




PCD - Resilience in Distributed Systems

Butnaru Gheorghiță, Pușcașu Bogdan, Ungureanu Alexandra

Resilience in a distributed system refers to its ability to maintain an acceptable level of functionality and performance despite various types of faults, failures, or attacks.

A resilient distributed system is designed to anticipate, detect, and recover from failures, ensuring continuous operation even under adverse conditions.

Key Aspects of Resilience in Distributed Systems

1.  **Fault Tolerance**
 - A resilient distributed system is fault-tolerant, meaning it can continue to function properly even in the presence of faults or failures.
 - "Fault-tolerant systems are designed to continue functioning correctly in the presence of faults, such as hardware failures or software errors." (Tanenbaum & Van Steen, 2007)¹
2.  **Scalability and Elasticity**
 - Resilient systems are designed to scale gracefully to accommodate changing workloads and demand.
 - "Scalable and elastic systems can dynamically adapt to changes in workload, scaling resources up or down as needed to maintain performance." (Armbrust et al., 2010)²
3.  **Redundancy and Replication**
 - Redundancy and replication of data and services are essential for resilience, ensuring that there are backup mechanisms in place.
 - "Redundancy and replication provide backup mechanisms to ensure data availability and service continuity in the event of failures." (Bondi, 2000)³

4. **Isolation and Containment**

- Resilient systems isolate and contain failures to prevent them from spreading and affecting the entire system.
- "Isolation and containment mechanisms prevent failures from propagating throughout the system, limiting their impact." (Chen et al., 2014)⁴

5. **Adaptability and Self-Healing**

- Resilient distributed systems are adaptive and self-healing, capable of recovering from failures automatically.
- "Adaptive and self-healing systems can detect and recover from failures automatically, minimizing downtime and disruption." (Kephart & Chess, 2003)⁵

6. **Monitoring and Management**

- Continuous monitoring and management are crucial for resilience, enabling proactive detection and response to issues.
- "Effective monitoring and management systems provide real-time insights into system health and performance, enabling proactive response to issues." (Alonso et al., 2015)⁶

Case Study: Handling Failures in **Netflix's** Distributed Infrastructure

Failures

- Hardware failures at the server and network equipment level.
- Software errors and performance issues at the application level.
- Security incidents and DDoS attacks.
- Anomalies in data and network traffic.

Solutions - design around failure

- Exception handling
- Clusters
- Redundancy
- Fault tolerance
- Fall-back strategies

- Unit testing, integration testing, stress testing, test suites to simulate failures (hard to simulate and test a large scale distributed system)

The Netflix team uses tools to induce failures in a more controlled manner, reducing the unpredictability of system failures. This approach helps them understand when failures occur.

Tools that helps to produce failures in a controlled manner:



Chaos Monkey - randomly shuts off instances causing instances fail

Lesson from *Chaos Monkey*

- State is bad (For example, a stored session state or cached data on an instance means that if that instance is removed, user requests are redirected to another instance, which must either rebuild the state or face request failure upon retry).
- Clusters are good - services are deployed on AWS autoscaling groups, ensuring redundancy with at least three instances of each service across different availability zones.
- Surviving single instances failure is not enough - indicates failures that Chaos Monkey can not capture such as data center power outage.

Chaos Gorilla



Chaos Gorilla - it shuts down entire Amazon availability zones and make sure that the remaining availability zones can handle the capacity and the traffic

Lessons from *Chaos Gorilla*:

- Deployment topology is masked (instances were not wired to talk to instances in other availability zones)
- Infrastructure control plane can be a bottleneck (is not easy to bring up thousands of different instances)

- Large scale events are hard to simulate
- Rapidly shifting traffic is error prone
- Smooth recovery is a challenge
- Cassandra works as expected (used for handling large amounts of data across many servers)



Chaos Kong - it shuts down the entire region and makes sure that other regions can handle all the traffic



LATENCY MONKEY

Latency Monkey - injects an arbitrary latency into a client - server interaction

Lessons from *Latency Monkey*

- Startup resiliency is often missed - at the scaling up when the latency monkey is run, the additional instances don't have the needed dependencies and they are not functional.
- An ongoing unified approach to runtime dependency management is important (visibility & transparency gets missed otherwise) - when dependencies are managed separately or inconsistently, it becomes difficult to track and understand how different components interact, leading to potential issues and inefficiencies.
- Unknown dependencies.
- Fall backs can fail too.



Hystrix - framework that helps to control the interaction between services by providing fault tolerance and latency tolerance

- Monitors the latency of each successful request and the percentage of errors that passed through the circuit. When a trigger gets hit (latency or error rates goes above a threshold) then the circuit flips to an open state and serves up a fallback.
- It is designed to improve the resilience and fault tolerance of distributed systems by providing mechanisms for latency and fault tolerance, such as circuit breakers, fallbacks, and monitoring.



Janitor Monkey - automatically cleans up resources in the cloud environment that are no longer needed or are in an inconsistent state. It uses garbage collector techniques.

Lessons from *Janitor Monkey*

- Label everything (is hard to identify what something is)
- Mess accumulates

Key points

- Observability
 - Helps to understand how resilient the system is.
 - Allows teams to understand and learn from every instance where a failure impacts customers, providing valuable insights for improving system reliability and performance in the future.
- Regularly inducing failure in the production environment validates resiliency and increases availability.

Netflix Characteristics

- **Redundant and Fault-Tolerant Architecture**
 - Netflix employs a distributed architecture that includes multiple replicas of its services across different regions and data centers.
 - Failures at the server or regional level are managed by automatically switching to available replicas.
- **Advanced Monitoring and Proactive Failure Detection**
 - Netflix utilizes an advanced monitoring system that collects and analyzes real-time system health and performance data.
 - Any anomalies are quickly detected, and operations teams are alerted to investigate and address the issue.
- **Automation and Recovery Orchestration**
 - To reduce recovery time and user impact, Netflix has automated recovery and switching processes between services and regions.
 - Tools like *Chaos Monkey* are used to simulate failures in production and test the system's ability to recover.

- **Continuous Improvement Based on Operational Feedback**
 - Netflix has a culture of continuous learning, where every incident and failure is thoroughly analyzed to identify lessons learned and make system improvements.

Results

1. High Availability and Consistent Performance

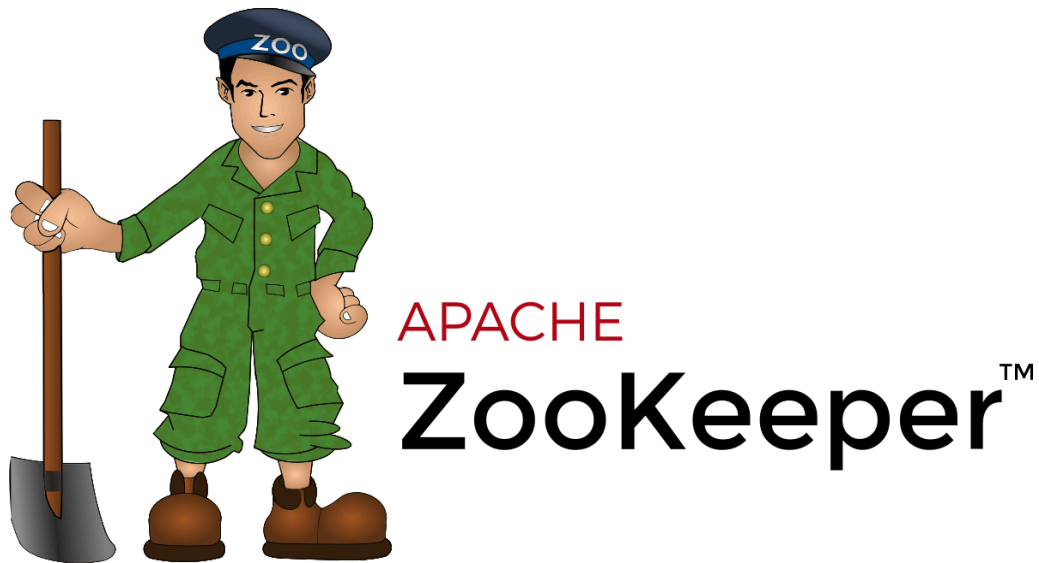
- Netflix's approach ensures reliable service for millions of users
- Architecture redundancy and proactive monitoring maintain uptime
- Users experience consistent streaming without interruptions

2. Reduced Recovery Time

- Automated recovery processes minimize downtime
- Swift detection and response to failures
- Services restored quickly during major incidents

3. Continuous Improvement

- Feedback-driven improvements enhance resilience
- Detailed analysis of failures identifies root causes
- Iterative enhancements to infrastructure and processes



Apache ZooKeeper is a centralized service for maintaining configuration information, providing distributed synchronization, and offering group services in large distributed systems. It acts as a highly available and reliable coordination service for distributed applications, enabling them to synchronize and manage their state across a cluster of machines.

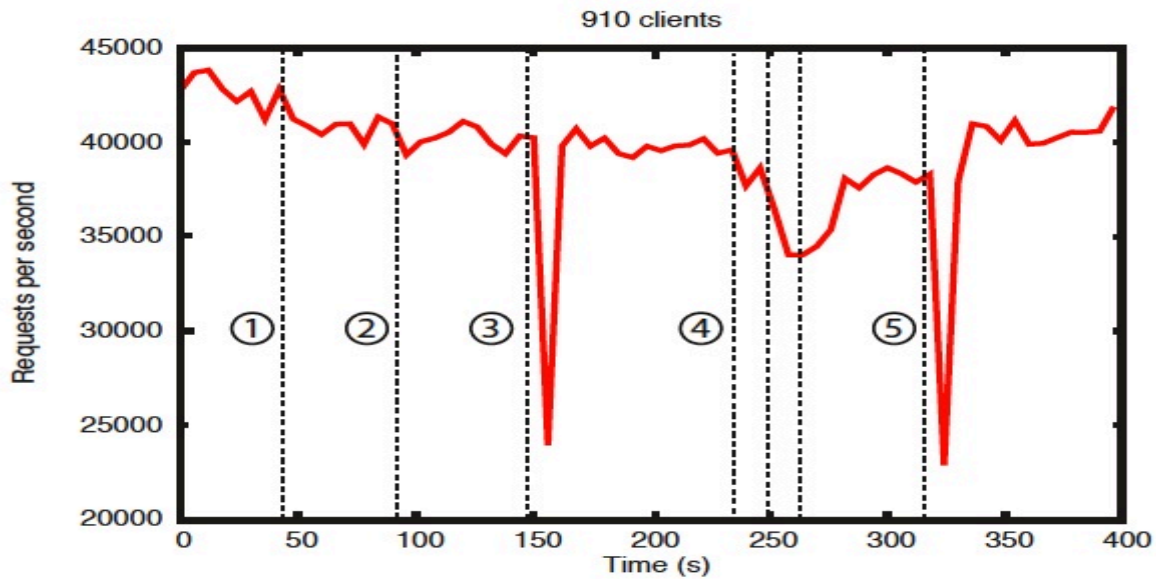
Design Principles:

Simplicity: ZooKeeper follows the principle of simplicity in design, providing a minimalistic API and focusing on core functionality such as distributed coordination and synchronization.

In order to keep the interface as simple as possible, the API only supports the following operations:

- create : creates a node at a location in the tree
- delete : deletes a node
- exists : tests if a node exists at a location
- get data : reads the data from a node
- set data : writes data to a node
- get children : retrieves a list of children of a node
- sync : waits for data to be propagated

Reliability: Reliability is a paramount concern in ZooKeeper's design. It achieves this through fault-tolerant mechanisms like data replication and leader election, ensuring that the service remains available even in the face of failures.

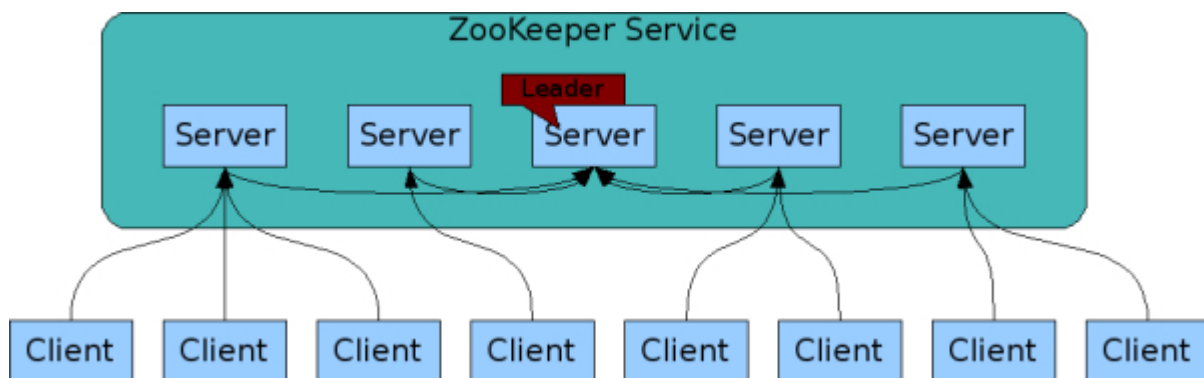


Scalability: ZooKeeper is designed to scale horizontally by adding more nodes to the ensemble. However, it's important to maintain a balance between the number of nodes and the overhead of coordination to ensure efficient operation.

Consistency: ZooKeeper provides strong consistency guarantees, ensuring that all clients see the same view of the system at any given time. This consistency is achieved through the use of distributed consensus algorithms like ZAB (ZooKeeper Atomic Broadcast).

Data Replication:

Ensemble: ZooKeeper operates in an ensemble mode where multiple ZooKeeper servers form a replicated cluster. These servers maintain copies of the same data and coordinate with each other to ensure consistency.



Quorum: ZooKeeper uses a majority-based approach for data replication. For any write operation to be considered successful, it must be acknowledged by a majority of the servers in the ensemble. This ensures that data remains consistent even if a minority of servers fail.

Error Detection:

Heartbeats: ZooKeeper uses heartbeats to detect the liveness of servers in the ensemble. Servers periodically exchange heartbeats to monitor each other's health. If a server fails to send heartbeats within a specified timeout period, it is considered to be down.

Quorum Checks: ZooKeeper employs quorum checks to ensure that a majority of servers are available and in sync before committing write operations. If a server is lagging behind or unreachable, it is temporarily excluded from the quorum to maintain consistency.

Recovery Mechanisms:

Leader Election: In the event of a server failure, ZooKeeper dynamically elects a new leader from the remaining servers in the ensemble. The leader is responsible for coordinating client requests and managing data replication.

Follower Catch-up: When a failed server recovers and rejoins the ensemble, it undergoes a catch-up process to synchronize its state with the rest of the ensemble. This involves fetching the latest updates from the leader and applying them to the local state.

Snapshot and Transaction Logs: ZooKeeper periodically takes snapshots of its data and maintains transaction logs to ensure durability and recoverability. In the event of a catastrophic failure, these logs can be used to restore the system to a consistent state.

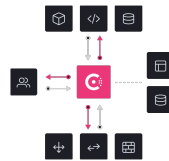
ZooKeeper Use Cases:

There are many open source projects, other Apache projects, as well as a number of companies that rely on Apache ZooKeeper in their implementations.

Some example of known companies that use ZooKeeper are:

- Facebook
 - Facebook uses the Zeus for configuration management which is a forked version of ZooKeeper, with many scalability and performance enhancements in order to work at the Facebook scale.
 - It runs a consensus protocol among servers distributed across multiple regions for resilience. If the leader fails, a follower is converted into a new leader.
- Pinterest
 - Pinterest uses the ZooKeeper for Service discovery and dynamic configuration. Like many large scale web sites, Pinterest's infrastructure consists of servers that communicate with backend services composed of a number of individual servers for managing load and fault tolerance.
 - Ideally, we'd like the configuration to reflect only the active hosts, so clients don't need to deal with bad hosts as often. ZooKeeper provides a well known pattern to solve this problem.
- Twitter
 - ZooKeeper is used at Twitter as the source of truth for storing critical metadata. It serves as a coordination kernel to provide distributed coordination services, such as leader election and distributed locking. Some concrete examples of ZooKeeper in action include:
 - ZooKeeper is used to store service registry, which is used by Twitter's naming service for service discovery.
 - Manhattan (Twitter's in-house key-value database), Nighthawk (sharded Redis), and Blobstore (in-house photo and video storage), stores its cluster topology information in ZooKeeper.

- EventBus, Twitter's pub-sub messaging system, stores critical metadata in ZooKeeper and uses ZooKeeper for leader election.
- Mesos, Twitter's compute platform, uses ZooKeeper for leader election.
- Yahoo!
 - ZooKeeper is used for a myriad of services inside Yahoo! for doing leader election, configuration management, sharding, locking, group membership etc.



HashiCorp Consul

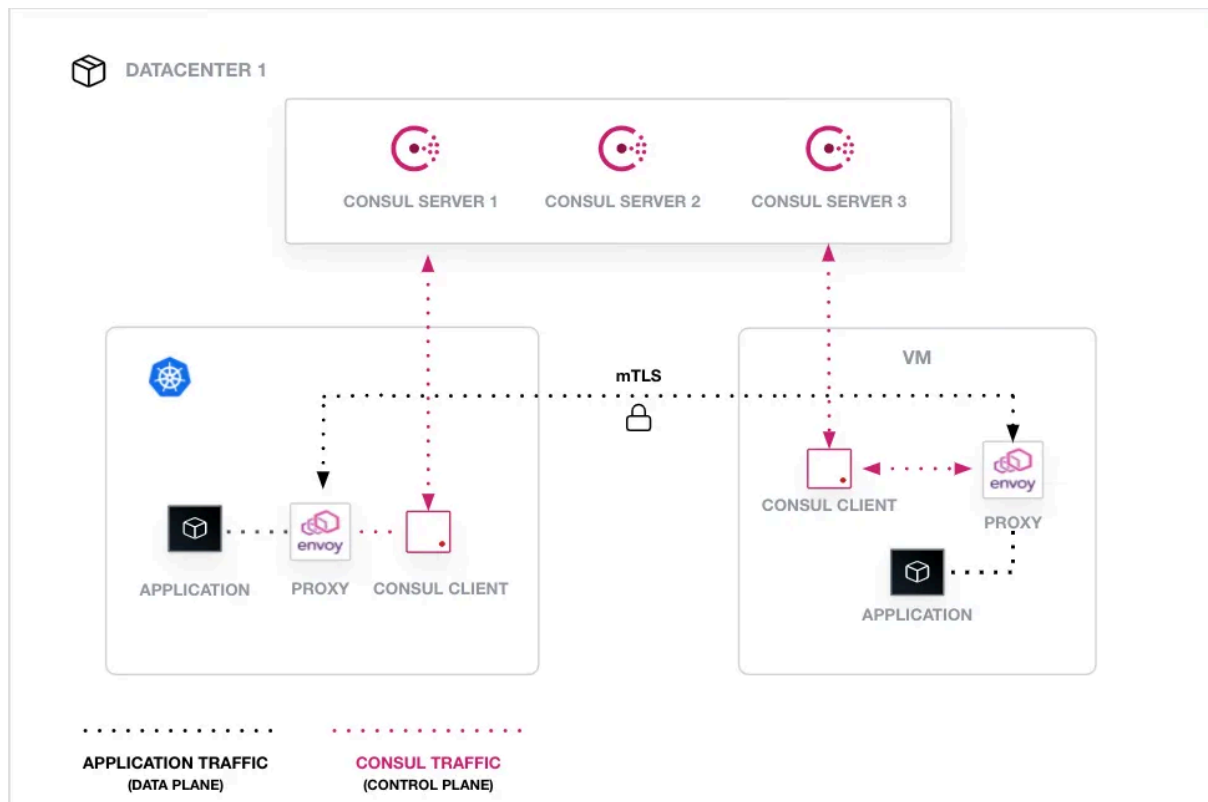
HashiCorp Consul is a service networking tool that automates service discovery, service connection, and security in distributed architectures, enabling efficient communication between services.

Primary use-cases:

- Global Service Registry & Service Discovery
- Service to Service Communication (Service Mesh)
- Network Infrastructure Automation

Consul workflow:

- **Register** - Teams register services in the Consul catalog, a central registry that enables services to automatically discover each other without the need for manual modification of application code, deploying extra load balancers, or hardcoding IP addresses.
- **Query** - Consul's DNS, based on identity, enables to locate healthy services in the Consul catalog, providing health status, access points, and additional data to manage data flow within your network.
- **Secure** - Once services identify upstreams, Consul ensures that communication between services is authenticated, authorized, and encrypted. Consul's service mesh secures microservice architectures using mTLS (mutual Transport Layer Security) and can control access based on service identities, regardless of variations in compute environments and runtimes.



Service A -> Proxy A -> Consul Client A -> Checks Control Plane Registry -> Proxy A
 -> Proxy B -> Consul Client B -> Proxy B -> Service B

Consul Servers:

- Shared address list of all other services (service catalog).
- Certificate data.
- Configuration data, like communication rules (inbound and outbound rules between services).
- Push out all the needed data to the Consul Clients like service registry information, configuration certificates.

Consul Clients (Agent):

- Responsible for service registration, health checking, routing, and interacting with the Consul system to ensure reliable and efficient communication between services in the network.
- It must be installed on every node within the Consul cluster and serves as the interface between server nodes for most operations.

Control Plane:

- Enables you to register, query, and secure services deployed across your network.
- Maintains a central registry to track services and their respective IP addresses.
- It is a distributed system that runs on clusters of nodes, such as physical servers, cloud instances, virtual machines, or containers.
- It can be seen as the manager of the network which makes decisions based on configurations (such as setting up connections between services, enforcing security policies, and managing configuration) and communicates these instructions to the Data Plane for execution.

Data Plane:

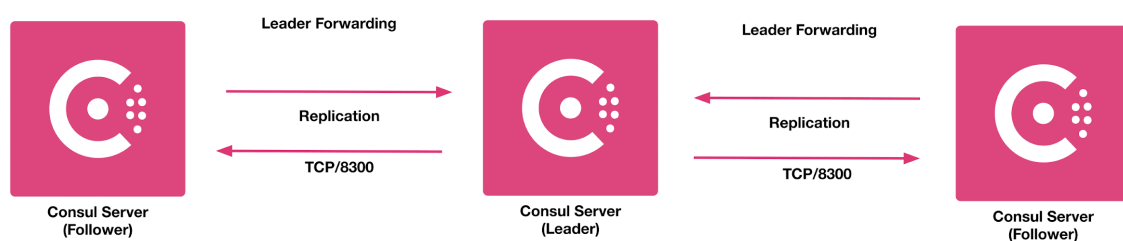
- Executes the instructions provided by the Control Plane.
- Handles the communication between services.
- Determines the routes for data packets and processes them accordingly.
- Enforces security measures, like encryption, as instructed by the Control Plane.
- Focuses on executing tasks efficiently to ensure fast and reliable data transmission.
- If something changes or gets updated, like the address of a service or new service gets added or removed, certificates get rotated, they will get the update from the Control Plane automatically.

Consul Server agents (Consul Servers)

- Store all state information, including details about services, node IP addresses, health checks, and configurations.
- It's recommended to deploy three to five Consul servers in a cluster for resilience and availability.
- More servers increase resilience and availability, as there are more copies of data stored across the cluster. However, having more servers can slow down the consensus process, which is critical for efficient information processing within Consul.

Consensus protocol & Data Replication

- It is the election of a single server to be the *leader*.

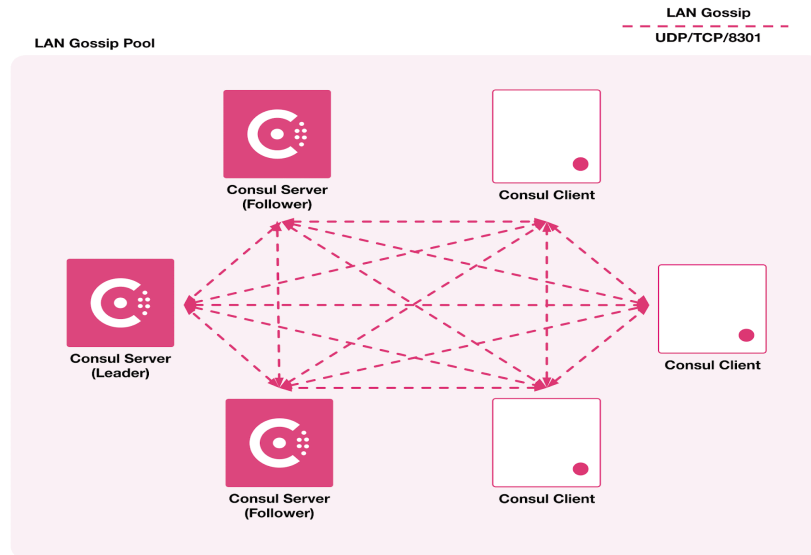


- The leader processes all queries and transactions, which prevents conflicting updates in clusters containing multiple servers.
- Servers that are not currently acting as the cluster leader are called *followers*.
- Replication ensures that if the *leader* is unavailable, other servers in the cluster can elect another leader without losing any data.

- Consul manages data replication by using the Raft consensus protocol to ensure that changes are replicated across multiple servers in the cluster. This ensures data consistency, fault tolerance, and high availability.

LAN Gossip Pool

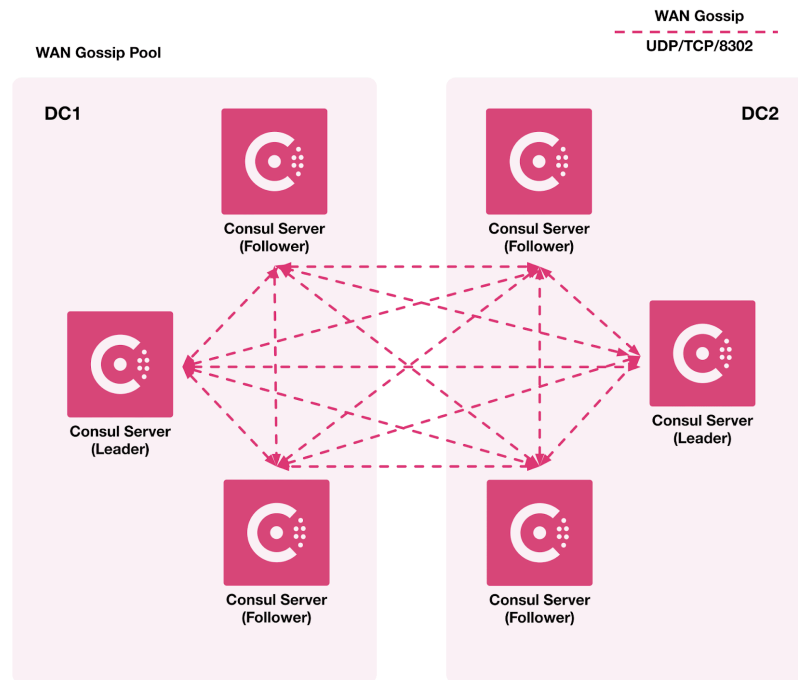
Client and server agents participate in a LAN gossip pool so that they can distribute and perform node *health checks*.



- Each datacenter in Consul has a LAN gossip pool containing all members.
- Membership information in the LAN pool enables automatic server discovery, reducing manual configuration.
- Failure detection is distributed across the cluster, ensuring quick response to failures.

WAN Gossip Pool

Servers can join a WAN gossip pool, optimized for Internet latency. This allows them to exchange information and gracefully handle connectivity loss during failures.



Consul Recovery Strategies

- Automatic Health Checks
 - *HTTP checks* - make an HTTP GET request to the specified URL and wait for the specified amount of time
 - *TCP checks* - attempt to connect to an IP or hostname and port over TCP and wait for the specified amount of time
 - *UDP checks* - send UDP datagrams to the specified IP or hostname and port and wait for the specified amount of time
 - *TTL checks* - passive checks that await updates from the service
 - ...
- Failure Detection
 - Quickly detects failures through its *health checks* and *gossip protocol*
- Service Discovery and Load Balancing
 - Automatically reroutes traffic away from failed services and balances load among healthy instances.
- Automatic Healing
 - Automatically restarts failed services or nodes, triggered by *health check* failures.
- Recovery Time Objective (RTO)
 - Minimizes recovery time by swiftly detecting and responding to failures according to set objectives.
- Manual Intervention
 - Operators can manually intervene when necessary to perform recovery tasks beyond Consul's automated capabilities.

Application Design - Datasets Data Quality

Functional requirements

The service makes data quality checks on datasets onboarded by users. For each dataset the service processes in a backfill style a selected time frame of data for determining the metric thresholds and then, depending on the rate on which data is being published to the dataset, performs scans for updating or determining new partitions of the dataset. After the dataset is scanned, an evaluation step follows that computes metrics and determines if values exceed the set limits.

The end users are dataset owners or subscribers that need to determine the existence of issues or the appearance of trends regarding data.

Datasets are retrieved from multiple providers.

These metrics include row count, sum, null, custom user queries and others.

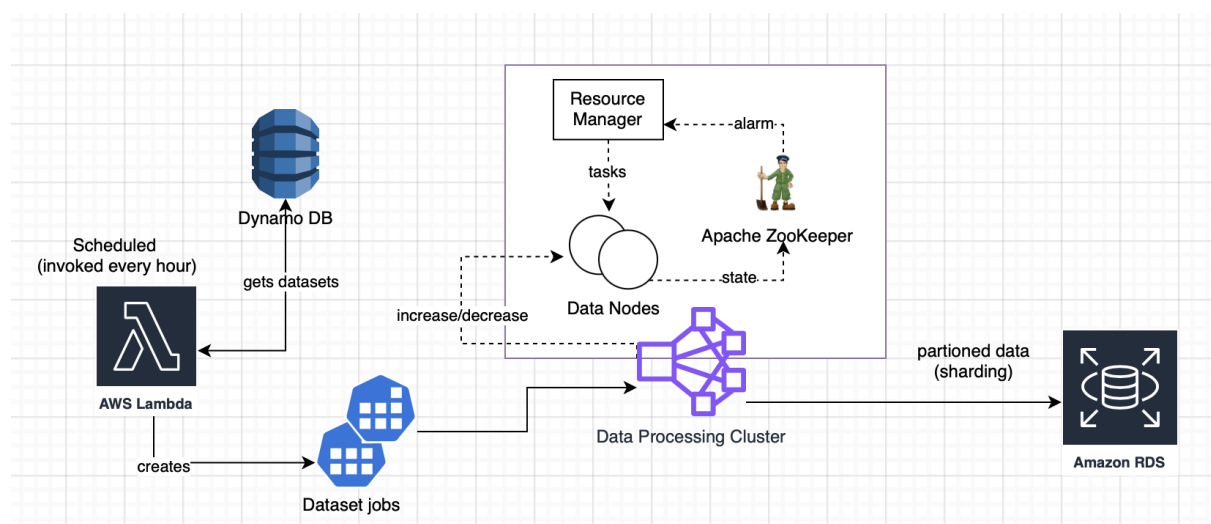
Users need a UI to see the metrics of their desired datasets.

Solution

The application will consist of multiple loosely coupled components:

Data Ingestion Service

This service deals with reading data from different providers and writing it in our own storage through a great number of scheduled jobs. These jobs will be processed by the **Data Processing Cluster** which will have under its umbrella **Data Nodes**, a **Resource Manager** and **ZooKeeper**.



Components:

A *Data Node* has the mission of processing one or more tasks concurrently and should report its status to the Resource Manager via Zookeeper.

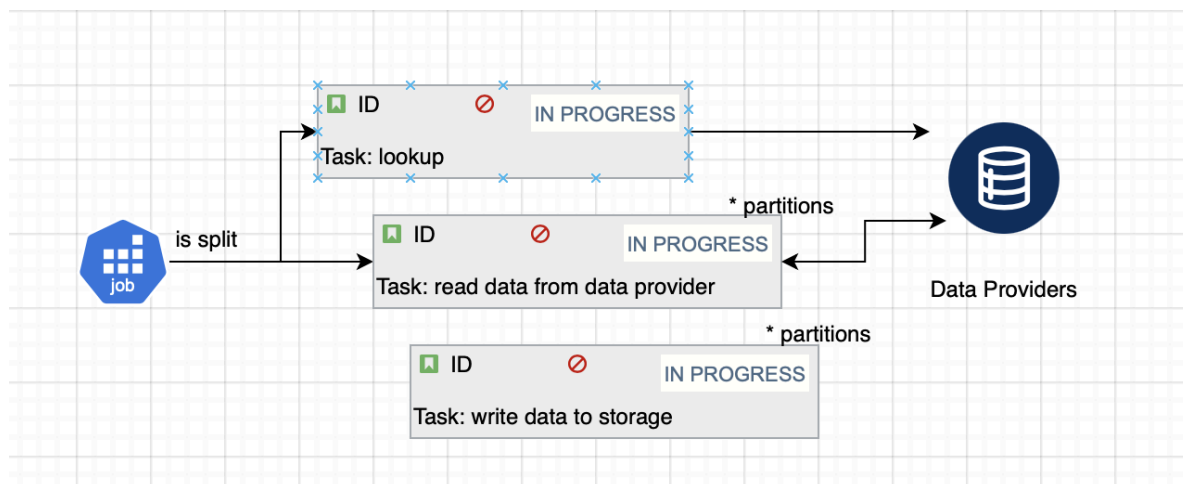
The Resource Manager will accomplish three main missions:

- **task allocation:** determines which node is best suited to take on a new task. This decision is based on the current workload of nodes and the nature of the tasks.
- **monitoring:** keeps track of all tasks being processed in the cluster. If a node becomes overloaded or goes down, the Resource Manager can redistribute the tasks to other nodes.
- **scalability management:** dynamically adjusts the allocation of tasks when new nodes are added or removed from the cluster. This ensures the cluster can scale up or down efficiently

ZooKeeper Ensemble will handle the following operations:

- **cluster coordination:** manages the session states of each node in the cluster. This is crucial for understanding which nodes are active, which are idle, and which may have failed.
- **metadata storage:** stores metadata about each task, including which node is processing it, the task status, and any dependencies between tasks.
- **failure detection and recovery:** detects node failures through heartbeat mechanisms. In the event of a failure, ZooKeeper can trigger alerts that inform the Resource Manager to reallocate tasks from the failed node to other nodes.
- **configuration management:** maintains a centralized repository of configuration settings for the cluster, which is critical when scaling or updating cluster settings.

Workflow:



The cluster receives jobs for processing. For this processing to be efficient, the job will be split into multiple tasks.

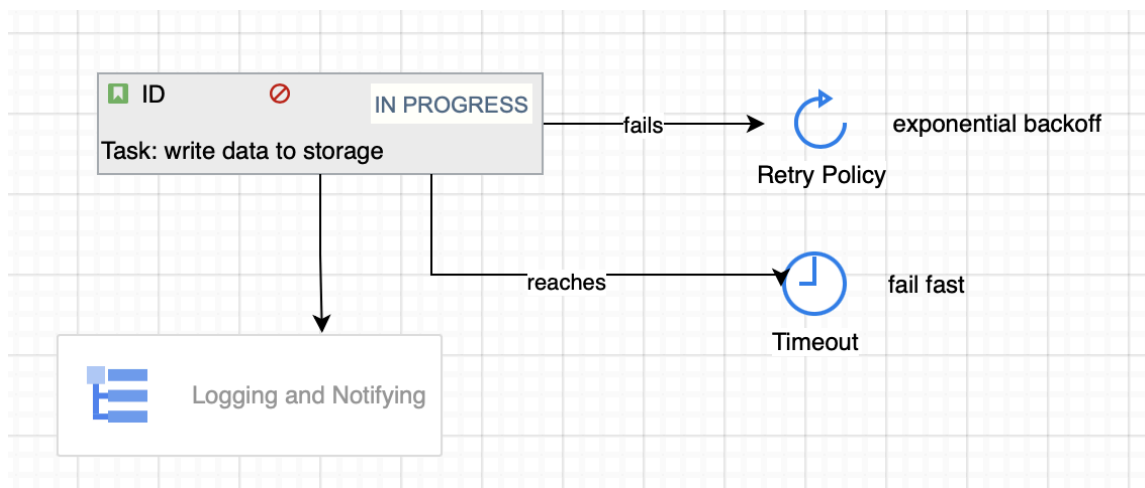
Each job will be dedicated to a dataset. Because the dataset can have multiple partitioning keys or to not be portioned at all, the first step after a job arrives is to determine that dataset partitions and for each partition detected to create a corresponding task. The partition lookup will be a task itself and will represent a request to the datasource. The Resource Manager, consulting with ZooKeeper, will assign these tasks to appropriate data nodes based on current load and task requirements. Data nodes begin processing their assigned tasks, periodically updating their progress to ZooKeeper. This information is

relayed back to the Resource Manager, which maintains an overview of job progress across the cluster.

The chosen storage for dataset data is RDS.

Handling failures in order to make a resilient system:

- If a node reports a failure or stops sending heartbeats, ZooKeeper alerts the Resource Manager. The Resource Manager then reallocates the tasks from the failed node to other nodes, potentially adjusting the cluster configuration to handle the redistribution of work.
- But also the task itself can fail, in which case we must have a **Retry Policy** to prevent infinite loops, resource wastage that would not permit other jobs to enter the processing because of not enough resources, or increased price as the system scales up wrongfully. First of all we should define a retry limit, in this case set to 2. We should also have an **Exponential Backoff**, meaning that there should be some time between retries of the tasks.
- **Error logging** and **notification systems** should be also implemented to understand the underlying issue: there could be a problem with service dependencies to the data sources, a problem with the actual data received or an issue related to the communication with the resource provider.
- A **Timeout Limit** should also be implemented, so if a task reaches a certain duration, we should **fail fast**.



Scaling Operations:

As the demand increases, additional nodes can be added to the cluster. ZooKeeper helps integrate these new nodes by distributing the configuration and current state information, allowing them to quickly begin processing tasks.

Once tasks are completed, nodes report this to ZooKeeper, which updates the overall job status. Successful completion of tasks triggers data writing operations, with final status reports managed through ZooKeeper for consistency and reliability.

After failure metrics

Fault tolerance metrics:

- **Failure Rate:** Track the frequency of node failures and task failures.
- **Recovery Time:** Measure how quickly the system recovers from a node or task failure, including the time taken to reallocate tasks and bring new nodes online.
- **Retry Counts:** Monitor the number of retries for each task due to failures to ensure that retry policies are effective but not causing resource exhaustion.

Scalability Metrics

- **Scale-up Time:** The time required to integrate new nodes into the cluster and have them fully operational, which impacts how quickly the system can respond to increased demand.
- **Node Elasticity:** The ability of the system to scale up or down efficiently based on the workload. This could be measured by the responsiveness of the Resource Manager in reallocating tasks when nodes are added or removed.

System Performance Metrics

- **Task Processing Time:** Measure the time taken to complete each task from initiation to completion. This helps in identifying performance bottlenecks.
- **Node Utilization:** Monitor the CPU, memory, and I/O usage of each node to ensure they are not overburdened or underutilized.

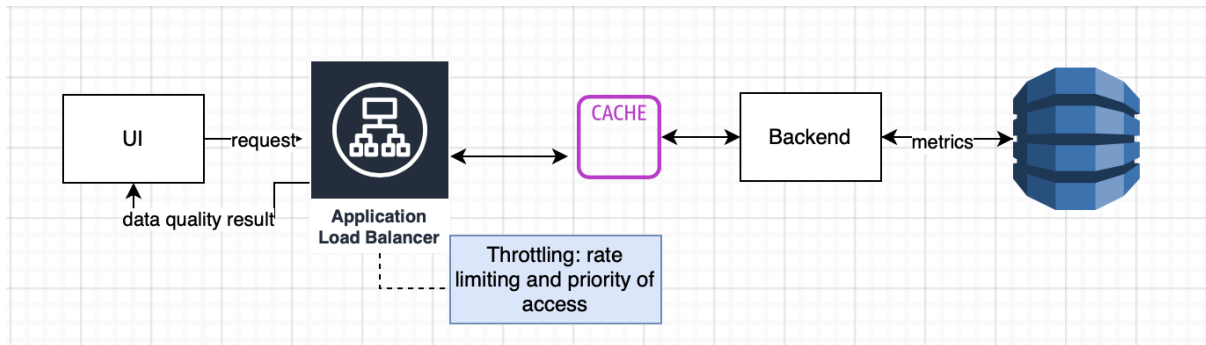
Data Quality Checks

This component will also include a cluster with a similar architecture as the previous one but with some modifications. The jobs and their corresponding tasks will read data from RDS, compute metrics and make checks against thresholds. The reading and subsequent computation could be done by multiple tasks, by further splitting the task not only based on partitions, but on size. The data could be split into multiple batches, each batch to be handled by a task. The final task of a job will have the mission to send a notification that the data quality check has finished for a specific partition.

Jobs creation

For the jobs creation we will have a scheduled lambda that takes the list of registered datasets from Dynamo and depending on the type of dataset, will create a scan job or skip it for the moment.

User interaction



The user interface will be using a backend for displaying the results of the metrics.

- **Load balancer** at user interaction application to distribute incoming requests across multiple backend servers. For load balancing there are multiple algorithms that we can try, but the chosen one in our case is “**least request**”.
- Because a dataset has multiple users that follow it, we will have a **cache** that stores the results of a query based on dataset and metric. The cache can be invalidated when the data quality check has finished (the last task) and it will depend on the type of dataset (more specifically, when it is updated at source)
- Another resilience strategy that comes inherently from the design is using the dataset partitions for storing data and computing metrics. In this way we achieve **sharding**. If one partition fails, in whatever step, the other partitions or datasets will not be affected, reducing the risk of a single point of failure.
- We could also use the cache system and not invalidate it if the database becomes unavailable. This is a very unlikely case, but if we encounter this issue, we stop the invalidation and show clients only data computed until that moment, in comparison to not showing any data at all, achieving in this way **graceful degradation**.
- **Throttling mechanism**: we should also have **rate limiting** to guard ourselves of Denial of Service Attacks. The implementation could have a **priority of access**, meaning that in high traffic dataset owners should have a better access to metrics than dataset followers, as dataset owners should be the first to get alerted by issues and start mitigating the problem.

Metrics

- Load balancer metrics:
 - request rate
 - latency
 - error rate
- Throttling metrics:

- rate limit exceeds
- rejected requests
- requests queued

References:

1. Tanenbaum, A. S., & Van Steen, M. (2007). Distributed Systems: Principles and Paradigms (2nd Edition). Pearson Prentice Hall.
2. Armbrust, M., et al. (2010). A View of Cloud Computing. Communications of the ACM, 53(4), 50-58.
3. Bondi, A. S. (2000). Characteristics of Scalable Systems. Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems.
4. Chen, M., et al. (2014). Big Data: A Survey. Mobile Networks and Applications, 19(2), 171-209.
5. Kephart, J. O., & Chess, D. M. (2003). The Vision of Autonomic Computing. IEEE Computer, 36(1), 41-50.
6. Alonso, G., et al. (2015). A Roadmap towards Resilient Enterprise Information Systems. Proceedings of the International Conference on Cloud Computing and Services Science.
7. Netflix Technology Blog: <https://netflixtechblog.com/>
8. D. Jacobson, et al. (2010). "Lessons Learned from Netflix". Communications of the ACM, 53(4), 60-65.
9. M. Nygard (2007). "Release It!: Design and Deploy Production-Ready Software". Pragmatic Bookshelf.
10. Consul Documentation: <https://developer.hashicorp.com/consul/docs>
11. Layer5 Website: <https://layer5.io/resources/service-mesh/service-mesh-consul>
12. Baeldung Website: <https://www.baeldung.com/introduction-to-hystrix>
13. ZooKeeper Documentation: <https://zookeeper.apache.org/documentation.html>
14. ZooKeeper Use Cases: <https://zookeeper.apache.org/doc/r3.8.4/zookeeperUseCases.html>