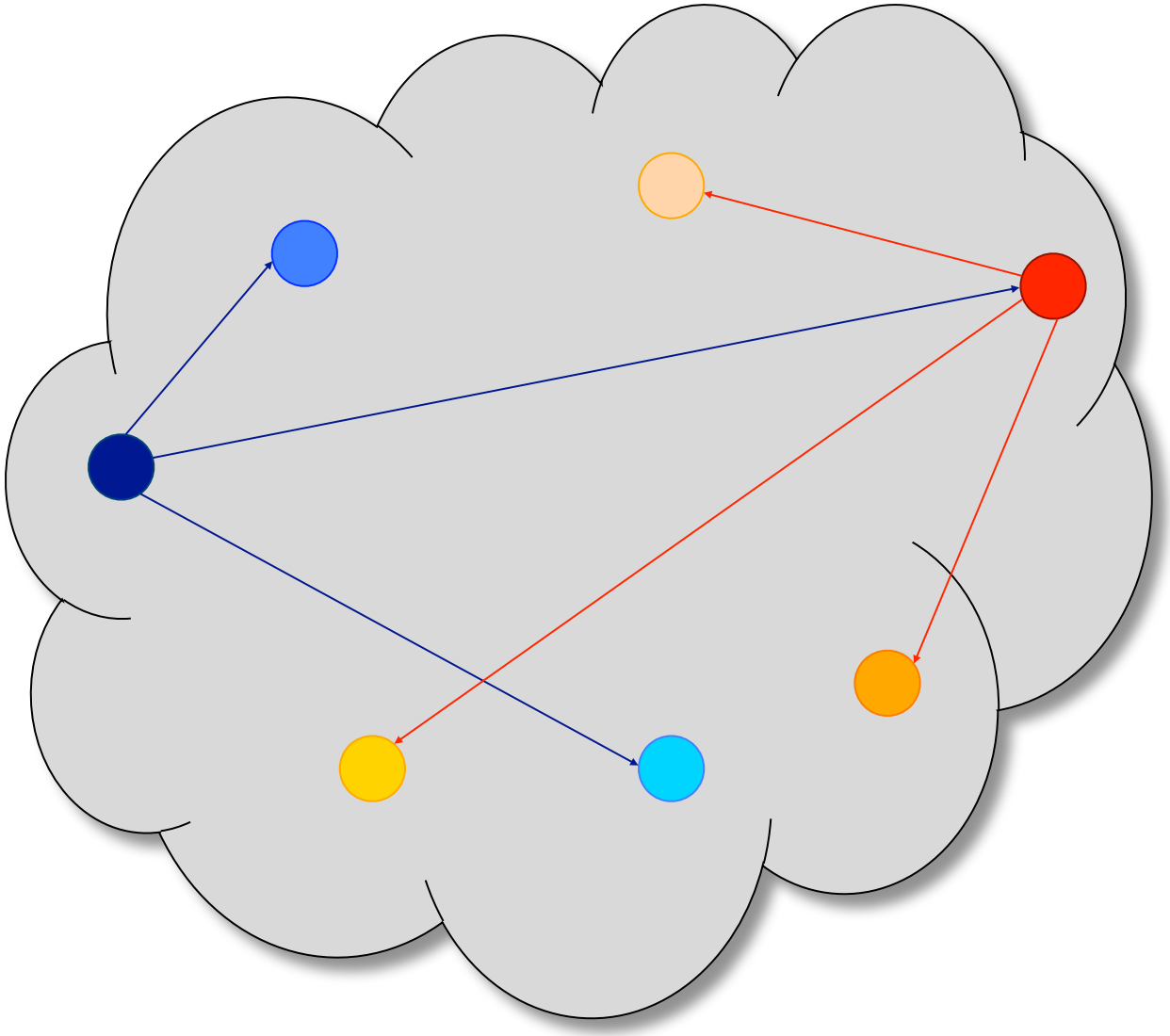
| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 4 |



| ID & Address | Time stamp |
|---|---|
| | 7 |
| | 10 |
| | 14 |

# CYCLON

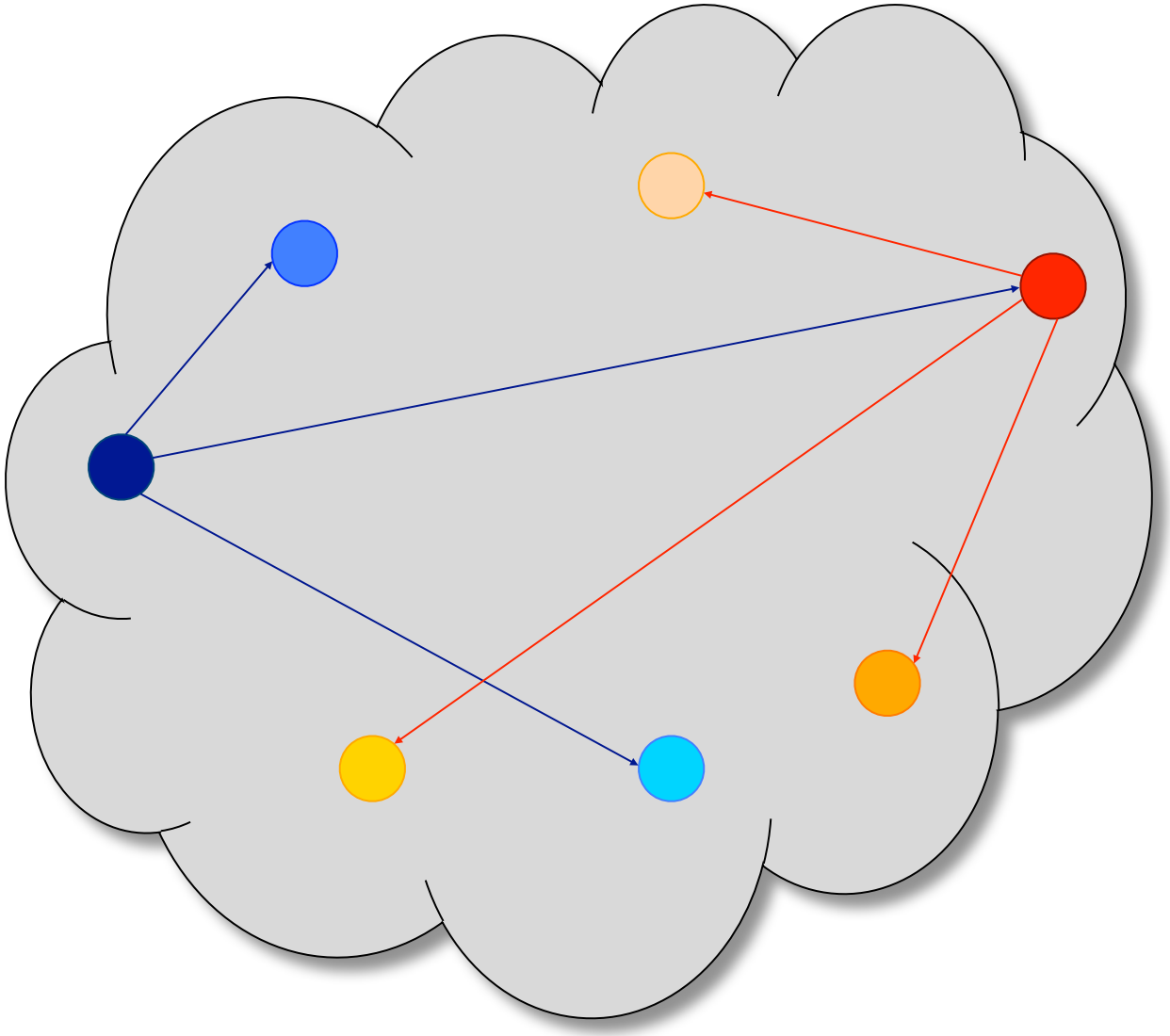
| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 4 |



| ID & Address | Time stamp |
|---|---|
| | 7 |
| | 10 |
| | 14 |

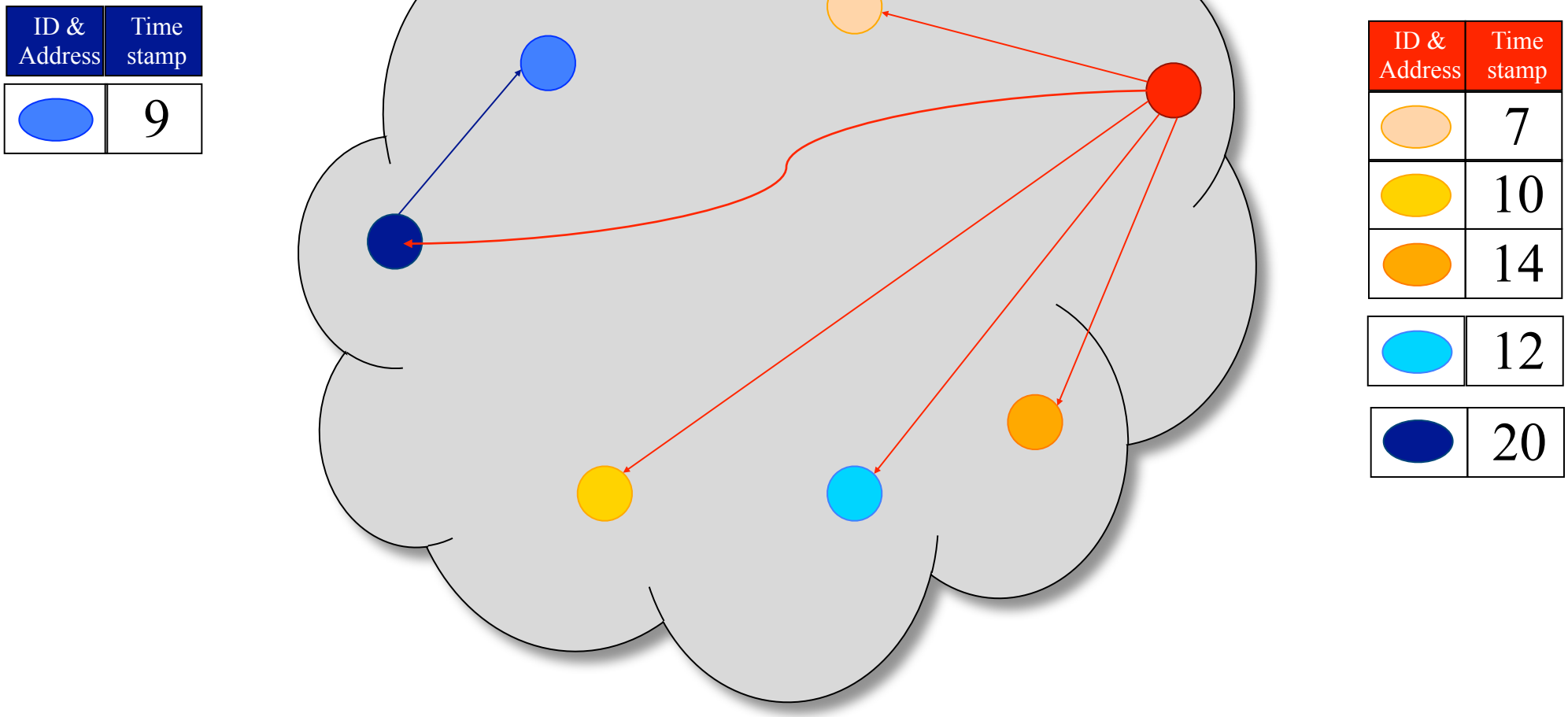
# CYCLON

| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 20 |



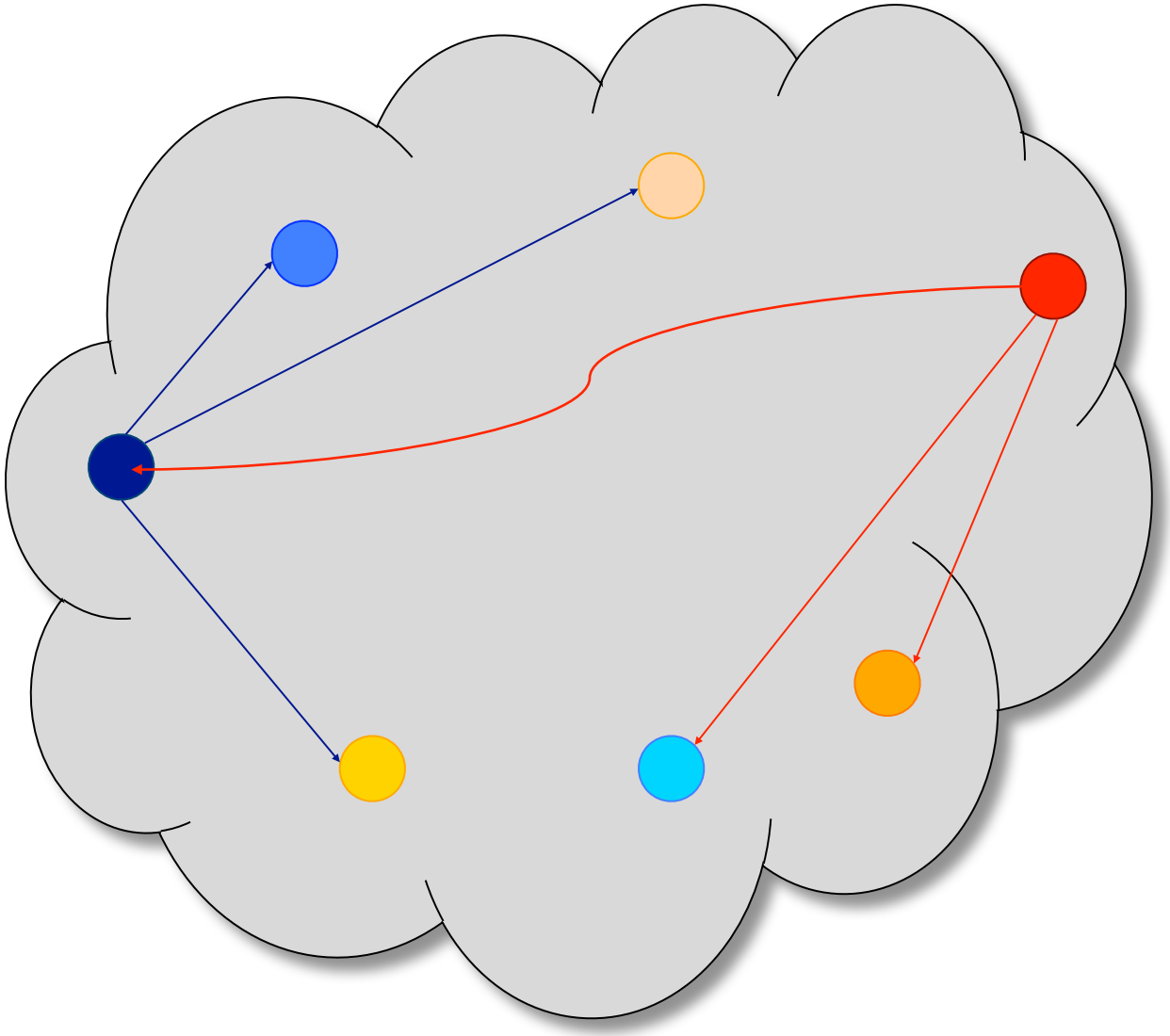
| ID & Address | Time stamp |
|---|---|
| | 7 |
| | 10 |
| | 14 |

# CYCLON



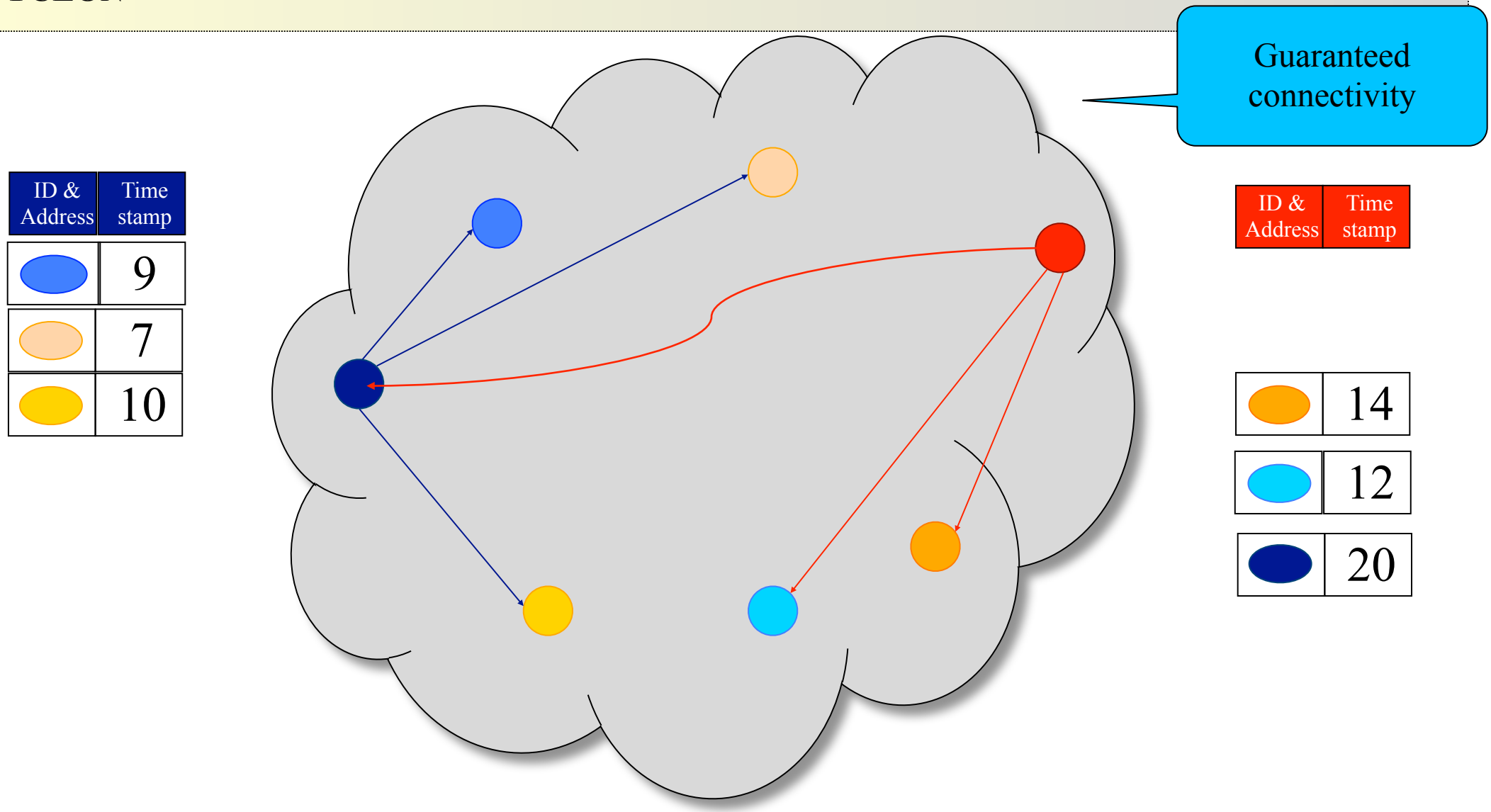
# CYCLON

| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 7 |
| | 10 |



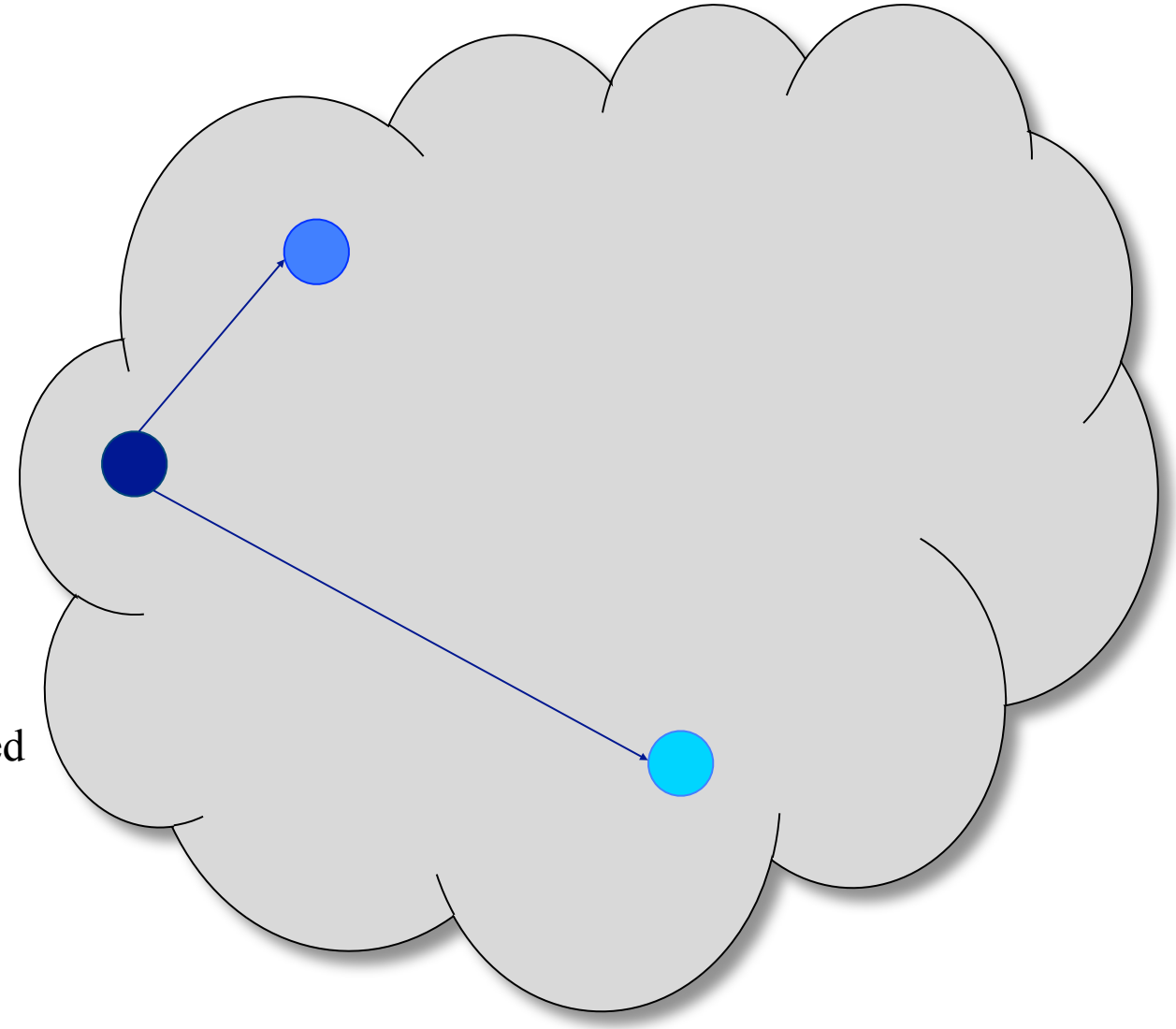
| ID & Address | Time stamp |
|---|---|
| | 14 |
| | 12 |
| | 20 |

# CYCLON



# CYCLON

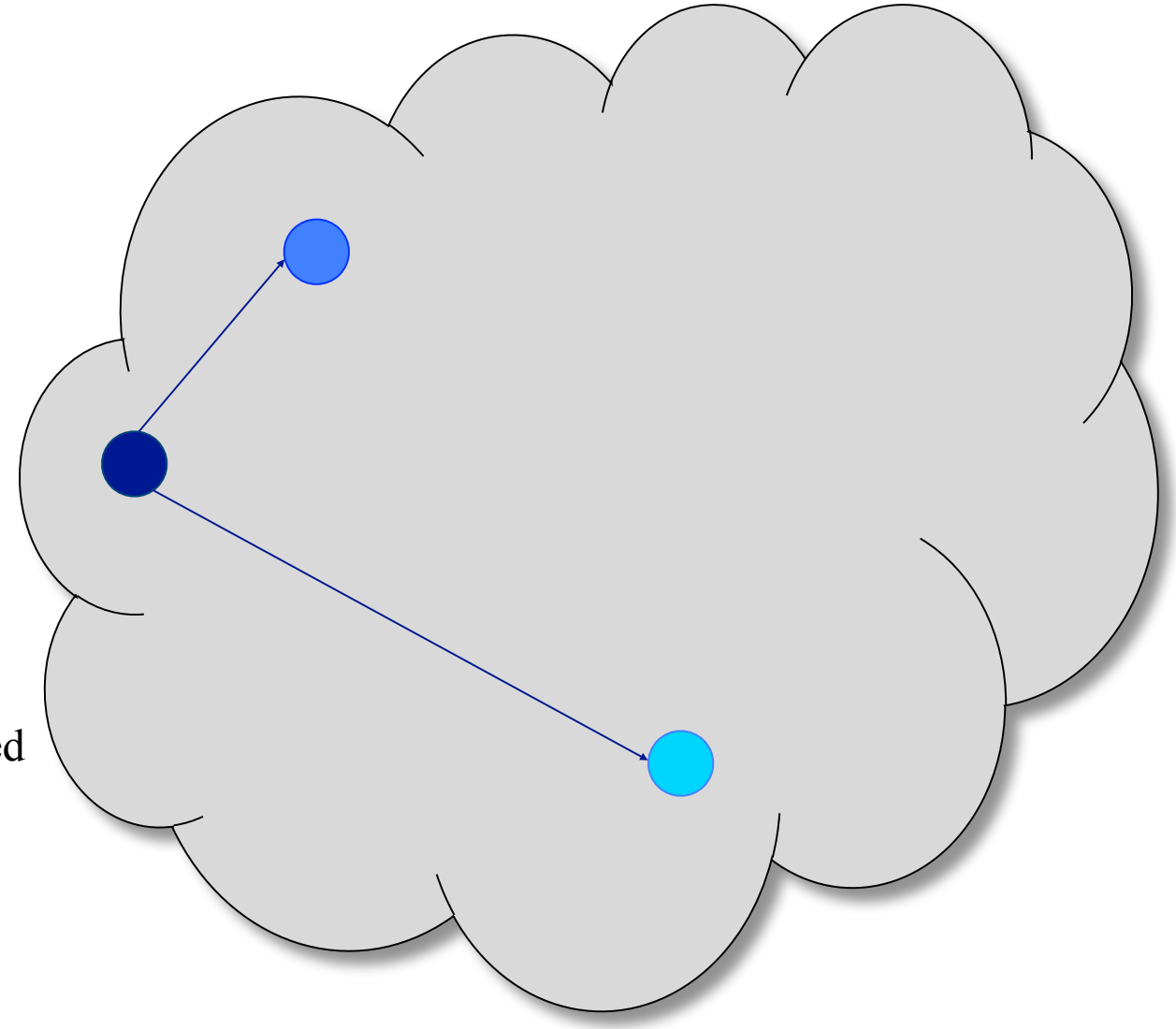
| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 4 |



Descriptors of crashed nodes will disappear

# CYCLON

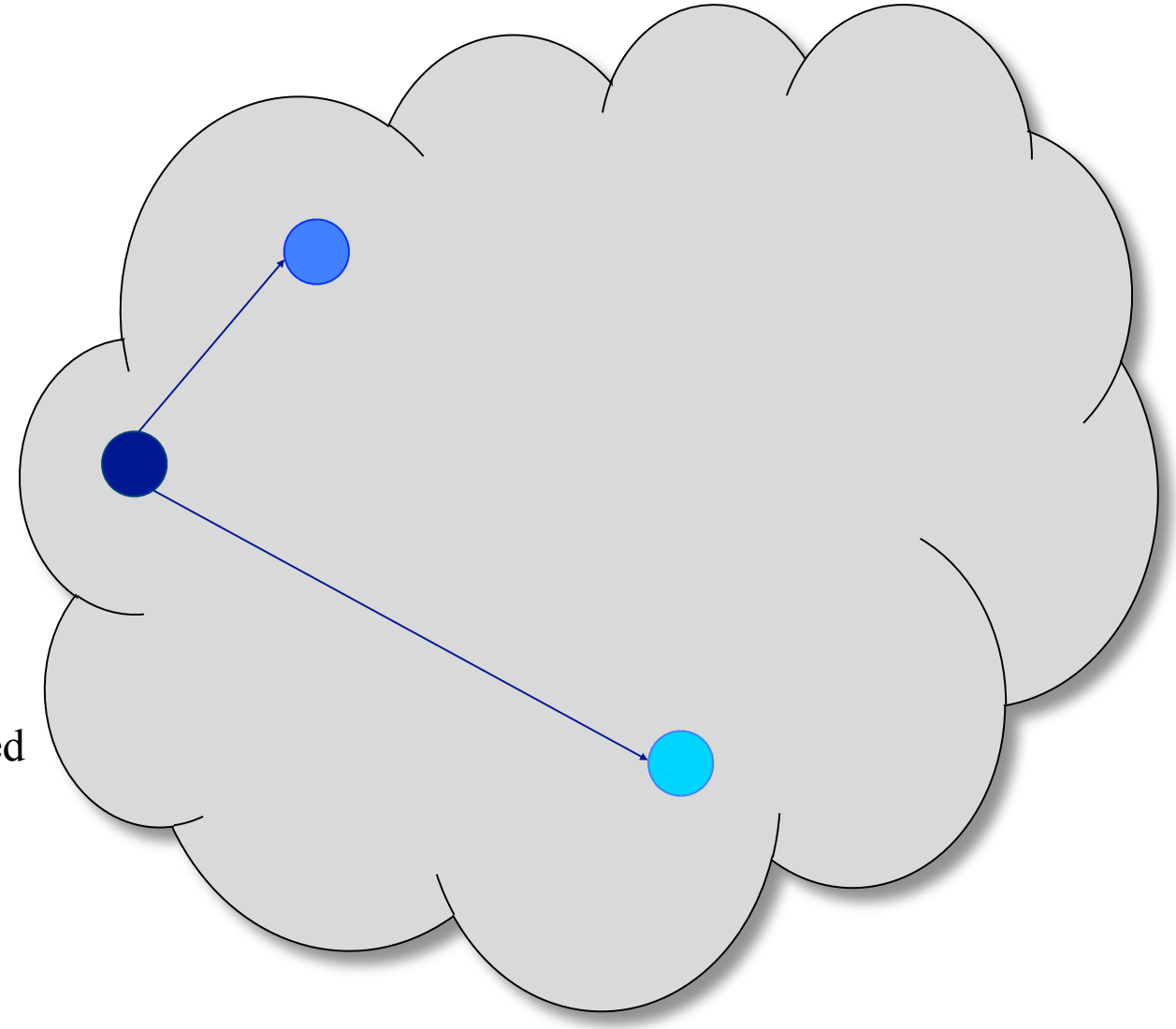
| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 4 |



Descriptors of crashed nodes will disappear

# CYCLON

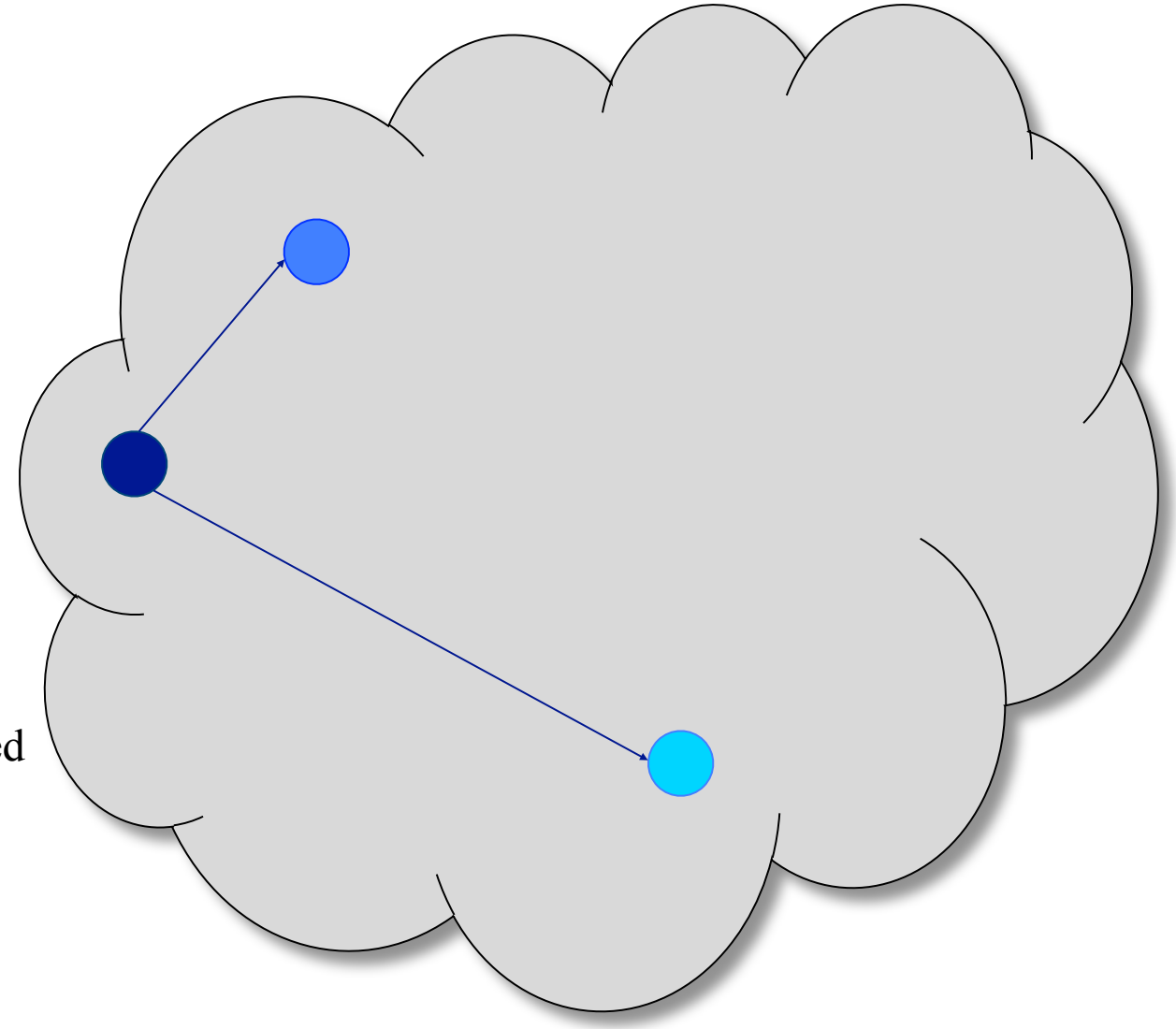
| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |
| | 20 |



Descriptors of crashed nodes will disappear

# CYCLON

| ID & Address | Time stamp |
|---|---|
| | 9 |

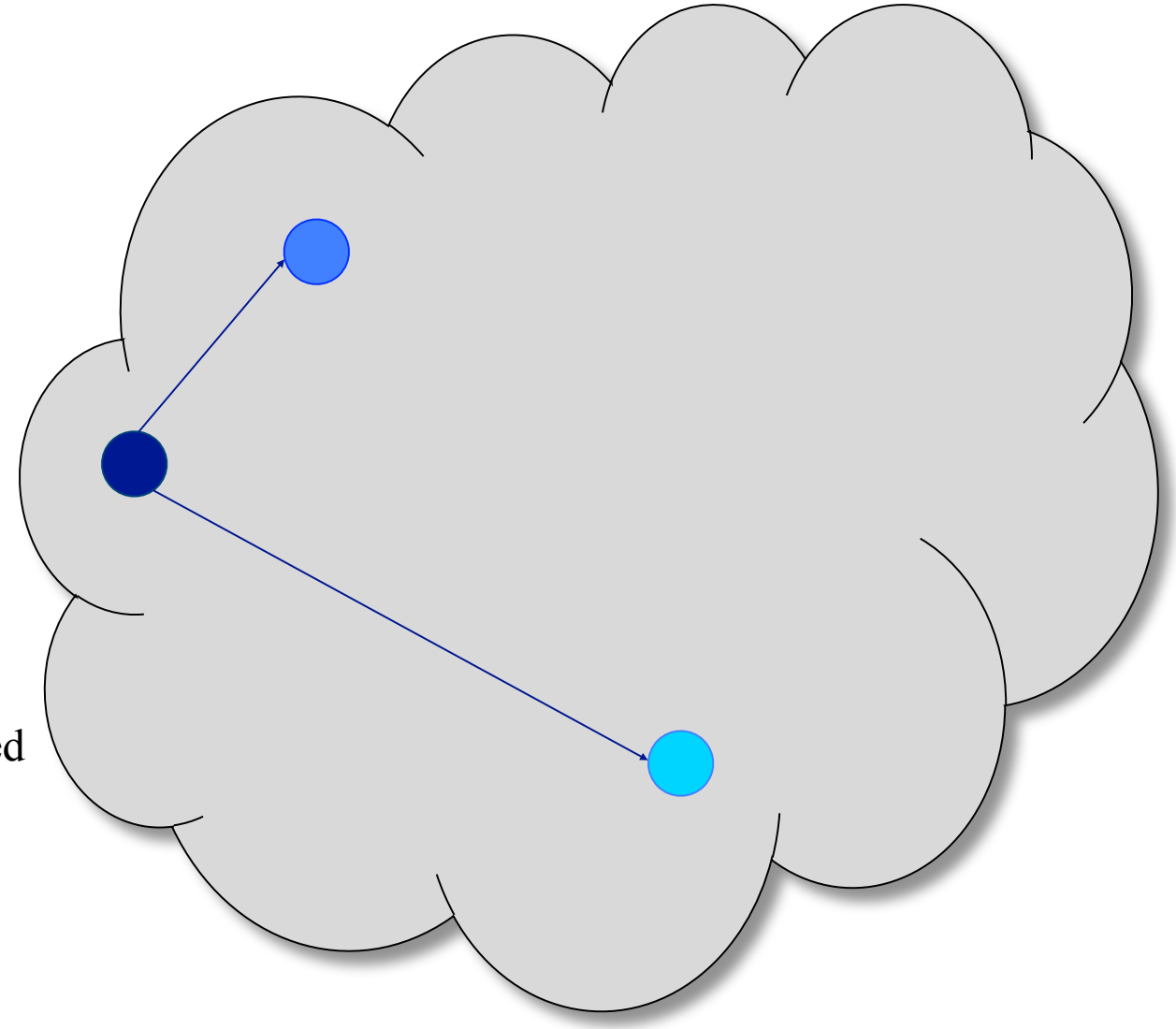


| | |
|---|---|
| | 12 |
| | 20 |

Descriptors of crashed nodes will disappear

# CYCLON

| ID & Address | Time stamp |
|---|---|
| | 9 |
| | 12 |



Descriptors of crashed nodes will disappear

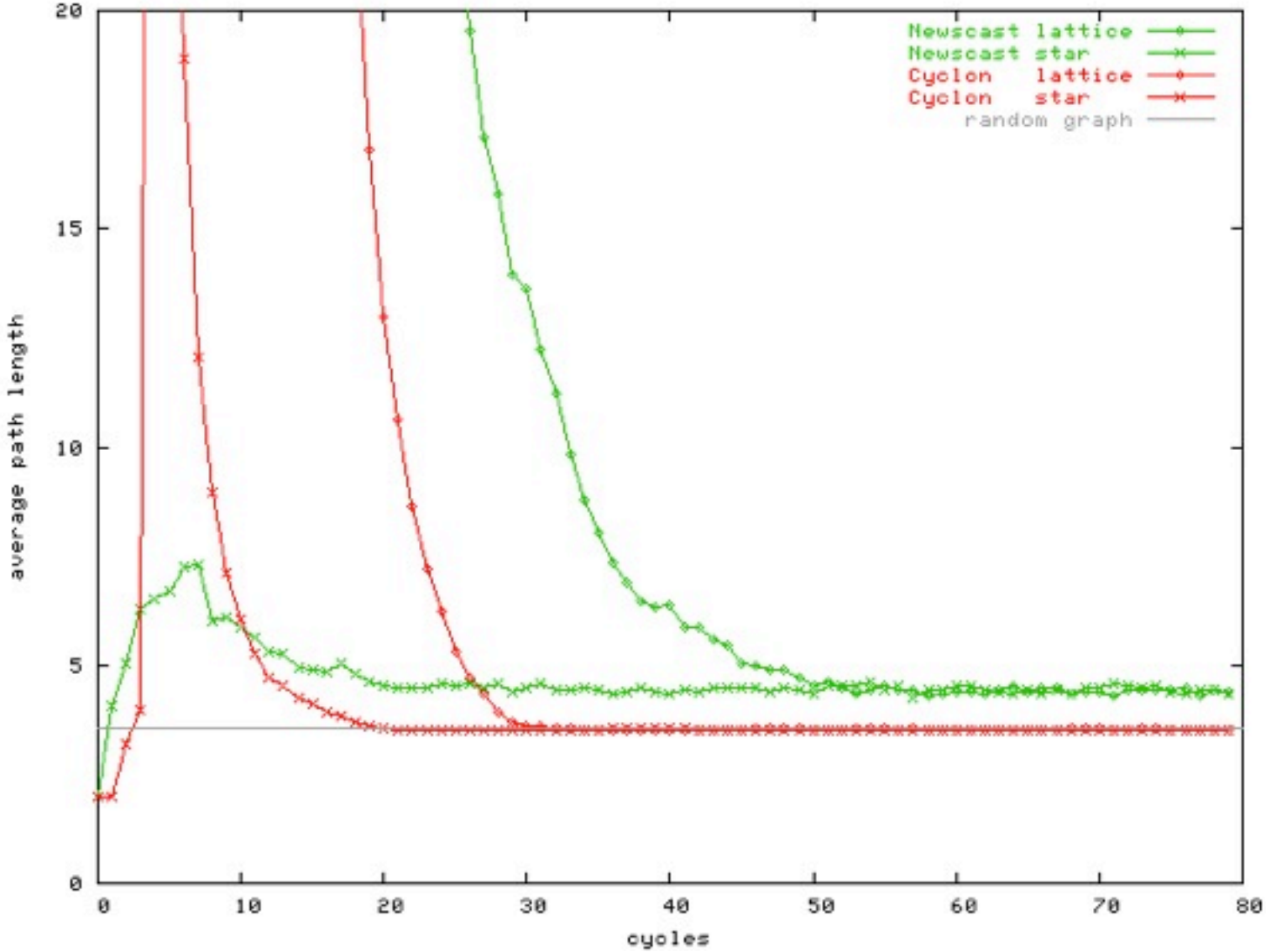
TABLE II
PARAMETERS USED IN THE EVALUATION.

| Parameter | Value | Meaning |
|---|---|---|
| $n$ | 100 000 | Total number of nodes |
| $\delta_{cyclon}$ | 10s | Cycle length of CYCLON |
| $c$ | 20 | Number of peer references in the view |
| $g$ | 5 | CYCLON message size |

- **Some advantages**
  - Connectivity is guaranteed
  - Uses small bandwidth
    - Only small part of the view is sent
    - Single message could be around 100 bytes (with overhead)
    - Each cycle: 4 messages expected
    - Bandwidth per node: 40 b/s
- **In the next slides**
  - Preliminary evaluation (without the cloud)

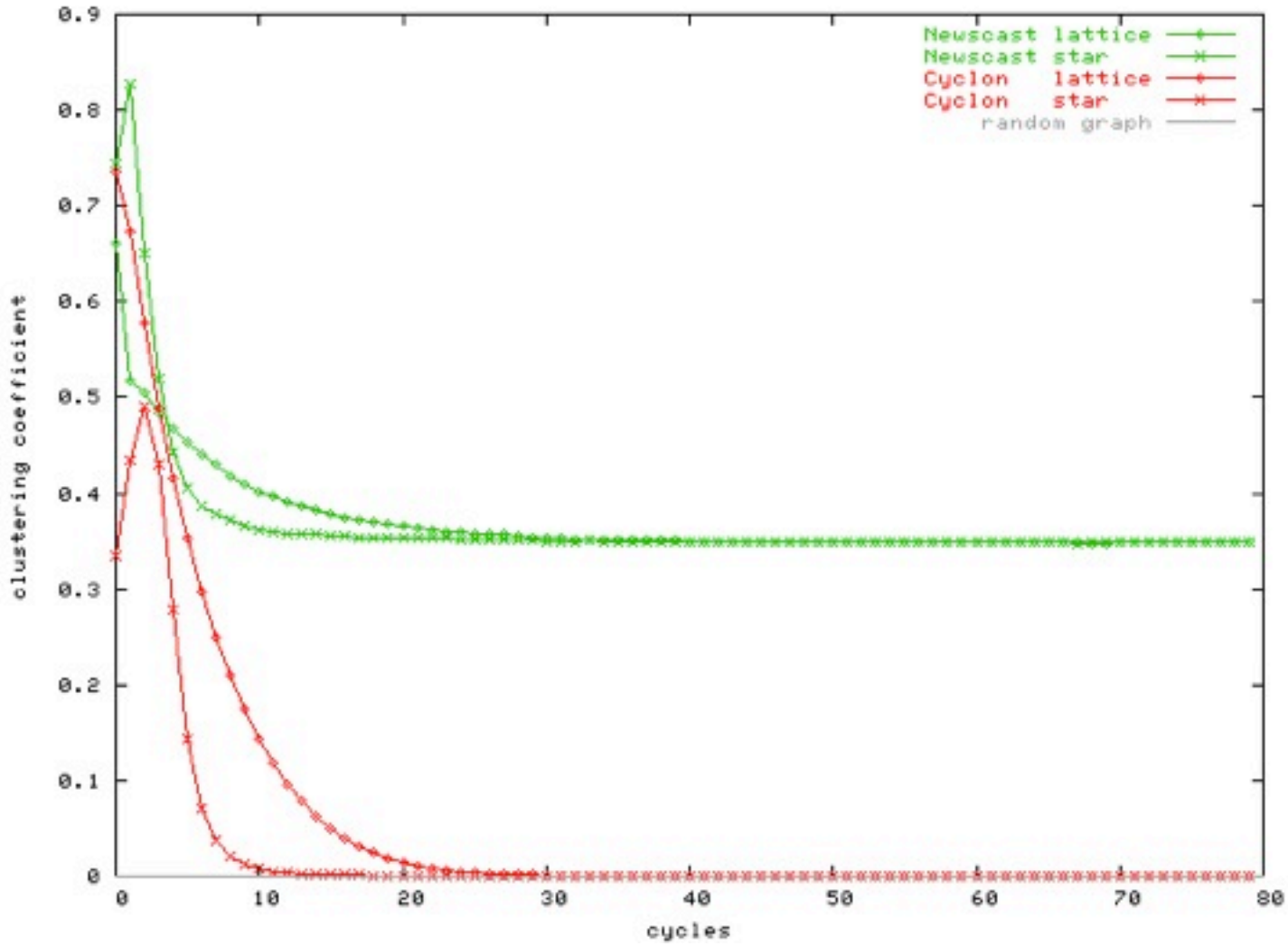
# Average path length

- **Indication of the time and cost to diffuse information**



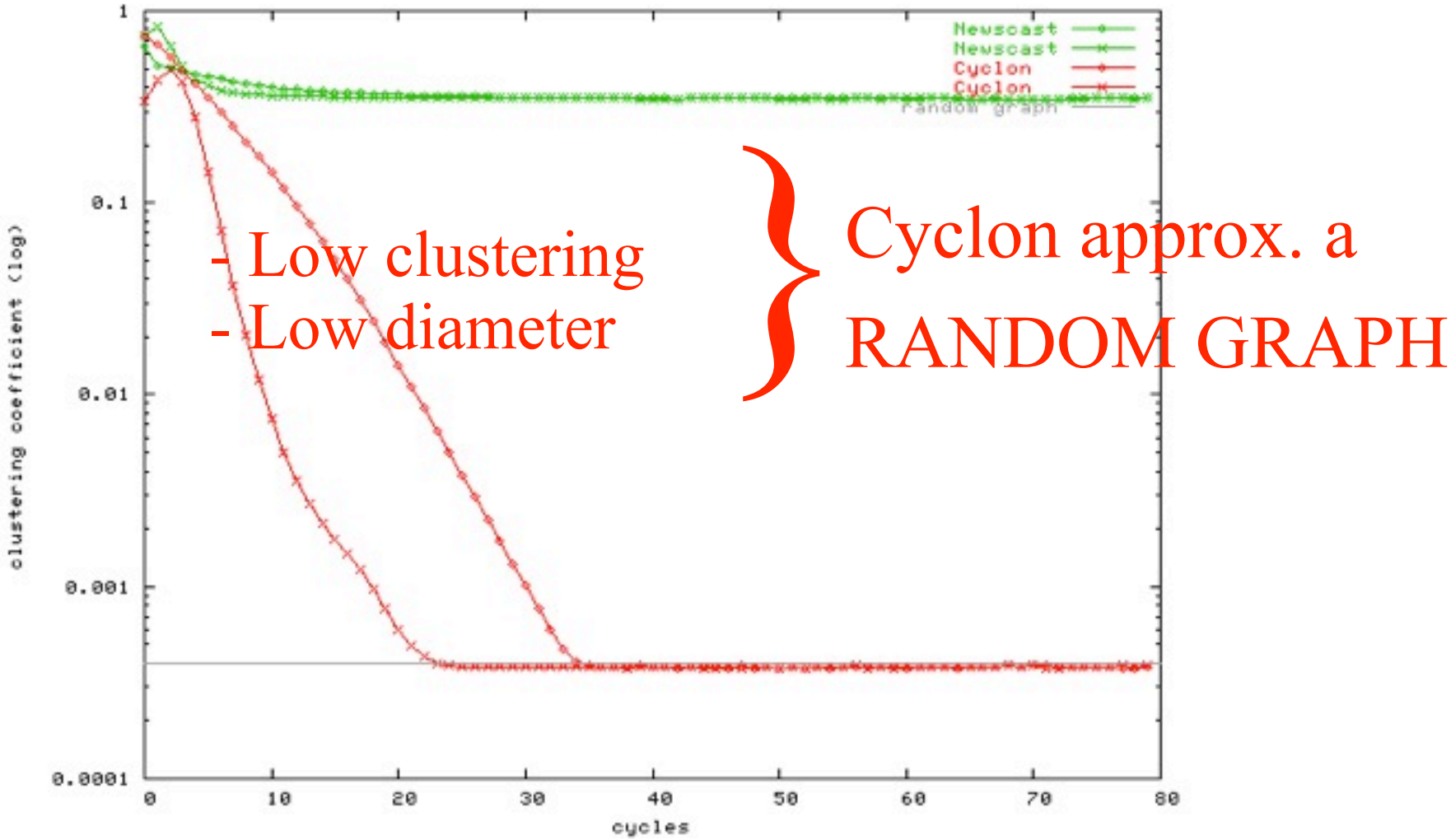
# Clustering coefficient

- **Low clustering coefficient is good for epidemic protocols**
  - Reduce the number of redundant messages

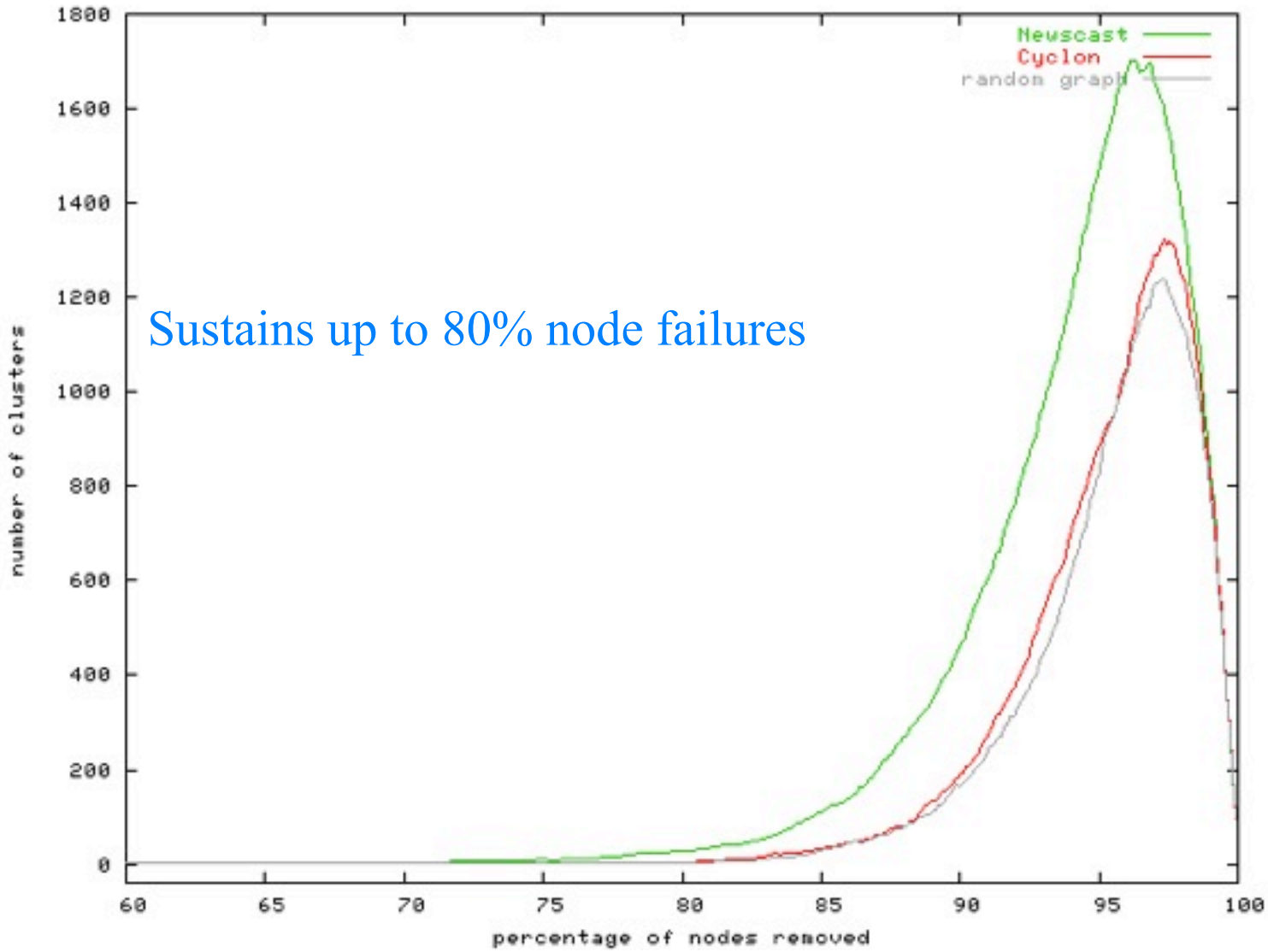


# Clustering coefficient

- **Low clustering coefficient is good for epidemic protocols**
  - Reduce the number of redundant messages



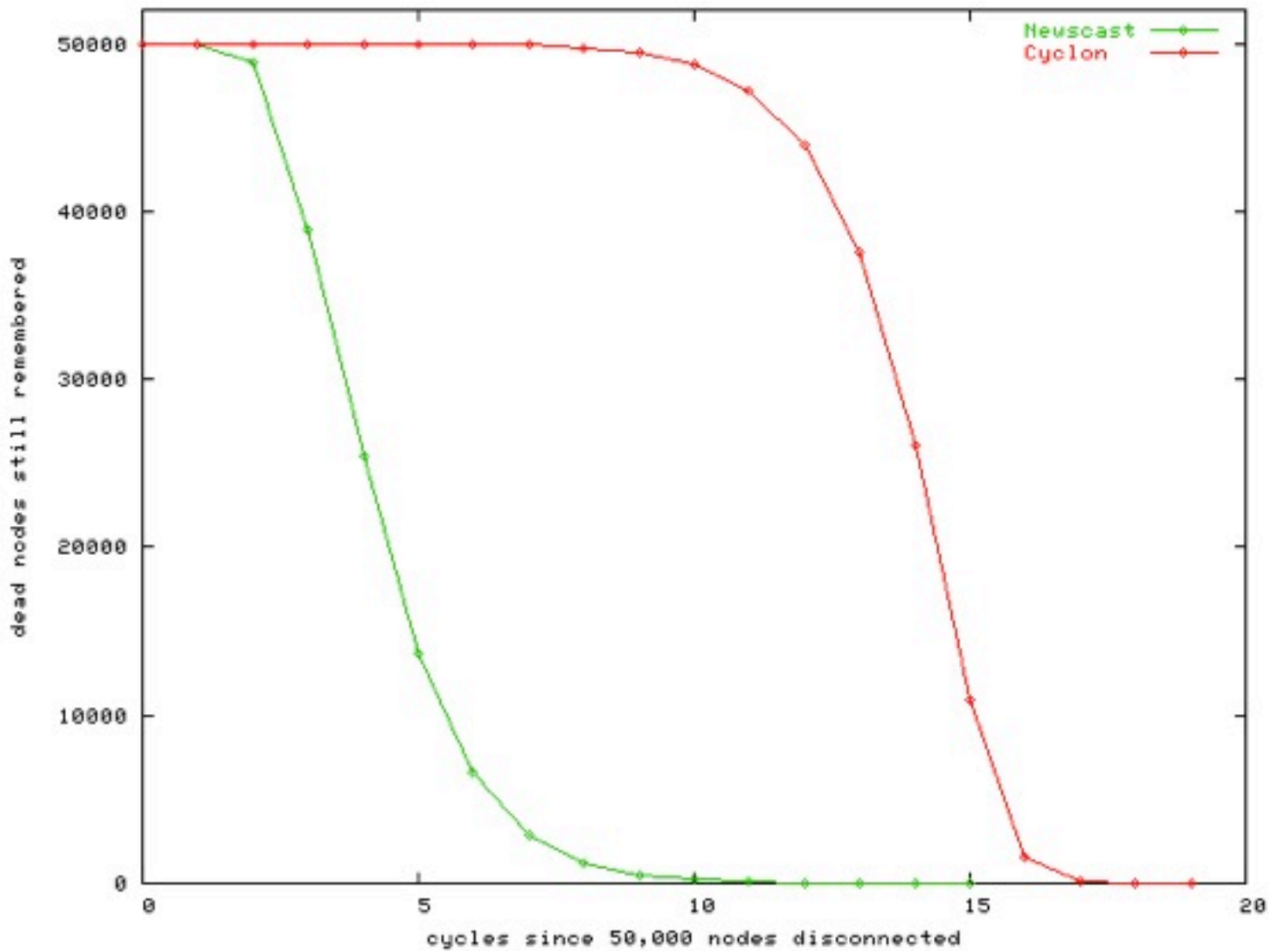
# Robustness




Sustains up to 80% node failures

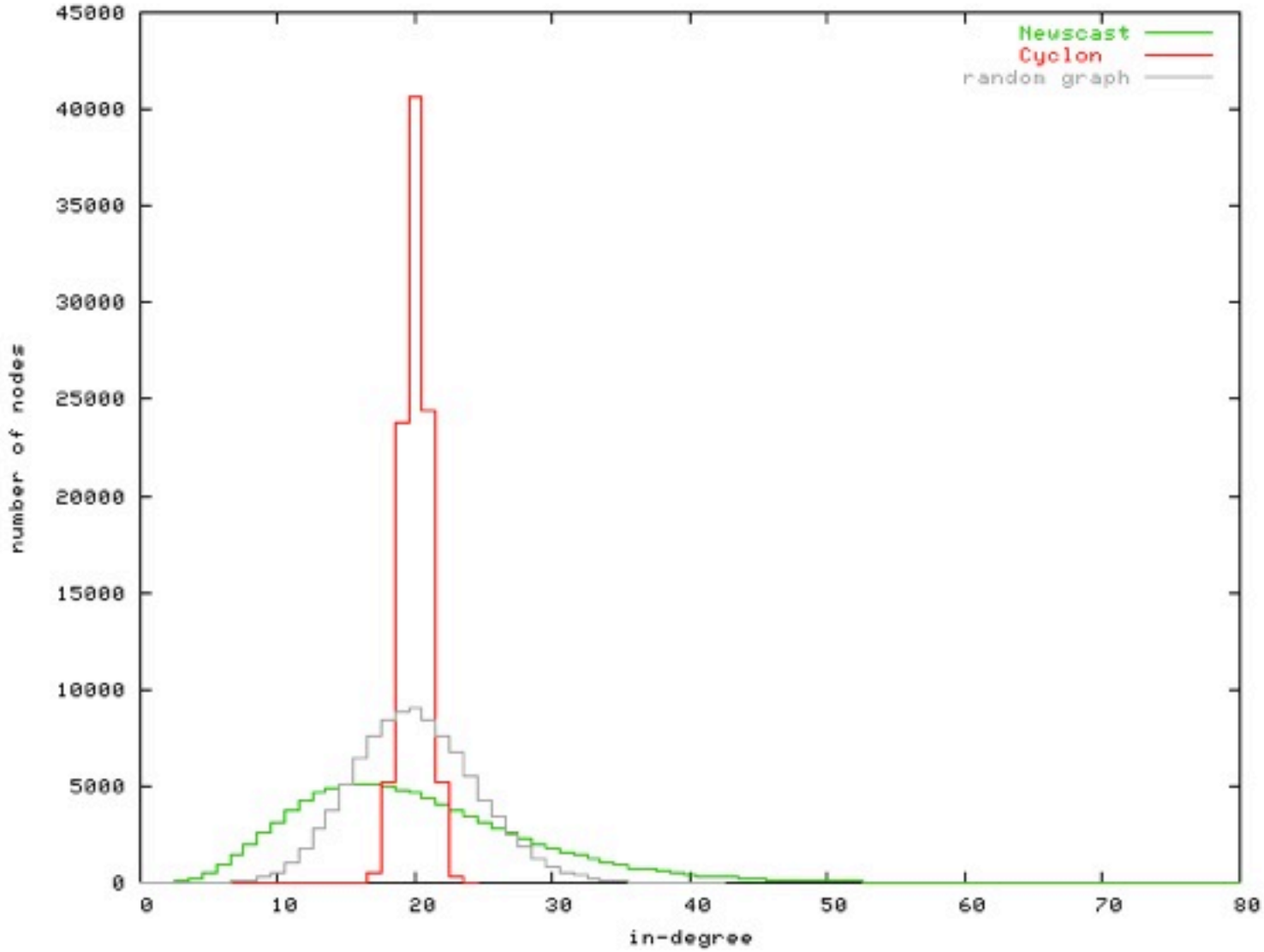
# Self-healing behaviour

- **If a node disappears, its descriptors will be progressively forgotten (self-cleaning)**



# In-degree Distribution

- **In-degree is almost constant, irrespectful of the size of the network**
  - Corresponds to the cache size: $N \cdot c / N = c$



## Adding the cloud to the view

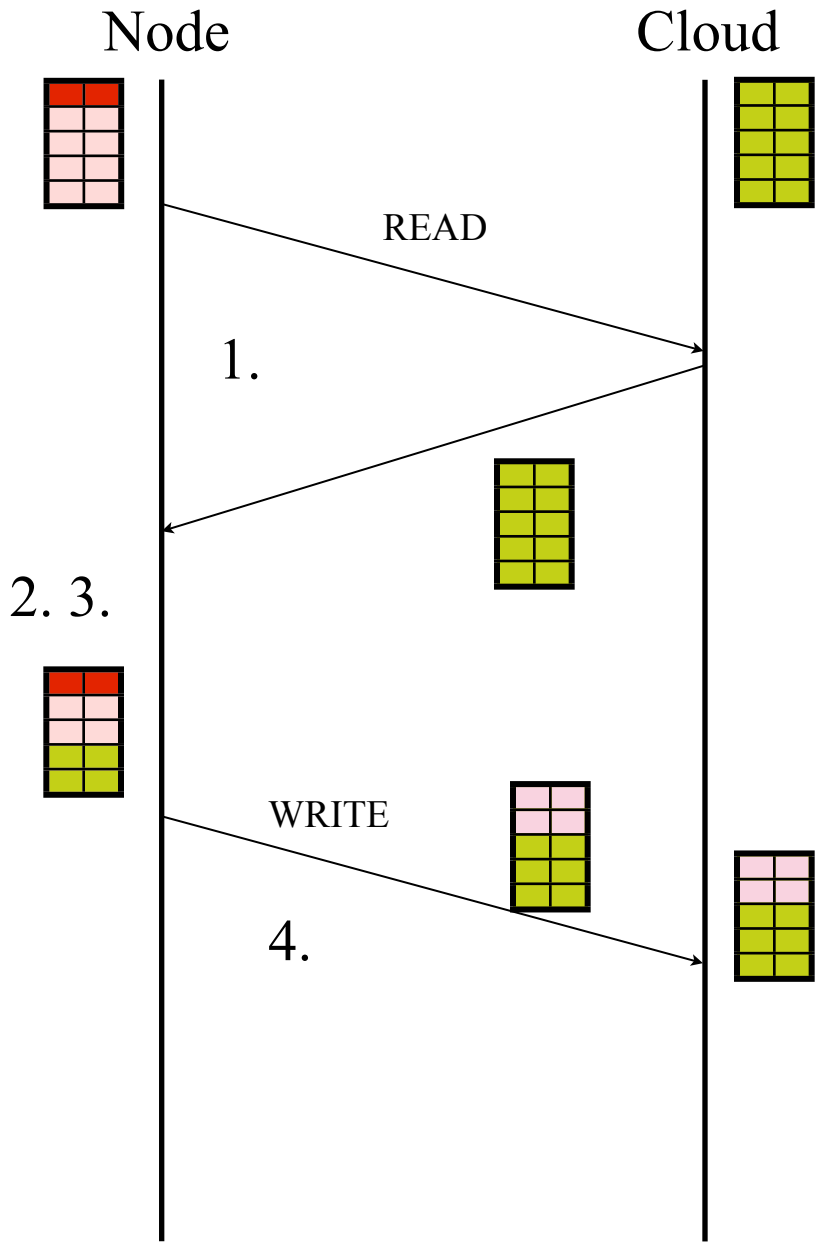
- **Monetary cost**
  - 744$/year for maintaining an EC2 instance always on
  - Bandwidth costs (# cycles per year · avg. number of contact per cycle)
- **Theoretically, how many times the cloud will be contacted per cycle?**
  - $c$ descriptors pointing to the cloud
  - $1/c$ is the probability of being the oldest
  - The expected number of contacts per cycle is 1
- **The cost is dominated by the EC2 instance**

## The novel approach

- **Switch to S3**
  - You pay only for the bandwidth
  - You can store everything there:
    - Partial views
    - Dilbert strips
    - Ashton's messages (really, who cares?)
  - You get high-availability and durability
- **The main issue:**
  - How we do guarantee "1 contact per cycle" if the cloud is a passive subject?

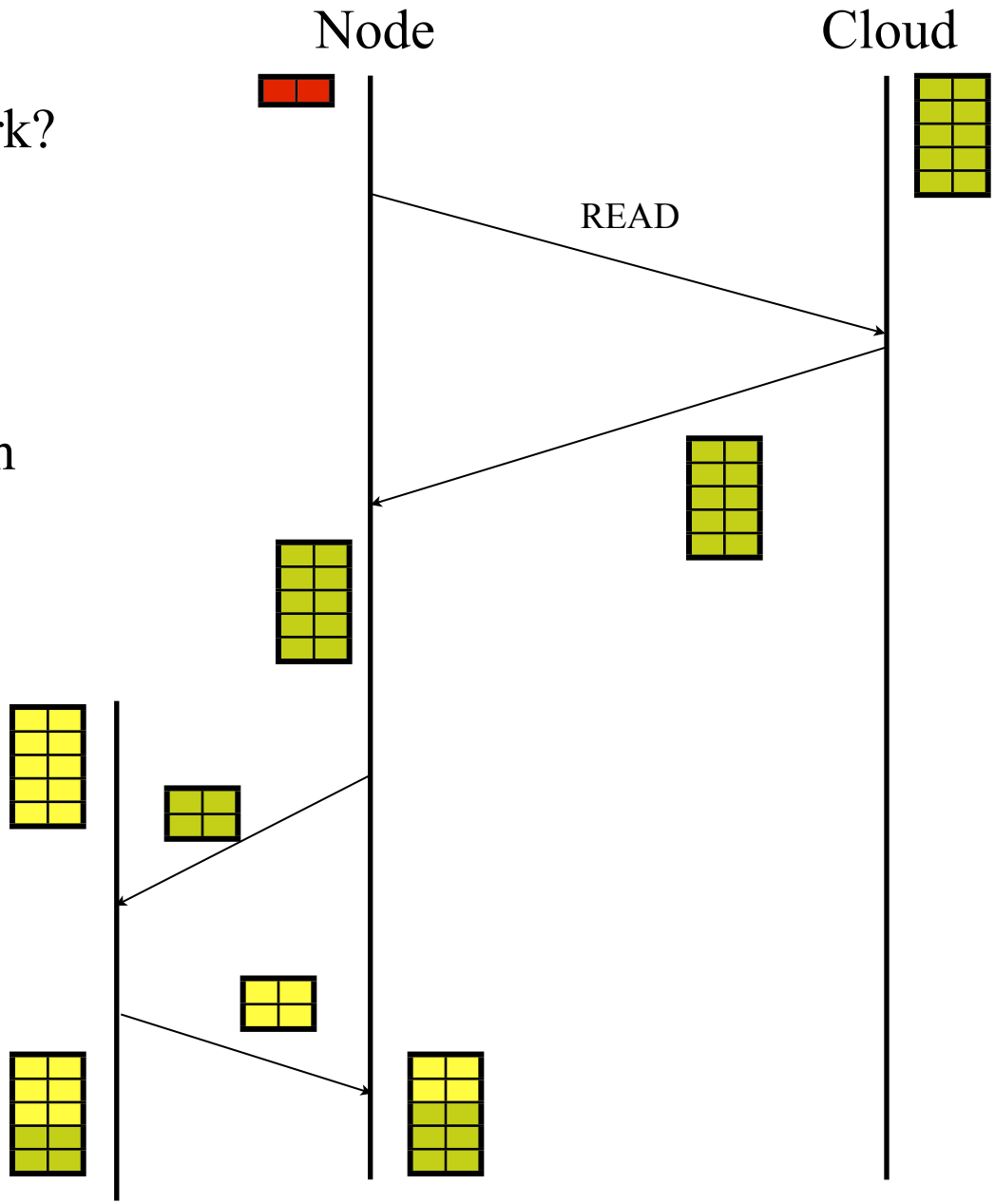
# Gossiping in the cloud

- **Issue:**
  - S3 cannot run an active thread
  - Cloud descriptors will be progressively forgotten
- **Solution:**
  - When a node contacts the cloud, plays an active interaction on behalf of the cloud
    1. Read the view from the cloud
    2. Randomly shuffle the local view and the cloud view
    3. Maintains a cloud descriptor, with a fresh timestamp
    4. Write the view to the cloud



# Gossiping in the cloud

- **Issue:**
  - How new nodes join the network?
- **Solution:**
  - The start with a cloud reference
  - They read the current view from the cloud
  - They start the protocol all over



## Gossiping in the cloud

- **In presence of high level of churn, some issues exist:**
  - The number of references may grow in an inexpected way
  - The number of refences may decrease in an expected way
- **Details of why this happen is beyond the goal of this talk**
- **Solution: control mechanism**
  - Cloud objects have “last modified” information
  - When contacting a cloud
    - If last modified is too close in time (¼ cycle), too many cloud references
      - do not maintain the cloud reference
    - If last modified is too far in time (4 cycles), too few cloud reference
      - maintain the cloud reference and create a new one
  - The system remain stable

# Scalability vs churn

- **Simulation length:**

  1 day

- **# of experiments**

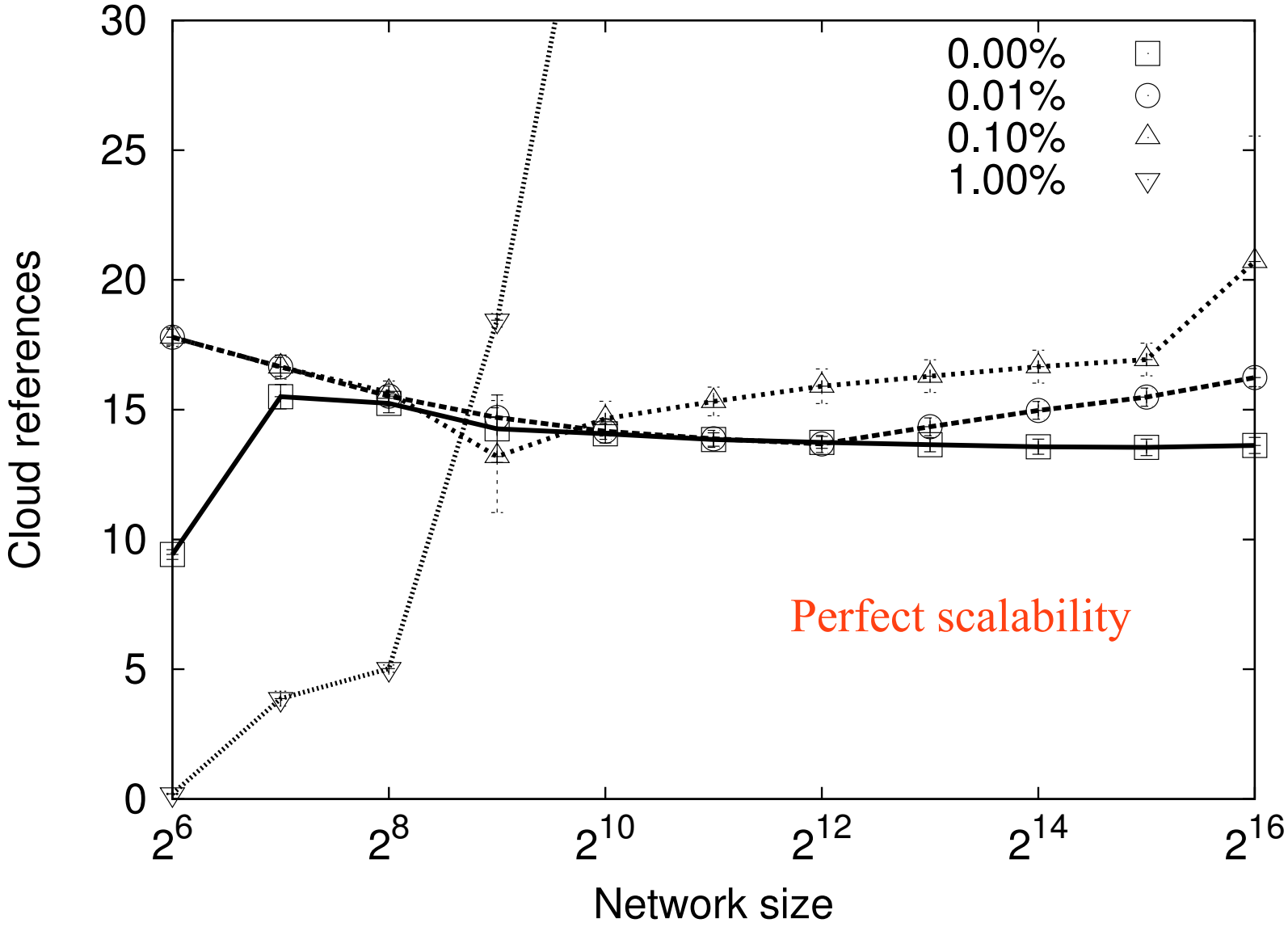
  50

- **Value shown**

  Cloud in-degree

- **Churn level**

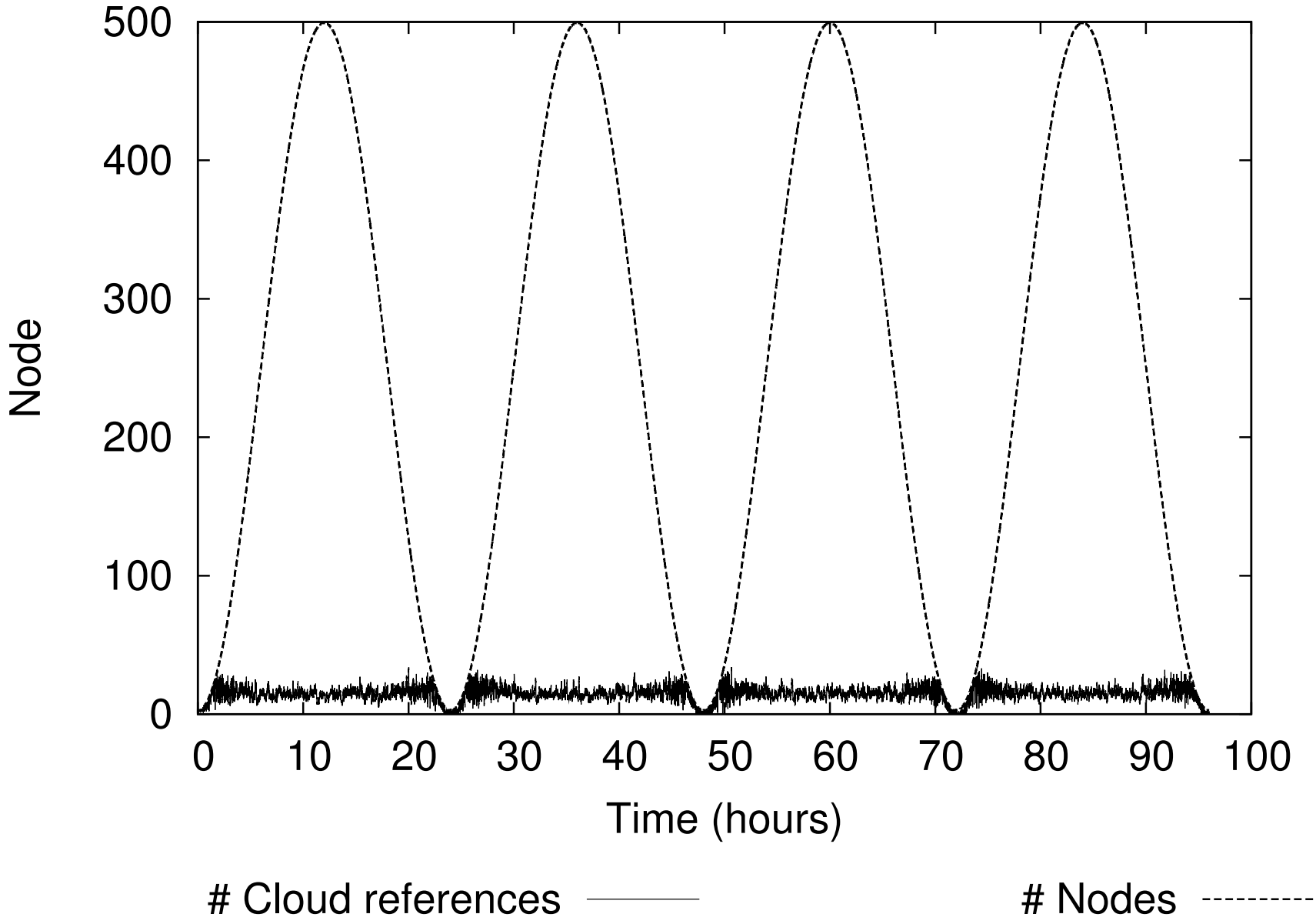
  *x*% probability of leaving per node per second



# Gossiping in the cloud

- **Issue:**
  - what happens if the number of cloud references goes to zero?
  - May happen
    - if the groups shrinks to empty
    - if the network becomes disconnected...
- **Solution**
  - If nobody has seen a cloud reference for a while, create a new one
  - How we do it?
    - We gossip the freshest cloud timestamp nodes have recently seen
    - Each node compare the timestamp with a threshold, then create a new one

# What happens when the size goes to zero?



# Cloud load

- **Simulation length:**

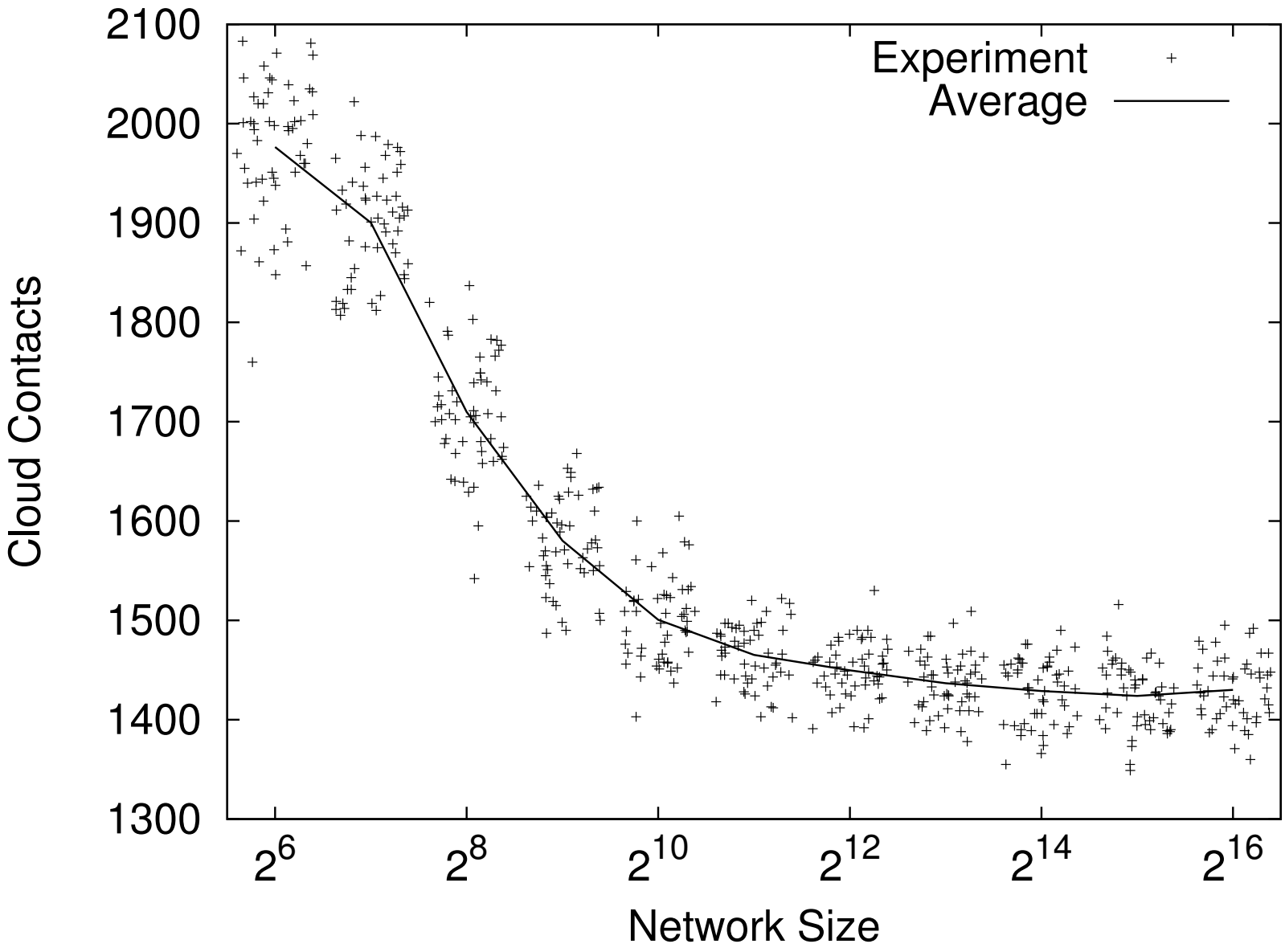
  1 day

- **# of experiments**

  50

- **Value shown**

  Number of contacts to the cloud in one day of simulation



# Check please!

- **GET/PUT requests**
  - 2000 GET · 365 days · 1 μ$ = 0.73 $ — 10.000 requests = 0.01$
  - 2000 PUT · 365 days · 10 μ$ = 7.30 $ — 1.000 requests = 0.01$
- **View download**
  - 2000 · 365 · 1000 byte · 0.15 n$ = 0.11 $ — 1GB transfer-out : 0.15$
- **View upload**
  - 2000 · 365 · 1000 byte · 0.10 n$ = 0.07 $ — 1GB transfer-in : 0.10$
- **Storing your view for one year - priceless!**
- **Total cost for peer sampling:**
  - 8.21 $ = 15.42 R$ per year (was 15.21 R$ yesterday, sorry)

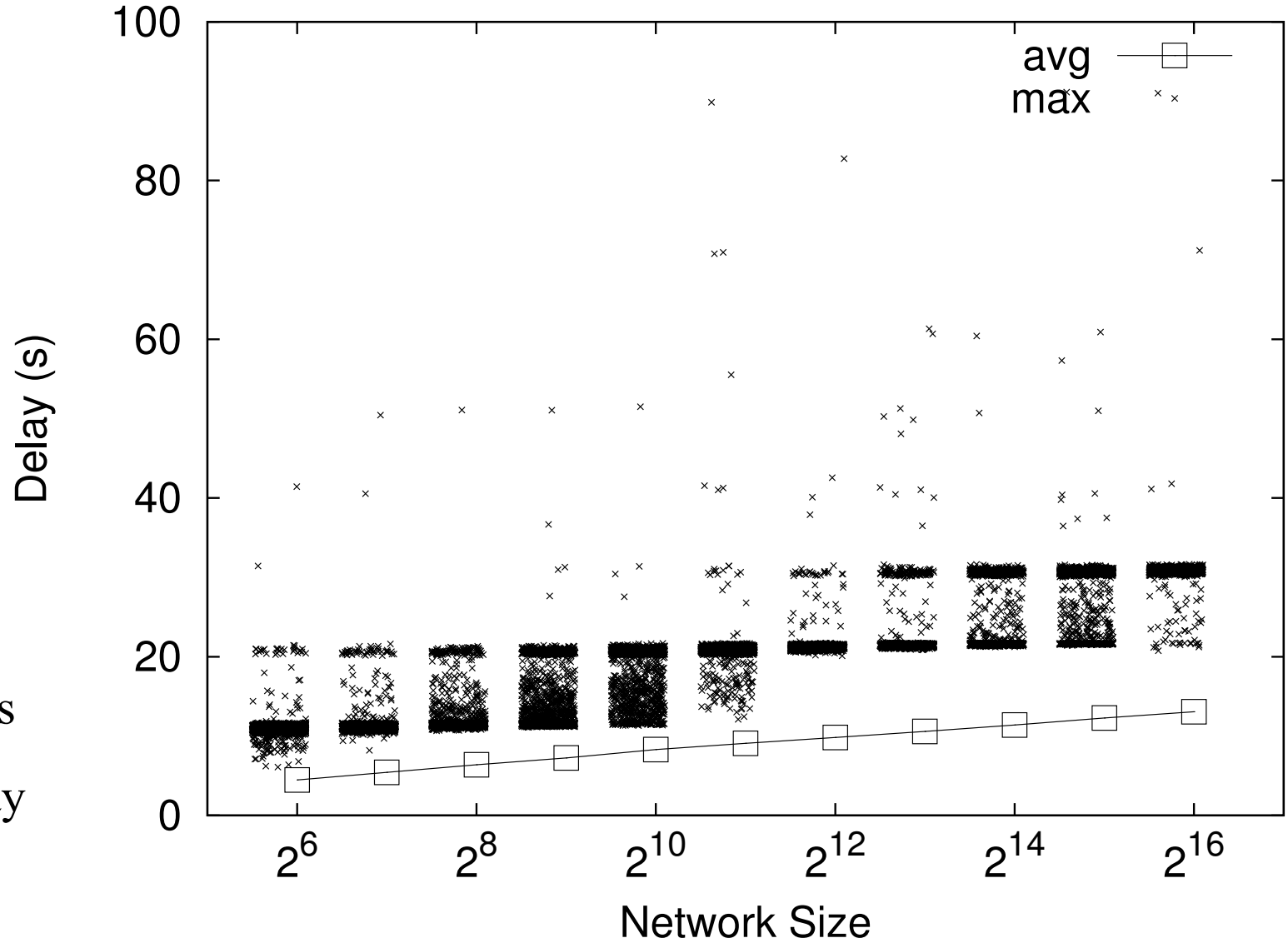
# Information dissemination

| Parameter | Value | Meaning |
|---|---|---|
| $n$ | 1-$2^{16}$ | Total number of nodes |
| $\delta_{cyclon}$ | 10s | Cycle length of CYCLON |
| $\delta_{rumor}$ | 1s | Cycle length of rumor mongering |
| $\delta_{entropy}$ | 10s | Cycle length of anti-entropy |
| $c$ | 20 | Number of peer references in the view |
| $g$ | 5 | CYCLON message size |

- **The creator of a new message**
  - Write the message in the cloud
  - Update a message counter in the cloud
  - Starts a rumor-mongering broadcasting on the new message
- **Rumor-mongering (push)**
  - Every node sends the message (the hot rumor) to a random peer
  - Stops forwarding the rumor with a given probability (0.2)
- **Anti-entropy (push-pull)**
  - Less frequently, node exchange summaries of the message received so far and download missing ones if needed

# Message diffusion

- **Length of the simulation**
  - 1 day
- **Messages**
  - 1 x minute
  - 1440 total
- **Value shown**
  - Average delay for all messages
  - Maximum delay for each msg



# Information dissemination

- **How the cloud is involved?**
  - In rumor-mongering: never
- **In anti-entropy:**
  - Each time the cloud is selected in an anti-entropy cycle
    - Read the message counter
    - If the message counter is new,
      - read the object
      - start a rumor-mongering push
  - The expected number of contacts per cycle is around 1

# Check please!

- **Dilbert - a total of 8.21$ +0.87$ + 1.2$ = $10.28 = 19.31 R$**
- **We did it!**

| Operation | Amount | Total Price |
|---|---|---|
| Strip upload | 365 x (50KB)=<br>18.25 MB + small overhead | Priceless! |
| Strip storage | 18.25 MB | Priceless<br>(or 1.2$) |
| Strip download (1) | 180.25 MB | $0.03 |
| GET message counter (2) | 730.000 GET<br>730 MB | $0.73<br>$0.11 |

(1) Assuming that each strip is downloaded at most 10 times
(maximum delay observed in our experiments 100s divided by δ=10s)

(2) Assuming the same (exceptional) overhead of peer sampling

# Check please!

- **Similar results with Ashton Kutcher**

| Operation | Amount | Total Price |
|---|---|---|
| Tweet upload | 365 x 8 x 140 B =<br>408 KB + small overhead | Priceless! |
| Tweet storage | 408 KB | Priceless<br>(or 1.2$) |
| Tweet download (1) | 4.08 MB | under $0.01 |
| GET message counter (2) | 730.000 GET<br>730 MB | $0.73<br>$0.11 |

(1) Assuming that each strip is downloaded at most 10 times
(maximum delay observed in our experiments 100s divided by δ=10s)

(2) Assuming the same (exceptional) overhead of peer sampling

# Conclusions

- **We have demonstrated a novel approach to mixing P2P and cloud computing**
- **Many open questions:**
  - There is obviously an unbalance between peer sampling and message diffusion
  - Avoiding so many GET/PUT requests
  - Make the system more adaptive - if the groups grows, the cloud is used less and less (just to guarantee durability)
- **What are the possible applications?**
  - Social networking: "*my friends will serve my content, unless none of them is around, in which case you can use the cloud*"
    - many friends - the cloud is not needed
    - few friends - feel free to use the cloud

# Obrigado!

- **Thanks**
  - to Marinho, Luciano, and Lisandro for the invitation
  - to Weverton and Flavio for taking good care of me
  - to Adler for the rafting
  - to my co-author Luca Abeni for the good work together
  - to you for listening!
- **Questions?**