Methods of Cloud Computing

Programming Cloud Resources 1: Scalable and Fault-Tolerant Applications

Complex and Distributed Systems
Faculty IV
Technische Universität Berlin

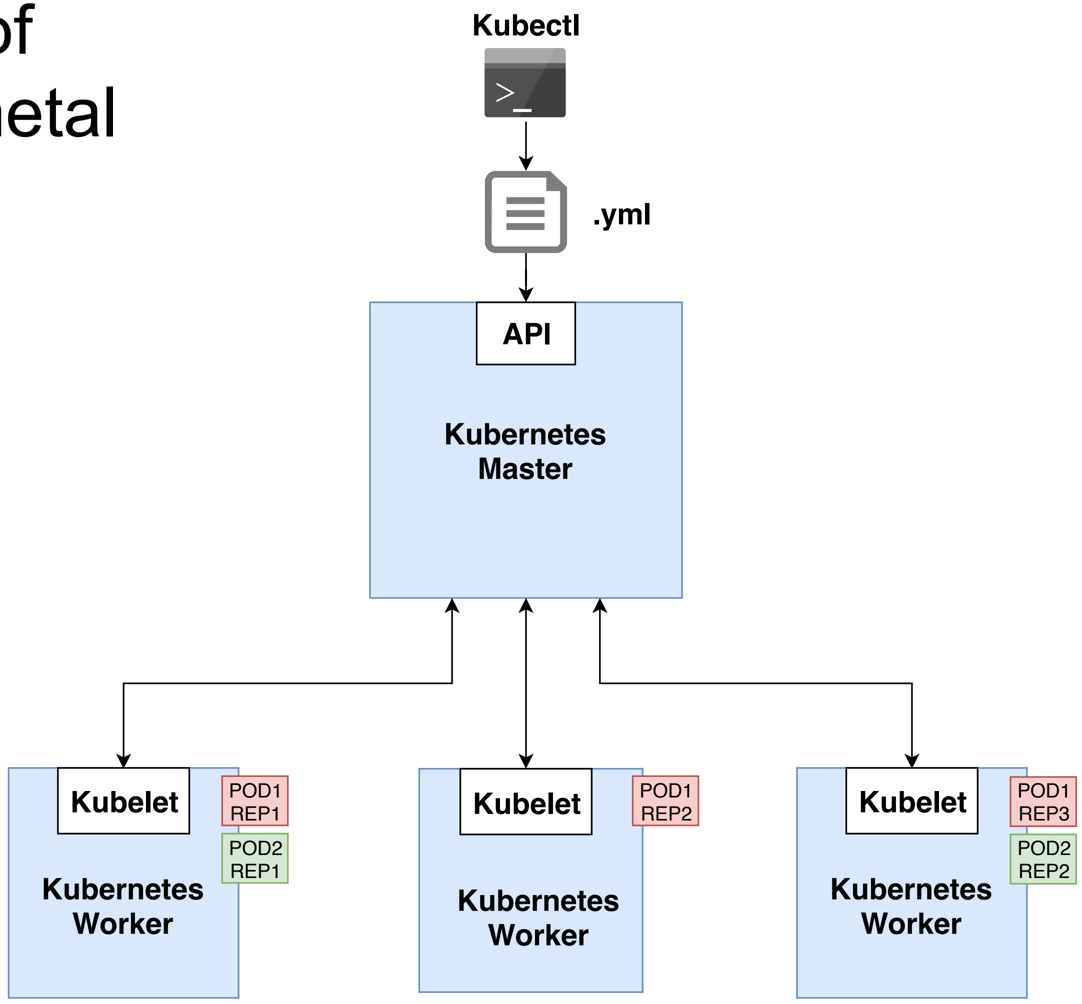
Operating Systems and Middleware
Hasso-Plattner-Institut
Universität Potsdam

Overview

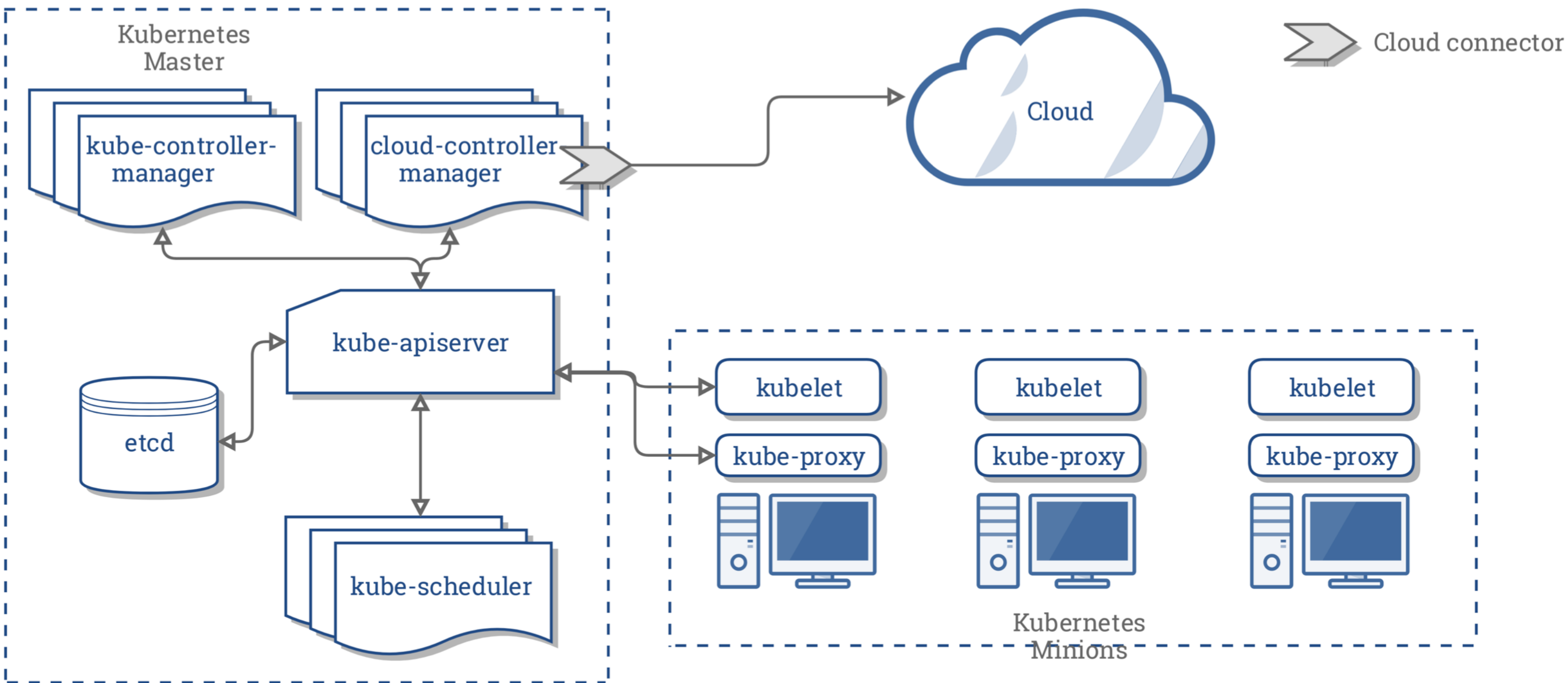
- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies
  - **Kubernetes Auto-Scaling**
  - Amazon DynamoDB
  - Blockchain

Recap: Kubernetes Cluster

- Manages collection of nodes (either bare-metal or virtual machines)
- Runs groups of replicated containers (called *Pods*)



Kubernetes Horizontal Pod Autoscaler

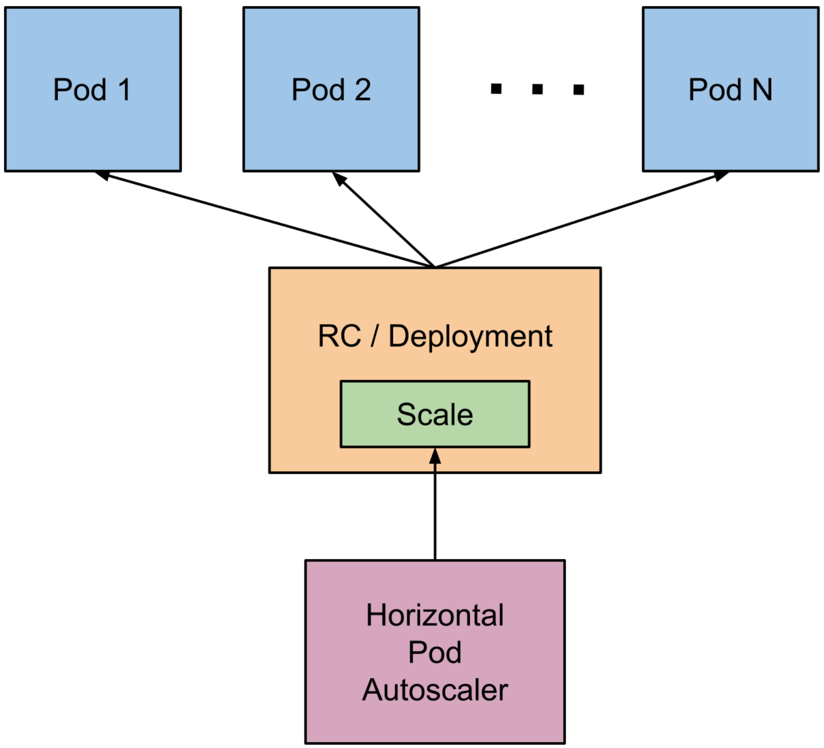


Kubernetes
Horizontal Pod Autoscaler

- Automatically scales number of pods in a replication controller
- Based on
  - CPU utilization or custom metrics
  - Target value for the metric

- Autoscaler checks metrics every 30s
- Sets the number of replications to optimize the metric towards the target value
  - Creating more pods
  - Or reducing the number of pods

Kubernetes Horizontal Pod Autoscaler



Kubernetes
Horizontal Pod Autoscaler

- Number of replicas is scaled by the quota of average current metric value and target value

```
desiredReplicas =
  ⌈ currentReplicas * (currentMetricValue/desiredMetricValue) ⌉
```

- No scaling if this quota is within 0.1 tolerance

JItter avoidance using tolerance

➔ This assumes linear scaling!

- Autoscaler scales to the highest number of desired replicas in a 5 minute sliding window
  - Quick responses to more load, but reduces *thrashing*

Sometimes metrics are not available

- Discard pods that are being shut down
- Normally running pods with missing metrics
  - If result without these would be to scale up: assume metric to be 0
  - If result without these would be to scale down: assume metric to be 1
- Assume metric to be 0 for pods that are not yet ready

Dampens the scaling

Overview

- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies
  - Kubernetes Auto-Scaling
  - **Amazon DynamoDB**
  - Blockchain

Amazon DynamoDB[4]

- Highly-available key-value store, provided as a managed service service by Amazon
- Primary design goals
  - High scalability
    - E.g. 1000s of servers
  - High availability
    - Particularly, support for “always write“
  - High performance
    - Particularly, small latency (single digit milliseconds) and number of requests (20 million requests per second)
- Sacrifices consistency to achieve these goals

Amazon DynamoDB: Design Principles

- Dynamo follows peer-to-peer approach
  - No server is more important than any other
  - No single point of failure
- Nodes can be added/removed incrementally at runtime
  - In terms of the CAP theorem, DynamoDB is AP
  - Weak consistency guarantee: eventual consistency
- No hostile environment, all servers obey rules
  - No security issues must be considered

How to Partition Data Among the Servers?

- Dynamo designed to be a key-value store
  - → No support for range queries needed
  - → No need for range partitioning

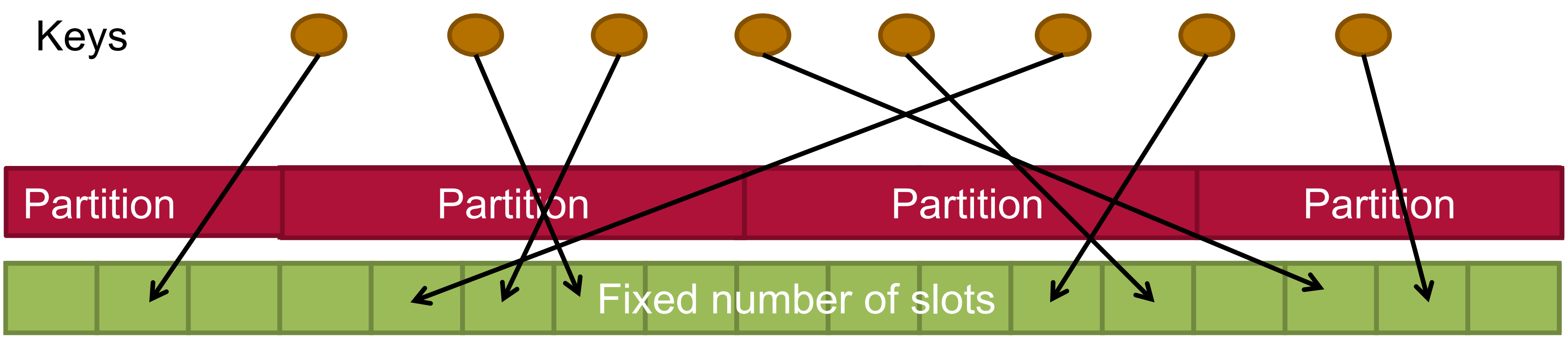
- Hash partitioning has good load balancing properties
  - But how to avoid data reorganization when servers are added or removed?

Consistent hashing: we assign virtual servers in the ring and only had to move data for the server lost instead of all the servers

- Solution: Hash function no longer maps to partitions
  - Instead function maps to a fixed number of slots
  - Variation of classic hashing called *consistent hashing*

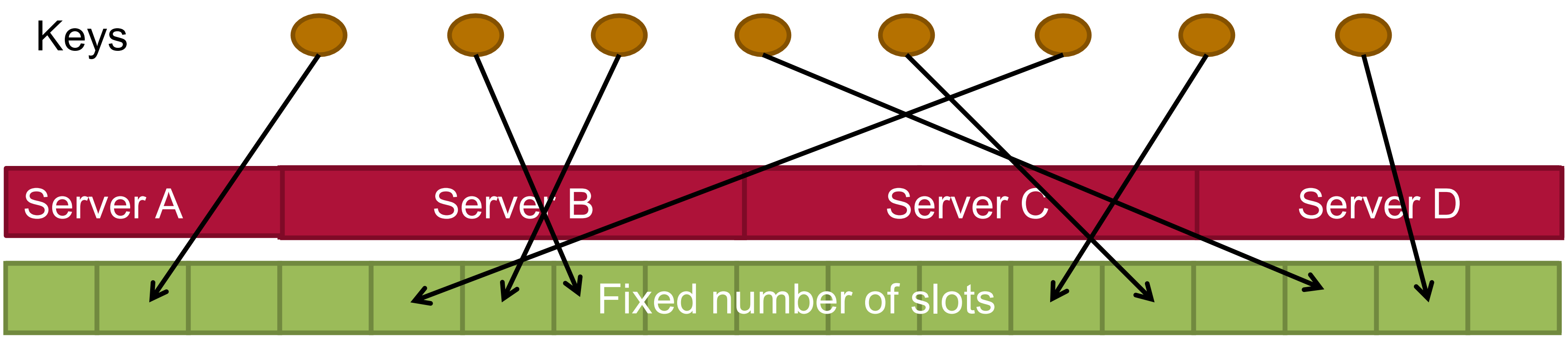
Consistent Hashing[6]

- Idea: Additional mapping between slots and partitions
  - Hash function maps keys to large but fixed number of slots (for example 2^{128} slots)
  - A partition now covers a consecutive number of slots
    - A server can be in charge of one or more partitions
    - Size of a partition is variable



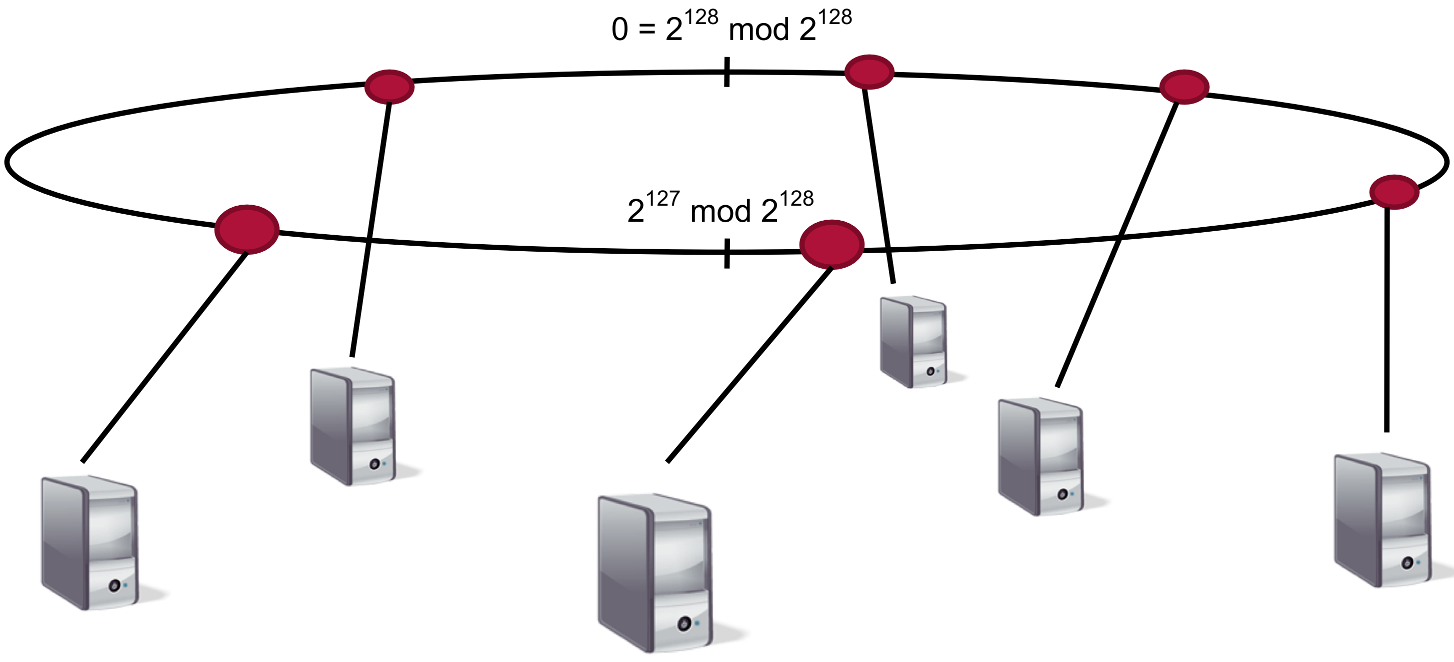
Distributed Hash Tables

- Consistent Hashing is basis for distributed hash tables (DHTs)
  - Each server takes at least one partition
  - Therefore each server is responsible for a continuous range of slots
  - Upon arrival/departure of server only O(#Keys/#Servers) data items must be reorganized



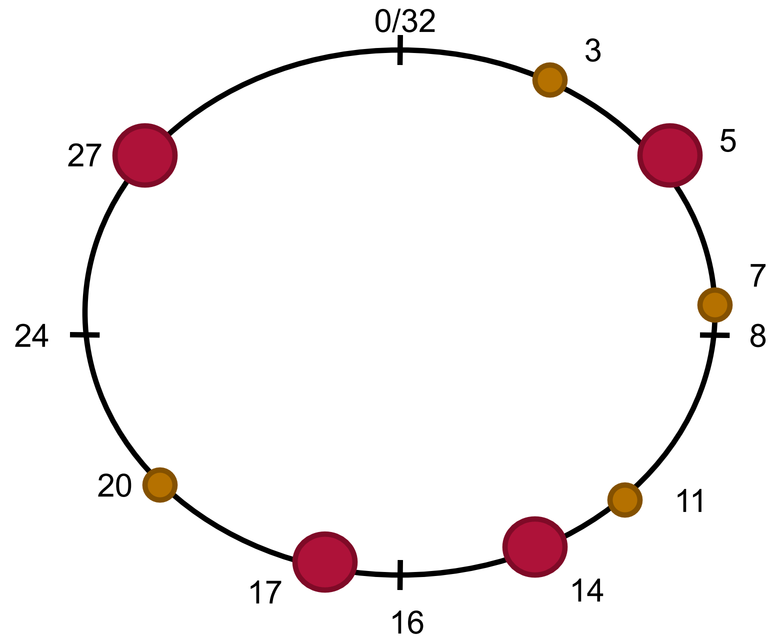
Amazon DynamoDB‘s Partitioning Algorithm (1/2)

- Slots form a circular ID space
  - All servers hash their ID with an MD5 function
  - Hashing result determines server‘s position on the ring



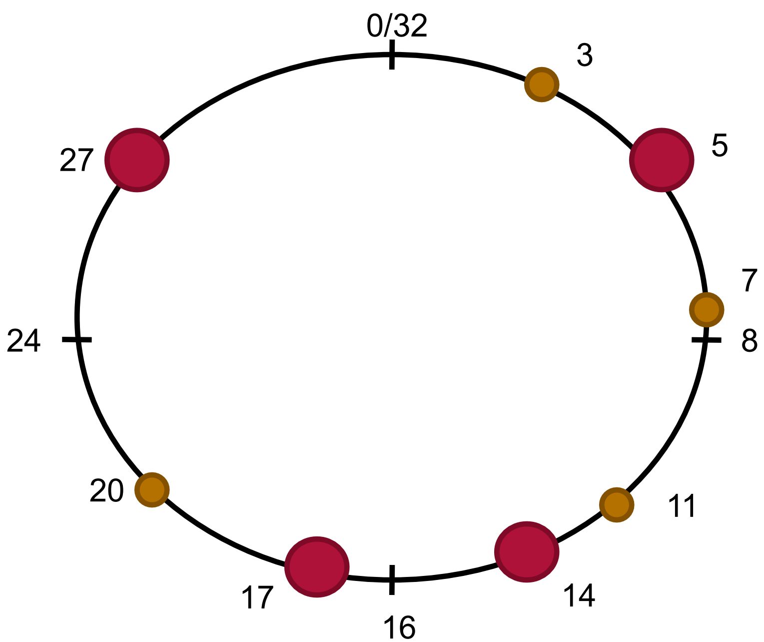
Amazon DynamoDB‘s Partitioning Algorithm (2/2)

- To distribute data, key of data also hashed with MD5
  - Result of hashing is a position in the circular ID space
  - Rule: Server is responsible for all preceding IDs (i.e. slots) up to and including its own ID



Routing in Amazon DynamoDB

- Each server maintains full routing table (ID to IP)
  - → Each server can determine which server is responsible for a data item based on routing table and mapping rule!
  - → One hop routing keeps latencies small

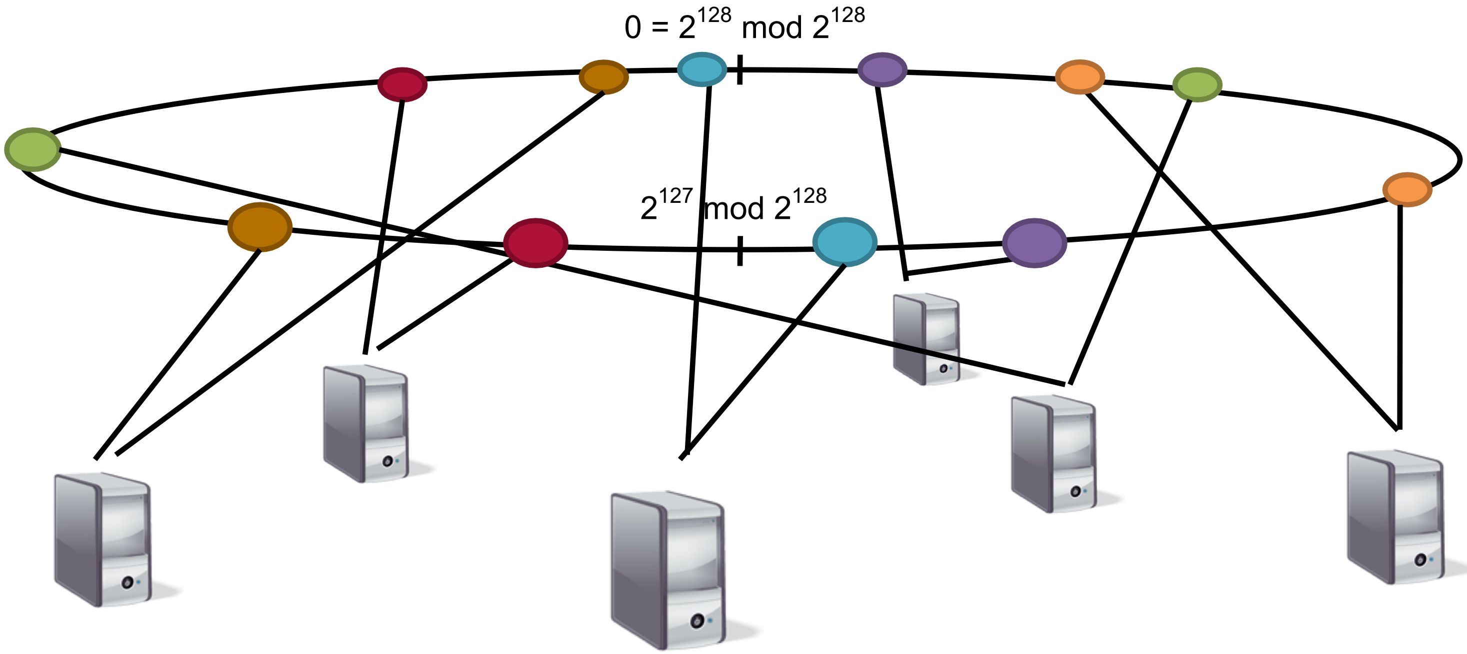


Example routing table in DynamoDB

ID	IP Address
5	192.168.1.2
14	192.168.1.18
17	192.168.1.115
27	192.168.1.98

Virtual Servers for Load Balancing

- Despite uniform distribution over ID space, the servers may receive skewed number of requests
- Idea: Each server appears on multiple positions on the ring (known as virtual servers)



Replication in Amazon DynamoDB

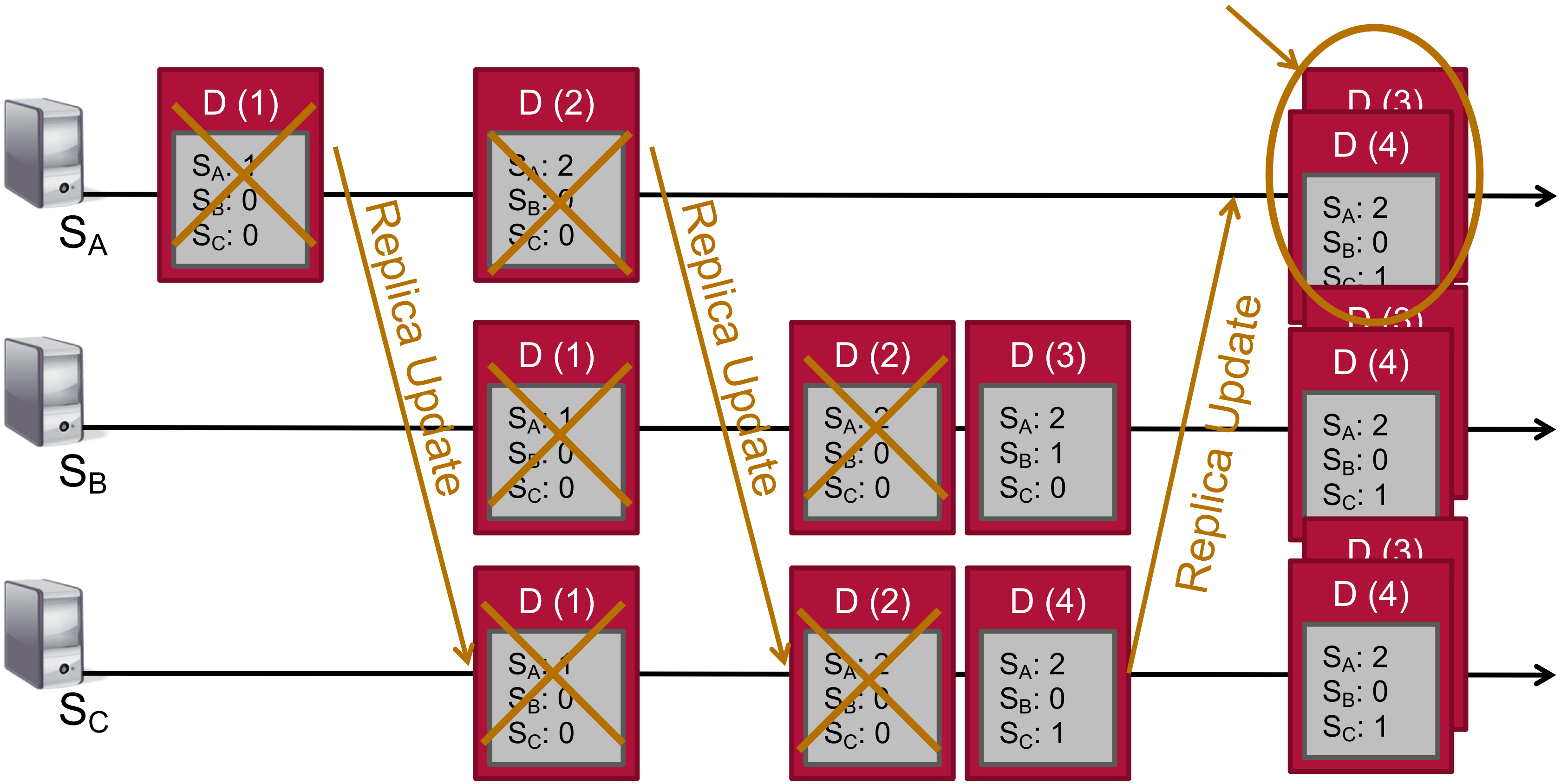
- Servers replicate data to their N successors on the ring
- Replication is adjusted whenever a server is added or removed from the ring
- Servers use heart beat protocol to determine availability
  - Periodic messages exchanged between ring neighbors
  - When server does not answer heart beat request in a given time span, it is considered gone
- Dynamo allows reads and writes on every replica!

Data Versioning in Amazon DynamoDB

- Dynamo allows „always write“ paradigm
  - Write operation allowed on every replica
  - put-() operation returns after one replica has been written
    - Lazy update of replicas in the background
- Result: Different version of a data item may exist
  - Dynamo treats each version of the data item as an immutable object
  - Vector clocks are used to reconcile different versions
  - When system cannot reconcile different versions, the versions are presented to client for reconciliation

Example of Version Reconciliation

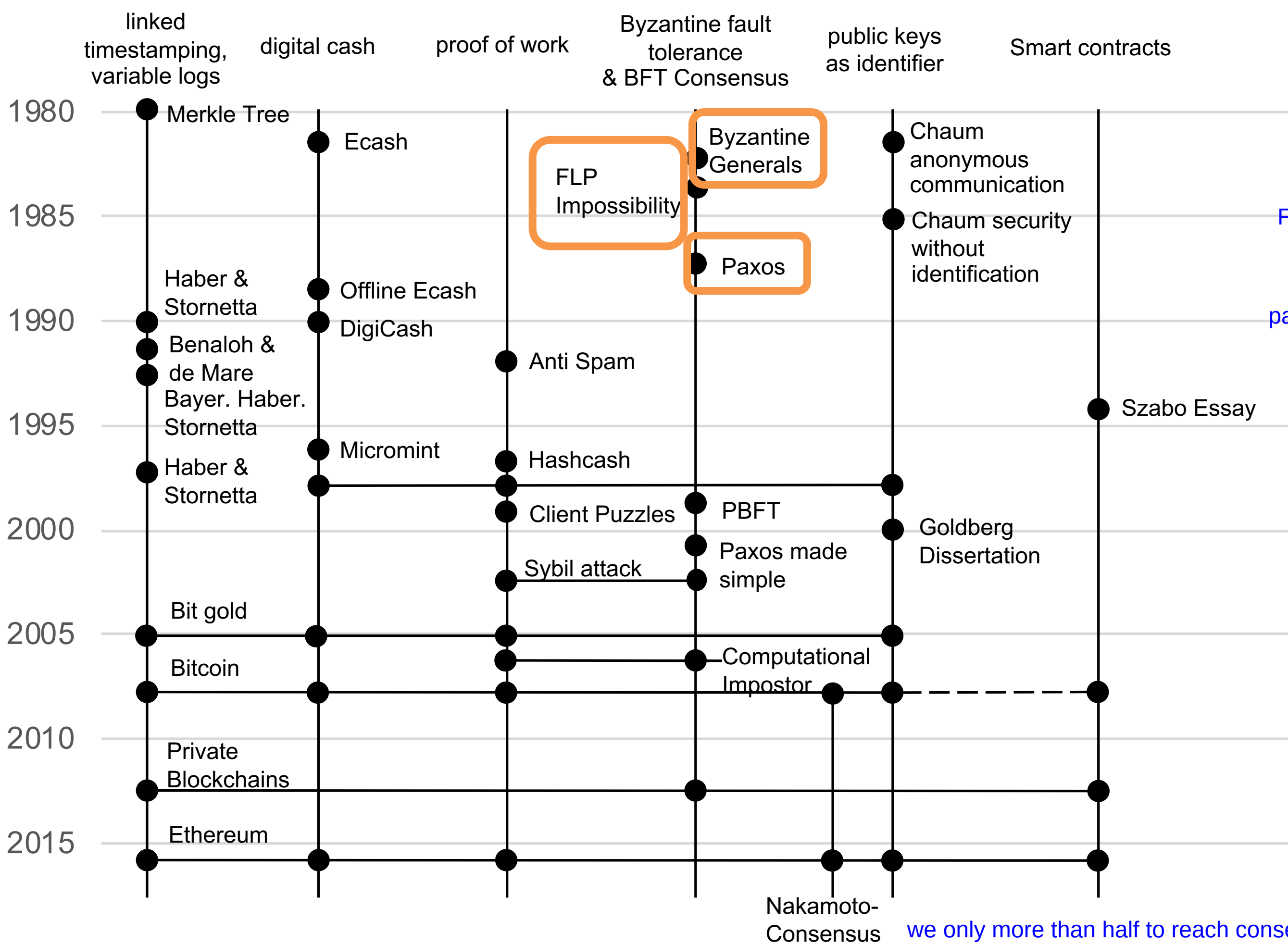
- Let's assume data item D is replicated among three servers: S_A, S_B, and S_C



Overview

- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies
  - Kubernetes Auto-Scaling
  - Amazon Dynamo Load balancing
  - **Blockchain** consistency

Blockchain: Background



FLP: Need 2/3 node to reach consensus

paxos/Raft: Leader selection

we only more than half to reach consensus
and required less requirments/resources

[Narayanan, Arvind, and Jeremy Clark. "Bitcoin's Academic Pedigree." *Queue* 15. 4 (2017): 20.]

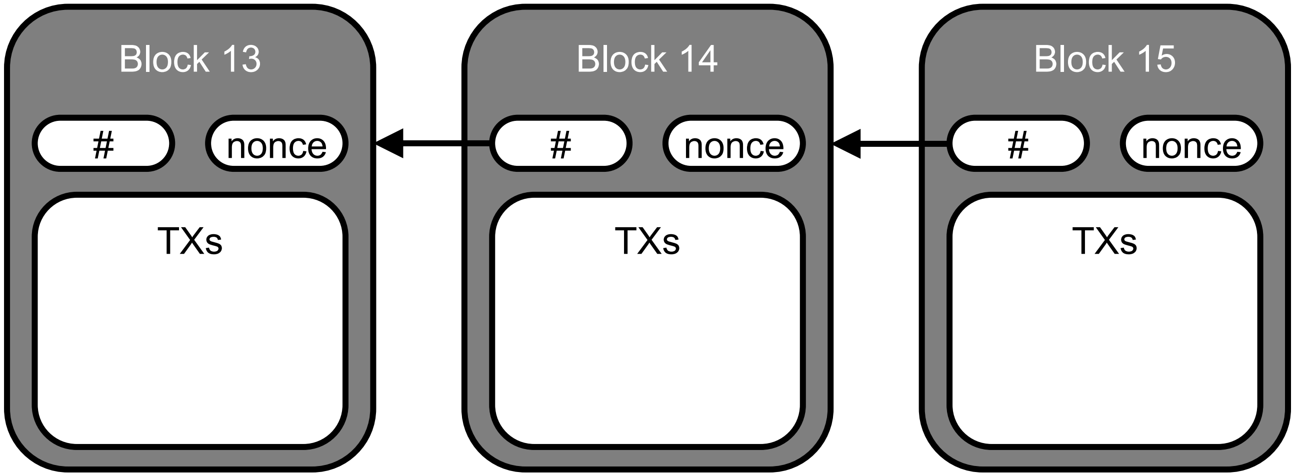
A Ledger

Amount	Sender	Receiver
…	…	…
2 BTC	2bf12	4c2dd
2 BTC	4c2dd	1156f
1 BTC	4c2dd	2bf12
…	…	…

1. Transactions are signed by the sender
2. Nobody is allowed to send more money than he has

– we are done?

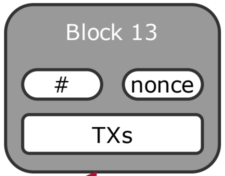
Blockchain



Nakamoto-Consensus

Nakamoto-Consensus based on Proof-of-Work and fixed consensus rules

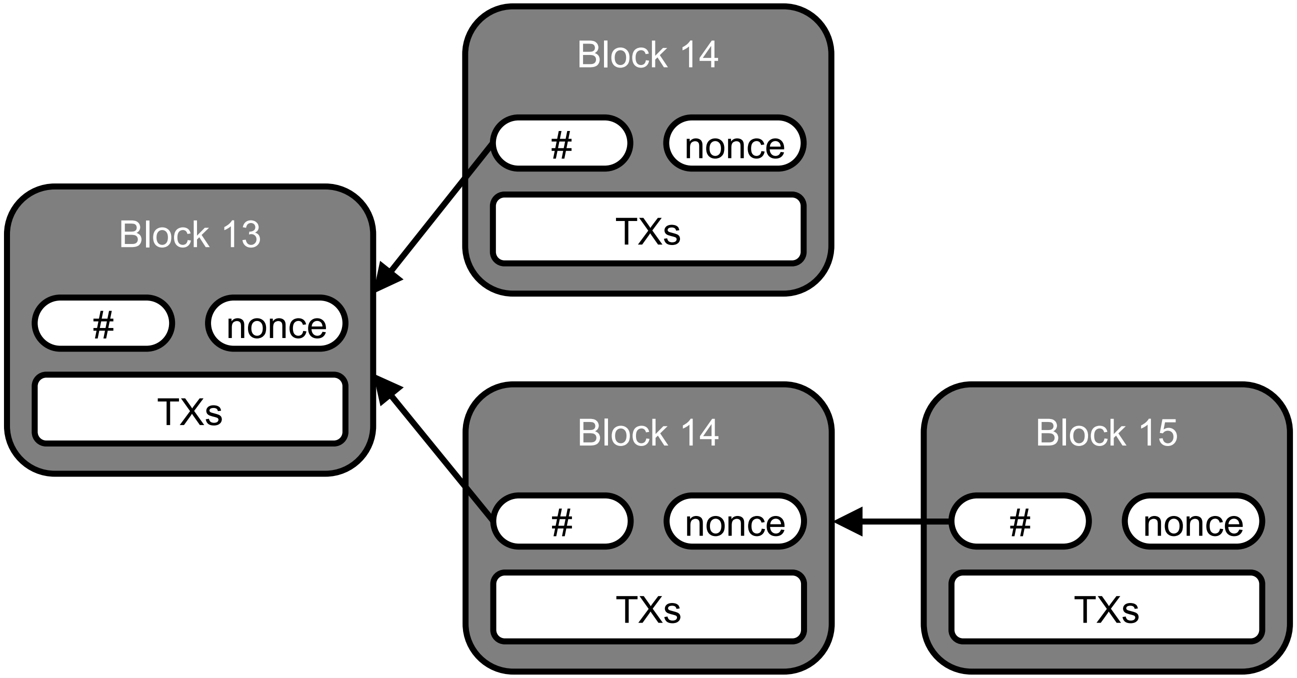
- Proof-of-Work:
  - For a given target:
  - Find a *nonce*, so that



$$\mathit{hash}(\ \mathrm{hash}(\mathit{block}_{n-1}) \oplus \mathrm{transactions} \oplus \mathrm{nonce}) < \mathrm{target}$$

➔ whoever is the first to find an appropriate nonce gets to propose the next block

Double Spends



once a week happens for block chain

stores want to find chain with 6 blocks since it's very highly unlikely to happen with 6 blocks

As as long 50% server are behaving we are good comparing to byzantain where we need 2/3

Nakamoto-Consensus

Voting is not explicit but implicit by signing blocks

→ voting weight proportional to computer performance

Overview

- Intro
- Partitioning
- Replication and Consistency
- CAP Theorem
- Case Studies
  - Kubernetes Auto-Scaling
  - Amazon Dynamo
  - Blockchain

Summary

- Scaling-out to more virtual resources: scalability and fault tolerance
  - Load balancing for replicated stateless components
  - Load balancing *and* data consistency models for replicated stateful components

- The higher the consistency level, the less scalable replicated services are
- System components fail eventually, decide on either availability or consistency

Literature and References

- Literature
  - A.S. Tannenbaum, M. Van Steen: “Distributed Systems: Principles and Paradigms”, Prentice Hall, 2016, Chapter 7
  - M. Kleppmann, “Designing Data-Intensive Applications”, 2017, Chapter 5 and 6

- References

[2] Eric. A. Brewer: “Towards Robust Distributed Systems”, PODC Keynote 2004, http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[3] S. Gilbert, N. Lynch: “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”, ACM SIGACT News, 33 (2), 2002

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels: “Dynamo: Amazon's Highly Available Key-Value Store”, in Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles, 2007

[6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin: “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”, in Proc. of the 29th ACM Symposium on Theory of Computing, 1997