The Raft Protocol Distributed Consensus for Dummies

Arnaud Bailly <abailly@murex.com> @abailly

2014-06

Who am I?

- Writing code since 1986

Who am I?

- Writing code since 1986
- Developping software since 1994

Who am I?

- Writing code since 1986
- Developping software since 1994
- Lead developer, Java/XP consultant at **Murex** since 2009

Who am I?

- Writing code since 1986
- Developping software since 1994
- Lead developer, Java/XP consultant at **Murex** since 2009
- Fascinated with distributed computing since ...

Who am I?

- Writing code since 1986
- Developping software since 1994
- Lead developer, Java/XP consultant at **Murex** since 2009
- Fascinated with distributed computing since ...
- By the way, **Murex is hiring!**

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*
- Systems may fail, and large systems may fail more often

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*
- Systems may fail, and large systems may fail more often
  - **fault-tolerance**

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*
- Systems may fail, and large systems may fail more often
  - **fault-tolerance**
- Yet we need to provide fast service reliably

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*
- Systems may fail, and large systems may fail more often
  - **fault-tolerance**
- Yet we need to provide fast service reliably
  - **high-availabilty**

Why Should I Care about Distributed Consensus?

- Real world is distributed (multicore chips, WWW)
  - Today's applications need to take care of **distribution**: *abstractions leak!*
- Systems may fail, and large systems may fail more often
  - **fault-tolerance**
- Yet we need to provide fast service reliably
  - **high-availabilty**
- **Consensus** is a basic building block for all kind of distributed systems features

Use Case: PaaS Configuration

- `etcd` is part of CoreOS, a linux distribution for clusters

Use Case: PaaS Configuration

- `etcd` is part of CoreOS, a linux distribution for clusters
- Provide consistent configuration for all *docker* containers hosted on CoreOS

Use Case: PaaS Configuration

- `etcd` is part of CoreOS, a linux distribution for clusters
- Provide consistent configuration for all *docker* containers hosted on CoreOS
- Uses on *Raft Distributed Consensus* implemented in Go

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

1. A room registration service instance starts

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

1. A room registration service instance starts
2. It registers itself as an *ephemeral node* in ZK

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

1. A room registration service instance starts
2. It registers itself as an *ephemeral node* in ZK
3. This triggers reconfiguration of HAProxy to this service in the cluster

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

1. A room registration service instance starts
2. It registers itself as an *ephemeral node* in ZK
3. This triggers reconfiguration of HAProxy to this service in the cluster
4. The service then can address other services using "dynamic" *HAProxy*-ed address

Use Case: Service Discovery

- Apache's `ZooKeeper` provides distributed consistent hierarchical key-value store
- AirBnB uses ZK to provide service discovery in their SmartStack solution
- Example scenario:

1. A room registration service instance starts
2. It registers itself as an *ephemeral node* in ZK
3. This triggers reconfiguration of HAProxy to this service in the cluster
4. The service then can address other services using "dynamic" *HAProxy*-ed address

- zab ensures distributed consensus across ZK nodes

Distributed Consensus is A Very Old Problem...

... And it is Hard

1. Horses and messengers can get killed...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...
4. Enemy can setup ambushes...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...
4. Enemy can setup ambushes...
5. Army corps can move...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...
4. Enemy can setup ambushes...
5. Army corps can move...
6. Nobody knows everything...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...
4. Enemy can setup ambushes...
5. Army corps can move...
6. Nobody knows everything...
7. You need to feed horses...

... And it is Hard

1. Horses and messengers can get killed...
2. Horses can travel only so fast...
3. You can send only so many horses at once...
4. Enemy can setup ambushes...
5. Army corps can move...
6. Nobody knows everything...
7. You need to feed horses...
8. Not all horses are created equal.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.

... Even in Distributed Computing

The 8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Fundamental Impossibility

Figure : The Fischer-Lynch-Paterson Theorem (aka. FLP)

In an Asynchronous Network...

> *It is not possible to reach distributed consensus with arbitrary communication failures*
> Distributed Algorithms, *Nancy Lynch, 1997, Morkan-Kaufmann*

In a Partially Synchronous Network...

> *It is possible to reach consensus assuming* f *processes fail and there is an upper bound* d *on delivery time for all messages, provided the number of processes is greater than* 2f
> *Nancy Lynch, op.cit.*

In Practice

Paxos

- Renowned consensus algorithm invented by *Leslie Lamport*

Paxos

- Renowned consensus algorithm invented by *Leslie Lamport*
- Provides foundations for several implementations: ZooKeeper (kinda...), Chubby

Paxos

- Renowned consensus algorithm invented by *Leslie Lamport*
- Provides foundations for several implementations: ZooKeeper (kinda...), Chubby
- But it is hard to implement correctly:

Paxos

- Renowned consensus algorithm invented by *Leslie Lamport*
- Provides foundations for several implementations: ZooKeeper (kinda...), Chubby
- But it is hard to implement correctly:

> *While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. Converting the algorithm into a practical system involved implementing many features some published in the literature and some not.*

Paxos

- Renowned consensus algorithm invented by *Leslie Lamport*
- Provides foundations for several implementations: ZooKeeper (kinda...), Chubby
- But it is hard to implement correctly:

> *While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. Converting the algorithm into a practical system involved implementing many features some published in the literature and some not.*
>
> *Paxos Made Live - An Engineering Perspective, T.Chandra et al.*

Raft

- In Search of an Understandable Consensus Algorithm, D.Ongaro and J.Osterhout, 2013

Raft

- In Search of an Understandable Consensus Algorithm, D.Ongaro and J.Osterhout, 2013
- Novel algorithm designed with *understandability* in mind

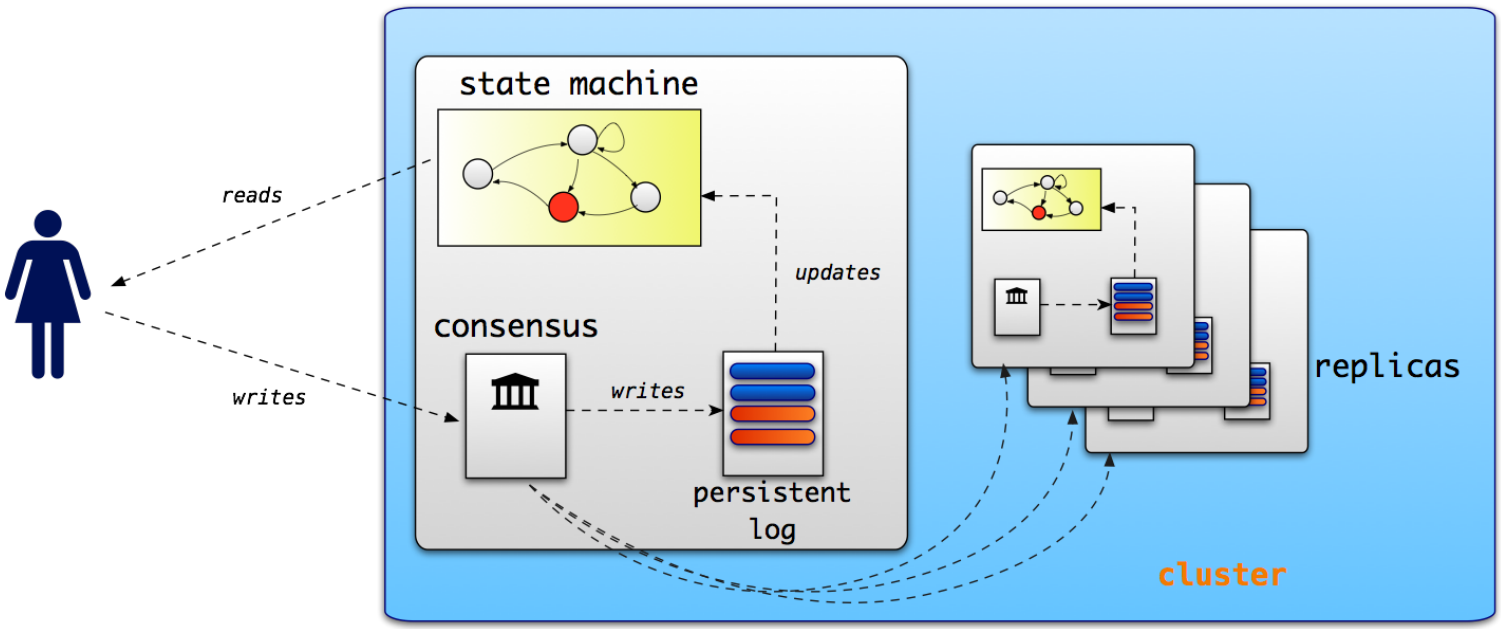
Raft

- In Search of an Understandable Consensus Algorithm, D.Ongaro and J.Osterhout, 2013
- Novel algorithm designed with *understandability* in mind
- Dozens of implementations in various language

Raft

- In Search of an Understandable Consensus Algorithm, D.Ongaro and J.Osterhout, 2013
- Novel algorithm designed with *understandability* in mind
- Dozens of implementations in various language
- Most prominent use is currently Go version for etcd distributed configuration system in CoreOS

Principle: Replicated State Machine With Persistent Log



Principles of Operation

- *Leader-follower* based algorithm: Leader is the single entry point for all operations on the cluster

Principles of Operation

- *Leader-follower* based algorithm: Leader is the single entry point for all operations on the cluster
- Each instance is a Replicated state machine whose state is uniquely determined by a linear *persistent log*

Principles of Operation

- *Leader-follower* based algorithm: Leader is the single entry point for all operations on the cluster
- Each instance is a Replicated state machine whose state is uniquely determined by a linear *persistent log*
- Leader orchestrates *safe log replication* to its *followers*

Raft Algorithm

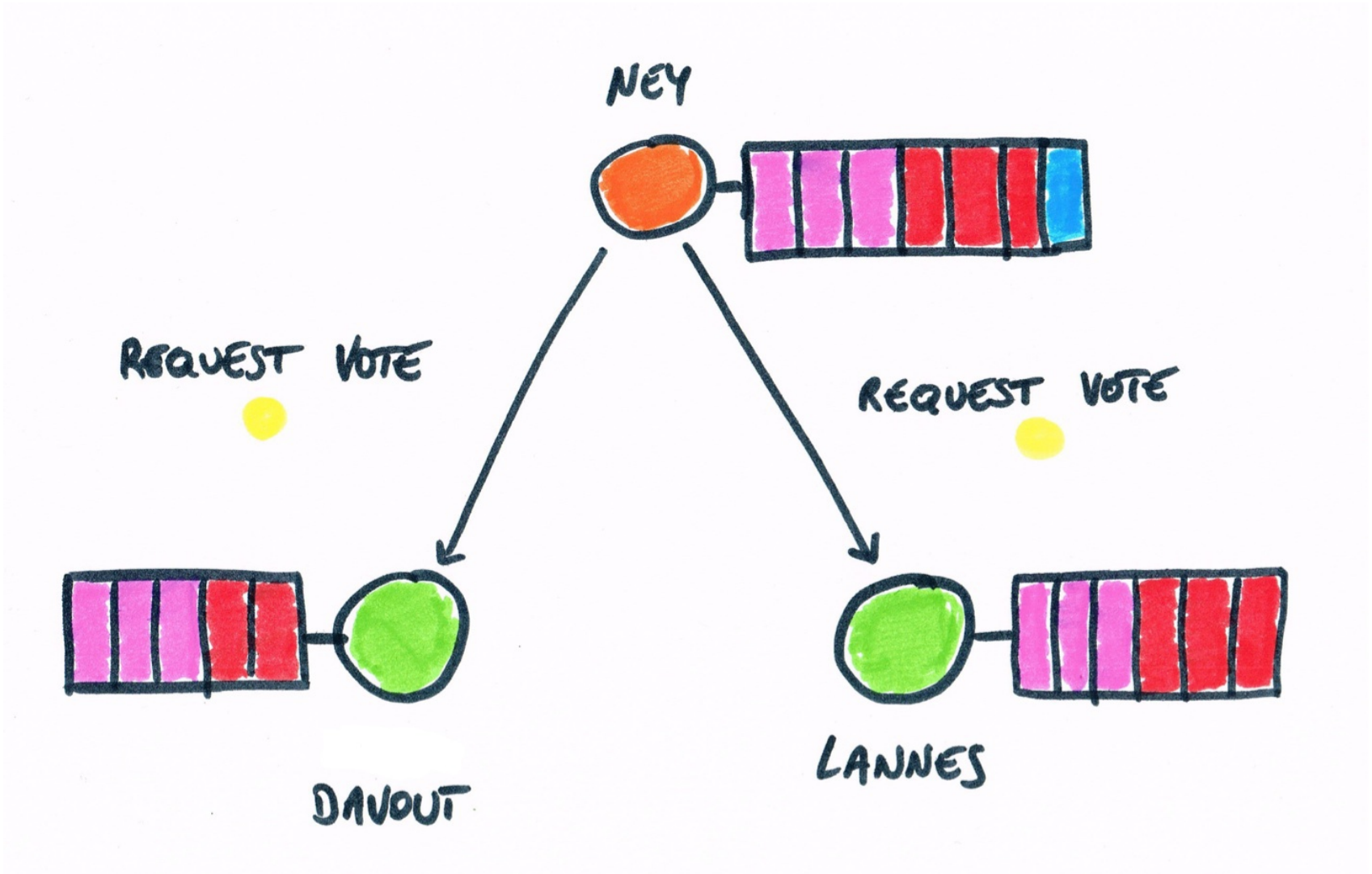


Figure : Ney requests being appointed leader

Raft Algorithm

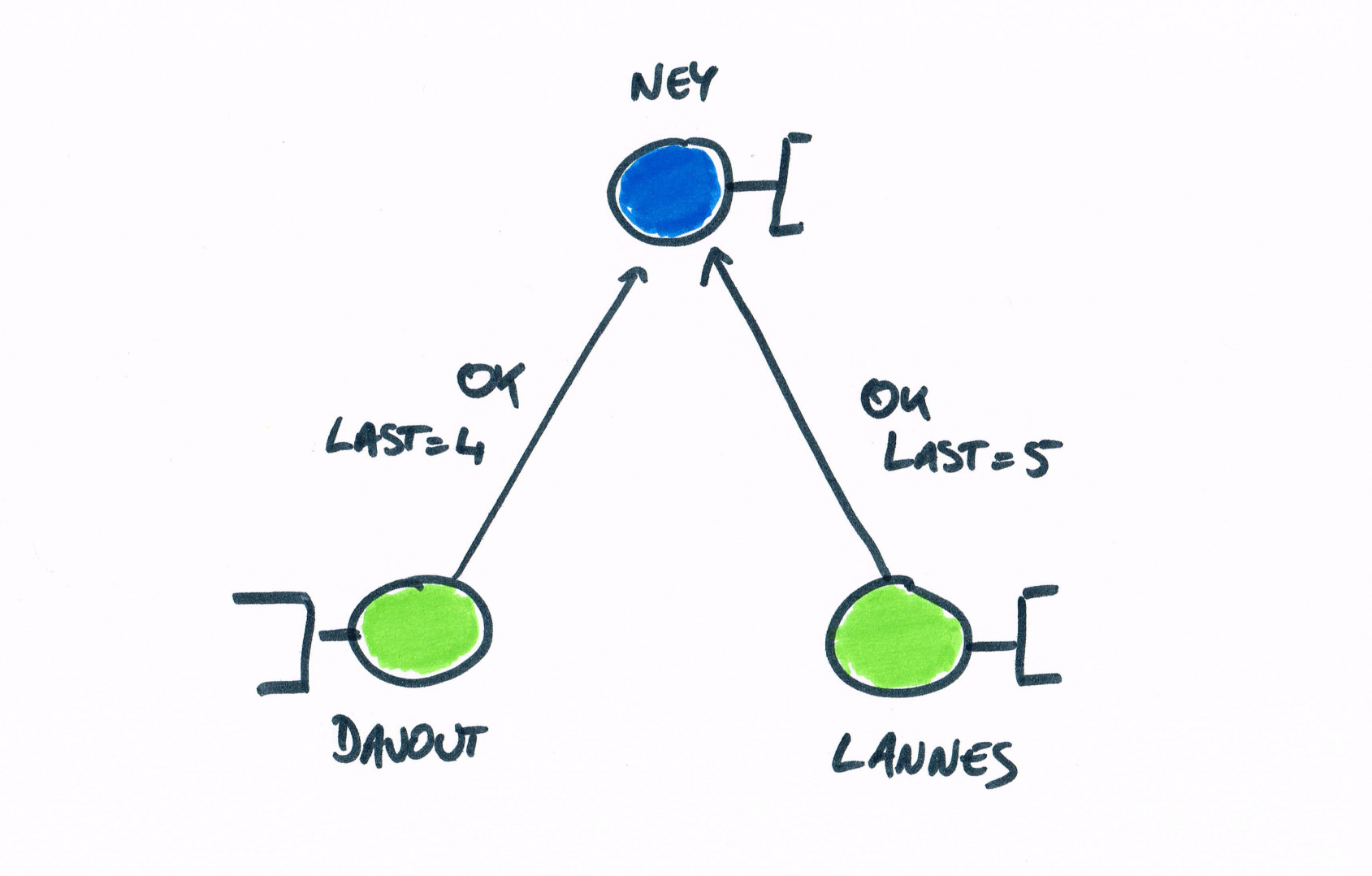


Figure : Ney becomes leader

Raft Algorithm

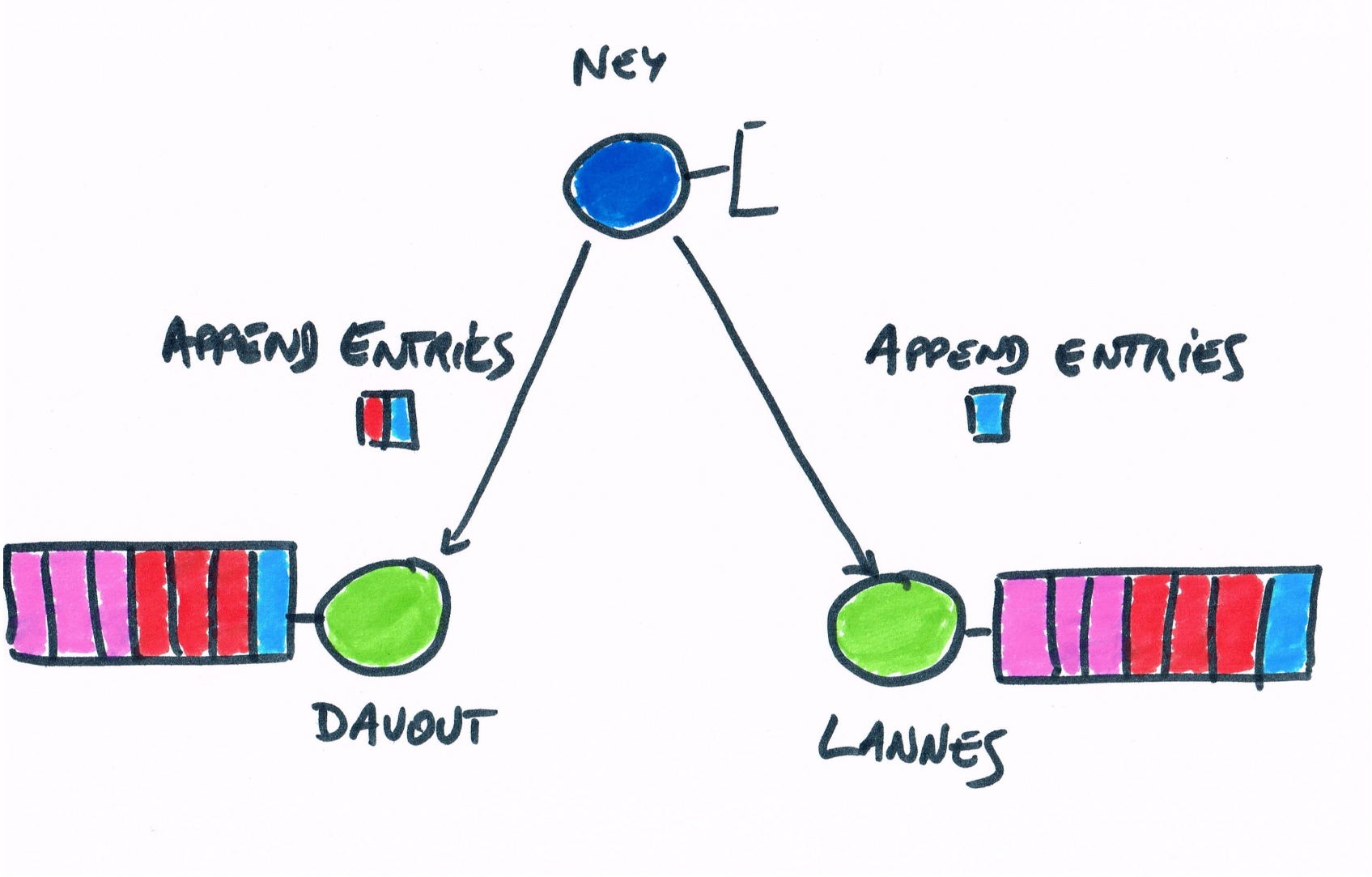


Figure : Leader replicates own log to followers

Raft Algorithm

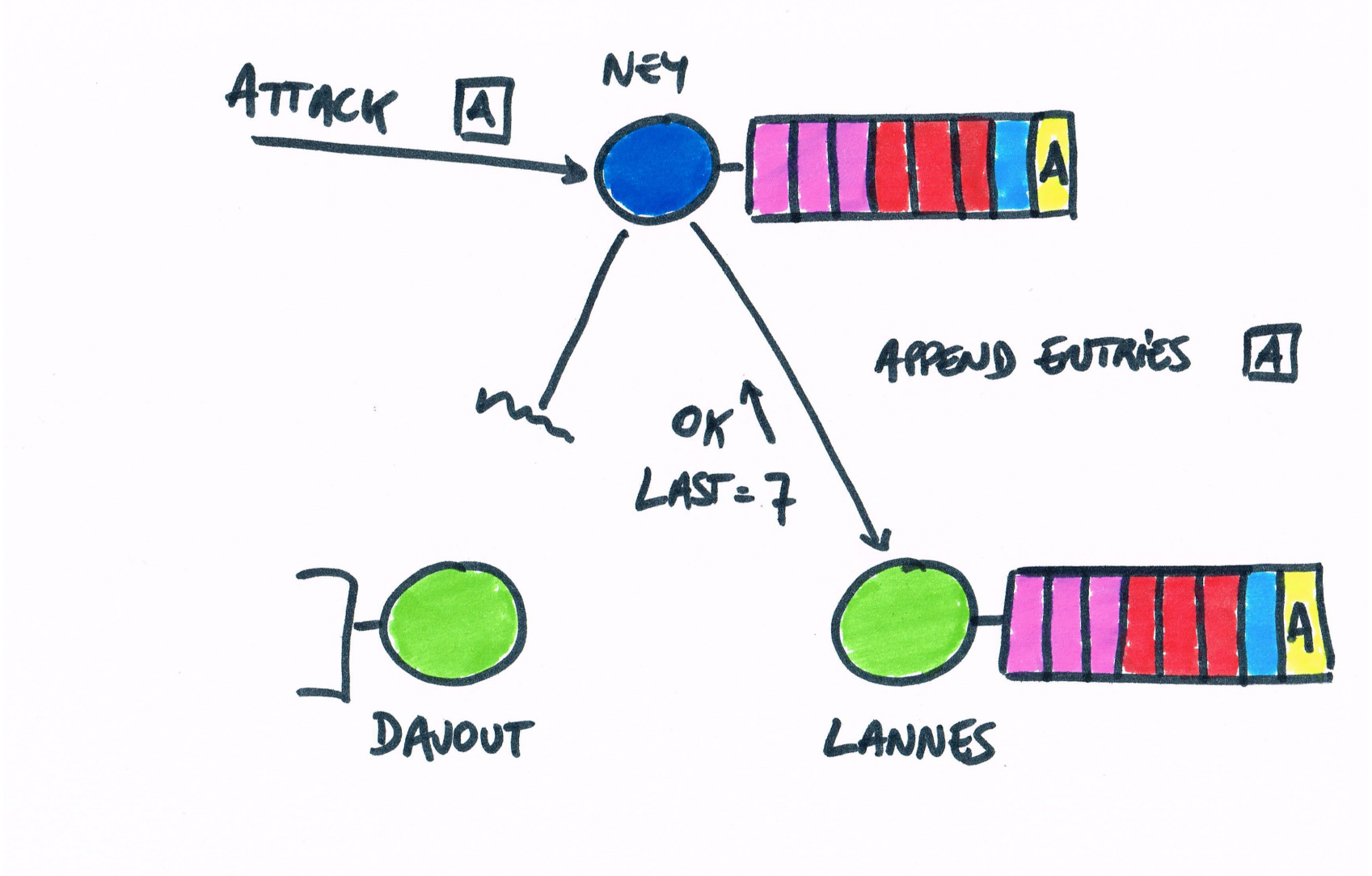


Figure : Ney receives attack order and propagates it

Raft Algorithm

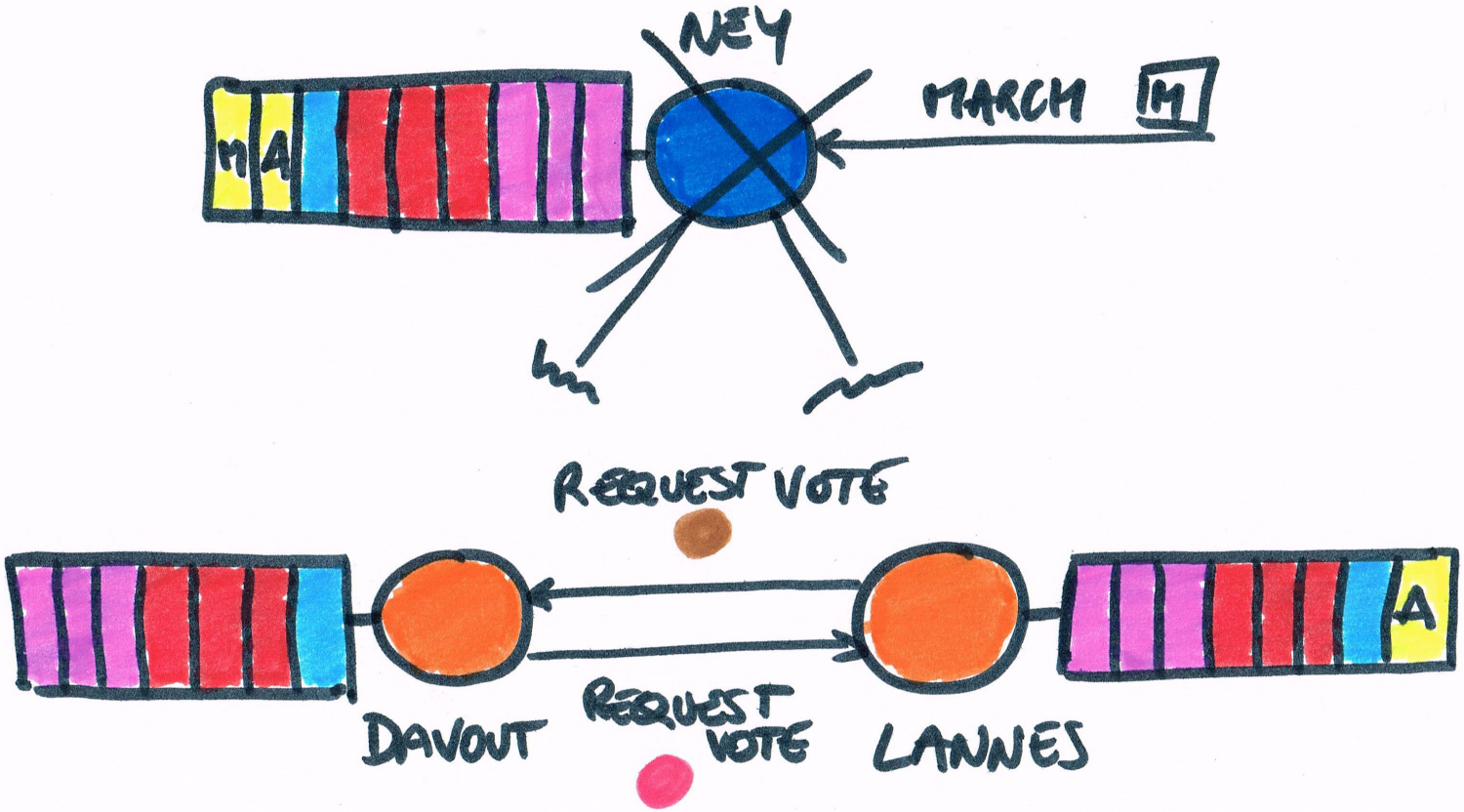


Figure : Ney receives march order but is isolated

Raft Algorithm

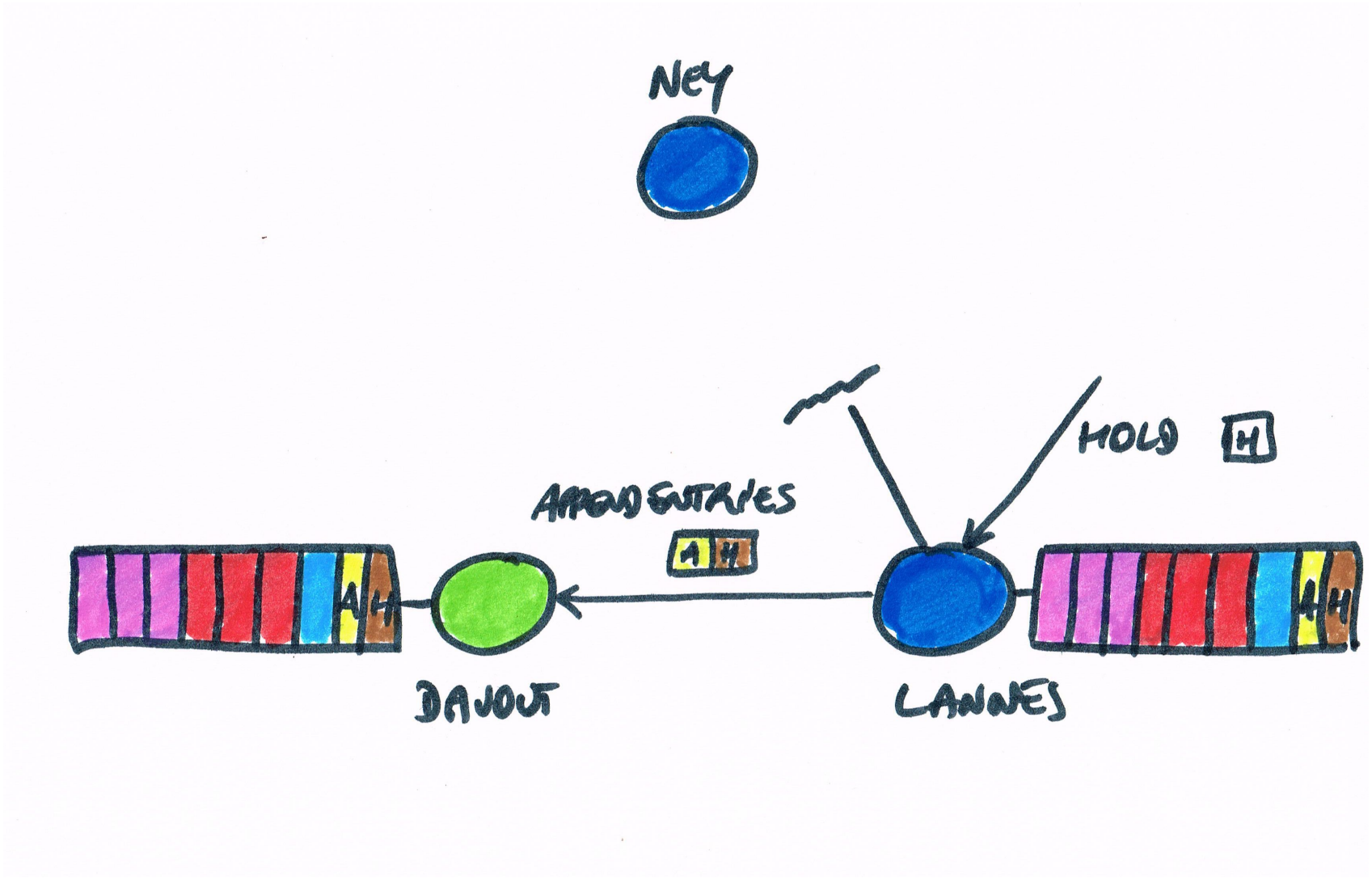


Figure : Lannes is appointed leader for new term

Raft Algorithm

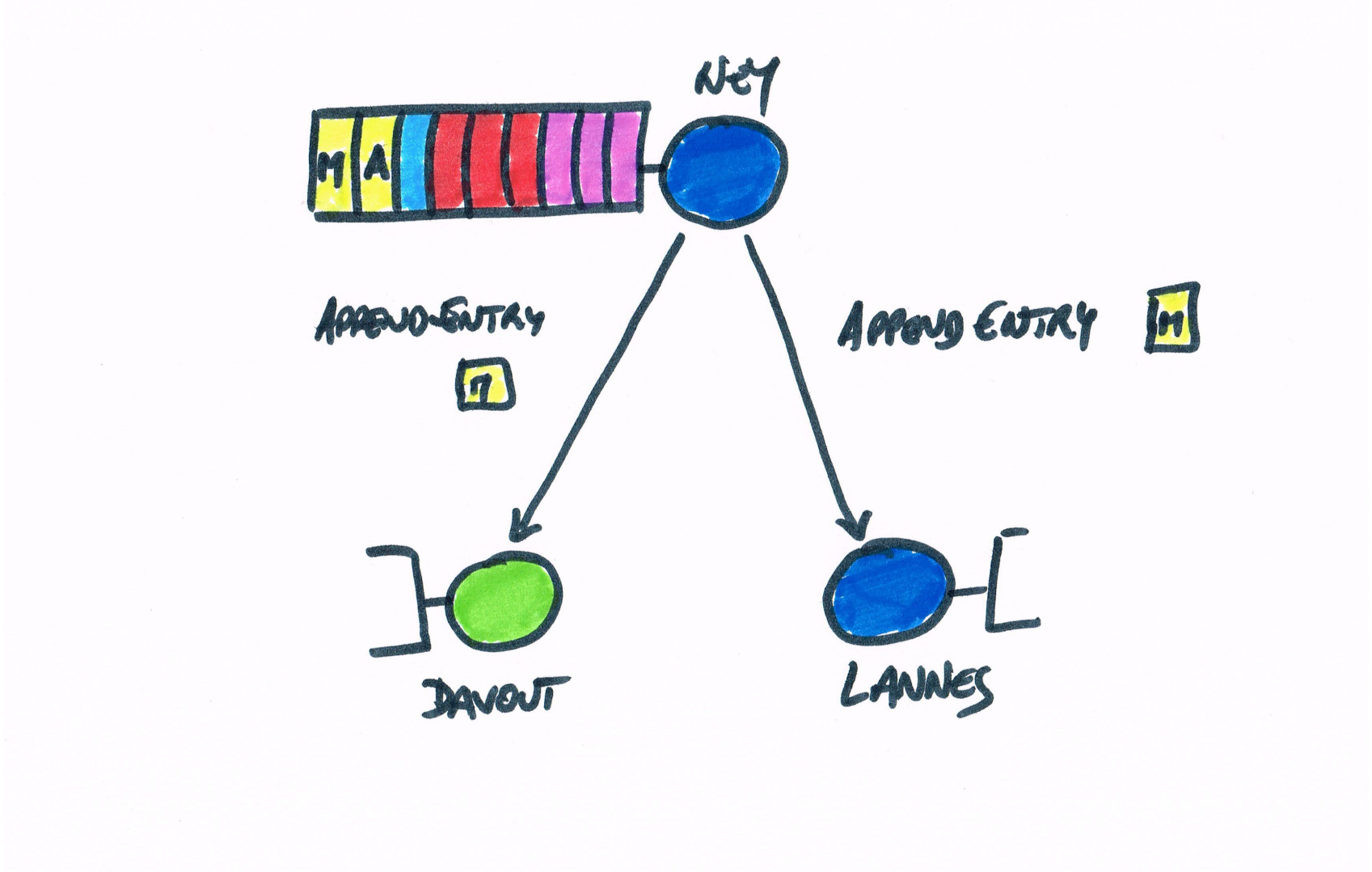


Figure : Ney comes back and tries to propagates march order

Raft Algorithm

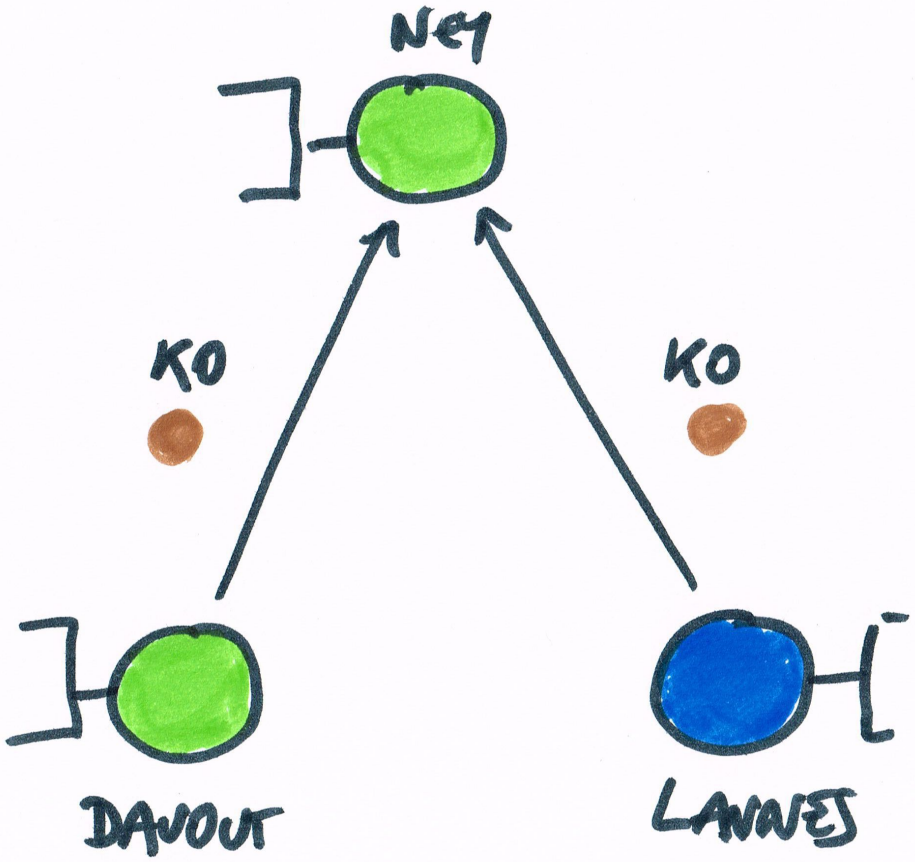


Figure : Ney fallback to follower state

Other Features

- **Cluster Reconfiguration** Supports cluster membership changes w/o service interruption

Other Features

- **Cluster Reconfiguration** Supports cluster membership changes w/o service interruption
- **Log compaction** Logs can grow very large on systems with high throughput, slowing down rebuild after crash and occupying unnecessary disk space

Other Features

- **Cluster Reconfiguration** Supports cluster membership changes w/o service interruption
- **Log compaction** Logs can grow very large on systems with high throughput, slowing down rebuild after crash and occupying unnecessary disk space
- *Snapshotting* replaces history prefix with a representation of the state

Java Implementation: Barge

https://github.com/mgodave/barge !

- OSS project started by Dave Rusek with contributions from Justin Santa Barbara and yours truly

Java Implementation: Barge

https://github.com/mgodave/barge !

- OSS project started by Dave Rusek with contributions from Justin Santa Barbara and yours truly
- Still very young but usable, provides 2 transport methods: Raw TCP and HTTP

Java Implementation: Barge

https://github.com/mgodave/barge !

- OSS project started by Dave Rusek with contributions from Justin Santa Barbara and yours truly
- Still very young but usable, provides 2 transport methods: Raw TCP and HTTP
- Feature complete w.r.t base protocol but missing *cluster reconfiguration* and *log compaction*

Java Implementation: Barge

https://github.com/mgodave/barge !

- OSS project started by Dave Rusek with contributions from Justin Santa Barbara and yours truly
- Still very young but usable, provides 2 transport methods: Raw TCP and HTTP
- Feature complete w.r.t base protocol but missing *cluster reconfiguration* and *log compaction*
- Friendly (Apache 2.0) License, *Pull Requests* are welcomed

Demo

Takeaways

- Understand your consistency requirements

Takeaways

- Understand your consistency requirements
  - Strong consistency $\Rightarrow$ **Consensus**

Takeaways

- Understand your consistency requirements
  - Strong consistency **Consensus**
- Lowered barrier of entry to use consensus at applicative level

Takeaways

- Understand your consistency requirements
  - Strong consistency **Consensus**
- Lowered barrier of entry to use consensus at applicative level
  - Raft is lightweight and understandable

Takeaways

- Understand your consistency requirements
  - Strong consistency **Consensus**
- Lowered barrier of entry to use consensus at applicative level
  - Raft is lightweight and understandable
- Not a Silver Bullet

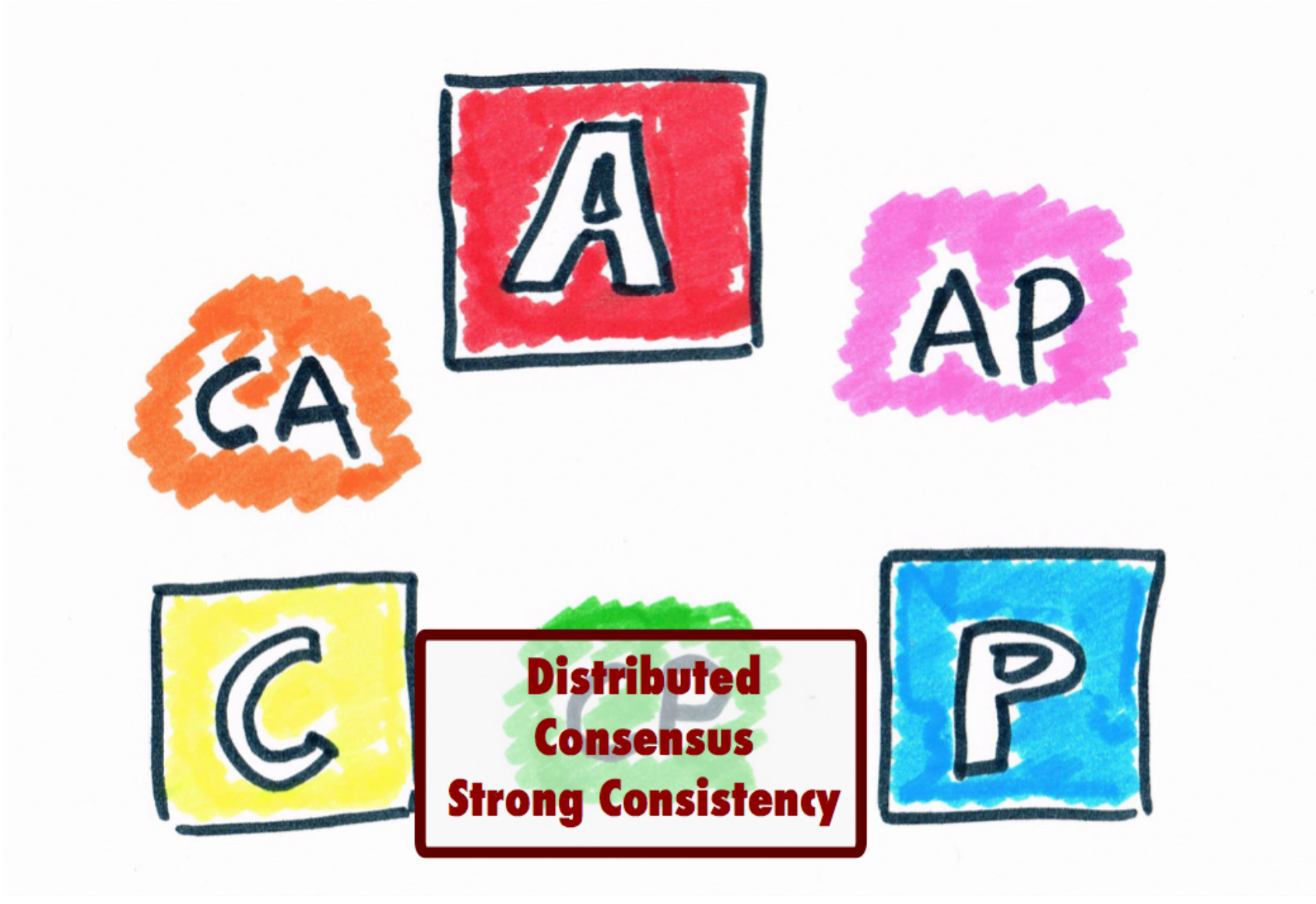
Takeaways

- Understand your consistency requirements
  - Strong consistency **Consensus**
- Lowered barrier of entry to use consensus at applicative level
  - Raft is lightweight and understandable
- Not a Silver Bullet
  - Strong Consistency has a cost you don't want to pay for high throughput and large data sets

Takeaways

- Understand your consistency requirements
  - Strong consistency **Consensus**
- Lowered barrier of entry to use consensus at applicative level
  - Raft is lightweight and understandable
- Not a Silver Bullet
  - Strong Consistency has a cost you don't want to pay for high throughput and large data sets
  - Sweet spot: Configuration data, synchronizing clients at key points

TINSTAAFL



Questions?

Credits & Links

- ETH Zurich Course on Distributed Systems

Credits & Links

- ETH Zurich Course on Distributed Systems
- Napoléon à Austerlitz

Credits & Links

- ETH Zurich Course on Distributed Systems
- Napoléon à Austerlitz
- Nancy Lynch at CSAIL