Intuitive distributed algorithms

with F#

Natallia Dzenisenka @nata_dzen
Alena Hall @lenadroid

“A tour of a variety of ~~intuitive~~ distributed algorithms used in practical distributed systems.

… and how to prototype with F# for better understanding”

Why distributed algorithms?

Sorting algorithms

Not **everyone** needs to know…

… But it can be really useful


Things
Will
Break

No blind troubleshooting

TUNING

Edge cases

… or even create your own new Distributed System ***

*** **WARNING**!

Riak

HIGH AVAILABILITY

Big Data applications require data all of the time.

SCALABILITY

Performance scales easily to meet your application requirements.

FAULT TOLERANCE

Reads and writes non-stop, regardless of outages or network partitions.

OPERATIONAL SIMPLICITY

Adds automation and efficiency to minimize manual processes.

Cassandra

What is Cassandra?

The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data.Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages.

PROVEN

Cassandra is in use at Constant Contact, CERN, Comcast, eBay, GitHub, GoDaddy, Hulu, Instagram, Intuit, Netflix, Reddit, The Weather Channel, and over 1500 more companies that have large, active data sets.

FAULT TOLERANT

Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.

PERFORMANT

Cassandra consistently outperforms popular NoSQL alternatives in benchmarks and real applications, primarily because of fundamental architectural choices.

DECENTRALIZED

There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical.

SCALABLE

Some of the largest production deployments include Apple's, with over 75,000 nodes storing over 10 PB of data, Netflix (2,500 nodes, 420 TB, over 1 trillion requests per day), Chinese search engine Easou (270 nodes, 300 TB, over 800 million requests per day), and eBay (over 100 nodes, 250 TB).

DURABLE

Cassandra is suitable for applications that can't afford to lose data, even when an entire data center goes down.

YOU'RE IN CONTROL

Choose between synchronous or asynchronous replication for each update. Highly available asynchronous operations are optimized with features like Hinted Handoff and Read Repair.

ELASTIC

Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.

PROFESSIONALLY SUPPORTED

Cassandra support contracts and services are available from third parties.

HBase

Welcome to Apache HBase™

Apache HBase™ is the Hadoop database, a distributed, scalable, big data store.

Use Apache HBase™ when you need random, realtime read/write access to your Big Data. This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware. Apache HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al. Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.

Download

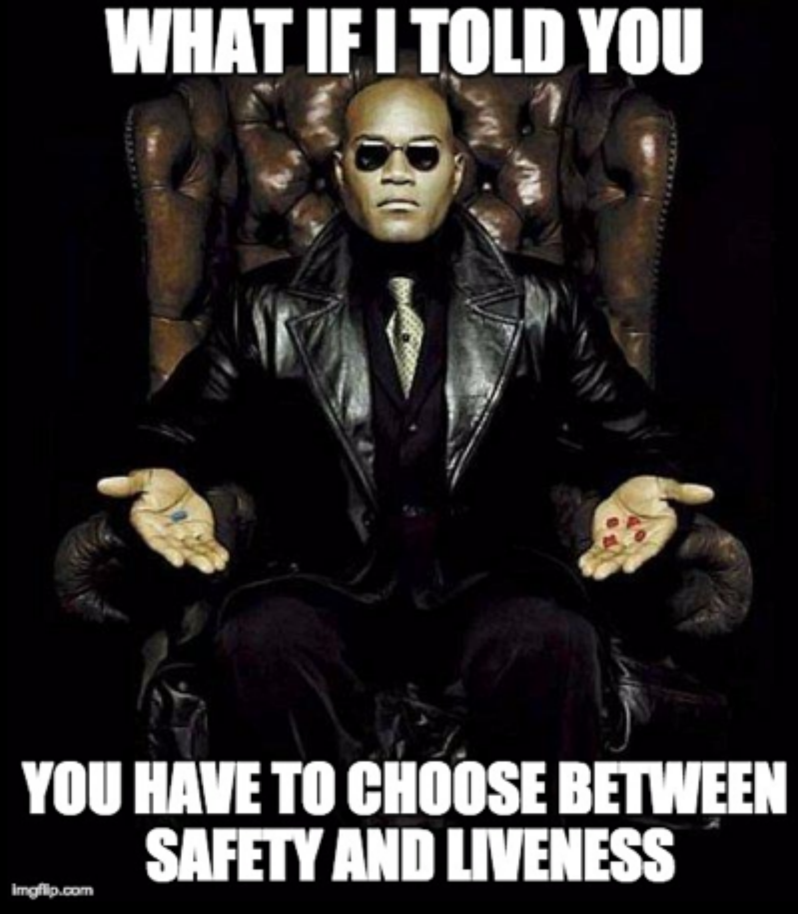
Click **here** to download Apache HBase™.

Features

- Linear and modular scalability.
- Strictly consistent reads and writes.
- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers.
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
- Easy to use Java API for client access.
- Block cache and Bloom Filters for real-time queries.
- Query predicate push down via server side Filters
- Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options
- Extensible jruby-based (JIRB) shell
- Support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia; or via JMX

Abstract words actually mean concrete algorithms


Liveness
Safety


WHAT IF I TOLD YOU
YOU HAVE TO CHOOSE BETWEEN SAFETY AND LIVENESS
imgflip.com

No “To Be Or Not To Be”

Will show algorithms that provide different types of guarantees for different purposes

System should be ready for events, changes and failures!

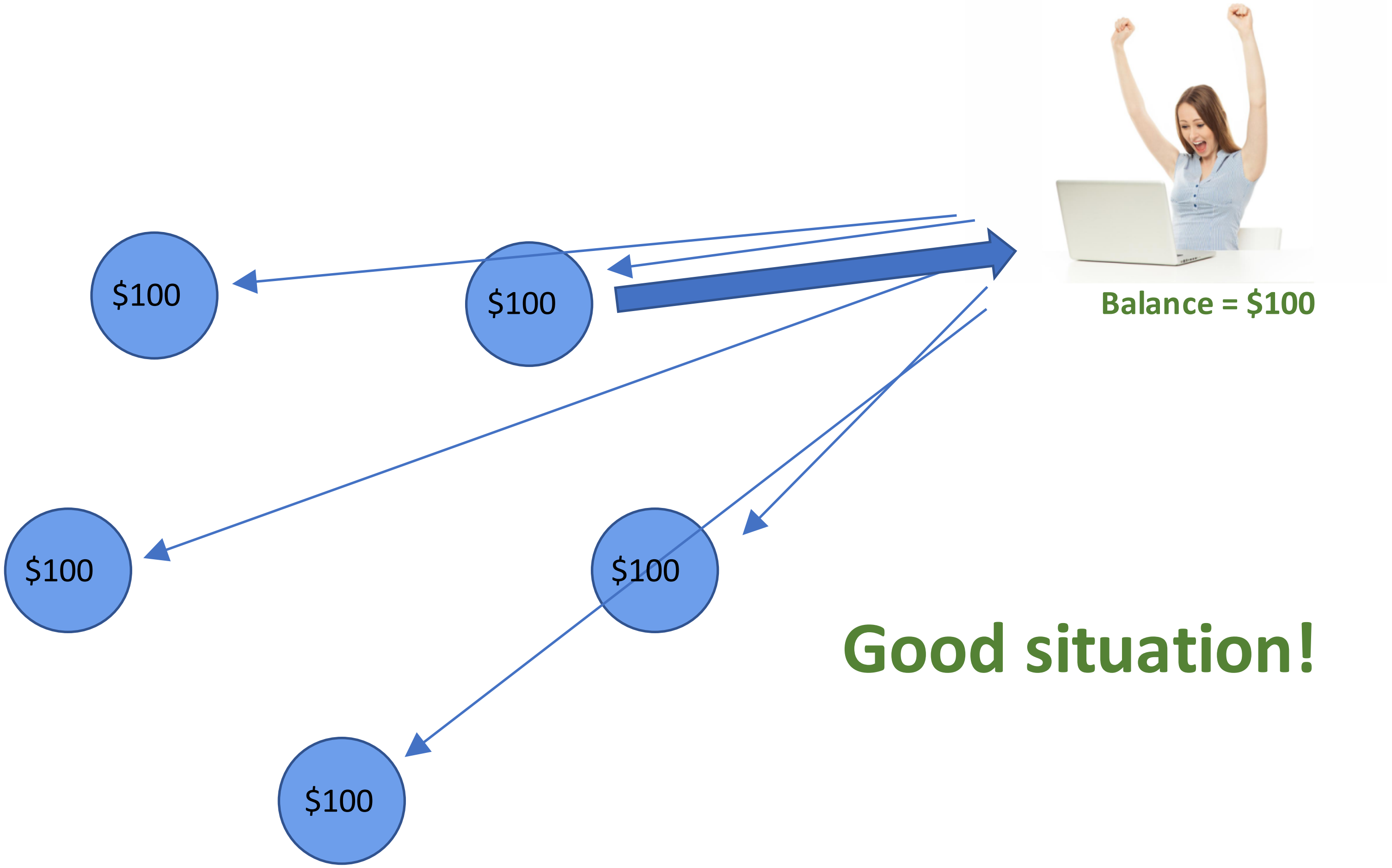
- A node can die — **[FAILURE DETECTION]**
- Node can join — **[GOSSIP, ANTI-ENTROPY]**
- Recover from failure — **[SNAPSHOTS, CHECKPOINTING]**

And a whole bunch of other things!

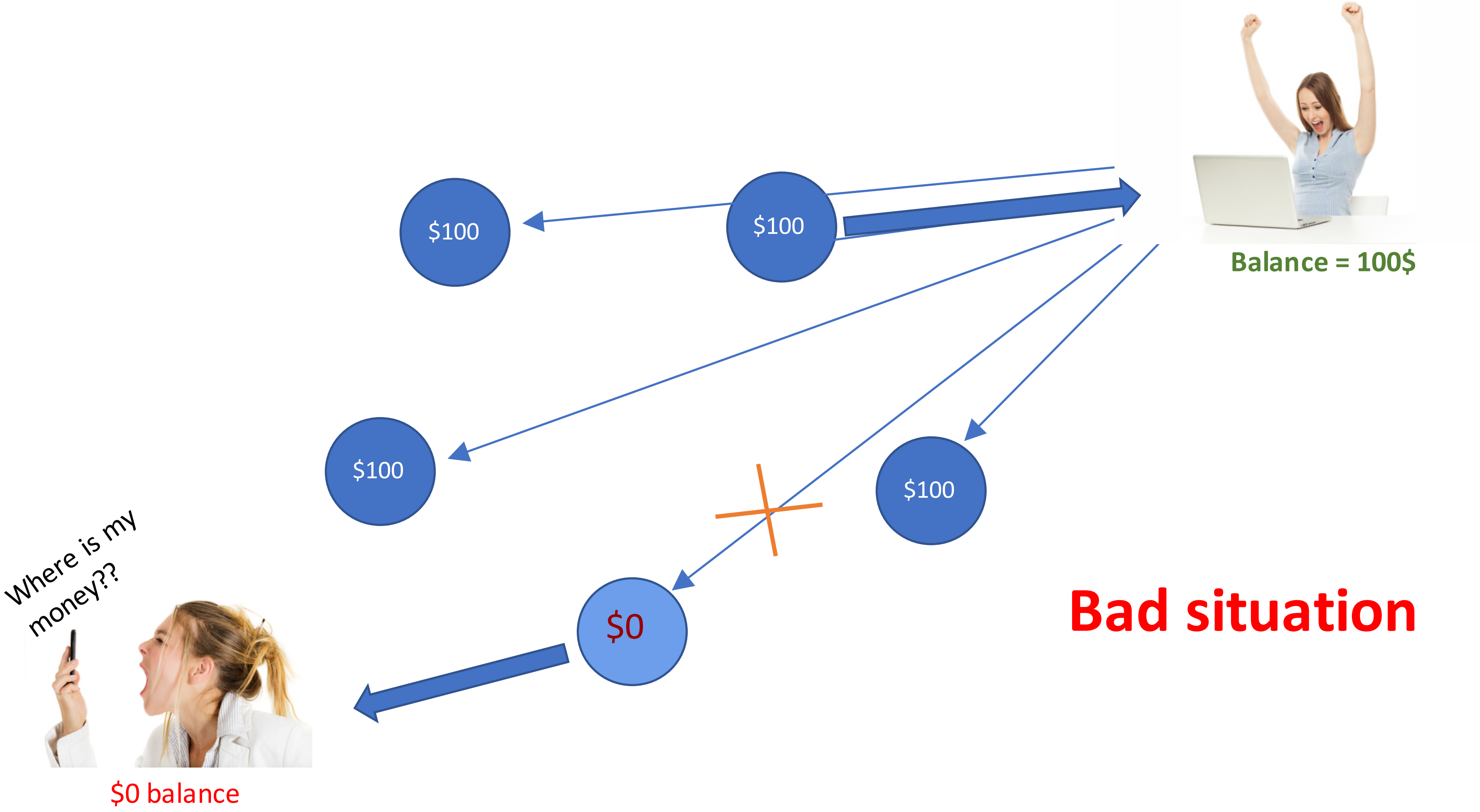
Contents

- ✓ Intro
- ➢ Consistency Levels
- ○ Gossip
- ○ Consensus
- ○ Snapshot
- ○ Failure Detectors
- ○ Distributed Broadcast
- ○ Summary

Consistency Levels



Good situation!



$100
$100
Balance = 100$
$100
$100
Where is my money??
$0
Bad situation
$0 balance

Good news: we can **choose** the level of consistency

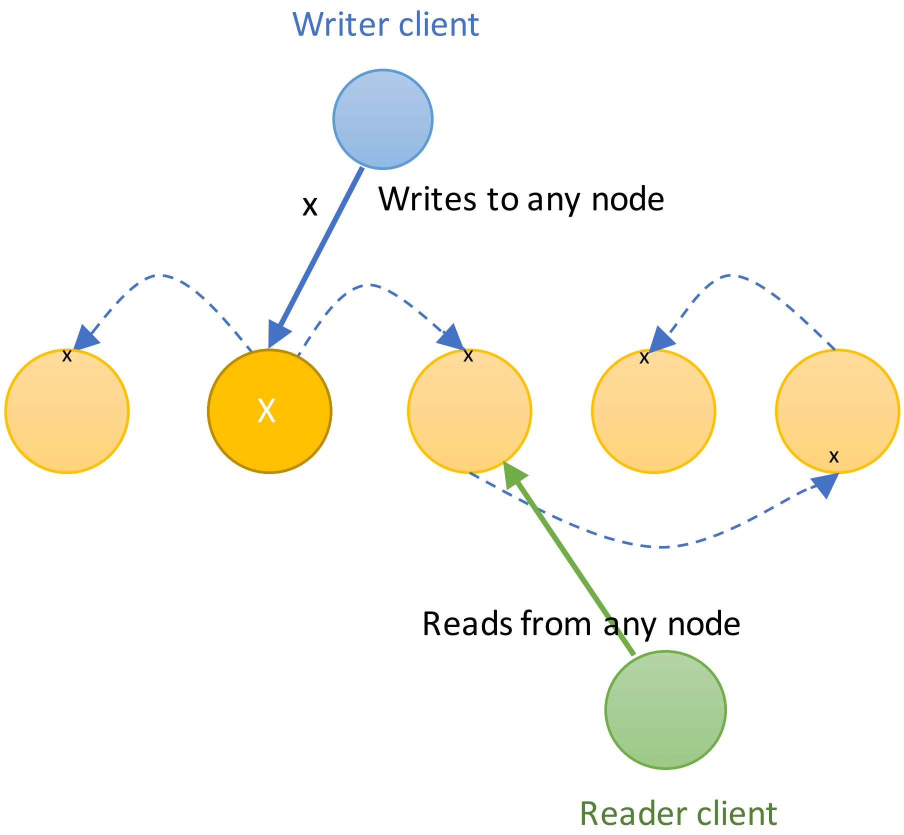
More consistency

→

Less availability

Let’s look at different consistency levels and how they affect a distributed system

The weakest consistency level

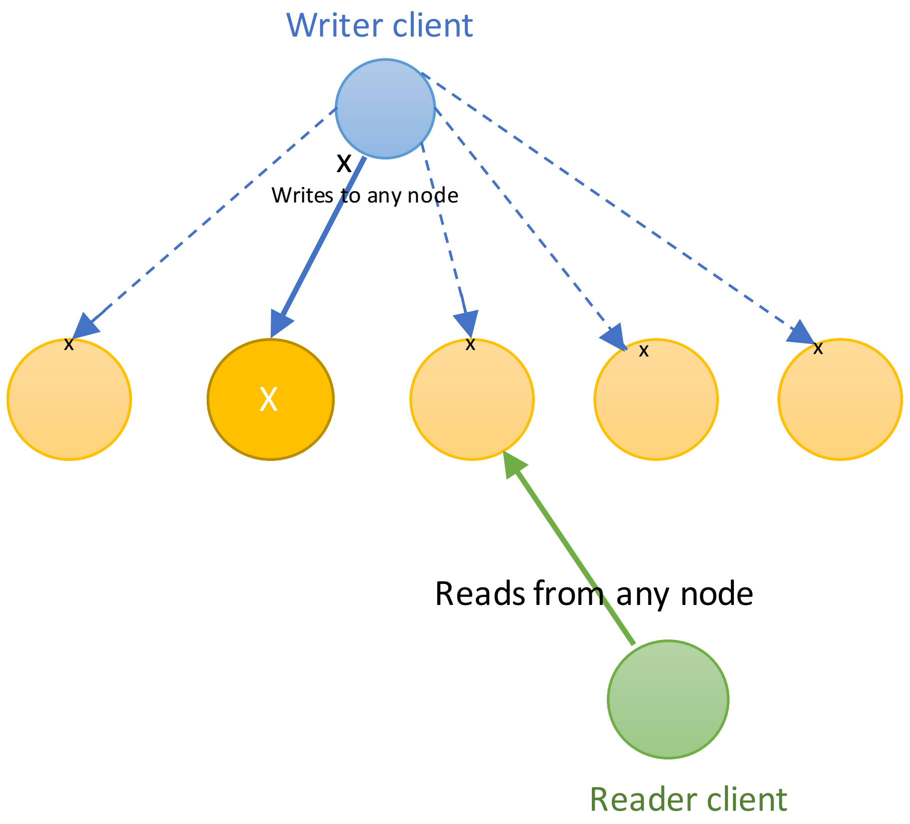


Weak Consistency

Advantages:	High availability, low latency
Trade-off:	Low consistency ← High propagation time

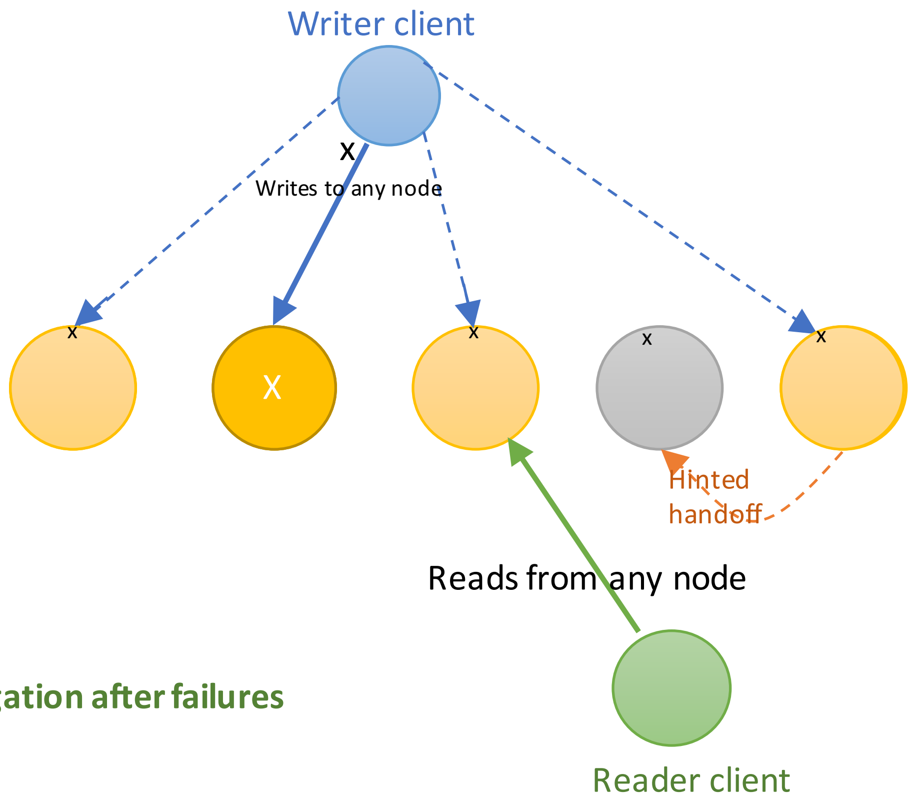
No guarantee of successful update propagation

Reducing propagation latency



NODE FAILS

Hinted Handoff

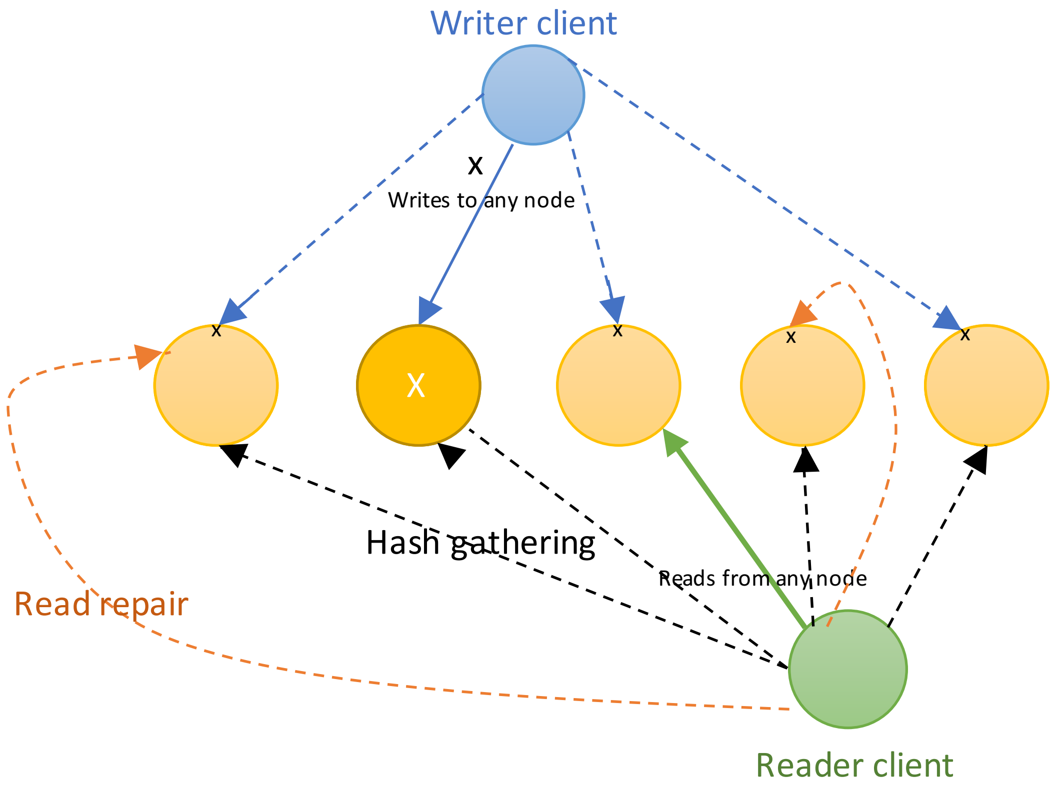


*** Faster update propagation after failures**

Coordinator fails

Hint file corrupted

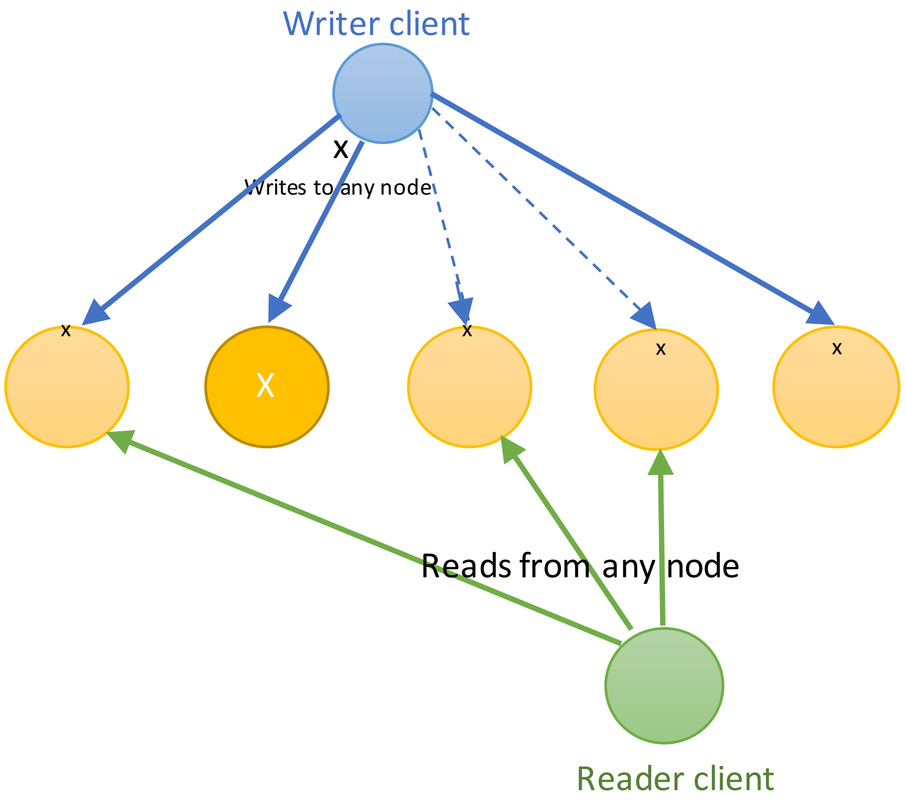
Read Repair



Writer client
X
Writes to any node
x
X
x
x
x
x
Hash gathering
Reads from any node
Read repair
Reader client

Can we get **stronger** consistency?

Quorum


Writer client
X
Writes to any node
x
X
x
x
x
Reads from any node
Reader client

Quorum

W – number of replicas where clients synchronously writes **[WRITE QUORUM]**

R – number of replicas from which client reads **[READ QUORUM]**

N – number of replicas

- $W > N/2$
- $W + R > N$

By following the rule we ensure that at least 1 replica will return fresh data during read operations.

Quintuplets and approximate estimation of Cassandra partition size

$$S_{\text{partition}} \approx \sum_i sizeOf(c_{k_i}) + \sum_j sizeOf(c_{s_j}) + N_r \times \left(8 + \sum_a sizeOf(c_a) + \sum_l sizeOf(c_{c_l}) \right)$$

How to **prevent** inconsistency?

Centralization

Consensus

… to be continued

Consistency levels in real distributed systems

- Cassandra and Riak use **weak consistency** level protocol for metadata exchange, **hinted-handoff** and **read repair** techniques to improve consistency.
- You can **choose read and write quorums** in Cassandra.
- HBase uses **master/slave** asynchronous replication.
- Zookeeper writes are also performed by **master**.

Contents

- ✓ Intro
- ✓ Consistency Levels
- ➢ Gossip
- o Consensus
- o Snapshot
- o Failure Detectors
- o Distributed Broadcast
- o Summary

Gossip

Common use cases of Gossip

- Failure detection: determining which nodes are down
- Solving inconsistency
- Metadata exchange: i.e. changes in distributed DB topology
- Cluster membership: new, failed or recovered nodes
- Many more

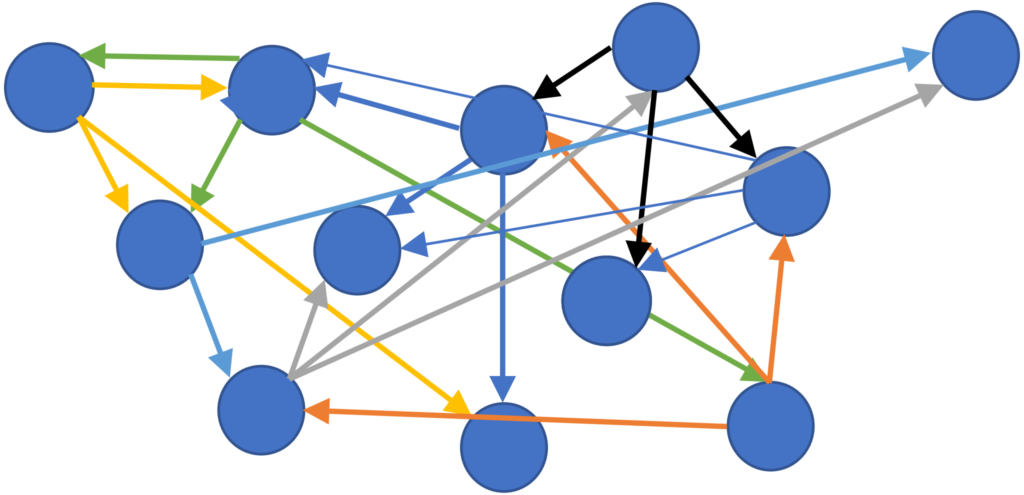
Use Gossip when system needs to be...

- Fast and scalable
- Fault-tolerant
- Extremely decentralized

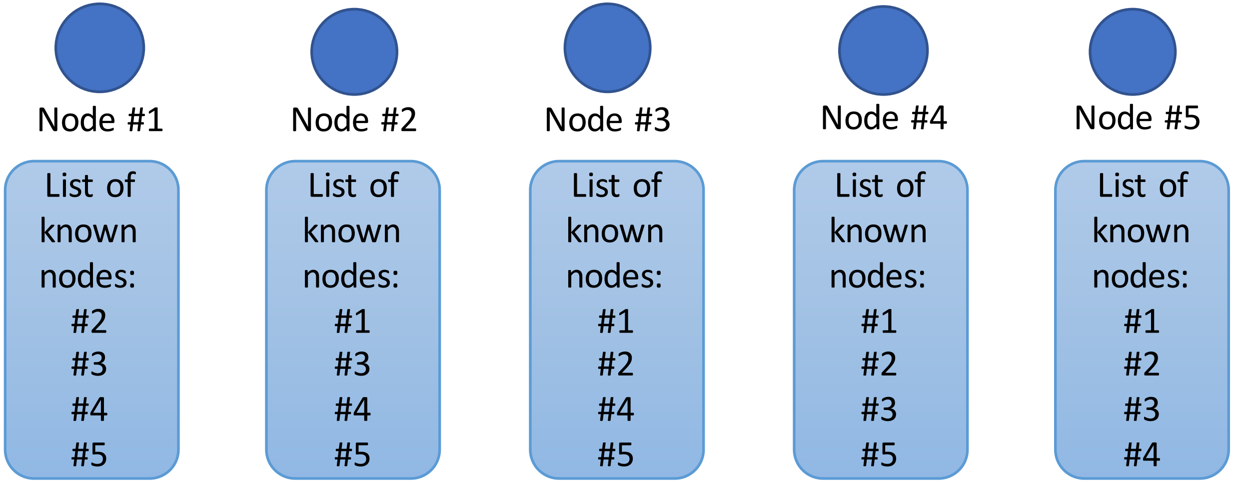
Or when it has huge number of nodes

General gossip

Endless process when each node periodically selects N others nodes and sends information to them



General gossip



Key: **42**
Value: “status: **green**”

Key: **42**
Value: “status: **red**”

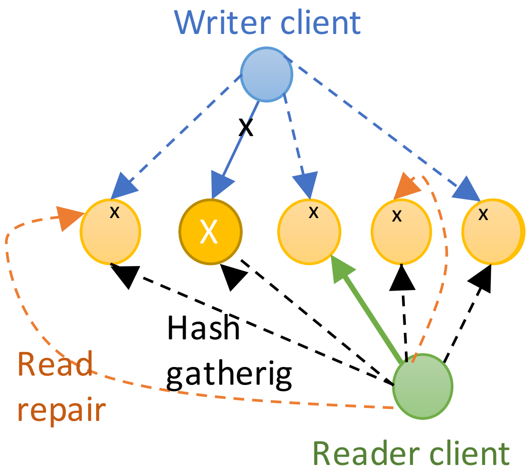
Key: **42**
Value: “status: **green**”
Timestamp: **1357589992**

Key: **42**
Value: “status: **red**”
Timestamp: **1357625899**

Anti-entropy repair

Solves the state of inconsistently replicated data

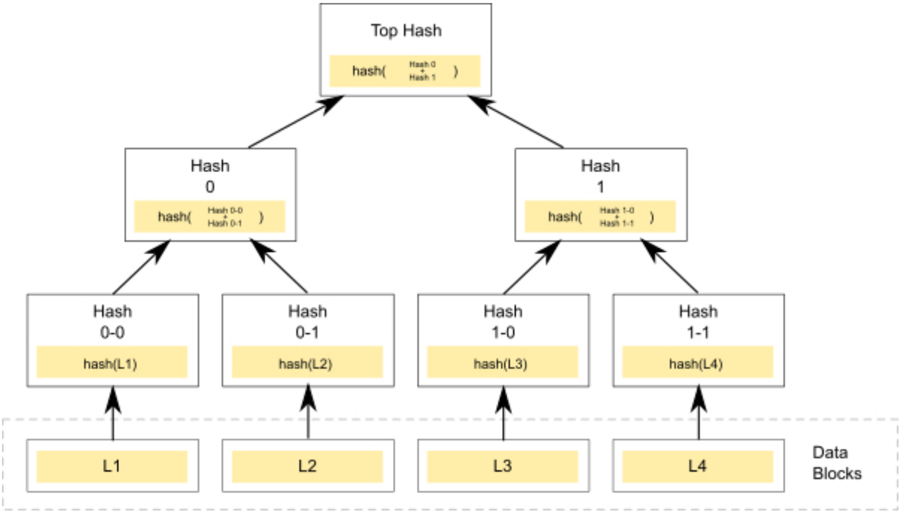
Read ⚒️ runs on reads…



Anti-entropy ⚒️ can run constantly, periodically or can be initiated manually.

Merkle trees for Anti-entropy

Tree data structure which has hashes of data on the leaves and hashes of children on non-leaves.



Cassandra, Riak and DynamoDB use anti-entropy technique using Merkle trees.

In case of hash inequality - exchange of actual data takes place

Merkle trees

Cassandra

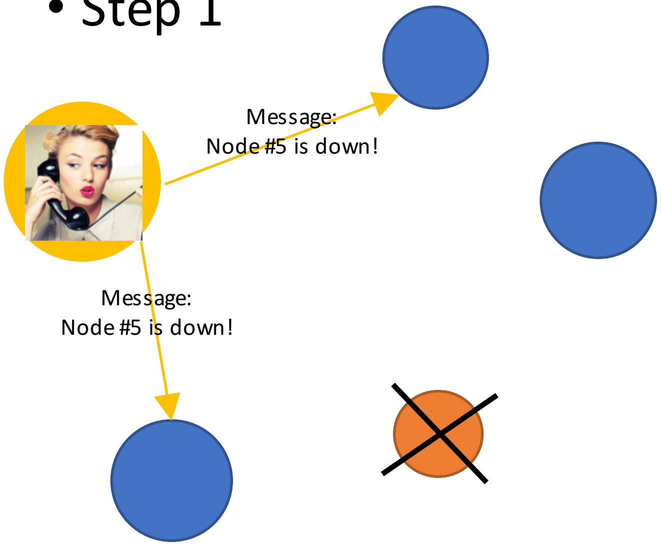
Short-lived in-memory

Riak

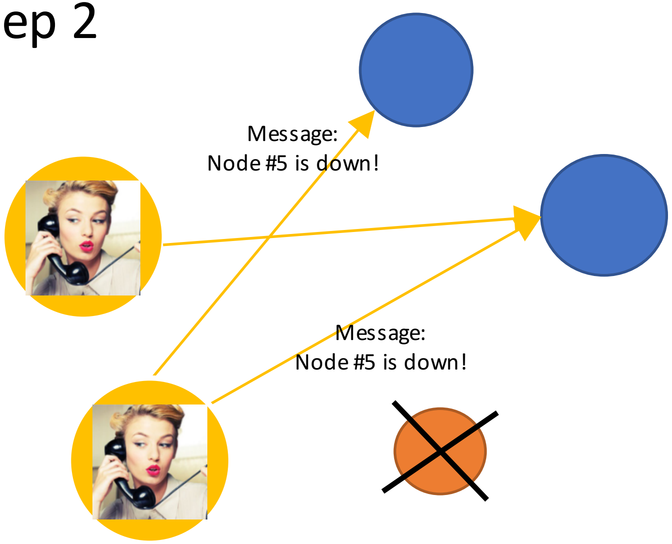
On-disk persistent

Rumor mongering

- Step 1



- Step 2



Real world use cases of Gossip

Riak

- Communicates ring state and bucket properties around the cluster

Cassandra

- Propagates information about database topology and other metadata
- For failure detection

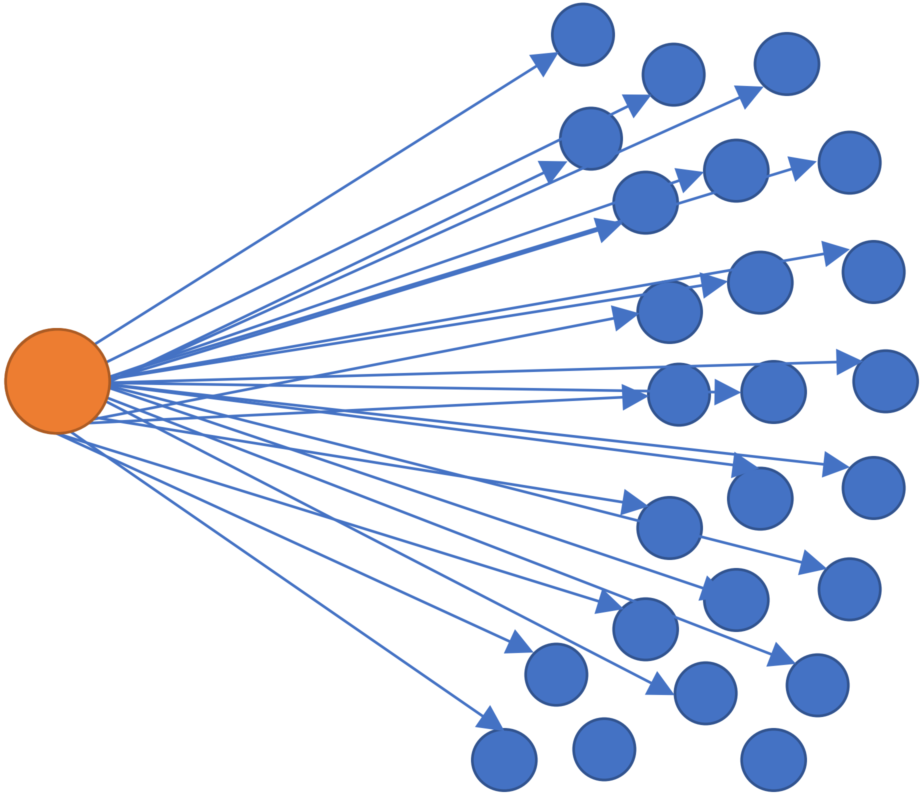
Consul

- To discover new members and failures and to perform reliable and fast event broadcasts for events like leader election. Uses SERF (based on "SWIM: *Scalable Weakly-consistent Infection-style Process Group Membership Protocol*")

Amazon S3

- To propagate server state to the system

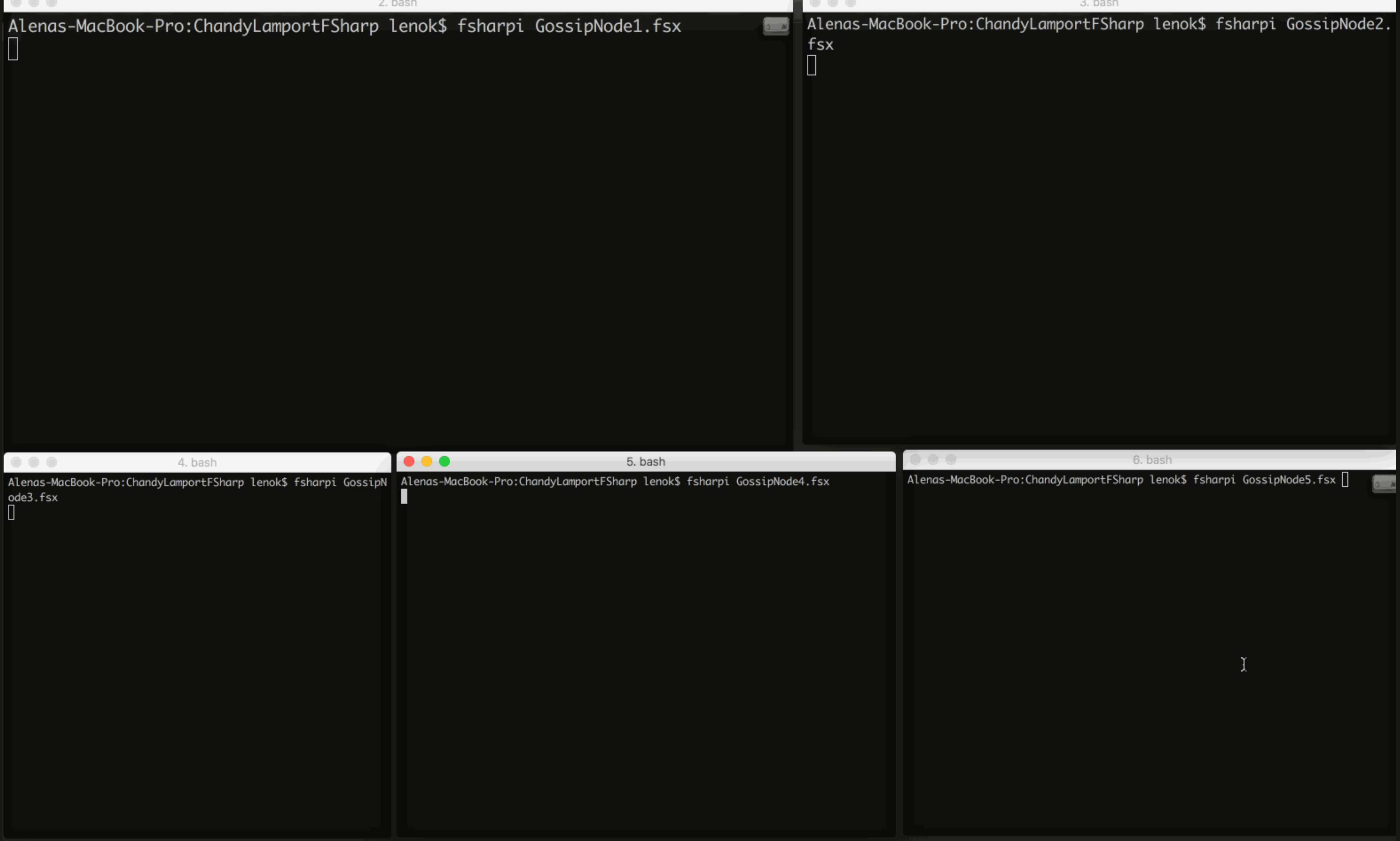
NOT gossip: one node is overloaded



Trade-offs with Gossip

- Propagation latency may be high
- Risk of spreading wrong information
- Weak consistency

General Gossip example


2. bash
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi GossipNode1.fsx
3. bash
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi GossipNode2.fsx
4. bash
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi GossipNode3.fsx
5. bash
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi GossipNode4.fsx
6. bash
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi GossipNode5.fsx

CODE

Contents

- ✓ Intro
- ✓ Consistency Levels
- ✓ Gossip
- ➢ Consensus
- o Snapshot
- o Failure Detectors
- o Distributed Broadcast
- o Summary

Consensus

“In a fully asynchronous system there is no consensus solution that can tolerate one or more crash failures even when only requiring the non triviality property”

- Fischer Lynch Paterson [FLP] result

Real world applications of Consensus

- Distributed coordination services, i.e. log coordination
- Locking protocols
- State machine and primary-backup replication
- Distributed transaction resolution
- Agreement to move to the next stage of a distributed algorithm
- Leader election for higher-level protocols
- Many more

PAXOS

… a protocol for state-machine replication in an asynchronous environment that admits crash-failures

Production use of Paxos [edit]

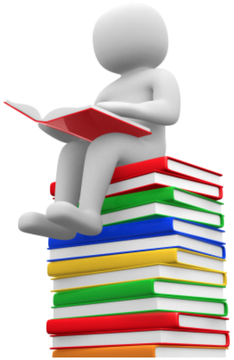
- The Petal project from DEC SRC was likely the first system to use Paxos, in this case for widely replicated global information (e.g., which machines are in the system).[20]
- Google uses the Paxos algorithm in their Chubby distributed lock service in order to keep replicas consistent in case of failure. Chubby is used by BigTable which is now in production in Google Analytics and other products.
- The Infinit peer-to-peer file system relies on Paxos to maintain consistency among replicas while allowing for quorums to evolve in size.
- Google Spanner and Megastore use the Paxos algorithm internally.
- The OpenReplica replication service uses Paxos to maintain replicas for an open access system that enables users to create fault-tolerant objects. It provides high performance through concurrent rounds and flexibility through dynamic membership changes.
- IBM supposedly uses the Paxos algorithm in their IBM SAN Volume Controller product to implement a general purpose fault-tolerant virtual machine used to run the configuration and control components of the storage virtualization services offered by the cluster.[*citation needed*]
- Microsoft uses Paxos in the Autopilot cluster management service from Bing.
- WANdisco have implemented Paxos within their DConE active-active replication technology.[21]
- XtreemFS uses a Paxos-based lease negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.[22]
- Heroku uses Doozerd which implements Paxos for its consistent distributed data store.
- Ceph uses Paxos as part of the monitor processes to agree which OSDs are up and in the cluster.
- The Clustrix distributed SQL database uses Paxos for distributed transaction resolution.
- Neo4j HA graph database implements Paxos, replacing Apache ZooKeeper from v1.9
- VMware NSX Controller uses Paxos-based algorithm within NSX Controller cluster.
- Amazon Web Services uses the Paxos algorithm extensively to power its platform.[23]
- Nutanix implements the Paxos algorithm in Cassandra for metadata.
- Apache Mesos uses Paxos algorithm for its replicated log coordination.
- Windows Fabric used by many of the Azure services make use of the paxos algorithm for replication between nodes in a cluster
- Oracle NoSQL Database leverages Paxos-based automated fail-over election process in the event of a master replica node failure to minimize downtime.[24]

Abstract

The Paxos algorithm, when presented in plain English, is very simple.

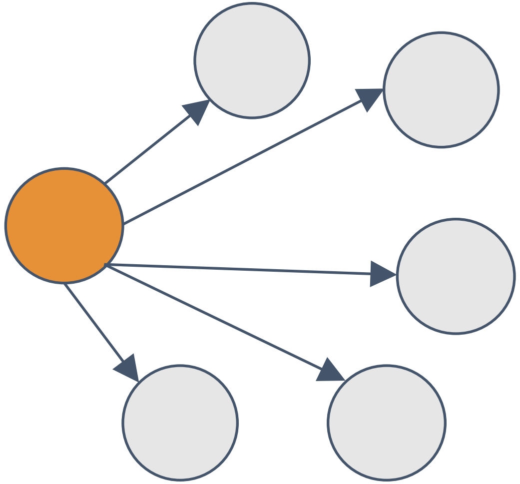
Proposers

Acceptors



Learners

“Prepare” request

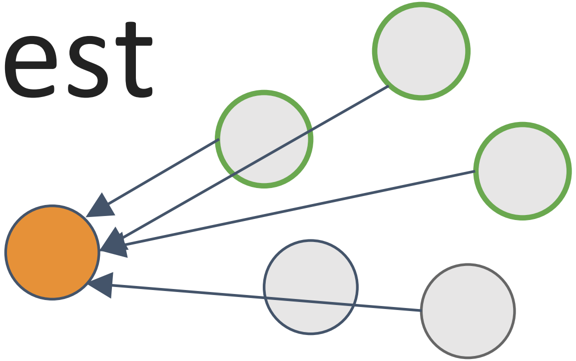


Proposer chooses a new proposal number **N**

Proposer asks acceptors to:

1. **Promise** to never accept a proposal # < **N**
2. Return a highest accepted proposal # < **N**

“Accept” request

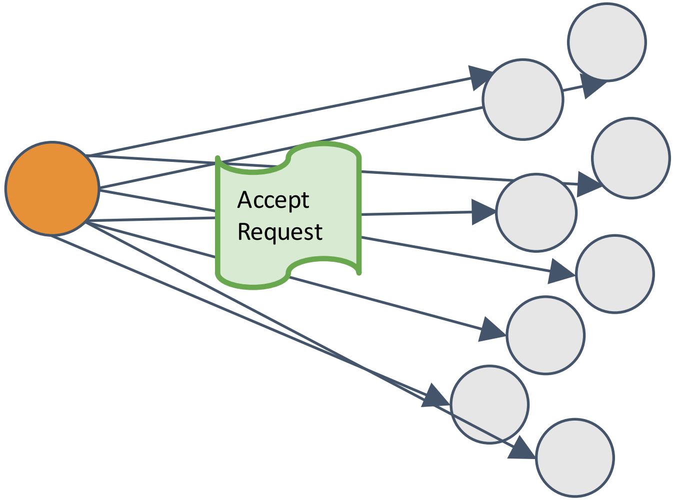


If majority of acceptors responded,

X = **value** **of the** **highest # proposal** **received from acceptors**

Proposer issues an accept request:

- With a **Proposal Number N**
- And a value **X**



Multi Paxos

Deciding on one value - running one round of Basic Paxos …
Deciding on a sequence of values - running Paxos many times

...Right?

Multi Paxos

Deciding on one value - running one round of Basic Paxos …
Deciding on a sequence of values - running Paxos many times

Almost.

We can optimize consecutive Paxos rounds by skipping prepare phase, assuming a stable leader

Table I. Types of Processes in Paxos

Process Type	Description	Minimum Number
Replica	Maintains application state Receives requests from clients Asks leaders to serialize the requests so all replicas see the same sequence Applies serialized requests to the application state Responds to clients	$f+1$
Leader	Receives requests from replicas Serializes requests and responds to replicas	$f+1$
Acceptor	Maintains the fault tolerant memory of Paxos	$2f+1$

Note: f is the number of failures tolerated.

Cheap Paxos

Based on Multi-Paxos

To tolerate **f** failures:

- **f + 1** acceptors (not **2f + 1**, like in traditional Paxos)
- Auxiliary acceptors in case of acceptor failures

Fast Paxos

Based on Multi-Paxos

Reduces end-to-end message delays for a replica to learn a chosen value

Needs **3f + 1** acceptors instead of **2f + 1**

Vertical Paxos

Reconfigurable

- Enables reconfiguration while state machine is deciding on commands
- Uses auxiliary master for reconfiguration ops
- A special case of Primary-Backup protocol

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [15, 16] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other con-

ZAB

Zookeeper's Atomic Broadcast

A bit stricter ordering guarantees…

If leader fails, new leader cannot arbitrarily reorder uncommitted state updates, or apply them starting from a different initial state

Contents

- ✓ Intro
- ✓ Consistency Levels
- ✓ Gossip
- ✓ Consensus
- ➢ Snapshot
- o Failure Detectors
- o Distributed Broadcast
- o Summary

Snapshots

Prototyping a Chandy-Lamport Snapshot in F#

```
member this.SendBasicMessage (amount: Contents) (delay: int) (node: ServerEndpoint) = async {

    let s = (!this.StateAtom)()
    printfn "Want to send %d, and my state is %d" amount s

    let! invariant = this.HasEnoughToTransfer amount
    if invariant then
        // Extracting the amount from current state.
        swap this.StateAtom
            (fun f ->
                (fun result () ->
                    if (result - amount >= 0) then result - amount else result
                ) <| f()
            ) |> ignore

        // Preparing a message with some amount to send.
        let messageToSend =
            {
                id = Guid.NewGuid().ToString()
                contents = amount
                address = ipAddress.ToString()
                port = port
                needAck = false
                delay = 0
            }
        printfn "Money sent %d, money left %d" amount ((!this.StateAtom)())

        // Sending a basic message to known endpoint.
        do! this.N.SendMessageToNeighbor node messageToSend
    else
        printfn "Not enough money to send %d: only %d left" amount s
}
```

```
member this.InitiateSnapshot () = async {
    // Capture own state into StateSnapshot
    // Send snapshot messages to all connected neighbors
    printfn "Initiating a snapshot!"
    if this.ShouldTakeASnapshot = Yes then
        do! this.PersistNodeState()
        do! this.SendSnapshotMessageToOutgoingChannels()
}
```

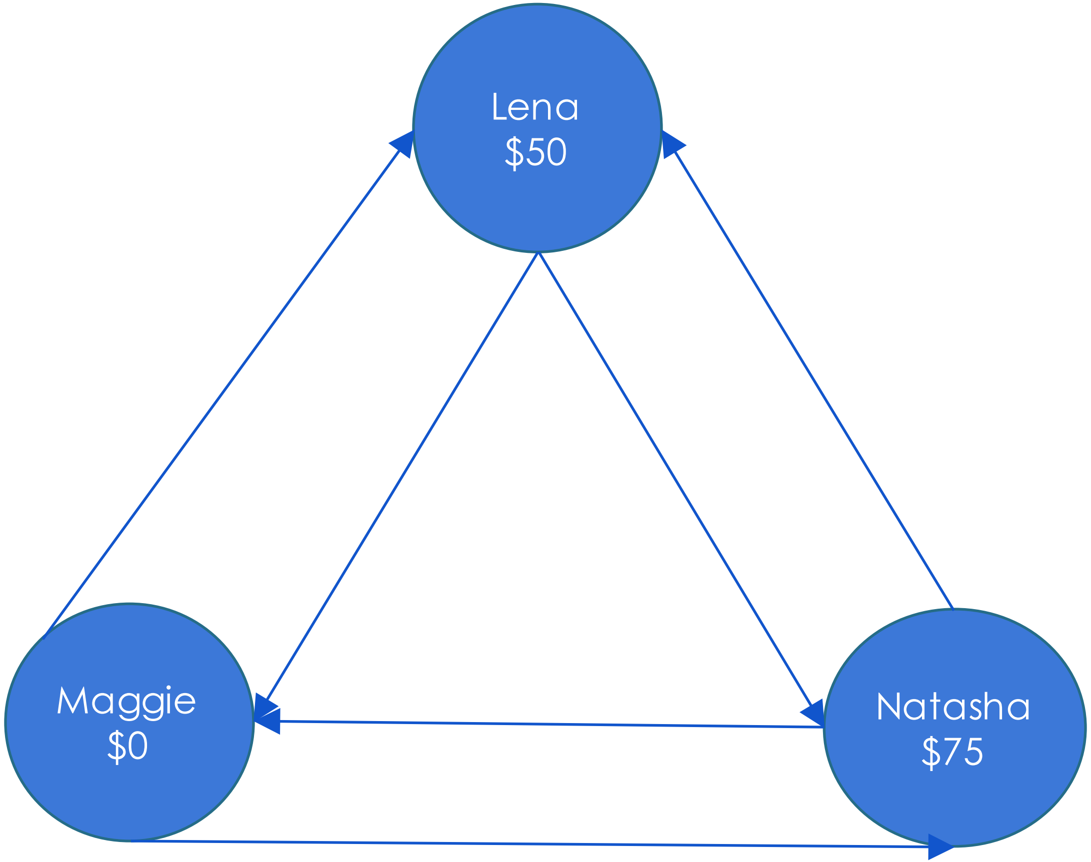
```
member this.SendSnapshotMessageToOutgoingChannels () = async {
    this.ConnectedNeighbors |> PSeq.iter (fun node ->
        async {
            printfn "Sending marker from %A:%A to %A:%A" ipAddress port node.Ip node.Port
            let snapshot =
                {
                    address = ipAddress.ToString()
                    port = port
                }
            do! this.N.SendMessageToNeighbor node snapshot
        } |> Async.Start)
}

member this.PersistNodeState () = async {
    this.ShouldTakeASnapshot <- AlreadyDone
    this.StateSnapshot <- (!this.StateAtom)()
    Console.ForegroundColor <- ConsoleColor.Blue
    printfn "*** Snapshot state of %A IS %A ***" id (this.StateSnapshot.ToString())
    Console.ForegroundColor <- ConsoleColor.White
}
```

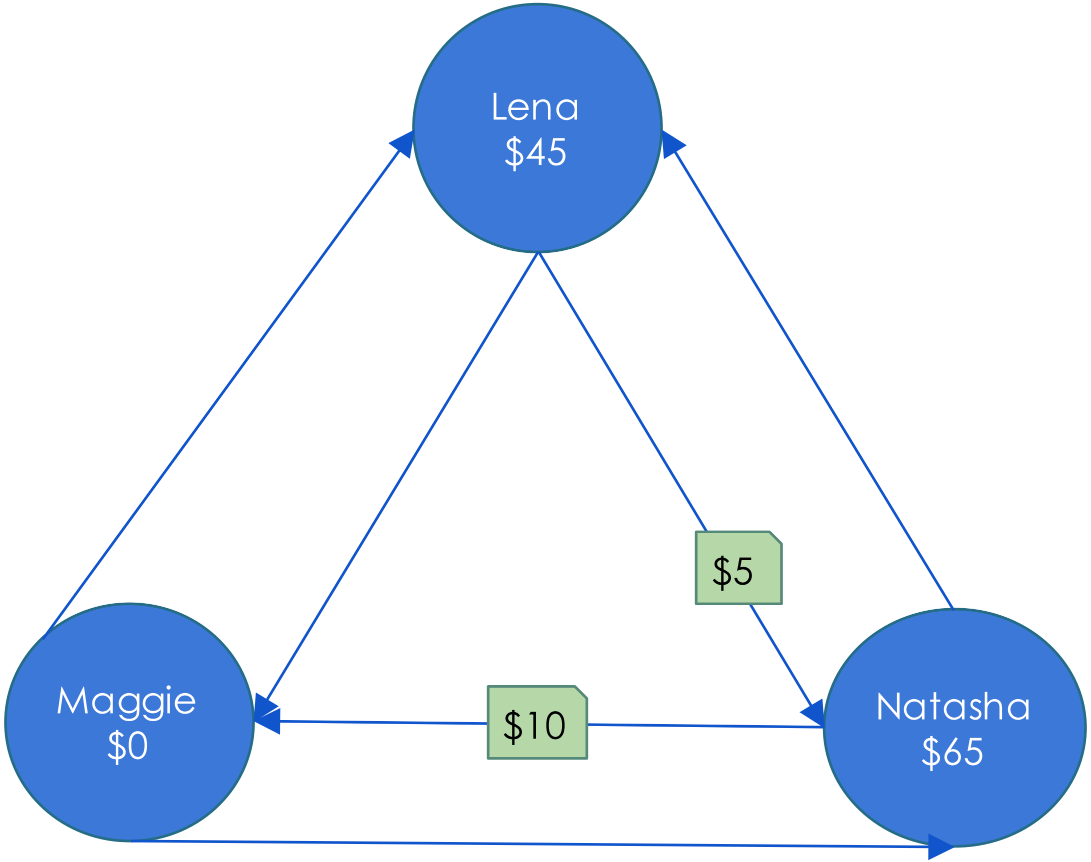
```
member this.ReceiveSnapshotMessage (s: SnapshotMessage) = async {
    let sender = { Ip = IPAddress.Parse s.address; Port = s.port }

    // Remember, that current node has already received a snapshot message from this sender.
    this.ReceivedMarkerFrom.[sender] <- true

    printf "Received a marker message from %s:%d" s.address s.port

    // In case current node hasn't taken a snapshot yet
    if this.ShouldTakeASnapshot = Yes then
        // Record it's own state.
        do! this.PersistNodeState()

        // Record the state of channel from sender to current node as {empty} set.
        do! this.PersistChannelState sender

        // And send snapshot messages to all outgoing channels.
        do! this.SendSnapshotMessageToOutgoingChannels()

    // In case current node has already recorded its local state by
    // receiving a snapshot message earlier from some other neighbor
    else
        // We need to record a state of incomming channel from sender to current node,
        // which is a set of messages received after current node recorded its state
        // and before current node received a marker from the sender.
        do! this.PersistChannelState sender
}
```

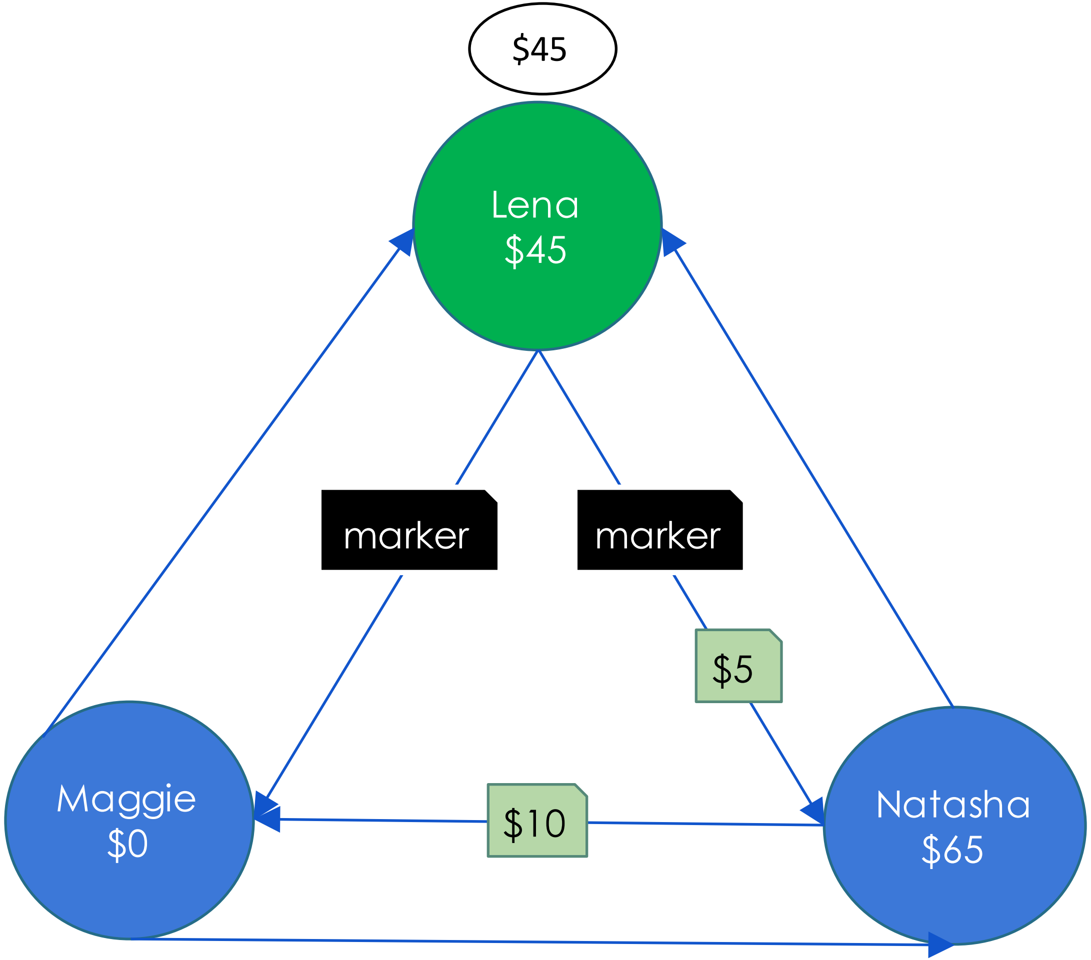
```
member this.ReceiveBasicMessage (m: Message) = async {
    let sender = { Ip = IPAddress.Parse m.address; Port = m.port }

    // 1. If current node has already taken its own snapshot, but didn't receive a marker from this sender yet:
    //    we need to record current message from this sender into messagesInTransit from sender to our node.
    printfn "Already received a marker from %A: %A!" sender (this.ReceivedMarkerFrom.ContainsKey(sender))
    if this.ShouldTakeASnapshot = AlreadyDone && not <| this.ReceivedMarkerFrom.ContainsKey(sender) then
        printfn "Adding message $%d from %s:%d to the channel state, because this node has already taken it's local snapsho
        this.ChannelStatesSnapshot.[sender].Add m
    else printfn "Received %d" m.contents

    // 2. If this node didn't take a snapshot yet:
    //    then it just processes a basic message and doesn't record it into snapshotted state
    //
    // 3. If this node has already taken its own snapshot, and already received a marker from sender:
    //    we should not record current message into messagesInTransit, because it's not a part of snapshotted state,
    //    so we can just do a normal processing on this basic message

    // Hence, message processing is necessary for all cases.
    do! this.ProcessBasicMessage m

    // That means, each node needs to store a list of nodes, who it has already received a marker from.
    // It can be a map with the key of sender endpoint and a value of messagesInTransit

}
```

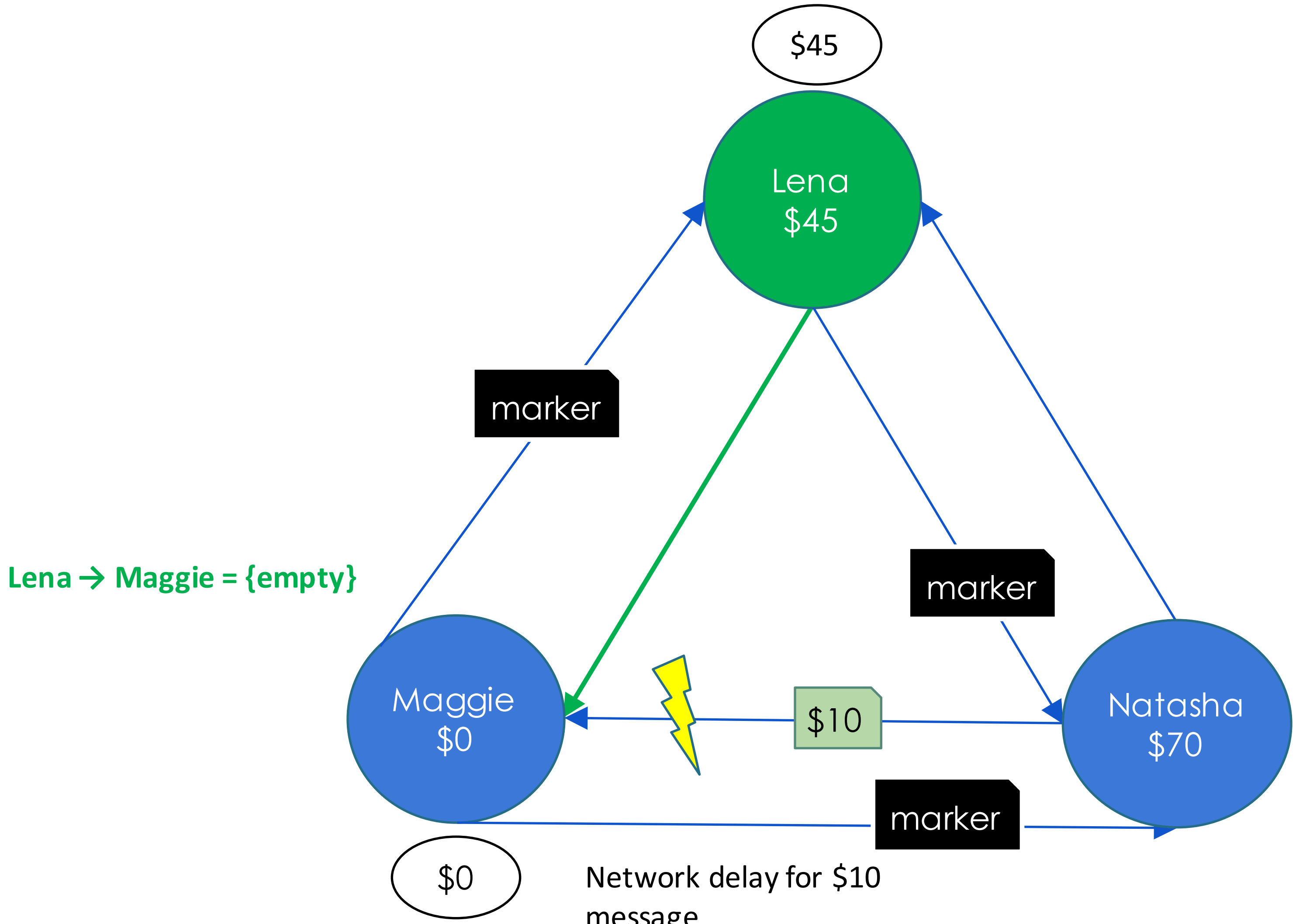
- Assumes that channels are FIFO
- All nodes can reach each other
- All nodes are available
- Basic Messages for regular data
- Marker Messages for snapshots
- Any node can initiate a snapshot at any moment

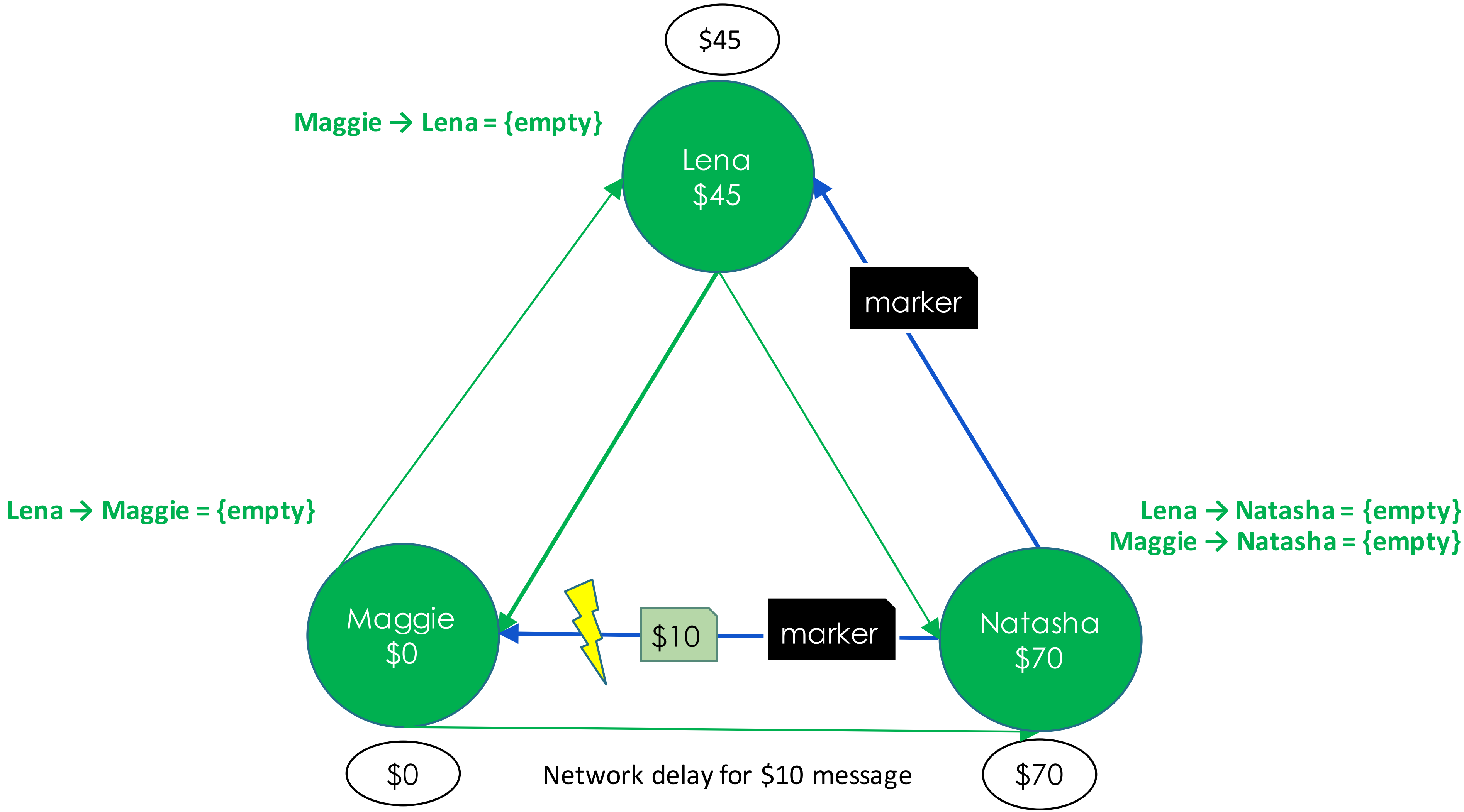
Initial state of the system



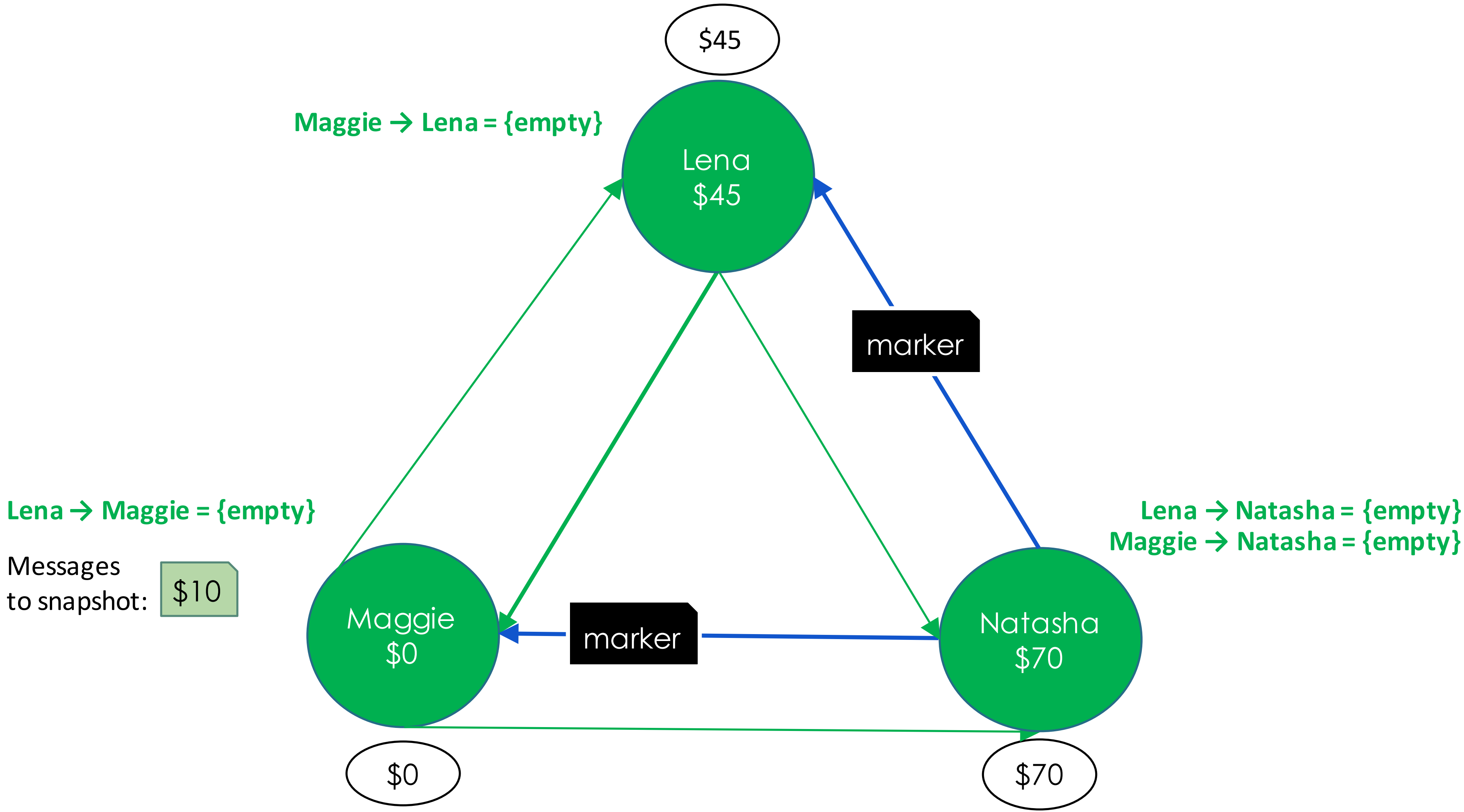

Lena
$45
Maggie
$0
Natasha
$65
$5
$10

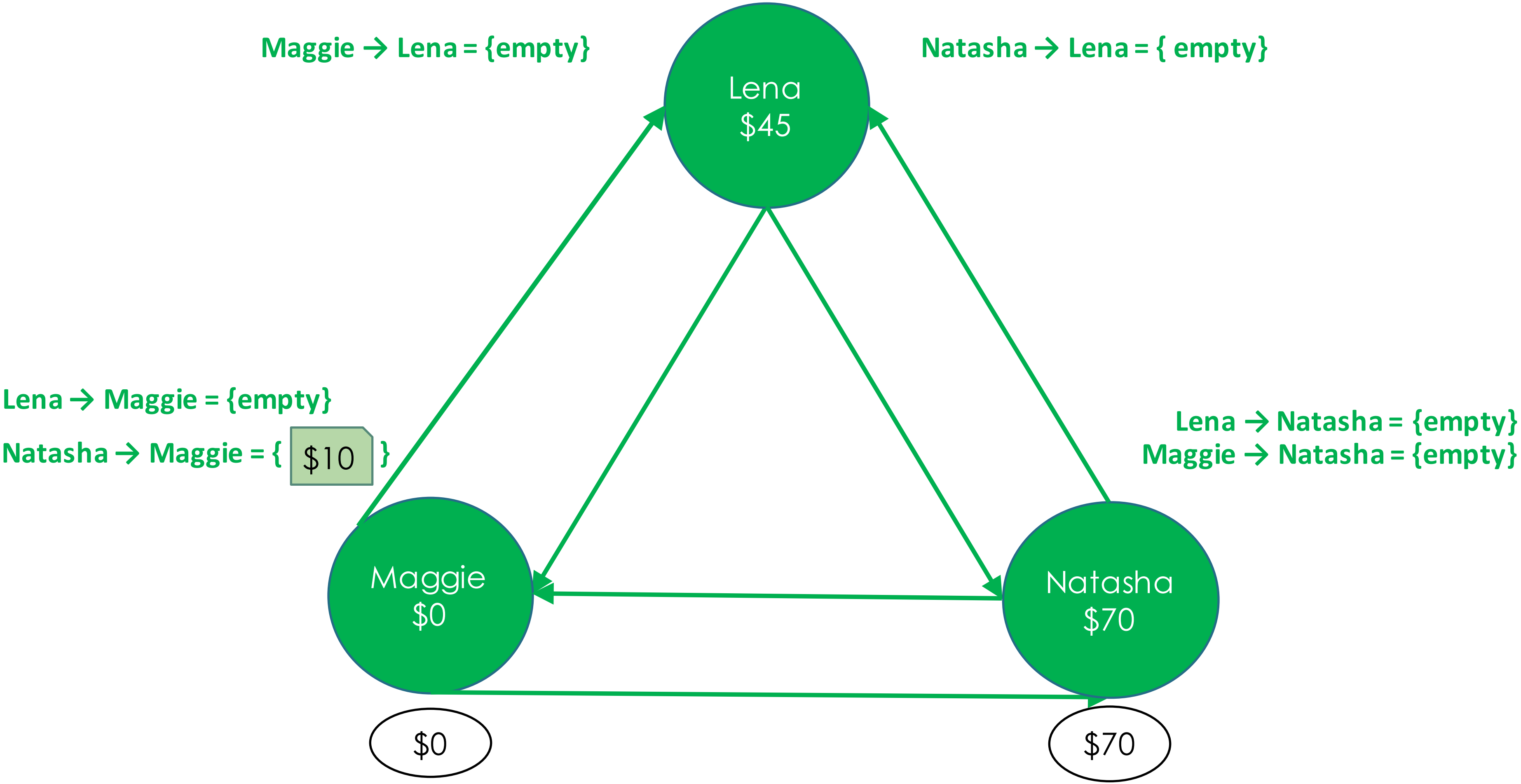

$45
Lena
$45
marker
marker
$5
Maggie
$0
$10
Natasha
$65


$45
Lena
$45
marker
Lena → Maggie = {empty}
marker
Maggie
$0
$10
Natasha
$70
marker
$0
Network delay for $10 message



$45
Maggie → Lena = {empty}
Lena
$45
marker
Lena → Maggie = {empty}
Lena → Natasha = {empty}
Maggie → Natasha = {empty}
Maggie
$0
$10
marker
Natasha
$70
$0
Network delay for $10 message
$70


$45
Maggie → Lena = {empty}
Lena
$45
marker
Lena → Maggie = {empty}
Lena → Natasha = {empty}
Maggie → Natasha = {empty}
Messages
to snapshot:
$10
Maggie
$0
marker
Natasha
$70
$0
$70


Maggie → Lena = {empty}
Natasha → Lena = { empty}
Lena
$45
Lena → Maggie = {empty}
Natasha → Maggie = { $10 }
Lena → Natasha = {empty}
Maggie → Natasha = {empty}
Maggie
$0
Natasha
$70
$0
$70

2. bash

```
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi SnapshotNode1.fsx
```

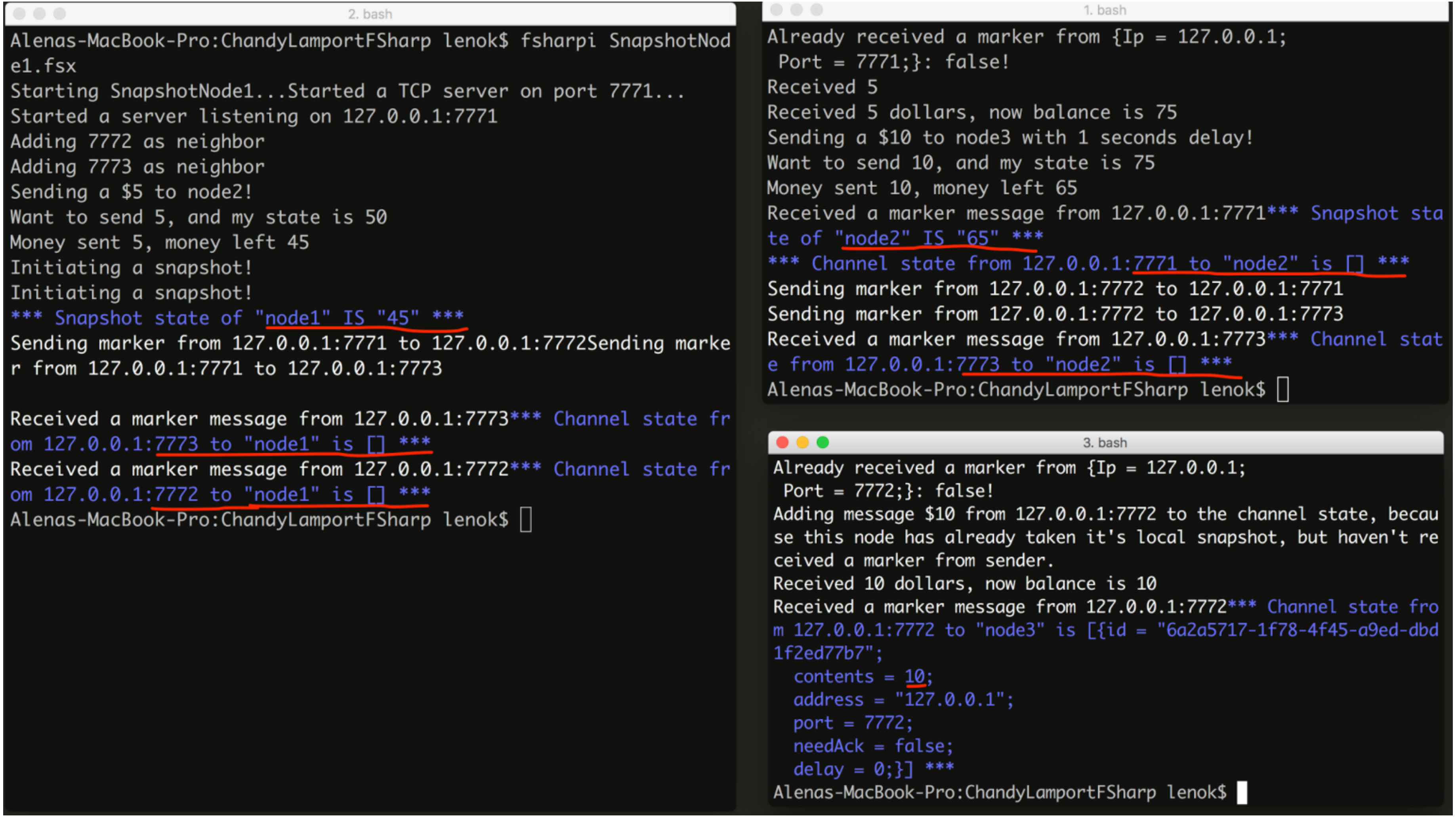
1. bash

```
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi SnapshotNode2.fsx
```

3. bash

```
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi SnapshotNode3.fsx
```

CODE



2. bash

```
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$ fsharpi SnapshotNod
e1.fsx
Starting SnapshotNode1...Started a TCP server on port 7771...
Started a server listening on 127.0.0.1:7771
Adding 7772 as neighbor
Adding 7773 as neighbor
Sending a $5 to node2!
Want to send 5, and my state is 50
Money sent 5, money left 45
Initiating a snapshot!
Initiating a snapshot!
*** Snapshot state of "node1" IS "45" ***
Sending marker from 127.0.0.1:7771 to 127.0.0.1:7772Sending marke
r from 127.0.0.1:7771 to 127.0.0.1:7773

Received a marker message from 127.0.0.1:7773*** Channel state fr
om 127.0.0.1:7773 to "node1" is [] ***
Received a marker message from 127.0.0.1:7772*** Channel state fr
om 127.0.0.1:7772 to "node1" is [] ***
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$
```

1. bash

```
Already received a marker from {Ip = 127.0.0.1;
 Port = 7771;}: false!
Received 5
Received 5 dollars, now balance is 75
Sending a $10 to node3 with 1 seconds delay!
Want to send 10, and my state is 75
Money sent 10, money left 65
Received a marker message from 127.0.0.1:7771*** Snapshot sta
te of "node2" IS "65" ***
*** Channel state from 127.0.0.1:7771 to "node2" is [] ***
Sending marker from 127.0.0.1:7772 to 127.0.0.1:7771
Sending marker from 127.0.0.1:7772 to 127.0.0.1:7773
Received a marker message from 127.0.0.1:7773*** Channel stat
e from 127.0.0.1:7773 to "node2" is [] ***
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$
```

3. bash

```
Already received a marker from {Ip = 127.0.0.1;
 Port = 7772;}: false!
Adding message $10 from 127.0.0.1:7772 to the channel state, becau
se this node has already taken it's local snapshot, but haven't re
ceived a marker from sender.
Received 10 dollars, now balance is 10
Received a marker message from 127.0.0.1:7772*** Channel state fro
m 127.0.0.1:7772 to "node3" is [{id = "6a2a5717-1f78-4f45-a9ed-dbd
1f2ed77b7";
  contents = 10;
  address = "127.0.0.1";
  port = 7772;
  needAck = false;
  delay = 0;}] ***
Alenas-MacBook-Pro:ChandyLamportFSharp lenok$
```

Contents

- ✓ Intro
- ✓ Consistency Levels
- ✓ Gossip
- ✓ Consensus
- ✓ Snapshot
- ➢ Failure Detectors
- o Distributed Broadcast
- o Summary

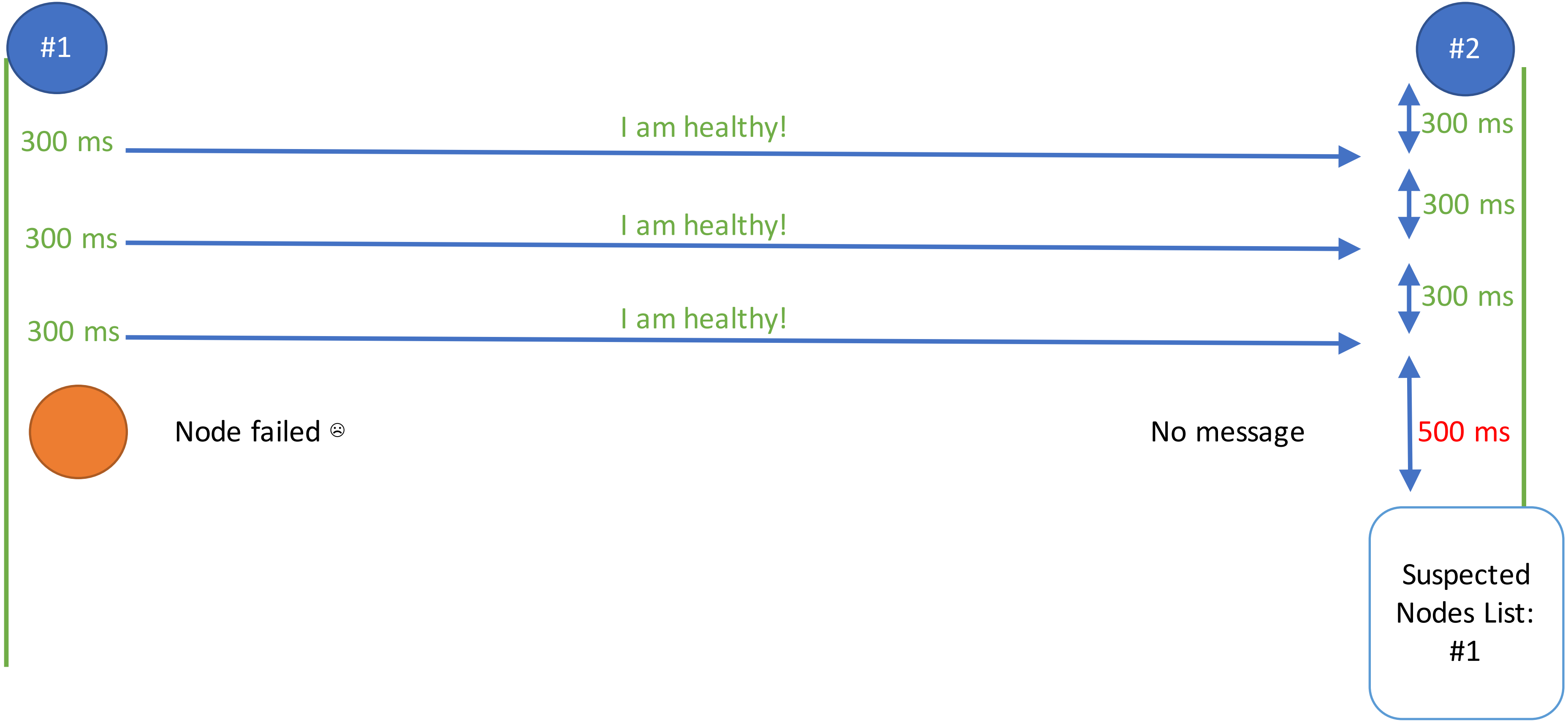
Failure detectors

Failure detectors are critical for:

- Leader election
- Agreement (e.g. Paxos)
- Reliable Broadcast
- And more

How are faults detected?

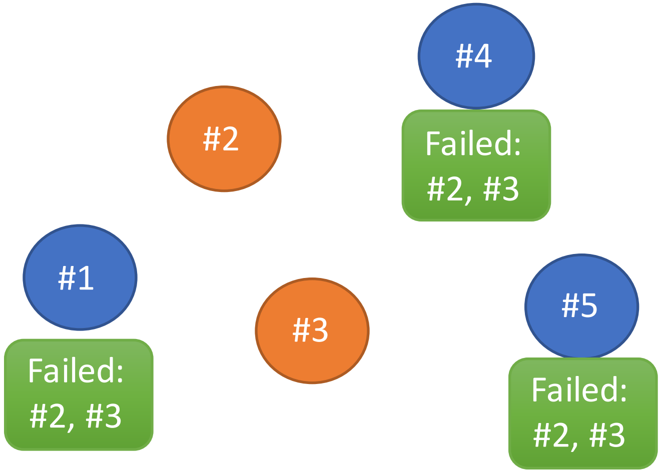
Heartbeat failure detection



Completeness

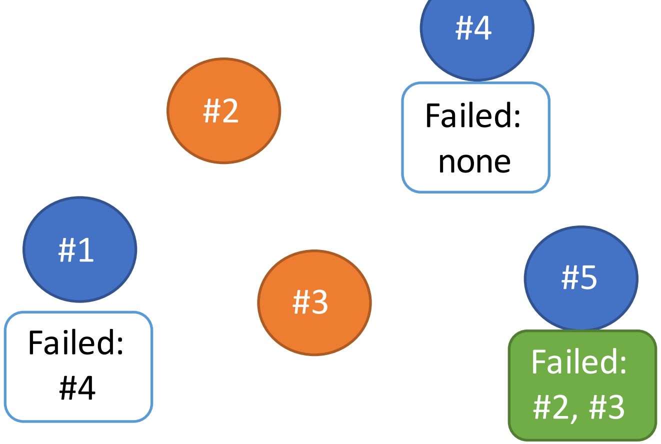
Strong

Eventually all failed processes are suspected by all correct once.

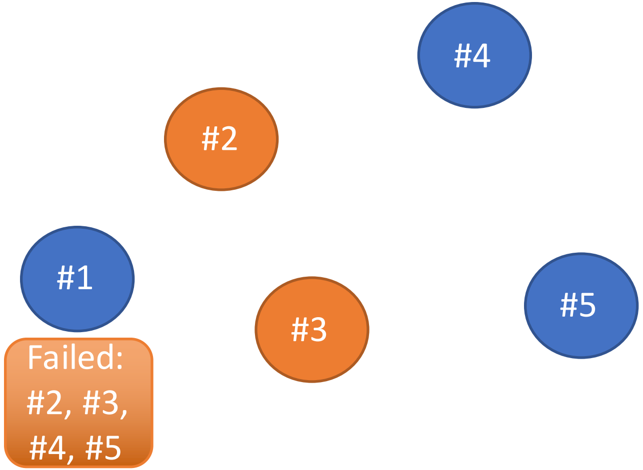


Weak

Eventually all failed processes are suspected by at least one correct one.



Detects every process as failed



Accuracy

Strong | **Weak** | **Eventually strong** | **Eventually weak**

We can say that accuracy…

Restricts the mistake or

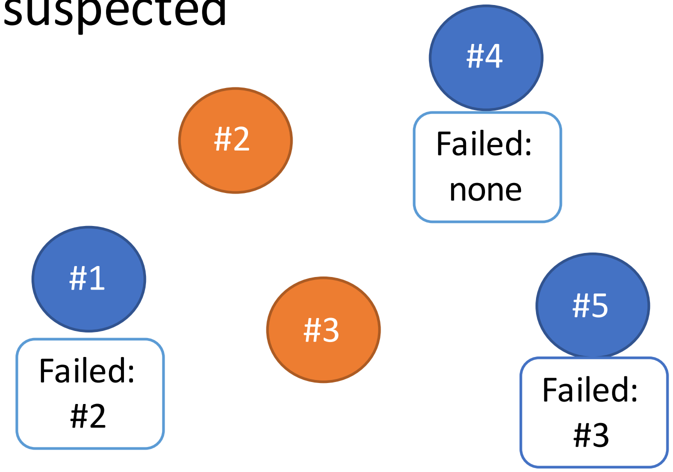
Limits number of false positives or, more precisely

Limits number of nodes that are mistakenly suspected by the correct nodes

Accuracy

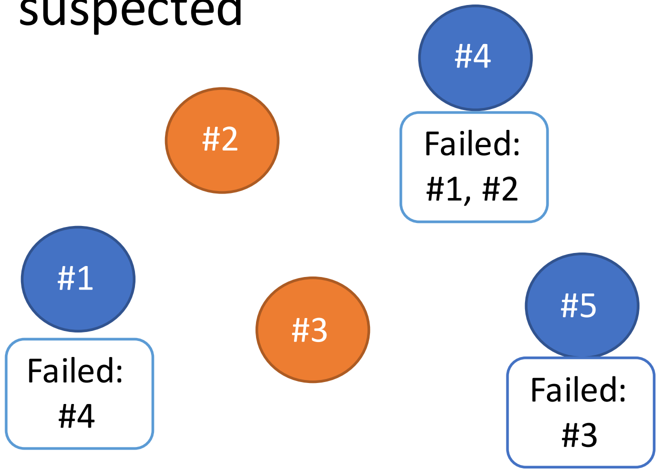
Strong

All correct processes are never suspected



Weak

At least 1 correct process is never suspected

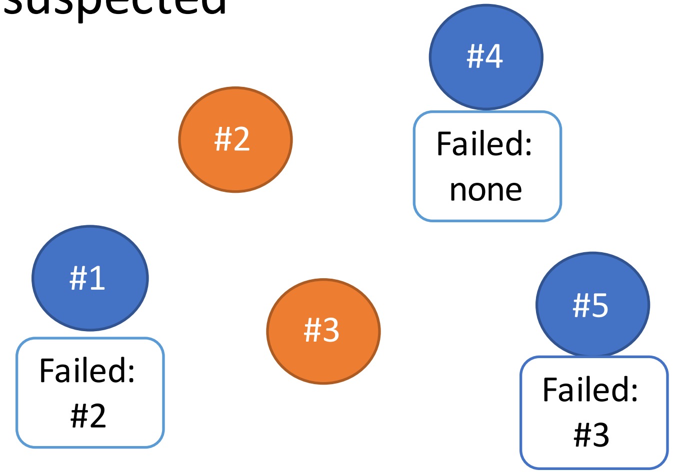


* Correct node #5 isn't in any suspected list ever

Accuracy

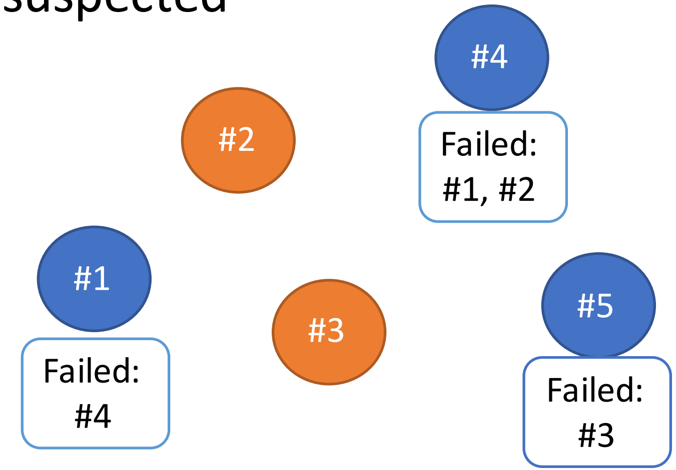
Eventual Strong

After some point in time,
all correct processes are never suspected



Eventual Weak

After some point in time,
at least 1 correct process is never suspected



* Nodes could have had wrong suspicions in the past

Failure detector classes

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

FIG. 1. Eight classes of failure detectors defined in terms of accuracy and completeness.

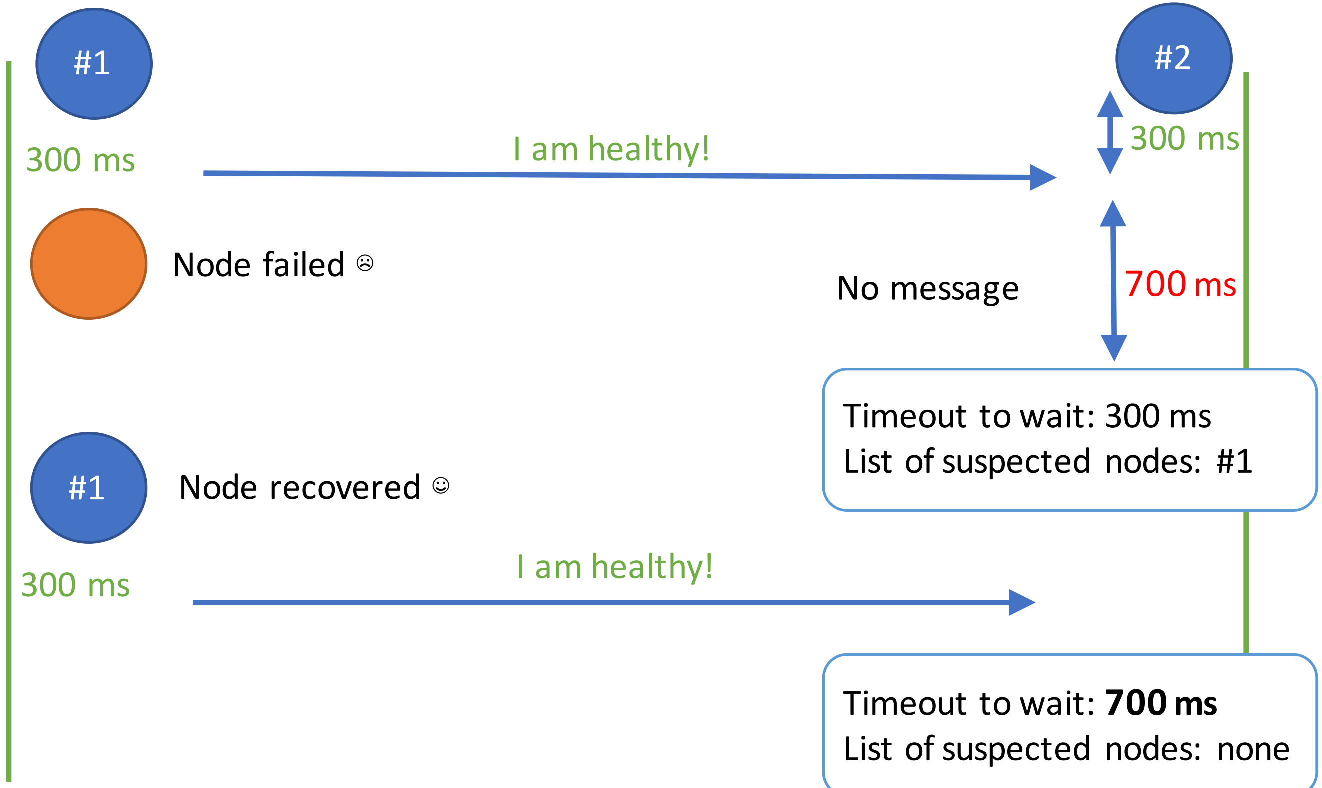
Perfect Failure Detector

- Strong Completeness + Strong Accuracy
- Possible only in synchronous system
- Uses timeouts (waiting time for a heartbeat arrival)

Eventually Perfect Failure Detector

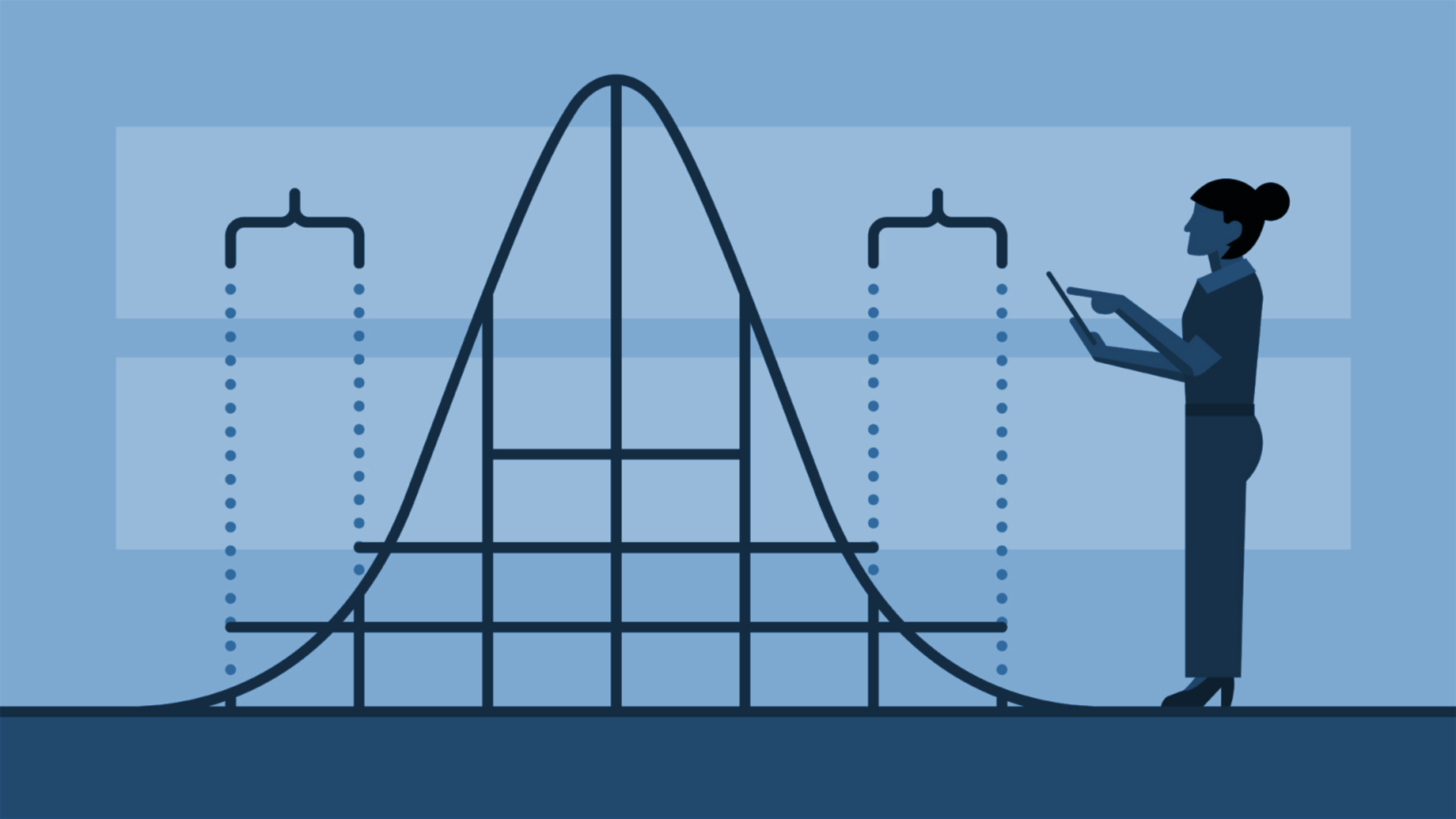
- Strong Completeness + Eventual Strong Accuracy
- Timeouts are adjusted to be maximum or average
- Eventually timeout will be that big that all assumptions will be correct

Eventually Perfect Failure Detector



Failure Detectors in Consensus

Failure detectors are present in almost every distributed system



SWIM

Scalable Weakly-consistent Infection-style process group Membership protocol

Strong completeness and configurable accuracy

1. Initiator picks random node from its membership list and sends a message to it.
2. If a chosen node doesn't send acknowledgement after some time, initiator sends ping request to other **M** randomly selected nodes to ask to send message to suspicious node.
3. All nodes try to get acknowledgement from suspicious node and forward it to the initiator. If none of the **M** nodes gets an acknowledgement, initiator marks the node as dead and disseminates about failed node.

Choosing bigger number **M** of nodes you make accuracy bigger.

Contents

- ✓ Intro
- ✓ Consistency Levels
- ✓ Gossip
- ✓ Consensus
- ✓ Snapshot
- ✓ Failure Detectors
- ➢ Distributed Broadcast
- ○ Summary

Distributed Broadcast

Sending message to every node in the cluster

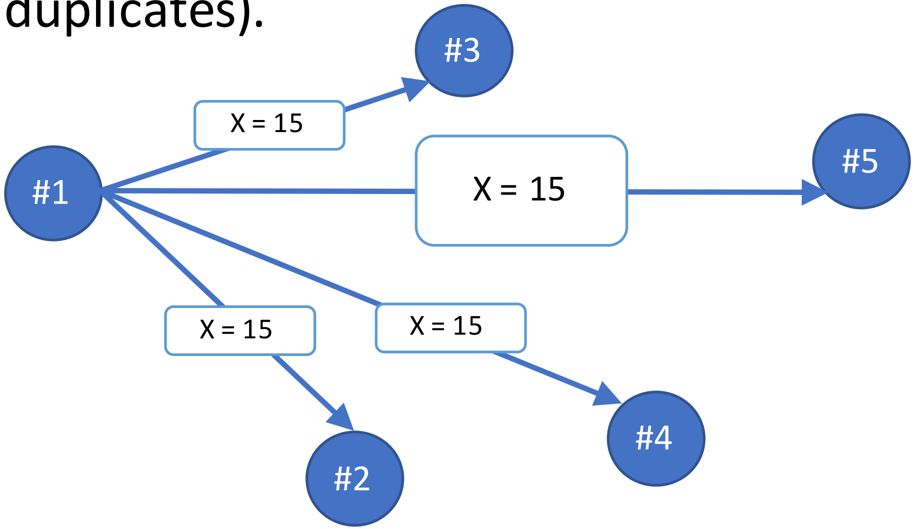
Messages can be lost

Nodes can fail

Various broadcast algorithms help us to choose between consistency of the system and its availability.

Best-effort Broadcast

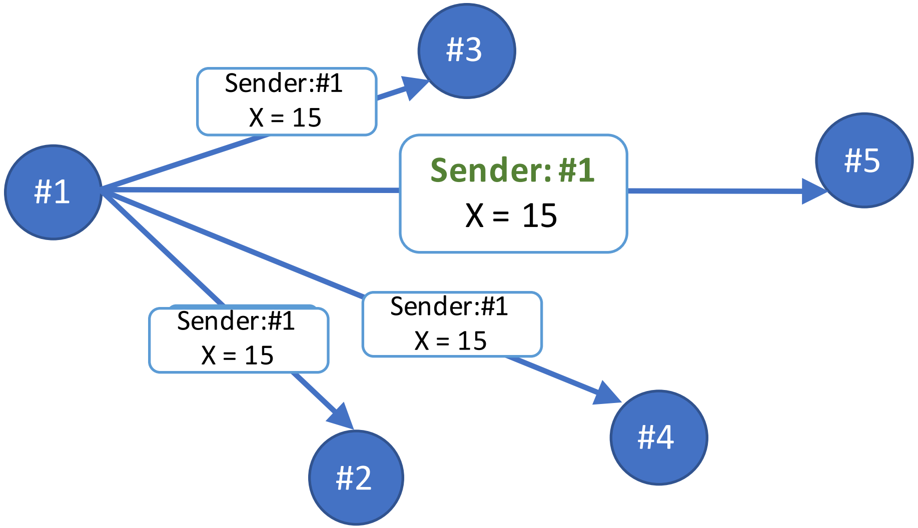
- Ensures the delivery of the message to all of the correct processes if sender doesn't fail during message exchange.
- Messages are sent using perfect point-to-point connection (no message loss, no duplicates).



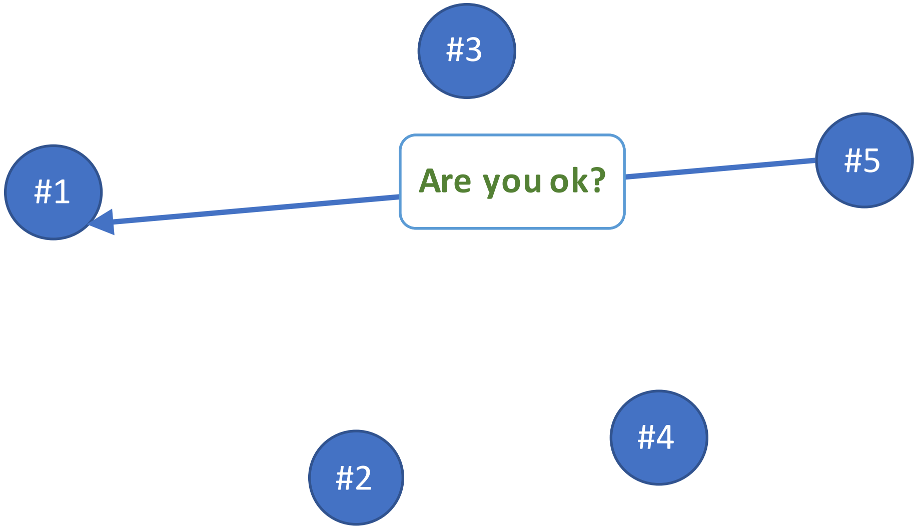
Reliable Broadcast

- If **any correct node** delivers the message, **all correct nodes** should deliver the message
- Normal scenario results in O(N) messages
- Worst-case scenario (when processes fail one after another) — O(N) steps with $O(N^2)$ messages.
- Uses failure detector

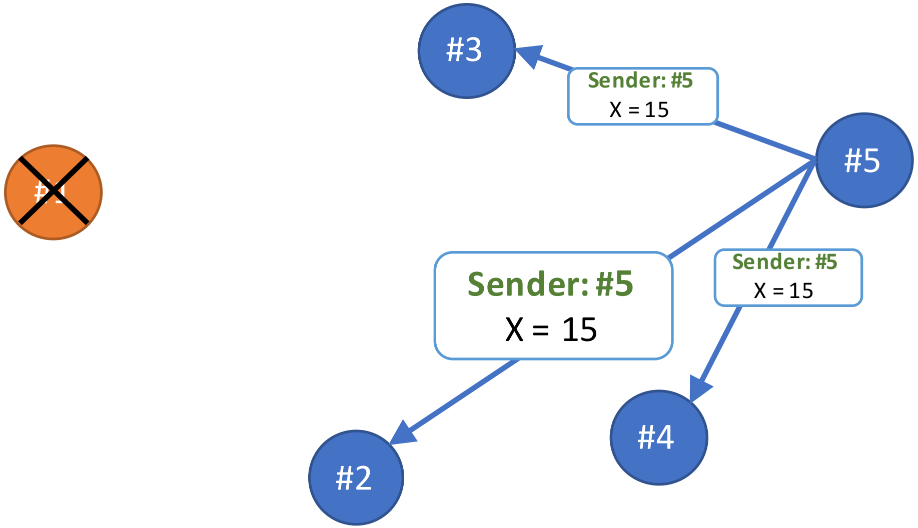
Message has sender ID



Nodes constantly check sender



If failure detected – message is relayed



They can even fail one by one, other nodes will detect failure and relay message

If failure detector marks sender failed when it’s not, unnecessary messages would be sent, that would impact performance, not correctness.

However, if process will not indicate failure, then needed messages won’t be sent.

Some reliable broadcasts **do not** use failure detectors…

What if a node processes a message, and then fails?...

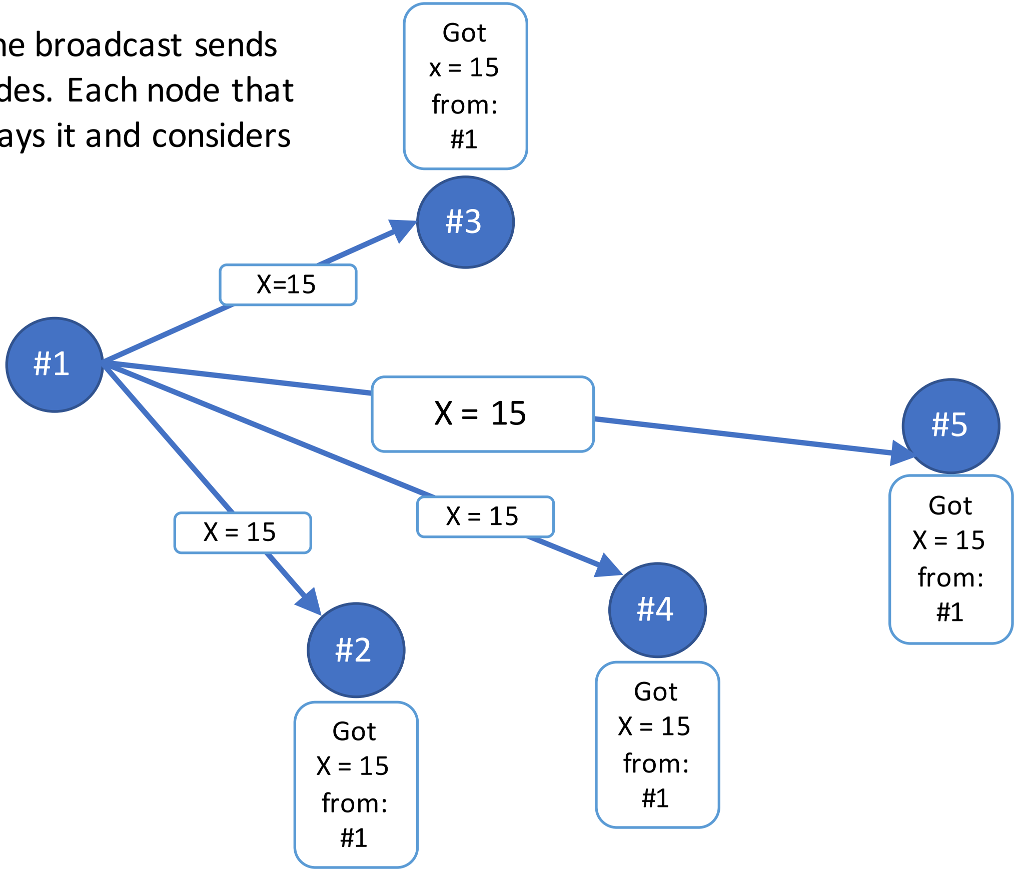
Uniform Reliable Broadcast

If **any node** (**doesn't matter correct or crashed**) delivers the message, **all correct nodes** should deliver the message.

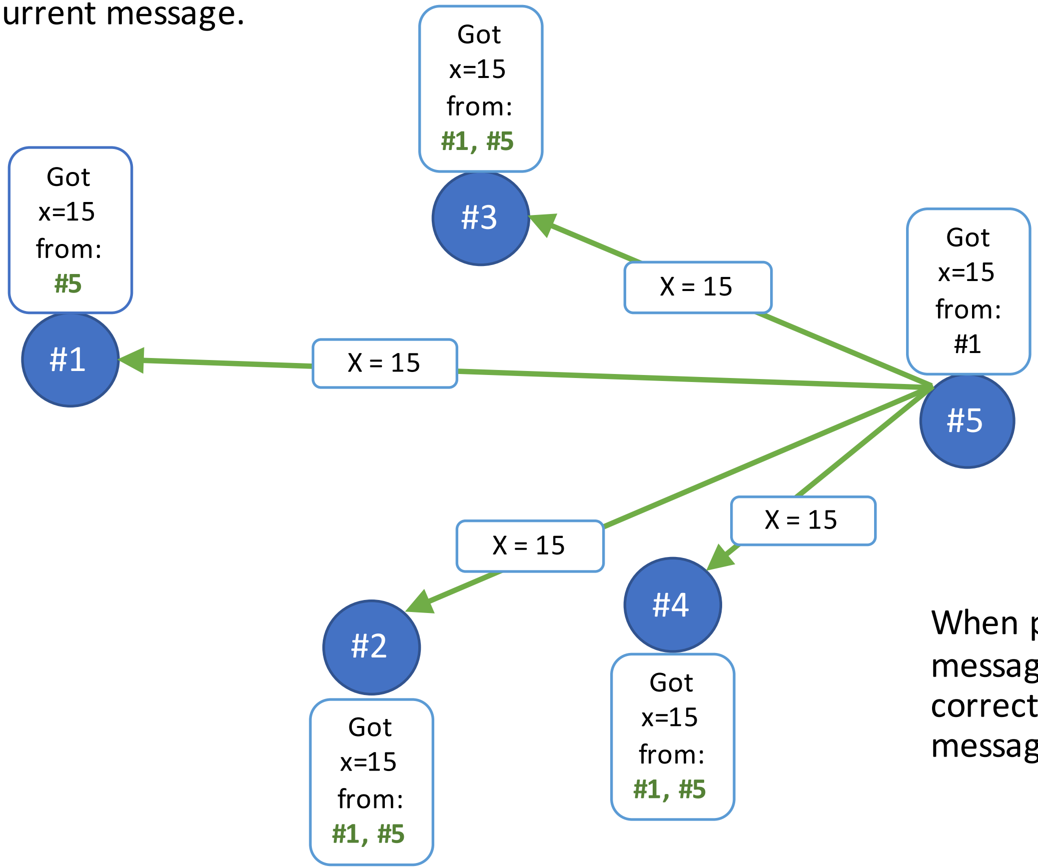
Works with failure detector, this way nodes don't wait for the reply from failed nodes forever.

All-Ack Algorithm

Node that initiated the broadcast sends messages to all of nodes. Each node that gets the message, relays it and considers pending for now.



Each node keeps track of nodes from which it received current message.



When process gets current message from all of the correct nodes, it considers message delivered.

Contents

- ✓ Intro
- ✓ Consistency Levels
- ✓ Gossip
- ✓ Consensus
- ✓ Snapshot
- ✓ Failure Detectors
- ✓ Distributed Broadcast
- ➢ Summary

Now I know distributed algorithms

Thank you and reach out!

Natallia Dzenisenka @nata_dzen

Alena Hall @lenadroid