CLOUD COMPUTING

Coordination & Serverless computing

Lorenzo Leonini
lorenzo.leonini@unine.ch

Objectives

- Coordination services
- PaaS
- Serverless computing
- AWS

Coordination services

Indroduction

- Many distributed computer systems require some form of coordination
  - Agree on a parameter value
  - Decide which component is in charge of some processing
  - Know the processes that participate to a collective task
  - Lock access to a specific resource
  - … and many more

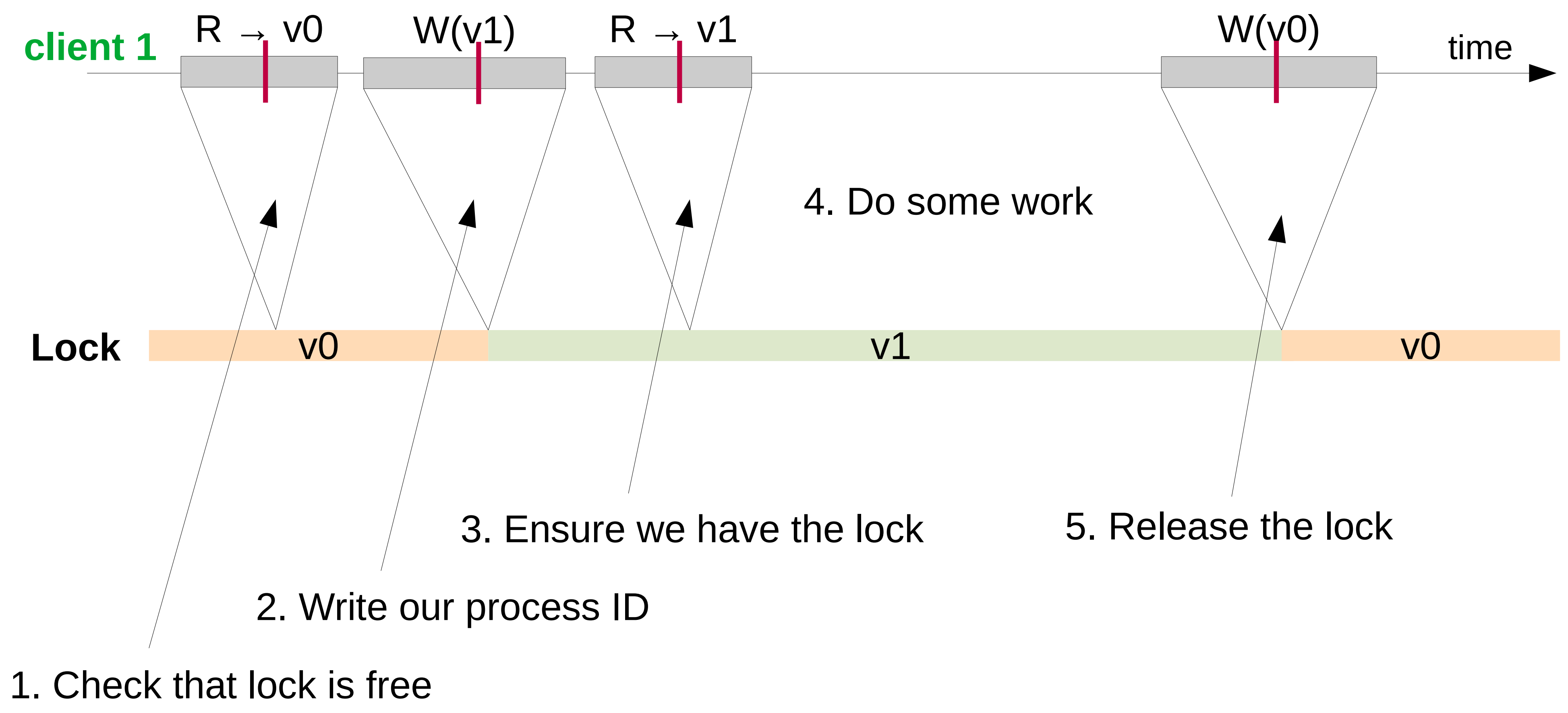
Coordination is difficult

- Many fallacies to stumble upon
  - The network is not reliable
  - The latency is not known in advance
  - The bandwidth is finite
  - There are multiple administrators
  - Difficult to know which node is alive, or not
- Many impossibilities results (FLP, CAP, ...)
- Several requirements across applications
  - Duplicating is bad, duplicating poorly is even worse

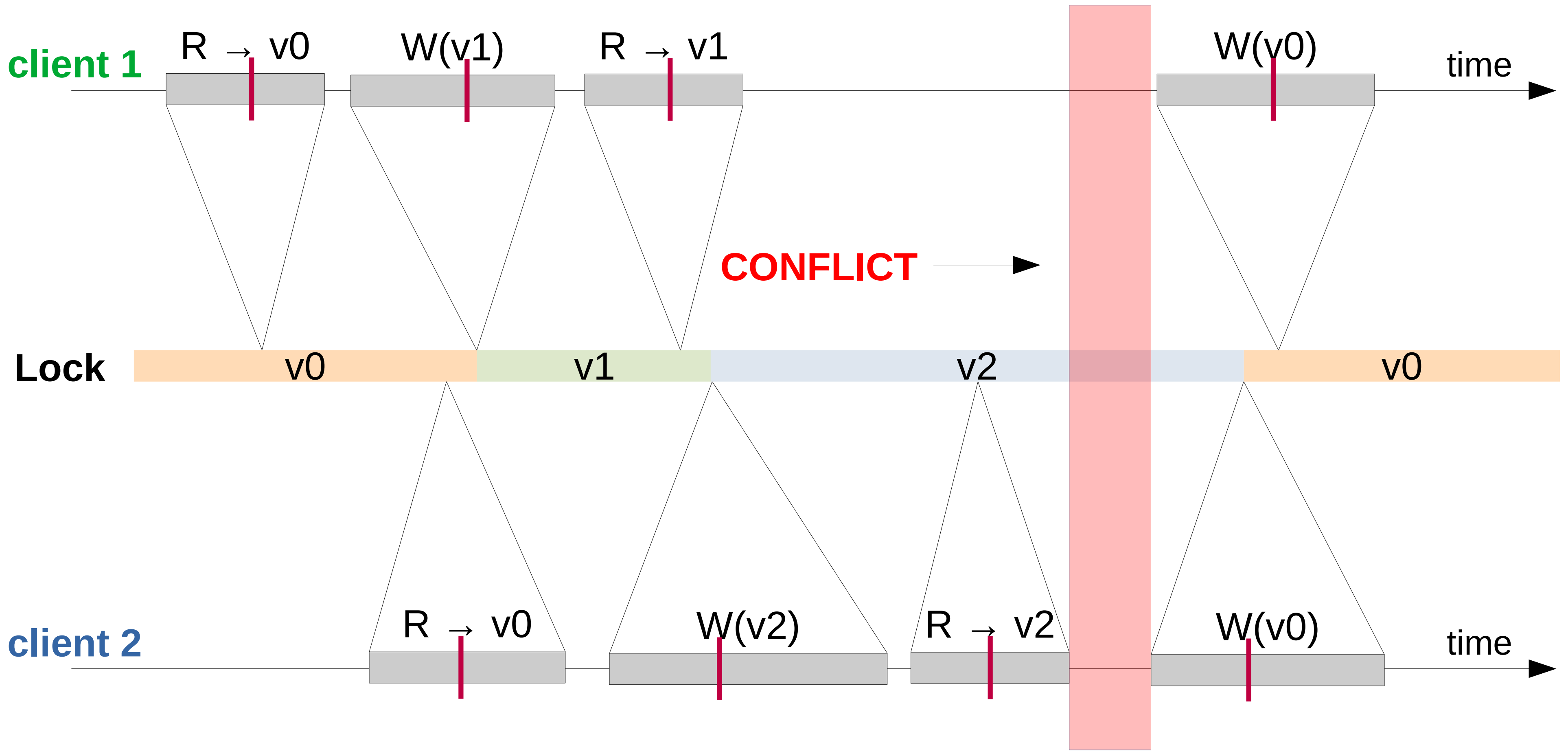
Example: Implementing mutual exclusion

- A groups of computers are running in parallel
  - They all have a unique ID
- They must exclusively access an external resource
- How to implement a *distributed lock* ?
  - We could use a NoSQL key/value store providing **linearizable consistency** !

Linearizable consistency



Linearizable consistency



Implementing mutual exclusion

- Strong consistency model is not sufficient !
  - How are implemented real locks ?

```c
int lock(int *lock)
{
    while (test_and_set(lock) == 1);
}
```

```c
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

- Using atomic "check & set" operations !

Coordination is difficult

- Locks
- Queues
- Leader election
- Group membership
- Configuration
- PubSub
- …

All require the knowledge of specific, complex protocols or systems

All are difficult to get right and bug-free

Common problems faced by many engineers.
Ad hoc solutions = lots of duplicated effort

- Coordination service: Implement it once and well
  - Following: examples of specific coordination services

Specific coordination services (1)

- **Mutual exclusion**
  - A resource must be accessed by only one node at a time
    - Mutual exclusion (mutex)
    - A lock associated to shared resource(s)
  - Must take the *lock* before access, and release the *lock* after access
- Dedicated service:
  - *Google's Chubby*

Specific coordination services (2)

- **Configuration**
  - List of operational parameters
    - Example: `IP:port` addresses of the servers supporting a given functionality
  - Static ....
    - Set by a master before starting application
  - ... or dynamic
    - Changes must be seen be all nodes in a consistent, comprehensive and fault-tolerant manner
- Dedicated service:
  - *Akamai's ACMS (Configuration Management System)*

Specific coordination services (3)

- **Queuing**
  - Processes must agree on an order
    - Order of messages to process in a redundant processing system
    - Order of modifications to apply to a shared replicated state
    - Order of processes for accessing to a resource
- Dedicated service:
  - *Amazon's Simple Queue Service (SQS)*

Specific coordination services (4)

- **Group membership and leader election**
  - Need to know which others nodes are alive
  - Nodes need to decide which *single* node takes charge of a particular operation (leader)
- Dedicated service:
  - *Microsoft's Centrifuge*

Specific coordination services (5)

- **Locks / mutual exclusion**
  - *Google's Chubby*
- **Configuration**
  - *Akamai's ACMS (Configuration Management System)*
- **Queuing**
  - *Amazon's Simple Queue Service (SQS)*
- **Group membership and leader election**
  - *Microsoft's Centrifuge*
- All services target one particular synchronization problem

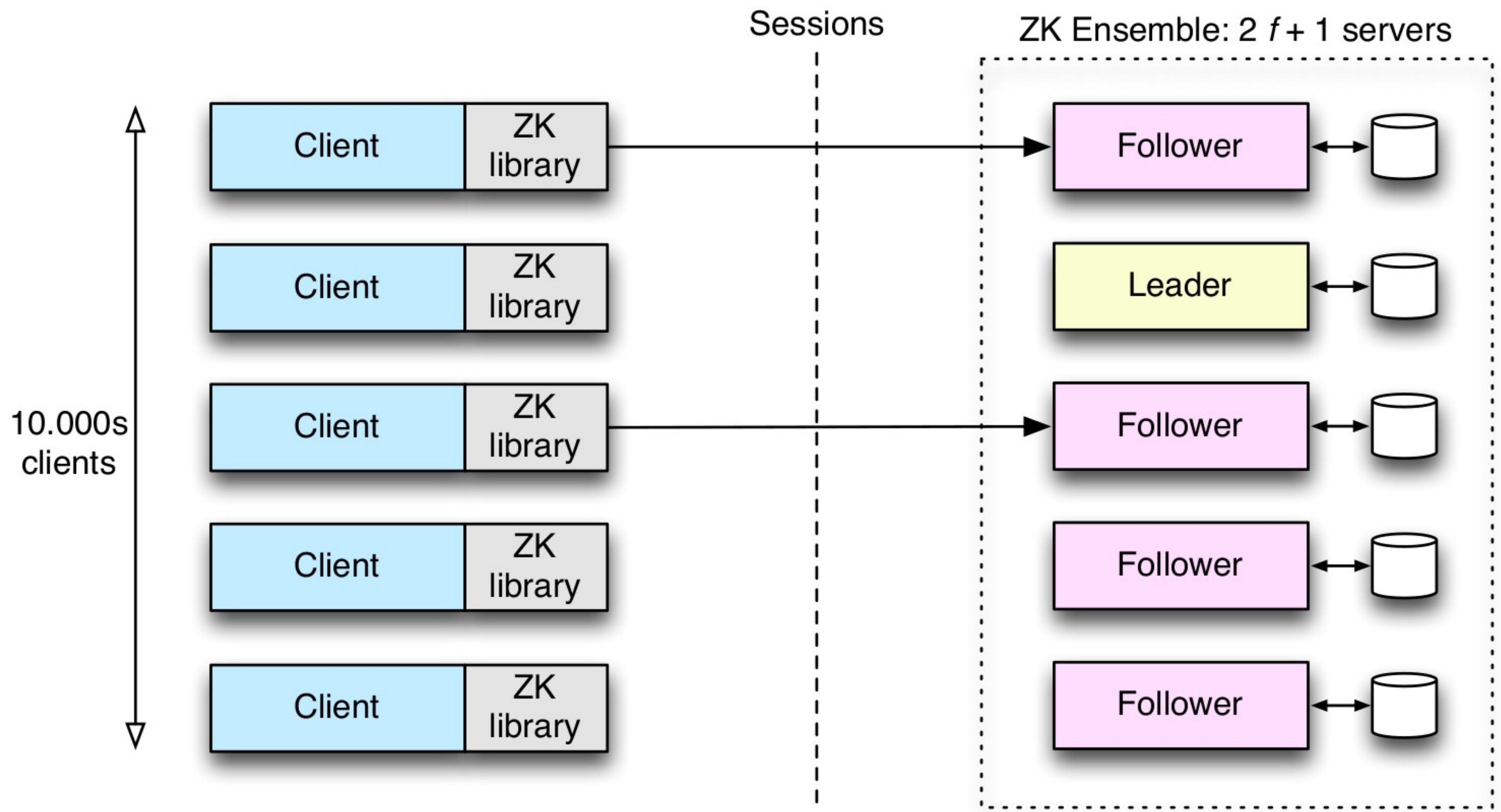
Universal coordination service

- Using different coordination services for different coordination primitives remains complex
- Can we build a generic or universal coordination service?
  - Also called a coordination *kernel*
  - Allows building advanced coordination mechanisms using a combination of simple primitives with associated guarantees

Apache ZooKeeper

- Proposed by Yahoo! as part of Hadoop
  - Now an Apache project
  - Written in Java
- A Coordination kernel
  - Provides low-level synchronization primitives at server side
  - Can be used at client side to build complex primitives
- Used in production at many companies, inc. Yahoo!
  - Crawling engine (called fetching service)
  - Hadoop Map/Reduce
  - Yahoo! message broker and HedWig (messaging systems)
  - ...

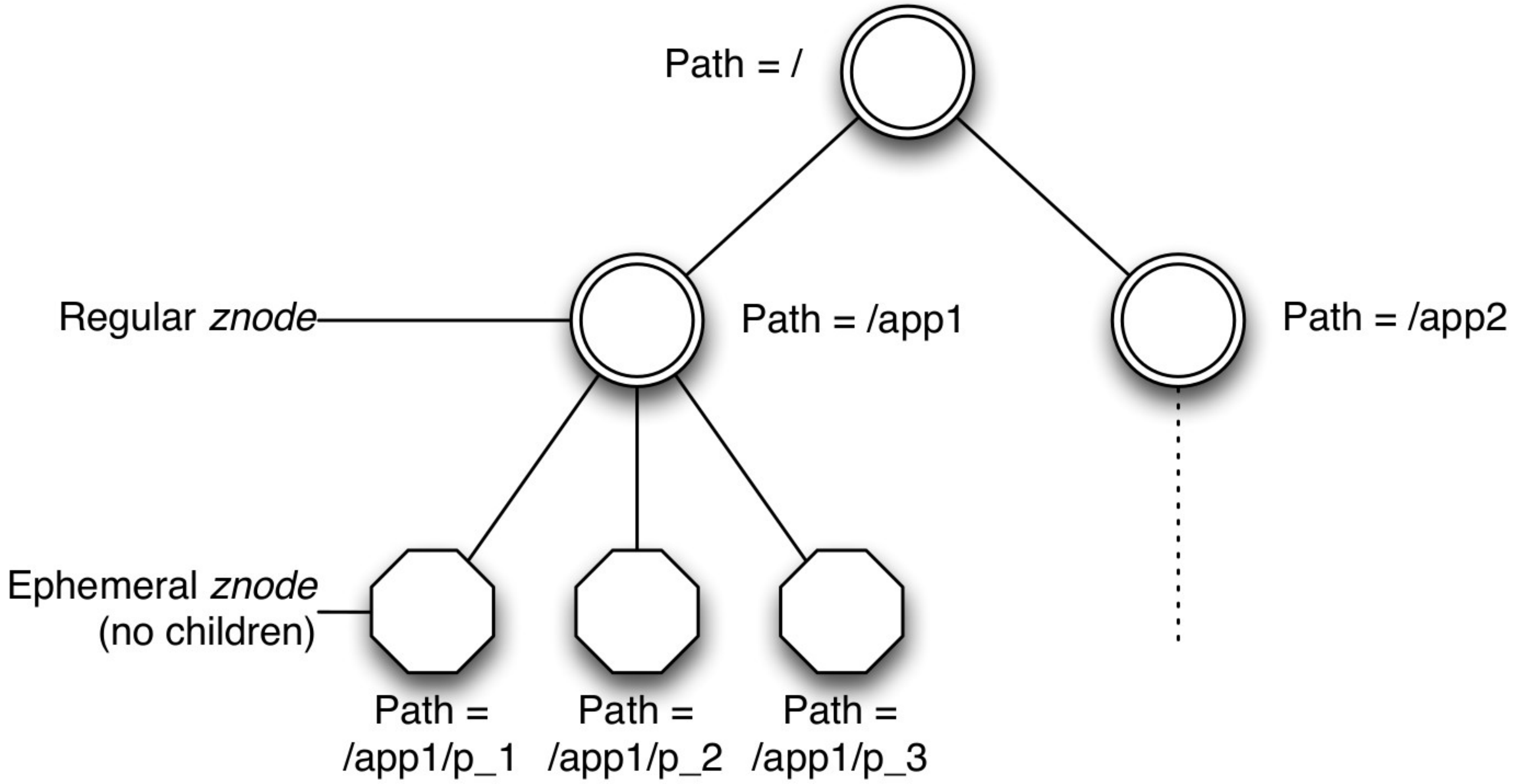
Zookeeper - The big picture



ZooKeeper data model

- Session-based semantics
  - Each client permanently connected to one of the ZK servers
  - Each client monitored for faults (with *keep-alive* messages)
- Can create/delete/update shared *znodes*
  - *znodes* can carry data if necessary
  - *znodes* organized hierarchically, like a file system
  - Two types of *znodes*
    - Regular: persistent
      - + can have children *znodes*
    - Ephemeral: disappear if creator’s session ends
      - no children *znodes*
- Coordination implemented by reading and writing *znodes*

ZooKeeper data tree



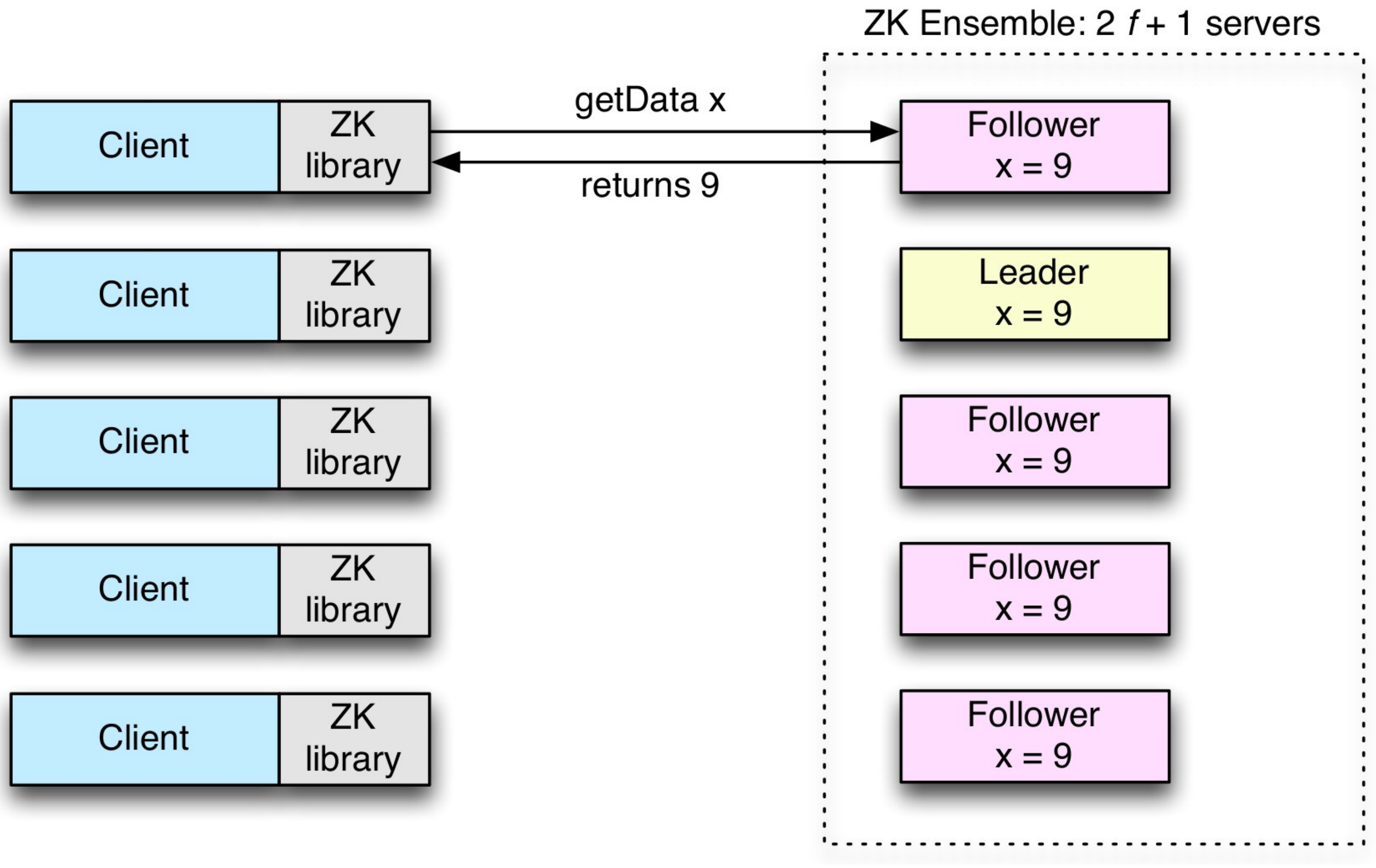
ZooKeeper - Watches and notifications

- Setting the watch flag allows the client to be notified upon the first modification to the znode
  - Modification to its value
  - Deletion
    - In particular, if session ends for an ephemeral node
  - Addition or deletion of child node if not ephemeral
- A watch is a one-time trigger
  - If the client wishes to continue monitoring changes to the znode, must re-read and set the watch at the same time

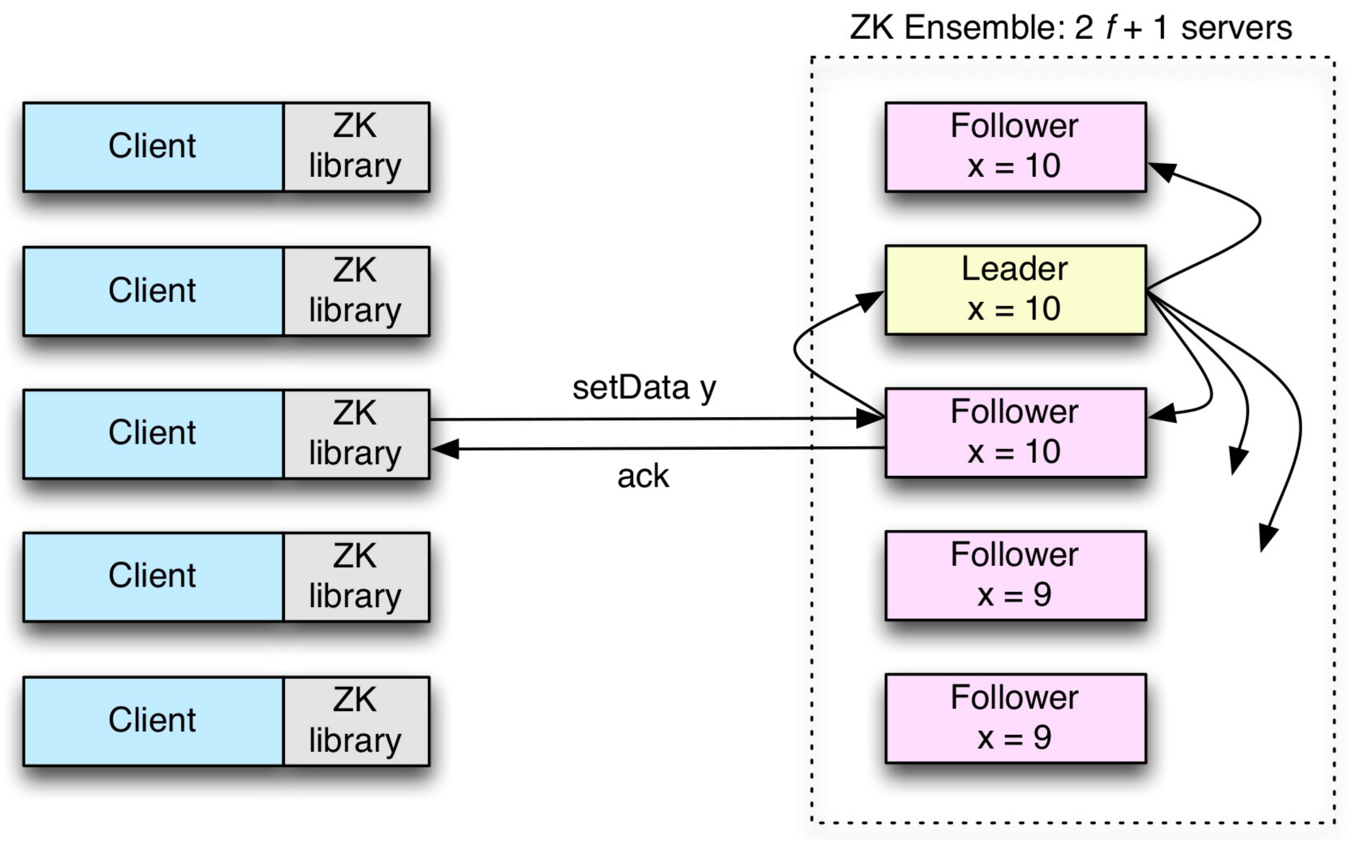
ZooKeeper consistency model

- FIFO per-client evaluation order
  - TCP channel between client and one ZK server
- Linearizable writes
  - Writes happens atomically during invocation - real time ordering
- Reads are not linearized
  - Can be served by any servers
    - Sequential consistency
  - *sync* option to also be linearized
- Ordering guarantee on notification
  - A client always receives the notification of change before it sees the new state after the change

ZooKeeper reads



ZooKeeper writes

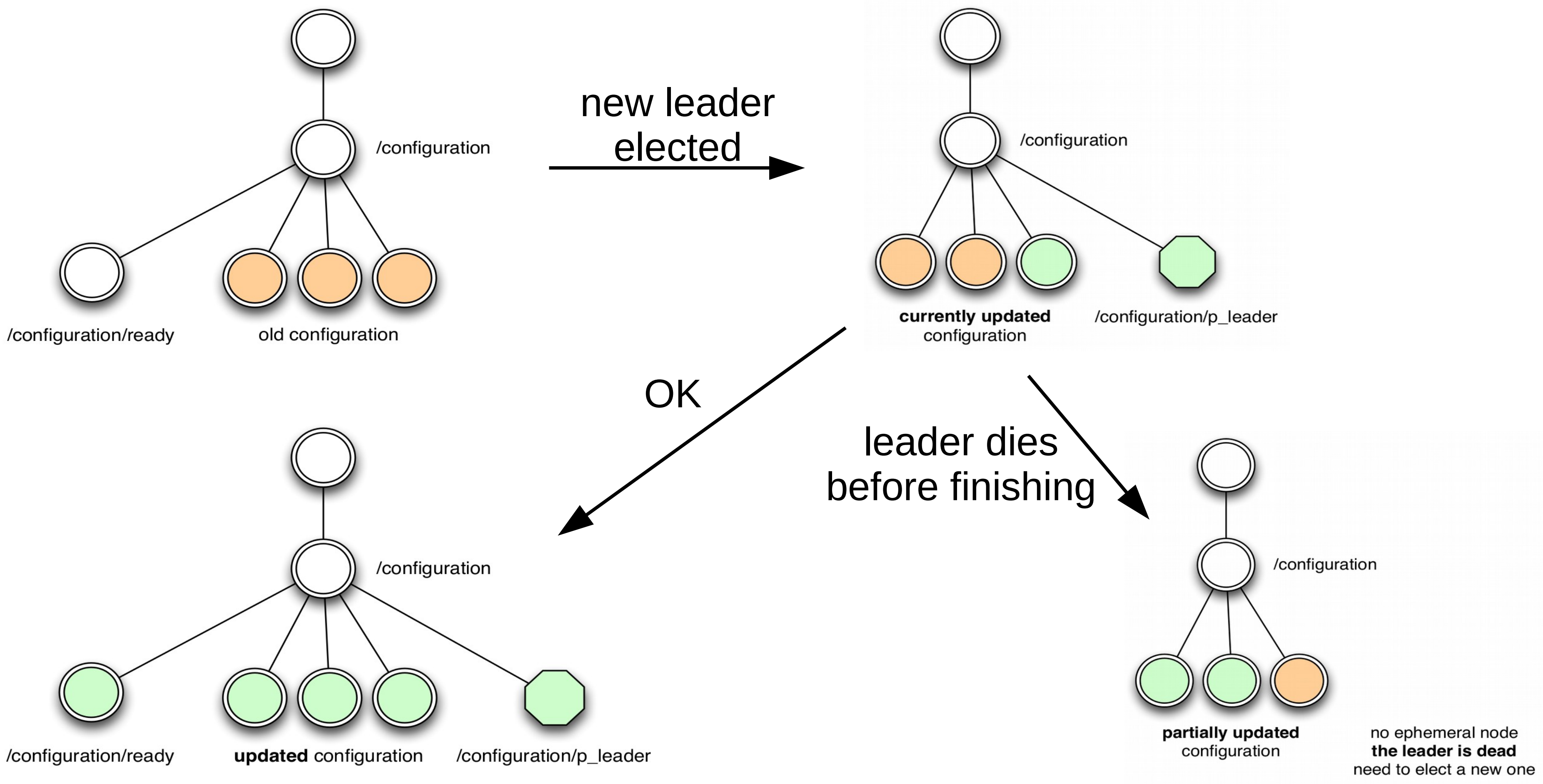


Write is acknowledged after a majority of replicas are updated !

ZooKeeper - Implementing shared configuration (1)

- Scenario: a set of client processes have elected a new leader
  - The leader must update the common configuration
  - We do not want other processes to use the configuration while it is being changed
  - If the new leader crashes before finishing, we do not want a process to use this partial configuration
- Znode "ready"
  - Deleted by the leader before changing parameters
  - Recreated by the leader after all changes sent
- Ephemeral znode containing the leader identity and created by the leader
  - Allows other nodes to know if the leader is still alive
  - Automatically deleted if the leader process fails
    - Can trigger the election of a new leader

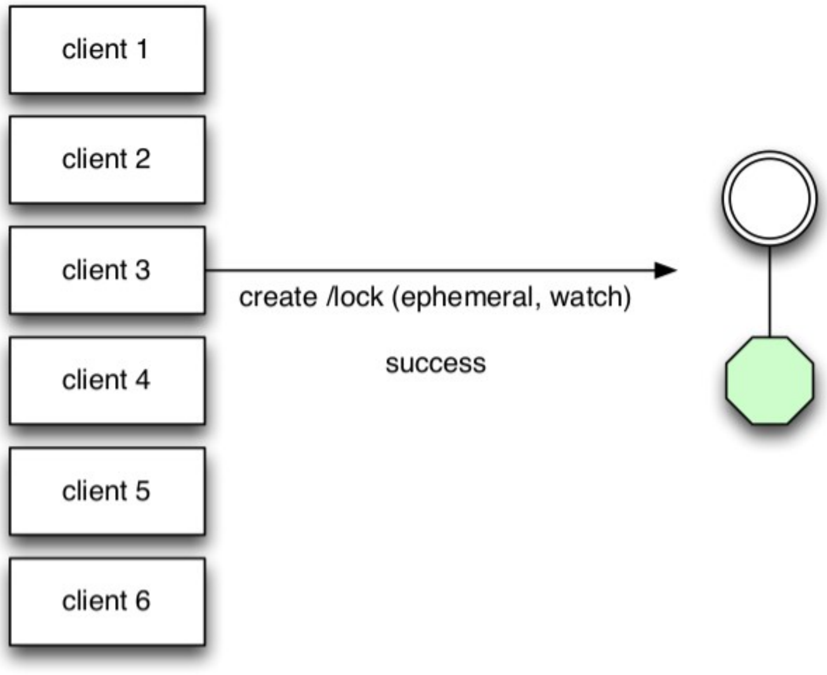
ZooKeeper - Implementing shared configuration (2)

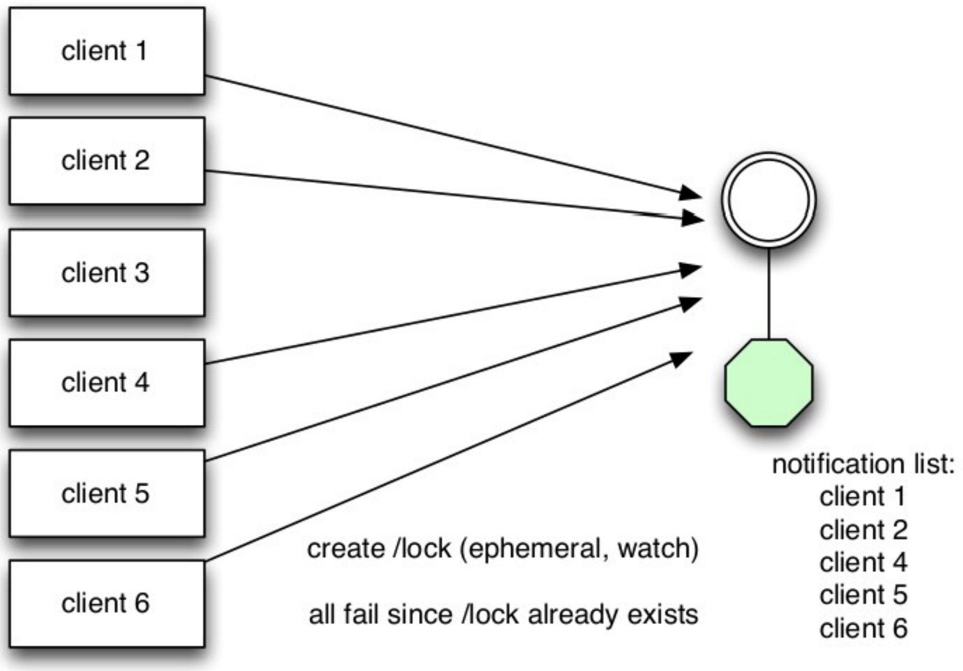


ZooKeeper - Implementing mutual exclusion (1)

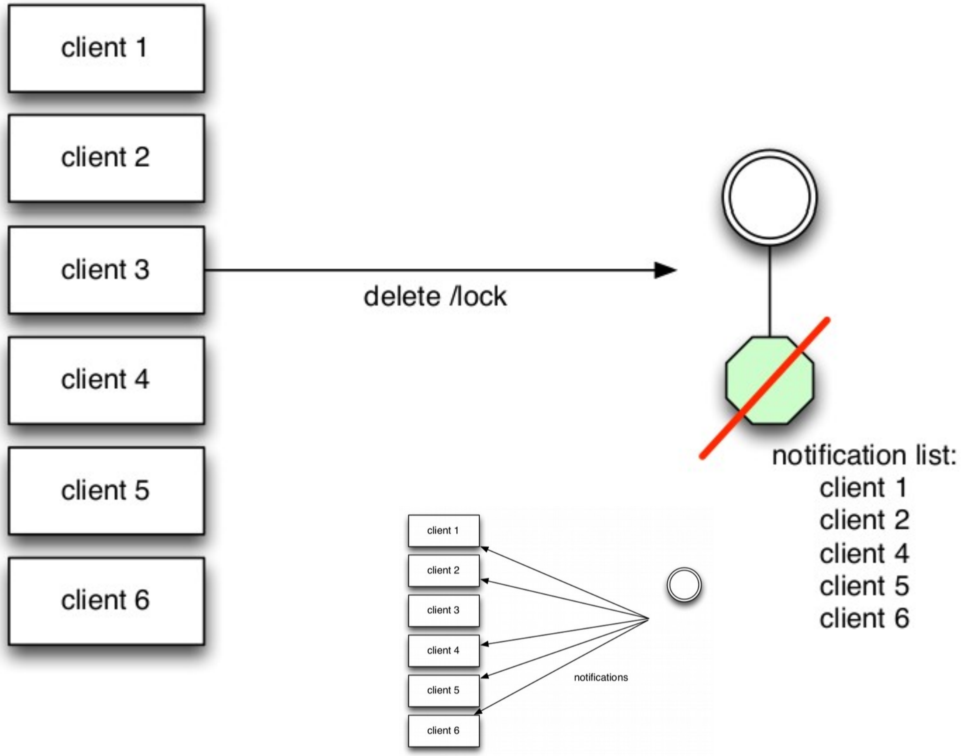
- Locks can be implemented on top of wait-free synchronization
- Simple version: create an ephemeral node named `/app_1/lock`, with the *watch* attribute set
  - *success*: hold the lock (file did not exist)
  - *failure*: wait for notification and retry taking the lock
- Problem: subject to the *herd effect*
  - When lock is released, all clients notified and all try to create the lock
  - Peak of requests, and only one succeeds

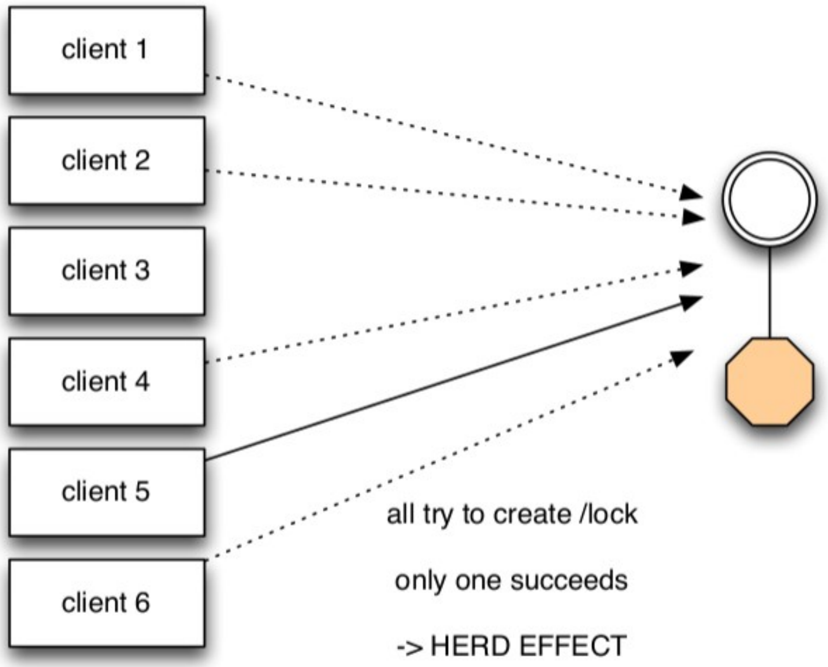
ZooKeeper - Implementing mutual exclusion (2)





ZooKeeper - Implementing mutual exclusion (3)

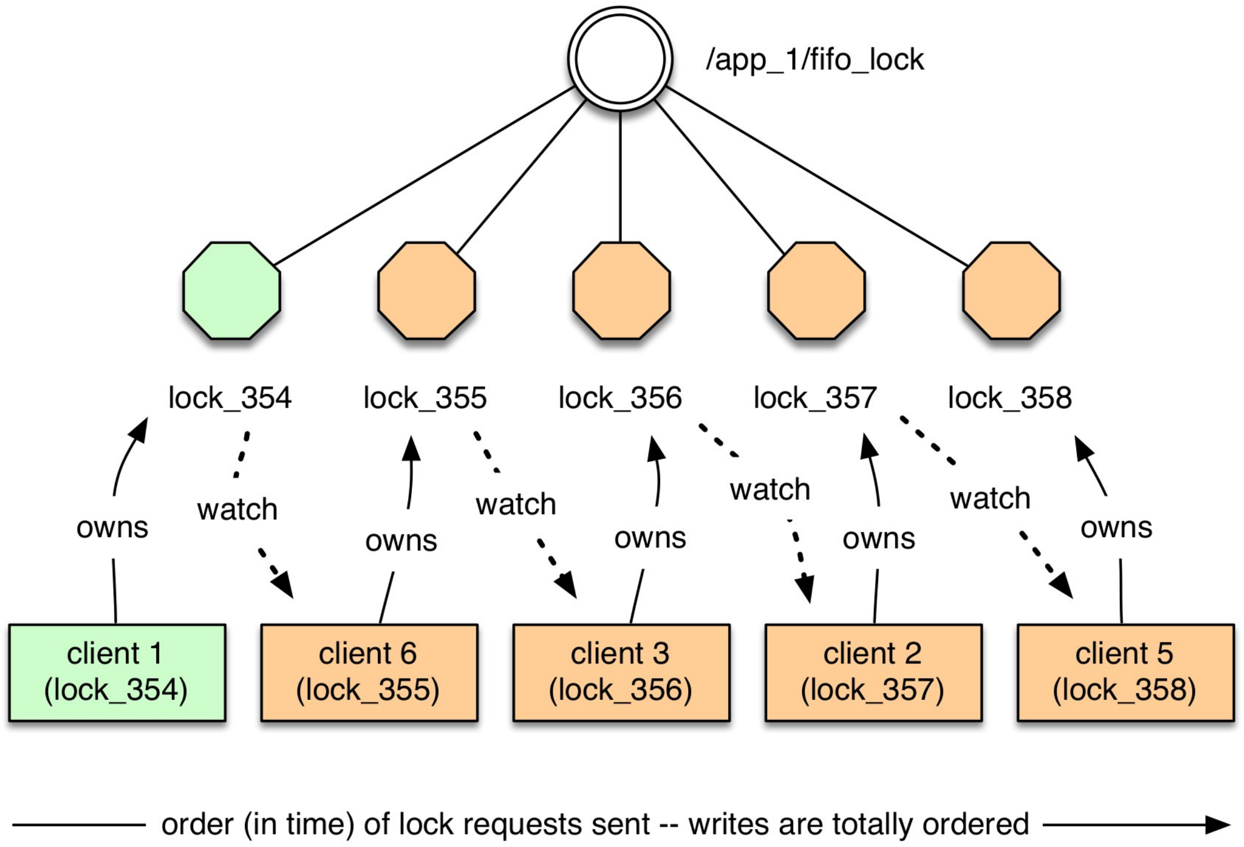




Solution to the herd effect (1)

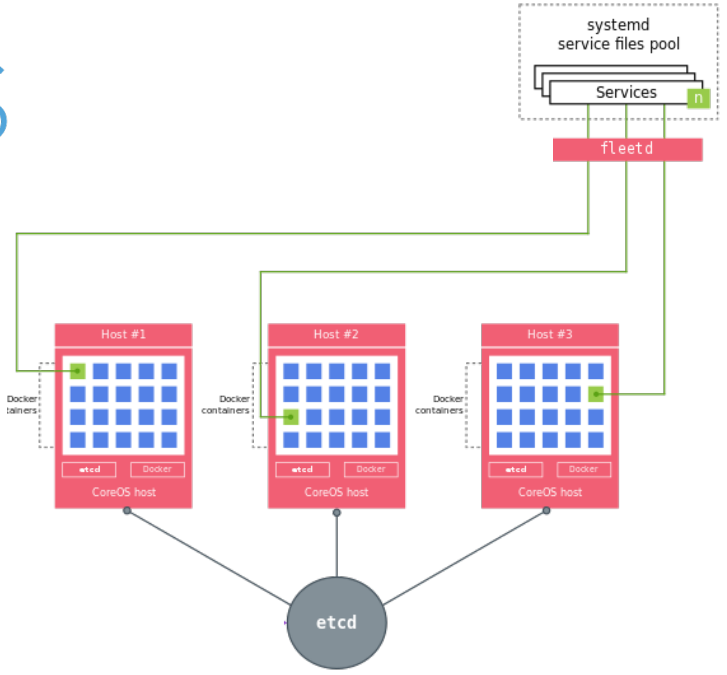
- Each node creates its own lock *znode*
  - With ephemeral and sequential flags set
- Only effectively hold lock when no *znode* of lower identifiers
  - To know that, set a watch on the *znode* of highest identifier before itself
- Only one client notified when a lock *znode* is deleted
  - Lock granted in order of requests
  - Also notified if owner of the ephemeral *znodes* dies

Solution to the herd effect (2)



CoreOS

- CoreOS is an operating system for clusters
  - Notable projects from CoreOS
    - Container Linux
      - Compact Linux distribution to run containers on bare metals clusters or VMs
      - Updates on a separate partitions (ability to rollback)
    - rkt
      - Container execution environment
    - etcd
      - Analogy to /etc (where UNIX stores its configuration file) and *d* for distributed
    - fleetd
      - Manage containers → now officialy replaced by kubernetes
- CoreOS bought by Redhat early 2018



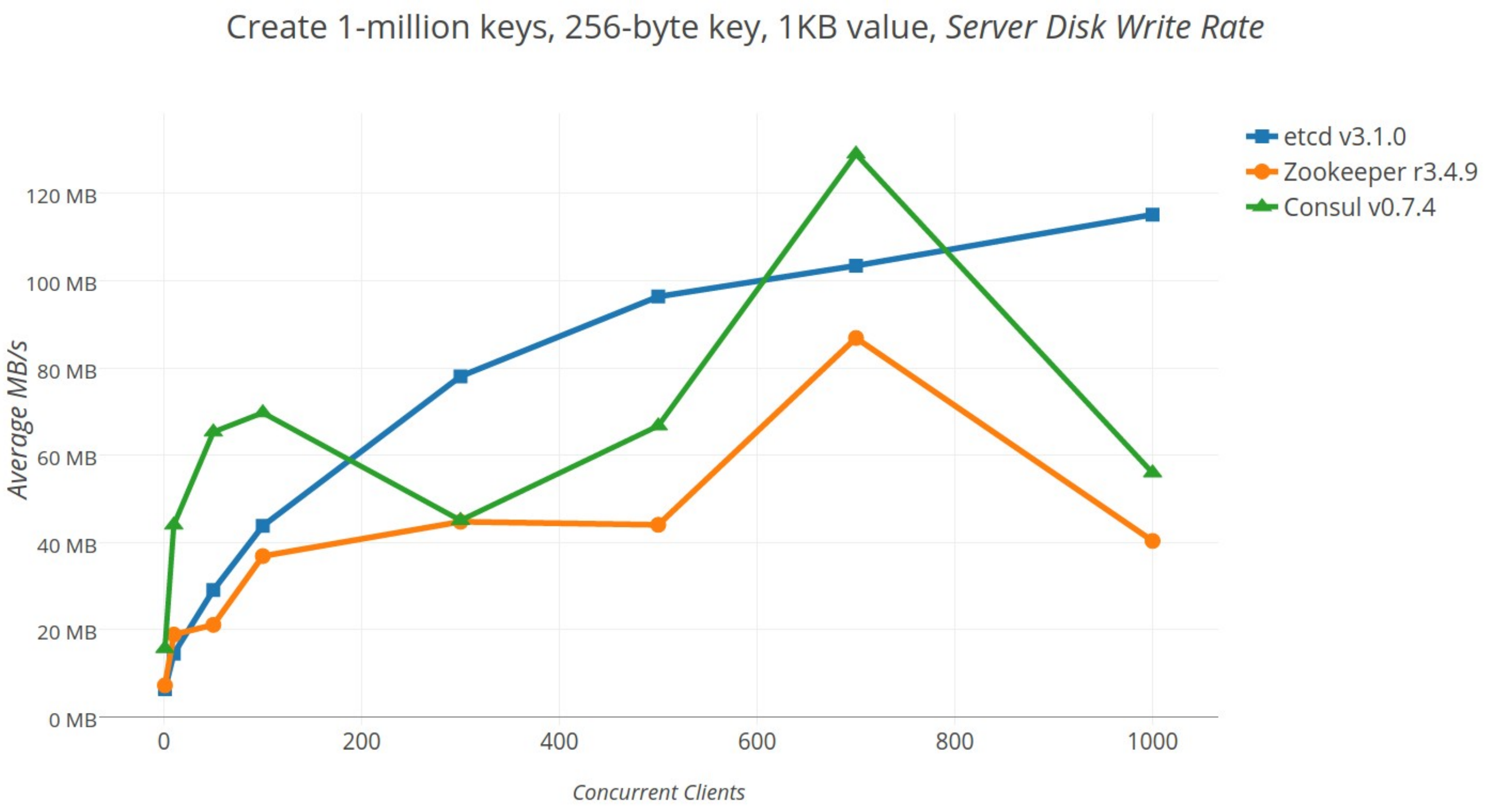
etcd

- Written in Go
  - Designed for performances
    - Do not suffer of some performances problems of the JVM (garbage collection)
    - Also more efficient than ZooKeeper for snapshoting (saving in memory data to FS)
- Key/Value oriented
  - Revisions based on a global clock (increment at each changes)
  - Ranges
  - Transactions
  - Watch
    - Event
    - Stream

etcd vs ZooKeeper

	ETCD	ZOOKEEPER
Concurrency Primitives	Lock RPCs, Election RPCs, command line locks, command line elections, recipes in go	External curator recipes in Java
Linearizable Reads	Yes	No
Multi-version Concurrency Control	Yes	No
Transactions	Field compares, Read, Write	Version checks, Write
Change Notification	Historical and current key intervals	Current keys and directories
User permissions	Role based	ACLs
HTTP/JSON API	Yes	No
Membership Reconfiguration	Yes	>3.5.0
Maximum reliable database size	Several gigabytes	Hundreds of megabytes (sometimes several gigabytes)

etcd vs ZooKeeper - Performances



https://coreos.com/blog/performance-of-etcd.html

Coordination services in cloud

- Group membership
- Leader election
- Configuration
- Service discovery
- Distributed synchronization
  - Locks / Semaphores
  - Queues
  - PubSub

Behind the scene used to implement all cloud services

One can use an SQL DB (ACID) to implement that !

Some are also provided as-a-service

Intermediate conclusion

- Coordination is often required in cloud systems
  - Big Data processing, deployment management, fault handling, etc.
  - Often requires mastering complex distributed systems concepts
    - Application developers may end up making difficult-to-detect errors
- Most cloud services use coordination kernels behind the scene
  - Complexity of coordination is hidden by using these high-level services
  - Also, a few services provide some coordination primitives, like queues, to the user
  - In case where more advanced coordination is needed, the user has to deploy a dedicated solution like ZooKeeper or etcd

PaaS

Platform-as-a-Service

- A cloud service model providing developers with a managed environment where they can create and deploy applications
  - Often Web-based or mobile applications, but not only
  - Relief from administration and maintenance aspects
    - No need to install and maintain OS, database server, web server, etc. by hand (this is just overhead for developer)
  - Less flexibility than IaaS
    - Must accommodate options offered by PaaS provider
  - Greater vendor lock-in than with IaaS
    - Easier to relocate VMs or containers between IaaS providers than code targeted at a specific PaaS platform

Blurring the line between IaaS and PaaS

- Several providers are providing a combination of IaaS with PaaS features (or reversely)
  - Windows Azure was initially pure PaaS, to which IaaS features and unmanaged environments (VMs & containers) were added later
  - Amazon Web Services (AWS) was initially pure IaaS; PaaS-like managed features added later (storage, elasticity, etc.)
  - Red Hat's OpenShift is a managed environment (PaaS) but lets the developer decide on some aspects of the environment and configuration of the OS

Some representative players in the PaaS market

heroku

App Engine

Elastic Beanstalk

Engine Yard™

af appfog

OPENSHIFT

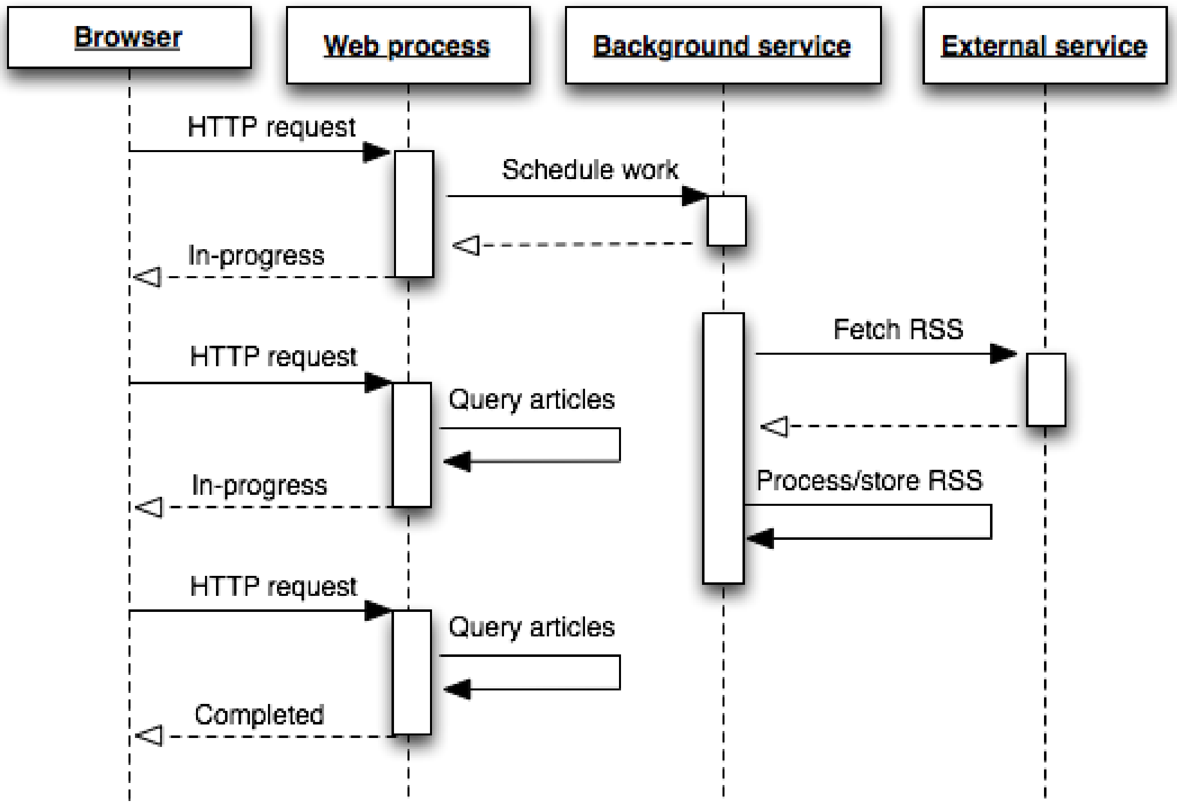
Comparing PaaS offerings

- More diversity than for IaaS as provided services are more high-level and allow more variety and specialization
  - Programming language(s) supported
  - Support for specific frameworks (RoR, …)
  - Support for background processes, CRON tasks
  - Fixed or extensible set of libraries
  - Data storage options
    - File-based, managed database, document stores, …
  - Communication options
    - Message queues, support for mobile clients, …
  - Support for development tools
    - Integration with IDE (Visual Studio, Eclipse, IntelliJ, …)
    - Integration with source code management tools (git)
    - Deployment facilities
  - Level of support (community versus dedicated personnel, 24/7, etc.)
  - Costs

Heroku (1)

- One of the first PaaS (2007), running on AWS
  - Now owned by Salesforce
- Started as a platform for Ruby developers (Ruby on Rails)
  - Now supports Python, Java, Scala, Clojure, Node.js, Go and PHP
  - "Twelve Factor Apps" methodology and best practices for scalable Web applications, see http://12factor.net/
- Application structured as dynos, which are abstract computing components
  - **Web dynos**: respond to HTTP requests
  - **Worker dynos**: execute task requests from a task queue
- Transfer computationally intensive tasks outside the Web layer and the user request/response cycle

Heroku (2)



https://devcenter.heroku.com/articles/background-jobs-queueing

Heroku (3)

- Heroku is tightly integrated with the git source code management tool
  - And in particular with the github service
  - Ability to ‘push’ an application to Heroku directly using github
- Manages dependencies automatically and provides features for agile deployment and testing directly in the cloud
- **PostgreSQL** database (only)
- Third-party add-ons for other services (including other storage) available
- Deployment and management via CLI: *heroku*
  - Scalabily managed via command line (autoscale available for advanced dynos)
    - However, scalability still limited by the DB

Google App Engine (1)

- Completely managed environment
  - Application runtime, libraries, OS, development tools
  - Target: distributed Web applications
  - Language version and libraries are limited to Google’s choices
    - Java, Python, PHP, Go and Node.js
- Guaranteed scalability when following GAE "playbook"
  - Target is applications with low response time, split in small and stateless components
- Deployment by CLI *gcloud*
  - Zero downtime

Google App Engine (2)

- Sandboxing from execution environment
  - No direct interaction with OS
  - Code only runs when handling a Web request or when scheduled as a cron job (i.e., scheduled at a specific time)
  - Must respond within 60 seconds
- Encourages stateless and simple components
  - No access to a file system
  - Communication using Web interfaces or platform-provided channels
  - Managed scalable database services (**BigTable**)
- Access to Google services
  - Authentication using Google accounts
  - Sending emails, APIs for iOS/Android, Google search, ...
- Used by large apps: Rovio (Angry birds!), Sony Music, Ubisoft, ...

AWS Elastic Beanstalk (1)

- AWS PaaS solution (2011)
- Built on top of standard AWS components
  - EC2: computing instances
  - RDS: **MySQL** or **PostgreSQL** DB
  - VPC / ELB / Cloudwatch / autoscaling groups
- Application is deployed on EC2 using familiar software stacks such as the Apache HTTP Server for Node.js, PHP and Python, Passenger for Ruby, IIS 7.5 for .NET, and Apache Tomcat for Java
- Elastic Beanstalk can be seen as a high level layer over low level components
  - Simplifies the setup, like a "wizard"

AWS Elastic Beanstalk (2)

- No specific charges for Elastic Beanstalk
  - You pay only for the AWS resources needed to store and run your applications
  - You can select EC2 and RDS DB instances size
  - Auto-scaling settings
    - Default based on bandwith
- Complete control ("open the hood")
  - You see all instanciated resources
    - Example: connect with SSH on EC2
- Alternative
  - AWS CloudFormation
    - Declarative templates to deploy complex architectures

PaaS pricing

- Different pricing models
  - Based on instances size (memory/cpu)
    - Can fluctuate if auto-scaling
  - Persistent storage
  - Network traffic
  - ...
- Different free tiers
  - AWS
    - Free tier depending of EC2 and RDS sizes
  - Heroku
    - Sleeping dynos !
- Not easy to compare !
  - Google let you set a spending limit / day !

Intermediate conclusion

- These are just a few of the many PaaS offerings available
  - Some target Web development, some target enterprise software (Red Hat, SAP, IBM, etc.)
  - Environments can differ a lot:
    - A wise choice is necessary before starting coding
    - Risk of vendor lock-in if solutions appear to be inappropriate
    - May create strong dependency on PaaS provider (sustainability, reliability, application and traffic growth)
- PaaS relieves developers from the overheads of maintaining and administrating a complex production deployment environments
  - Ideal (and often used) for creating a Web or mobile startup
- Containers were not existing…

Serverless Computing

Serverless computing

- Cloud computing execution model
  - Execute code in reaction to **events**
  - Specialization from PaaS to **FaaS** (Function-as-a-Service)
- The provider manages all aspects of resources allocation
  - True **NoOps** platform !
- Price is based on the actual amount of resources used
  - For each individual calls
- Can work complementary to code deployed in traditional styles like microservices
  - But can also work completely standalone

Serverless advantages

- Cost
  - Pay-as-you-go model, no fees for idle time
  - Very effective, for light, inconstant loads
- Operations
  - All maintenance aspects are managed by the provider including fault tolerance
- Scalability
  - Scaling is also completely managed by the provider
- Productivity
  - As code is written as simple functions, the developper has not to worry about multithreading aspects

Serverless disadvantages (1)

- Resources limits
  - This model is not well suited for high performance computing
    - Total resources are limited by the provider
    - Cannot use shared memory
- Performances
  - First execution time can be slow due to the need of deploying the code on (new) compute instances and then to execute it
    - This is even more noticable with JVM based environment where startup delay can be important (JIT compiler) and code is optimized later at runtime

Serverless disadvantages (2)

- Vendor lock-in
  - Events from proprietary services
- Security
  - Each component (function) is an entry point
    - Each function has to be associated to specific minimal access rights
- Monitoring and debugging
  - Performance characteristics of the execution environment can defer from local execution
  - Not possible to attach debuggers, profilers, ...

Serverless usecases

- BaaS (Backend-as-a-Service)
  - REST APIs
    - Website
    - Mobile apps
    - IoT
- Data processing
  - Media transformation
    - Converting images (format, resolution, watermark,…)
    - Real time or batch

- Business logic
  - Orchestrating microservices
- Voice interpretation
  - AWS Alexa
- Chat bots
  - Answering queries

AWS Lambda

- AWS solution for **serverless computing**
  - Runs your code in response to **events**
  - Lambda runs your code on high-availability compute infrastructure
  - Automatically manages the underlying compute resources
    - Server, OS maintenance, runtime upgrade
    - Capacity provisioning and automatic scaling
    - Code monitoring and logging
  - **All you need to do is supply the code**

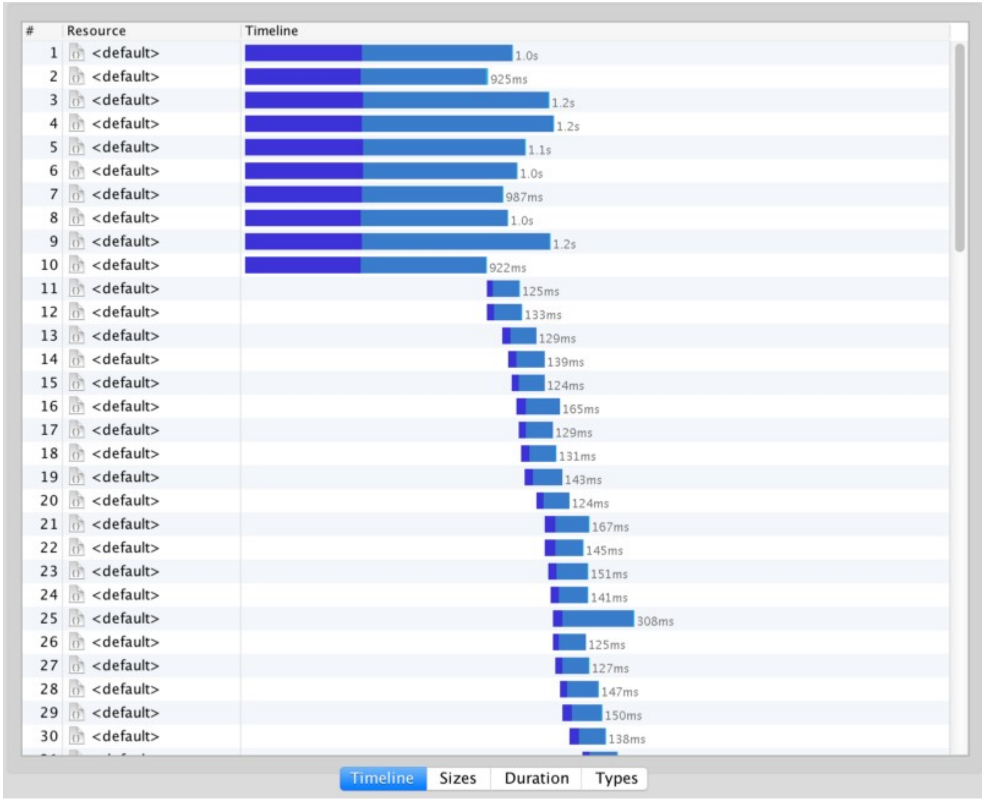
AWS Lambda - Execution

- Programming languages supported
  - Java (and JVM languages), Node.js, C#, Python and Go
- Executed with the AWS identity you provide
- Configurable memory limit and timeout
- 500MB of scratch disk space
- Synchronous or assynchronous operations

- You select the trigger events
  - HTTP requests via Amazon **API Gateway**
  - Modifications to objects in **Amazon S3** buckets
  - Table updates in Amazon **DynamoDB**
  - State transitions in **AWS Step** Functions
  - Stream of Amazon **Kinesis**
  - Messages in **SNS**, **SES** and **SQS**
  - **...**

AWS Lambda - Cold start

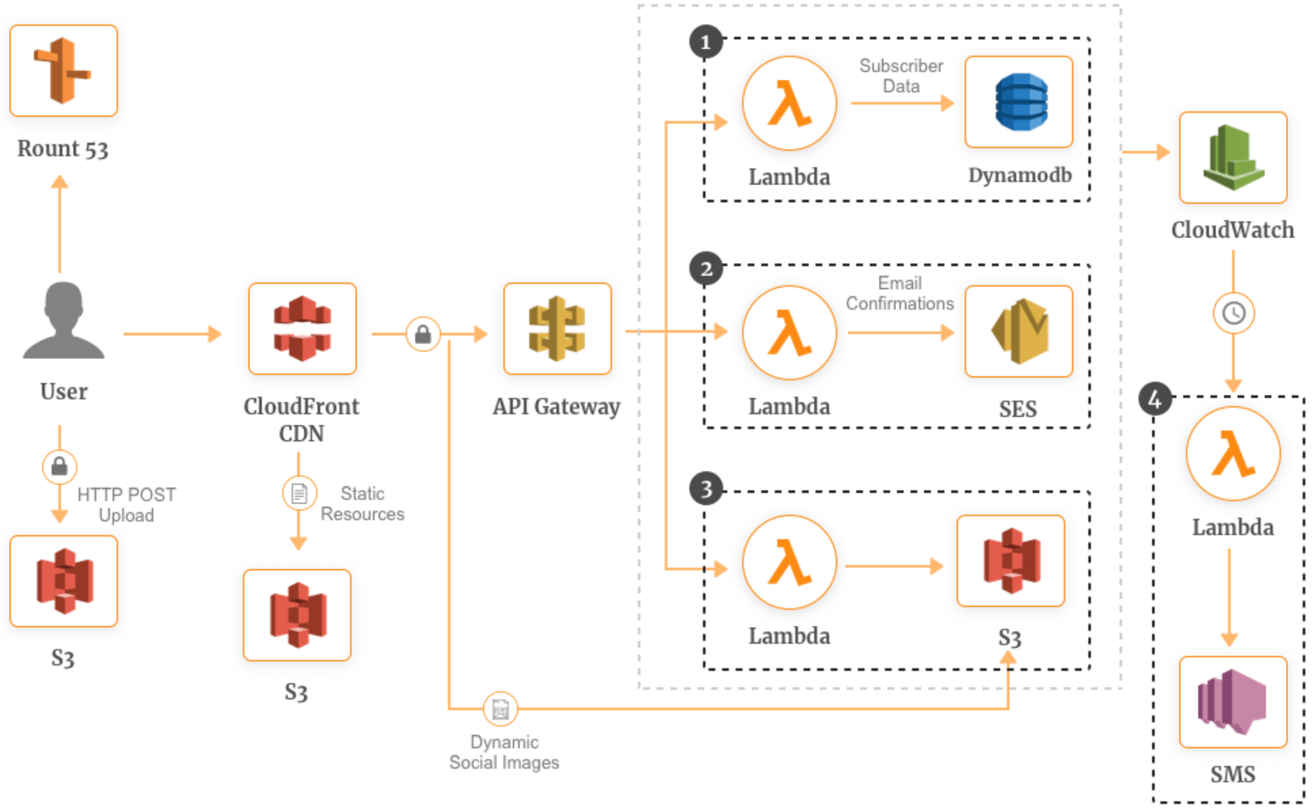
- How is the first request executed ?
  - Need to "deploy" your code in EC2
    - JVM
      - (Possibly) need to start
      - Hotspot optimizations
- Solutions
  - Use languages with fast startup
  - Keep your API warm using "ping" requests that maintain a number of parallel instances up



#	Resource	Timeline
1	<default>	1.0s
2	<default>	925ms
3	<default>	1.2s
4	<default>	1.2s
5	<default>	1.1s
6	<default>	1.0s
7	<default>	987ms
8	<default>	1.0s
9	<default>	1.2s
10	<default>	922ms
11	<default>	125ms
12	<default>	133ms
13	<default>	129ms
14	<default>	139ms
15	<default>	124ms
16	<default>	165ms
17	<default>	129ms
18	<default>	131ms
19	<default>	143ms
20	<default>	124ms
21	<default>	167ms
22	<default>	145ms
23	<default>	151ms
24	<default>	141ms
25	<default>	308ms
26	<default>	125ms
27	<default>	127ms
28	<default>	147ms
29	<default>	150ms
30	<default>	138ms

https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f

AWS Lambda website



1 Capture Data Change 2 Serverless Email System 3 Transform on Upload 4 Scheduled CRON Jobs

https://www.simform.com/serverless-examples-aws-lambda-use-cases/

AWS Lambda - Pricing

- Free (permanent) monthly tier then pay as you go
  - 1M (million) requests then 0.2$/M requests
  - 400'000 GB/s (memory/second) then $0.00001667/GB/s
- Example
  - 8M requests: memory 256M, average run time 0.5s
    - Requests
      - 8'000'000 - 1'000'000 = 7'000'000 * 0.2 = 1.4$
    - Memory
      - 8'000'000 * 0.5 * 256 / 1024 = 1'000'000GB/s
      - 1'000'000 - 400'000 = 600'000 * 0.00001667 = 10$
    - Total: 11.4$

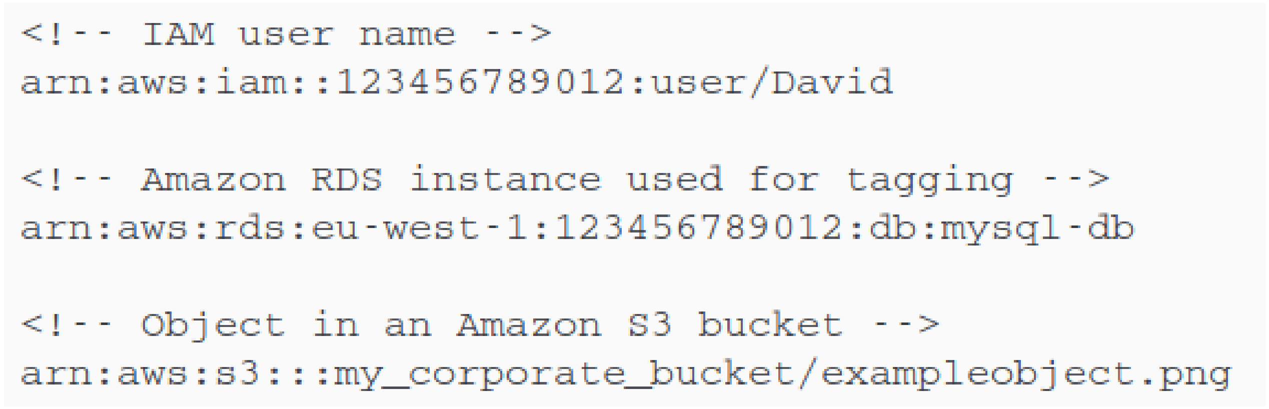
AWS

AWS IAM

- IAM = Identity and Access Management
  - Authentication & Access control
  - Provides MFA (Multi-Factor Authentication)
- IAM roles
  - Set of permissions
  - Can be associated to IAM users, applications or AWS services
- Access control to all AWS resources
  - All resources are provided as a set of functions
    - IAM permits to define which function(s) on witch resource(s) one can call

Amazon Resource Names (ARNs)

- ARNs uniquely identify AWS resources
- Required when you need to specify a resource unambiguously across all of AWS such as in IAM policies
- Formats
  - `arn:partition:service:region:account-id:resource`
  - `arn:partition:service:region:account-id:resourcetype/resource/qualifier`



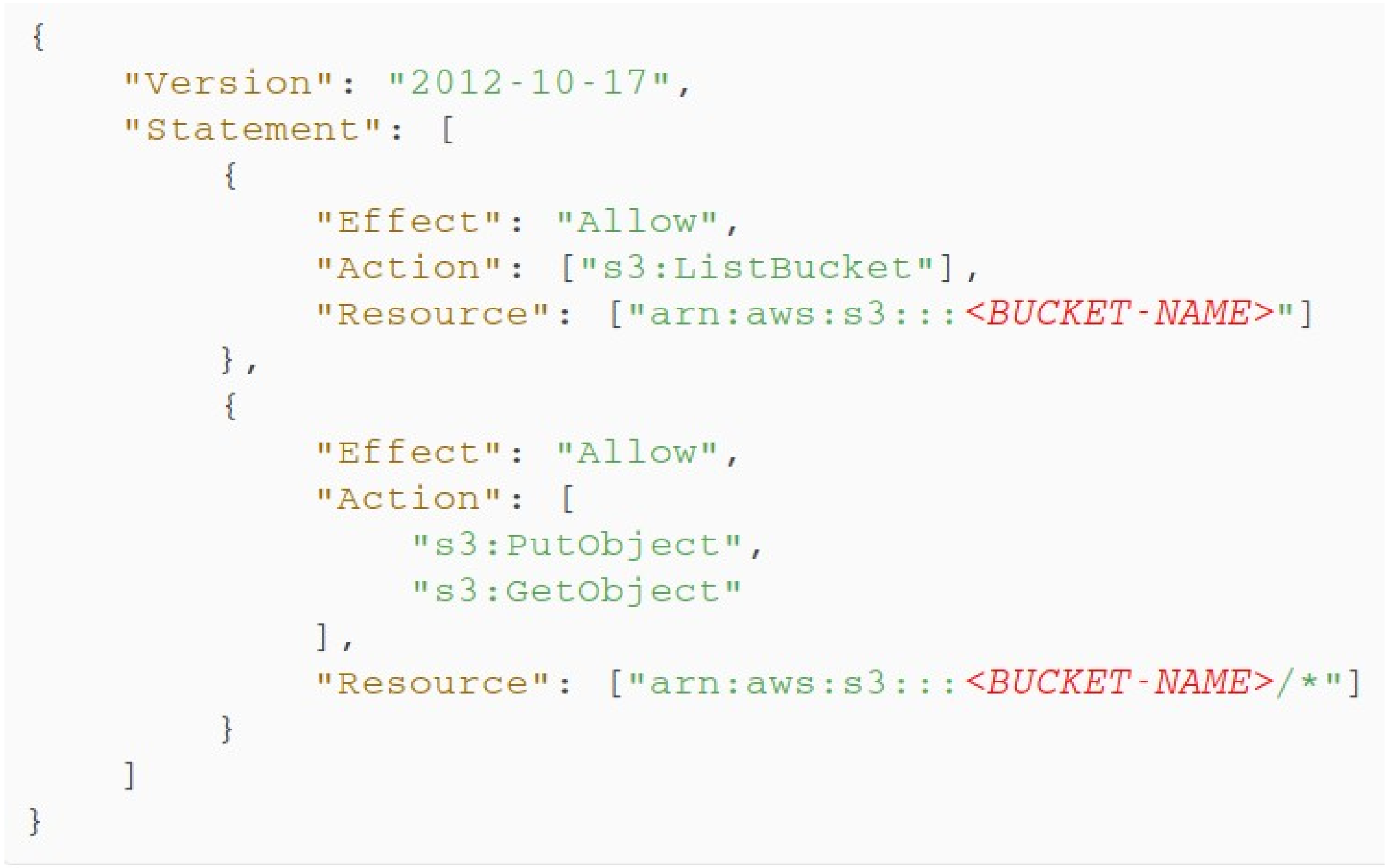
```
<!-- IAM user name -->
arn:aws:iam::123456789012:user/David

<!-- Amazon RDS instance used for tagging -->
arn:aws:rds:eu-west-1:123456789012:db:mysql-db

<!-- Object in an Amazon S3 bucket -->
arn:aws:s3:::my_corporate_bucket/exampleobject.png
```

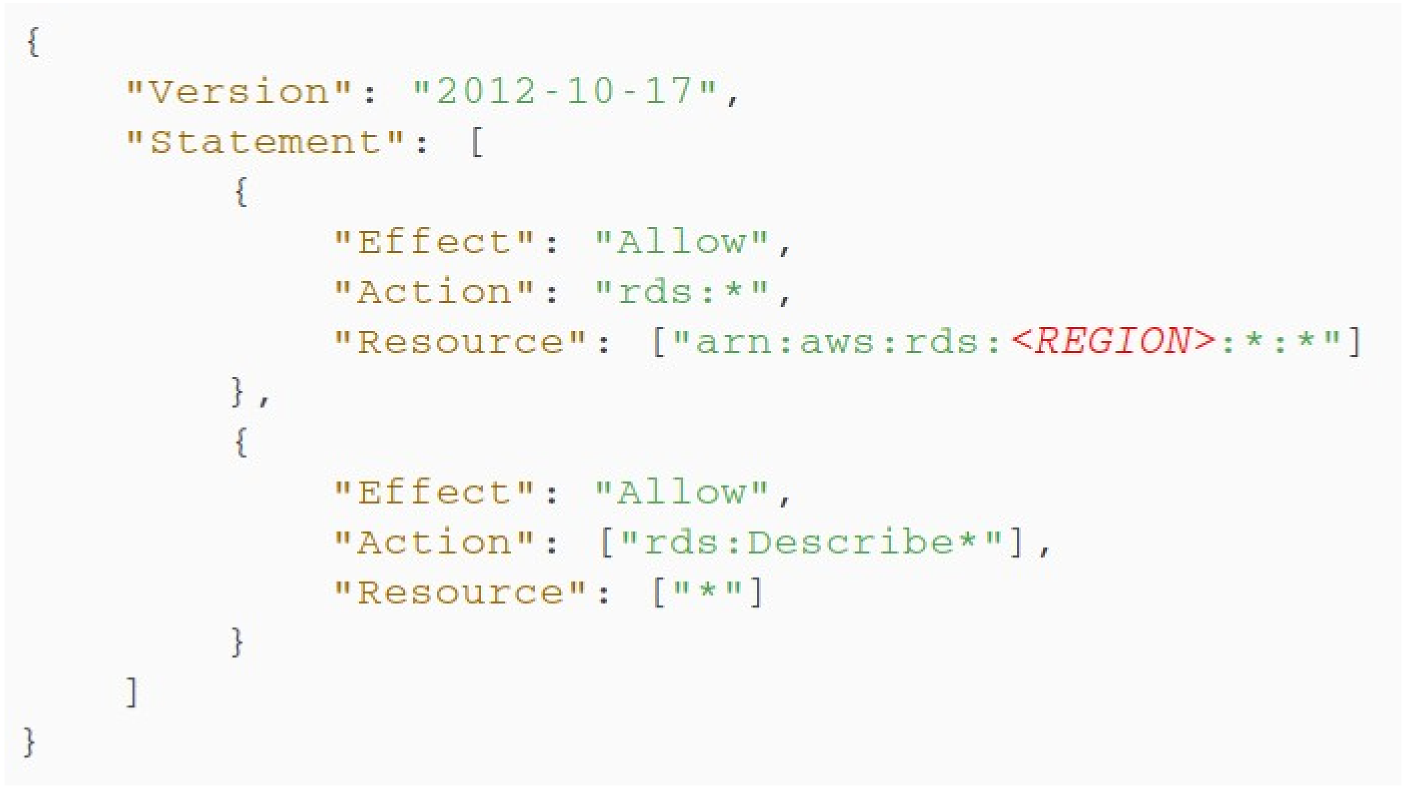
https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

AWS IAM - S3 policy example



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["s3:ListBucket"],
            "Resource": ["arn:aws:s3:::<BUCKET-NAME>"]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject"
            ],
            "Resource": ["arn:aws:s3:::<BUCKET-NAME>/*"]
        }
    ]
}
```

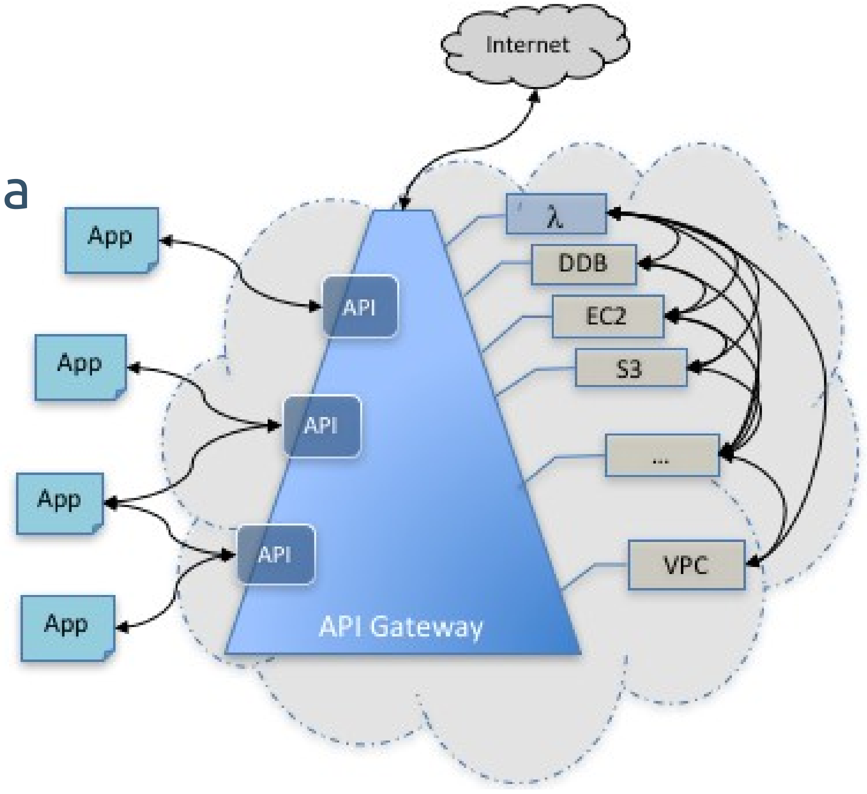
AWS IAM - RDS policy example



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "rds:*",
            "Resource": ["arn:aws:rds:<REGION>:*:*"]
        },
        {
            "Effect": "Allow",
            "Action": ["rds:Describe*"],
            "Resource": ["*"]
        }
    ]
}
```

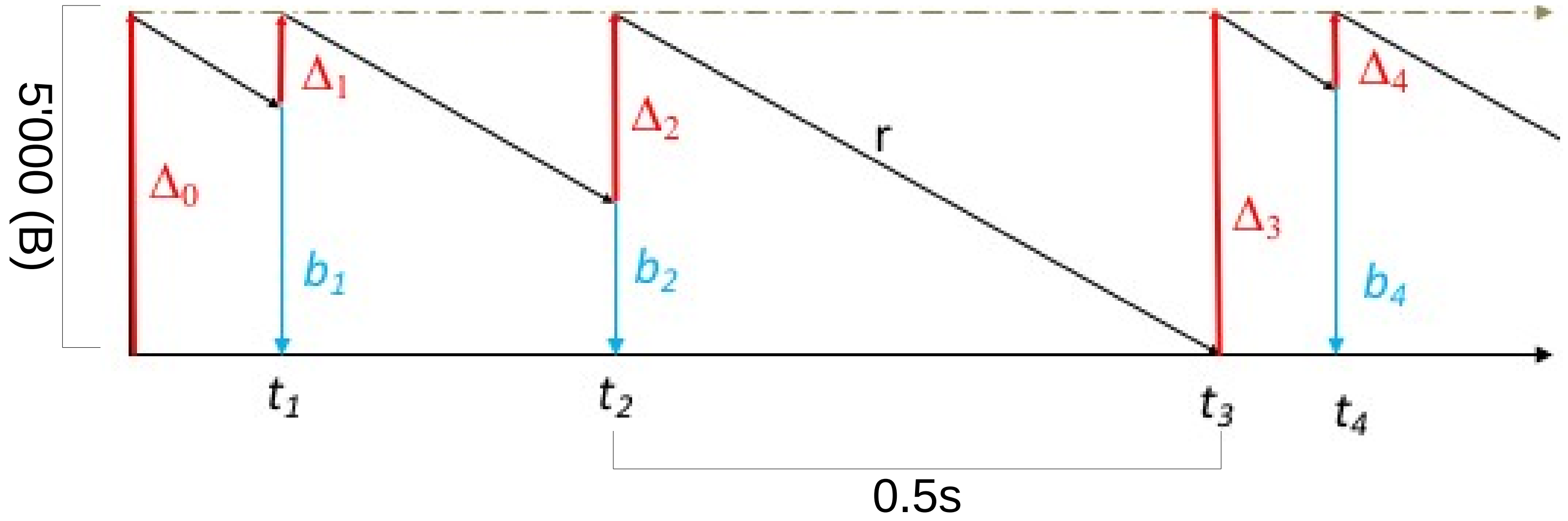
AWS API Gateway

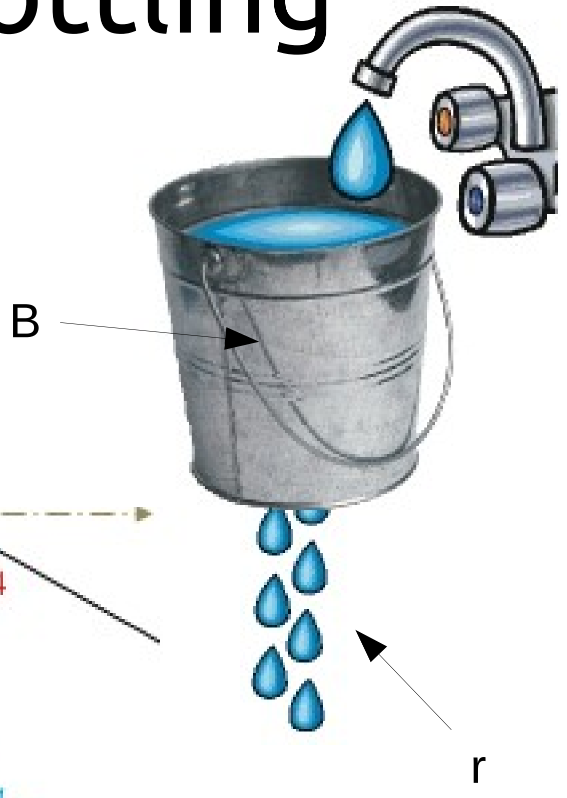
- "Front door" for application to access data and backend services functionalities
  - Facade to a microservices platform
  - Powerful integration with AWS Lambda
- Many features
  - Authentication (API key, ...)
  - Throttling
  - Caching
  - DDos protection
  - Logging (CloudWatch) or AWS X-Ray



AWS API Gateway - Throttling

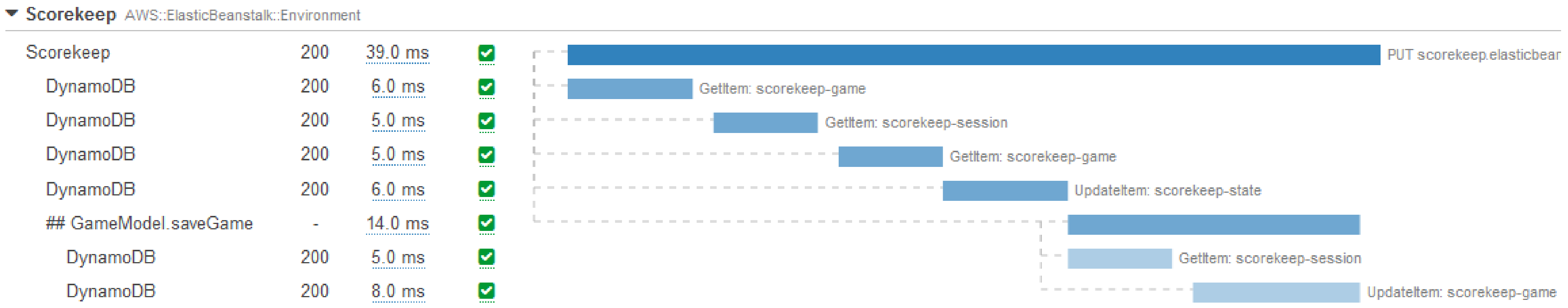
- Bucket algorithm
  - Max request/s: 10'000 (r)
  - Max **burst**: 5'000 (B)
  - If bucket full (B > 5'000) → `429 Too many requests`
    - Default, configurable by account or clients





AWS X-Ray

- "Stacktrace" requests throught differents components
  - Components aware of X-Ray will complete stack information
  - Typical example: API Gateway → AWS Lamda → DynamoDB



Scorekeep AWS::ElasticBeanstalk::Environment

Scorekeep	200	39.0 ms	✓	PUT scorekeep.elasticbear
DynamoDB	200	6.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-game
DynamoDB	200	6.0 ms	✓	UpdateItem: scorekeep-state
## GameModel.saveGame	-	14.0 ms	✓	
DynamoDB	200	5.0 ms	✓	GetItem: scorekeep-session
DynamoDB	200	8.0 ms	✓	UpdateItem: scorekeep-game

AWS CLI - Setup

$ sudo apt install awscli

- https://docs.aws.amazon.com/cli/latest/userguide/installing.html

$ aws configure

- Set default credentials, region and output format
  - Can also be given on command line or env variables
- These files will be provisioned:
  - ~/.aws/credentials
  - ~/.aws/config
    - You can edit them to set multiple profile
      - Then on command line, use `--profile my-profile` to select

References

- The Chubby lock service for loosely-coupled distributed systems, Google Research, 7th USENIX Symposium on OSDI - 2006
- ZooKeeper: Wait-free coordination for Internet-scale systems, Yahoo Research, USENIX Annual Technology Conference - 2010
- Spanner: Google's Globally Distributed Database
- Centrifuge: Integrated Lease Management and Partitioning
- Zab: High-performance broadcast for primary-backup systems, Flavio Paiva Junqueira, Benjamin C. Reed, Marco Serafini
- ZooKeeper: Wait-free Coordination for Internet-scale Systems, Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, Benjamin Reed - 2010
- Extensible distributed coordination, Tobias Distler, Christopher Bahn, Alysson Neves Bessani, Frank Fischer, Flavio Junqueira - 2015