Distributed State Management

Apache Zookeeper and Spring Distributed State Machine

Machine Learning + Big Data in Real Time + Cloud Technologies

=> The Future of Intelligent Systems

Where to Find The Code and Materials?

https://github.com/iproduct/course-ml

Agenda for This Lesson - I

- Synchronizing state in distributed systems.
- **Eventual Consistency** and **High Availability – HA**. **CAP** theoreme
- **Multi-Agent Systems – MAS**
- Real-time collaboration between multiple agents.
- Methods and protocols for achieving consistency in distributed systems.
- **Gossip** protocols.
- **Paxos** protocols for achieving consesnsus in a network of unreliable processors.
- **RAFT** algorithm for consesnsus achievement.

Agenda for This Lesson - II

- Production **RAFT** implementation - **BRAFT** of Baidu using brpc.
- Development of industrial-grade **HA** distributed systems using **BRAFT**.
- **Operational Transforms (OT)**
- **Conflict-free Replicated Data Types (CRDTs)**
- **CRDTs** based on **operations** and **state** – equivalence between them
- Examples for CRDTs implementation: **G-Counter (Grow-only Counter)**, **PN-Counter (Positive-Negative Counter)**, **G-Set (Grow-only Set)**, **2P-Set (Two-Phase Set)**, **LWW-Element-Set**, **OR-Set (Observed-Remove Set)**, **Sequence CRDTs**.

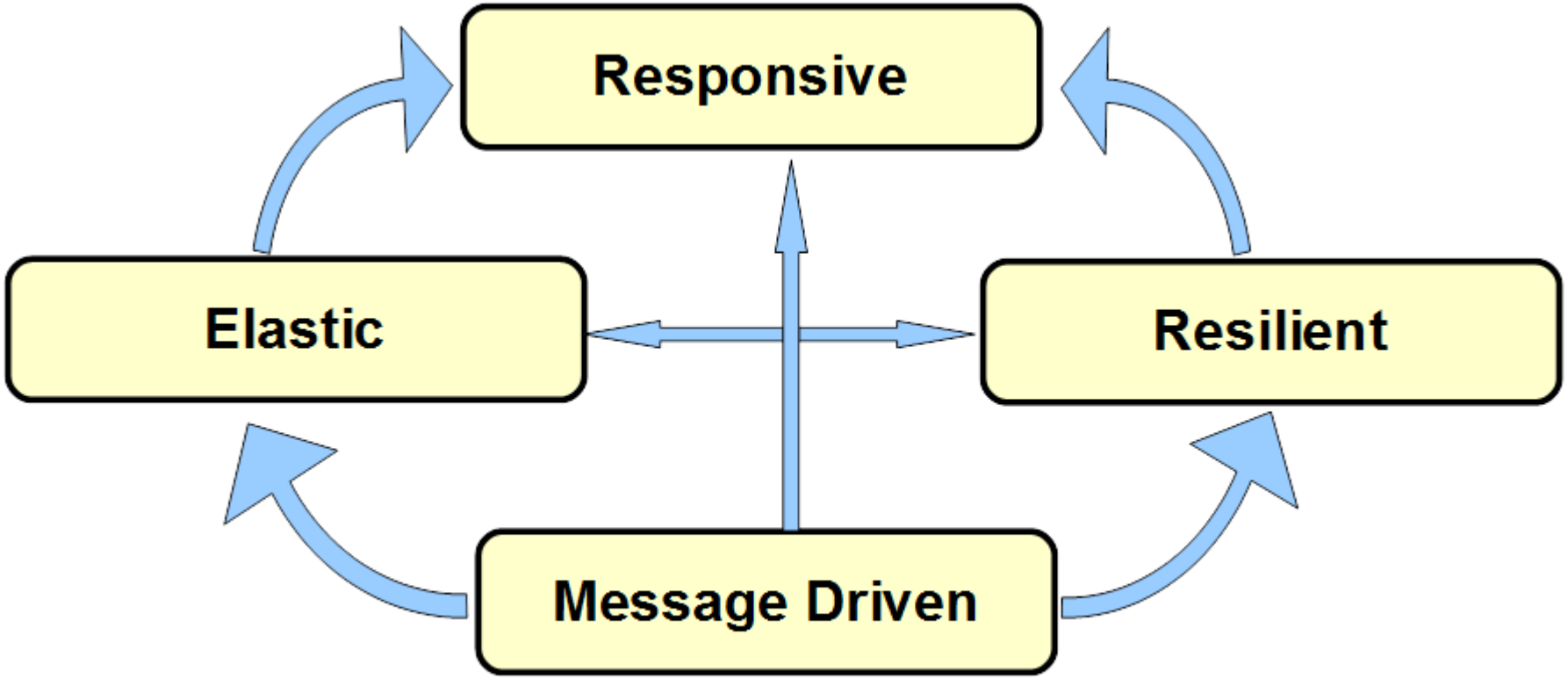
Synchronizing Distributed State

Distributed Systems and Consistency

- **Distributed system** – a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another – examples: SOA-based systems to massively multiplayer online games to peer-to-peer applications, distributed data stores (filesystems, databases, caches).
- **Consistency** – in classical deductive logic, a consistent theory is one that does not entail a contradiction. The lack of contradiction can be defined in either semantic or syntactic terms. The semantic definition states that a theory is consistent if it has a model, there exists an interpretation under which all formulas in the theory are true.
- In a distributed system, it may be understood as that plurality of data values in the nodes should be consistent.
- **Consensus algorithm** needed – when certain data is not available in a single node (host), other nodes (hosts) should ensure availability of that data.

Reactive Manifesto

[http://www.reactivemanifesto.org]



Consistency

- A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes.
- This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation
- Examples:
  - agreeing on what transactions to commit to a database in which order
  - state machine replication – a technique for converting an algorithm into a fault-tolerant, distributed implementation
  - atomic broadcasts
- Real-world examples: cloud computing, clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain, etc.

Types of Consistency

https://medium.com/cosmosgaminghub/consistency-e3e0fe41358d

- **Strong Consistency** - ensure that cluster state changes immediately after submission of state changes. Examples models:
  - Paxos
  - Raft（muti-paxos）
  - ZAB（muti-paxos）
- **Weak Consistency** = **Eventual Consistency (EC)** – the system does not guarantee change the cluster immediately after submission status change, but over time the final state will eventually be the same. Examples models:
  - DNS system
  - Gossip protocols

Examples:

- **Google's Chubby** distributed lock service, using the **Paxos** algorithm
- **etcd** distributed key-value database, using the **Raft** algorithm
- **ZooKeeper** coordination of distributed application services, **Chubby** open source implementation, using algorithms **ZAB**

CAP Theoreme

- CAP theorem [Eric Brewer, 1988], states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes
- When a network partition failure happens should we decide to
  - Cancel the operation and thus decrease the availability but ensure consistency
  - Proceed with the operation and thus provide availability but risk inconsistency
- **ACID** (Atomicity, Consistency, Isolation, Durability) → **BASE** (Basically Available, Soft state, Eventual consistency)

BASE (Basically Available, Soft state, Eventual consistency)

- **(B)asically (A)vailable**: basic reading and writing operations are available as much as possible (using all nodes of a database cluster), but without any kind of consistency guarantees (the write may not persist after conflicts are reconciled, the read may not get the latest write)
- **(S)oft state**: without consistency guarantees, after some amount of time, we only have some probability of knowing the state, since it may not yet have converged
- **(E)ventually consistent**: If the system is functioning and we wait long enough after any given set of inputs, we will eventually be able to know what the state of the database is, and so any further reads will be consistent with our expectations
- Eventual consistency is sometimes criticized as increasing the complexity of distributed software applications. This is partly because eventual consistency is purely a liveness guarantee (reads eventually return the same value) and does not make safety guarantees (an eventually consistent system can return any value before it converges)

Conflict Resolution and Strong Eventual Consistency

- To ensure replica convergence, a system must **reconcile differences** between multiple copies of distributed data. This consists of two parts:
  - exchanging versions or updates of data between servers (anti-entropy);
  - choosing appropriate final state when concurrent updates have occurred (reconciliation)
- Reconciliation approaches : "last writer wins", conflict handler, timestamps & vector clocks
- Time to reconcile:
  - Read repair - correction is done when a read finds an inconsistency
  - Write repair: correction takes place during a write operation
  - Asynchronous repair: correction is not part of a read or write operation.
- **Strong eventual consistency (SEC)** – adds the safety guarantee that any two nodes that have received the same (unordered) set of updates will be in the same state.
- If furthermore, the system is monotonic, the application will never suffer rollbacks. Conflict-free replicated data types (CRDT) are a common approach to ensuring SEC.

Paxos Strong Consensus Algorithm [Leslie Lamport, 1989]

- **No deterministic fault-tolerant consensus protocol can guarantee progress** in an asynchronous network (proved in a paper by Fischer, Lynch and Paterson)
- **Paxos guarantees safety (consistency)**, and the conditions that could prevent it from making progress are difficult to provoke.
- Assumptions:
  - **Processors** operate at arbitrary speed, may experience failures, may re-join the protocol after failures, do not collude, lie, or otherwise attempt to subvert the protocol (no Byzantine failures)
  - **Network** – processors can send messages to each other, messages are sent asynchronously and may take arbitrarily long to deliver, messages may be lost, reordered, or duplicated, messages are delivered without corruption.
- Number of processors – if there are **2*N + 1** processors, the consensus can be achieved despite the simultaneous failure of any **N** processors (the number of non-faulty processes must be strictly greater than the number of faulty processes).

Paxos Strong Consensus Algorithm - Roles

Client - issues a request to the distributed system, and waits for a response. For instance, a write request on a file in a distributed file server.

Acceptor (Voters) - act as the fault-tolerant "memory" of the protocol. Acceptors are collected into groups called Quorums. Any message sent to an Acceptor must be sent to a Quorum of Acceptors. Any message received from an Acceptor is ignored unless a copy is received from each Acceptor in a Quorum.

Proposer - advocates a client request, attempting to convince the Acceptors to agree on it, and acting as a coordinator to move the protocol forward when conflicts occur.

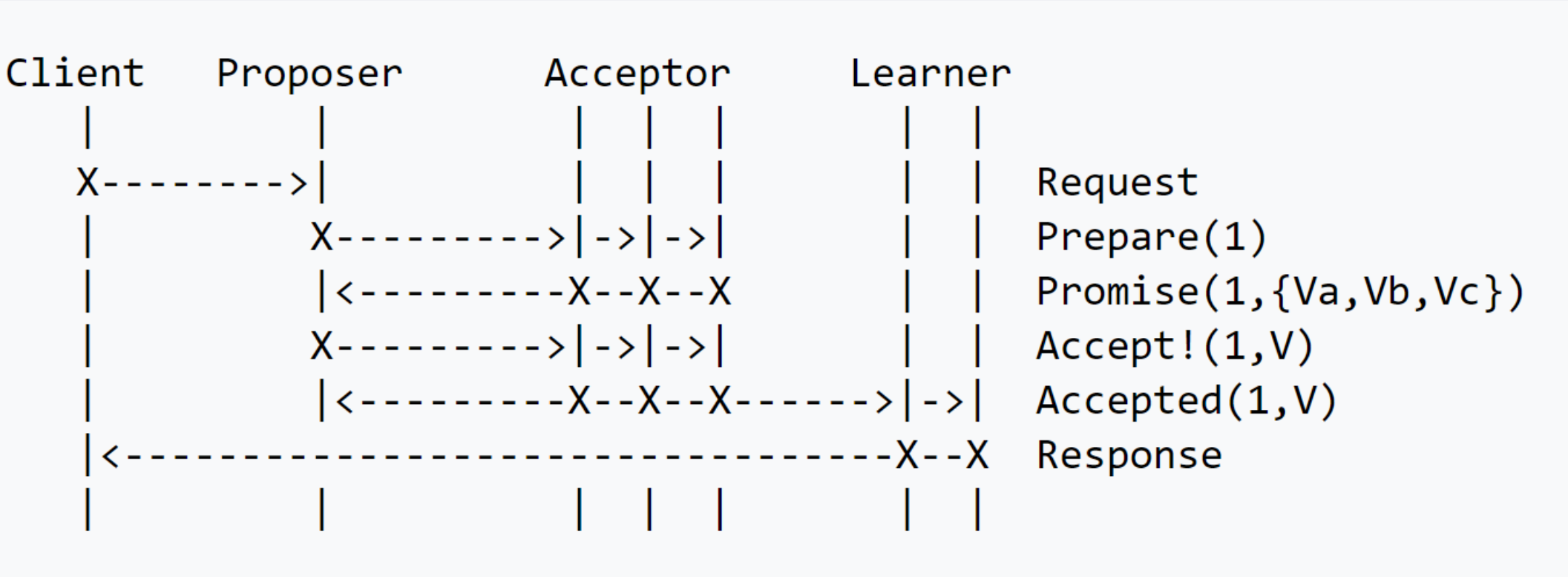
Learner - the replication factor for the protocol. Once a Client request has been agreed upon by the Acceptors, the Learner may take action (i.e.: execute the request and send a response to the client). To improve availability of processing, additional Learners can be added.

Leader - distinguished Proposer to ensure request progress. Many processes may believe they are leaders, but the protocol only guarantees progress if one of them is eventually chosen. If two processes believe they are leaders, they may stall the protocol by continuously proposing conflicting updates. However, the safety properties are still preserved in that case.

Paxos Strong Consensus Algorithm without Failures

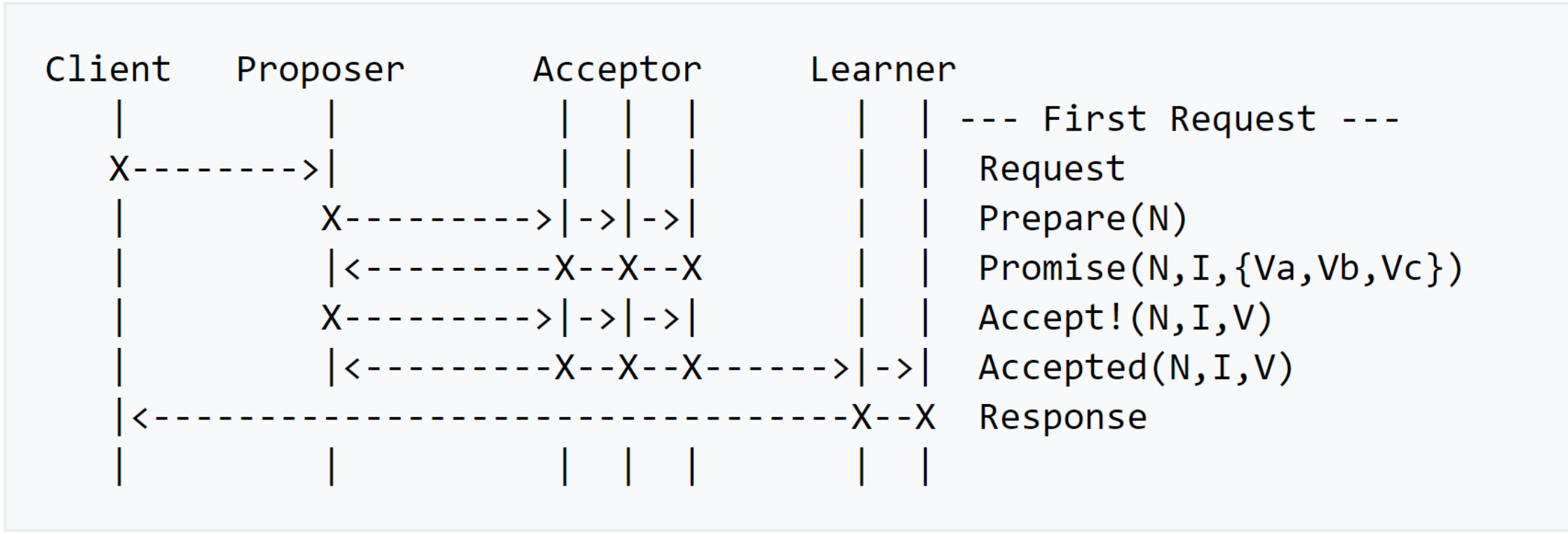
https://en.wikipedia.org/wiki/Paxos_(computer_science)

- Proposal proposed modification request, i.e., a distributed system, can be expressed as: **<[proposal number N, the content of the proposal value V]>**



Multi-Paxos when phase 1 can be skipped

https://en.wikipedia.org/wiki/Paxos_(computer_science)



Raft Algorithm

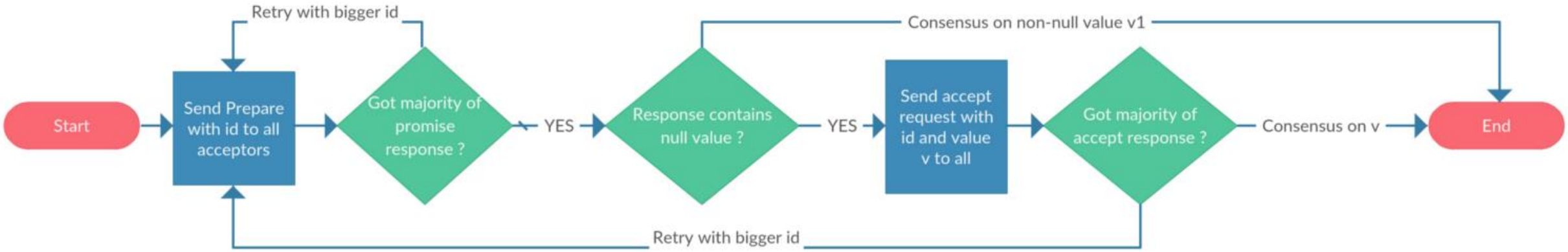
- **Raft** is a consensus algorithm designed as an alternative to the Paxos family of algorithms.
- It is named after Reliable, Replicated, Redundant, And Fault-Tolerant.
- It was meant to be more understandable than Paxos by means of separation of logic, but it is also formally proven safe and offers some additional features
- Raft offers a generic way to distribute a state machine across a cluster of computing systems, ensuring that each node in the cluster agrees upon the same series of state transitions.
- It has a number of open-source reference implementations, with full-specification implementations in Go, C++, Java, and Scala.

Safety rules in Raft

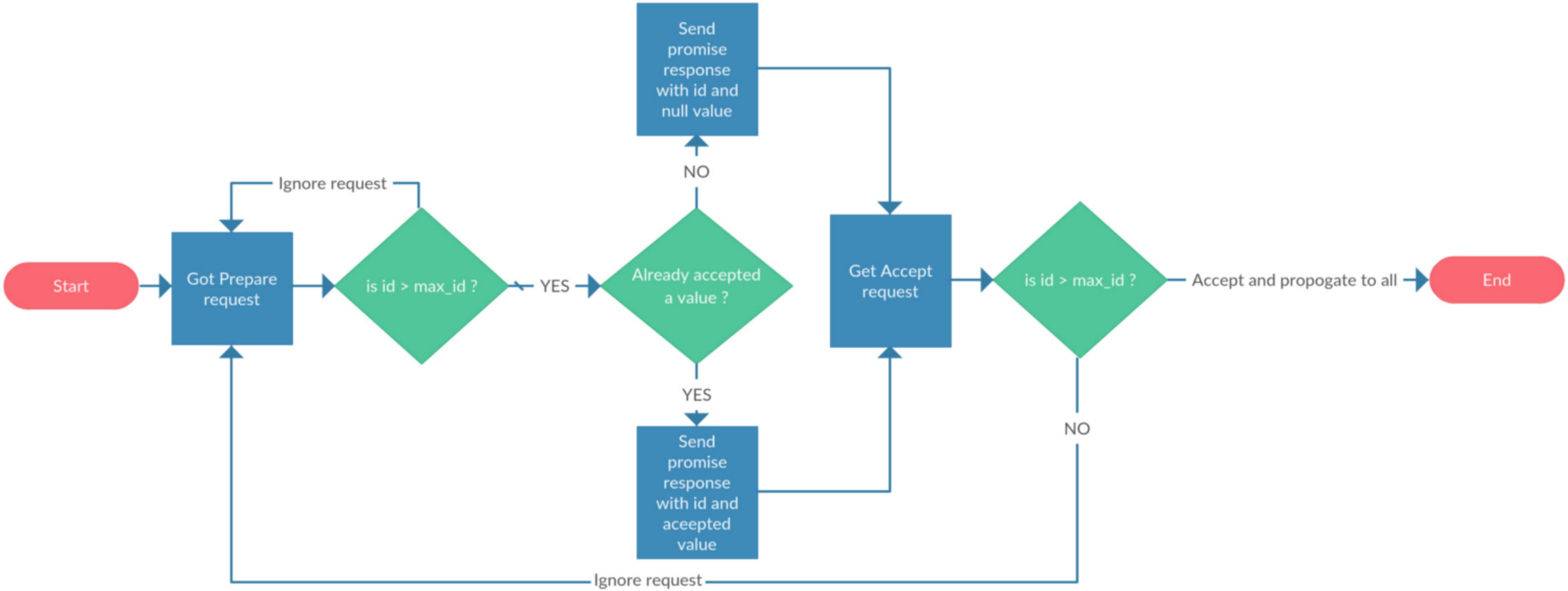
- Raft guarantees each of these safety properties :
  - Election safety: at most one leader can be elected in a given term.
  - Leader append-only: a leader can only append new entries to its logs (it can neither overwrite nor delete entries).
  - Log matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
  - Leader completeness: if a log entry is committed in a given term then it will be present in the logs of the leaders since this term
  - State machine safety: if a server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log.
- The first four rules are guaranteed by the details of the algorithm described in the previous section. The State Machine Safety is guaranteed by a restriction on the election process.

Raft Algorithm – Proposers

http://thesecretlivesofdata.com/raft/



Raft Algorithm - Acceptors



Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD