redhat.

AMQ Streams

Introduction to Apache Kafka

Ecosystem

Apache Kafka ecosystem

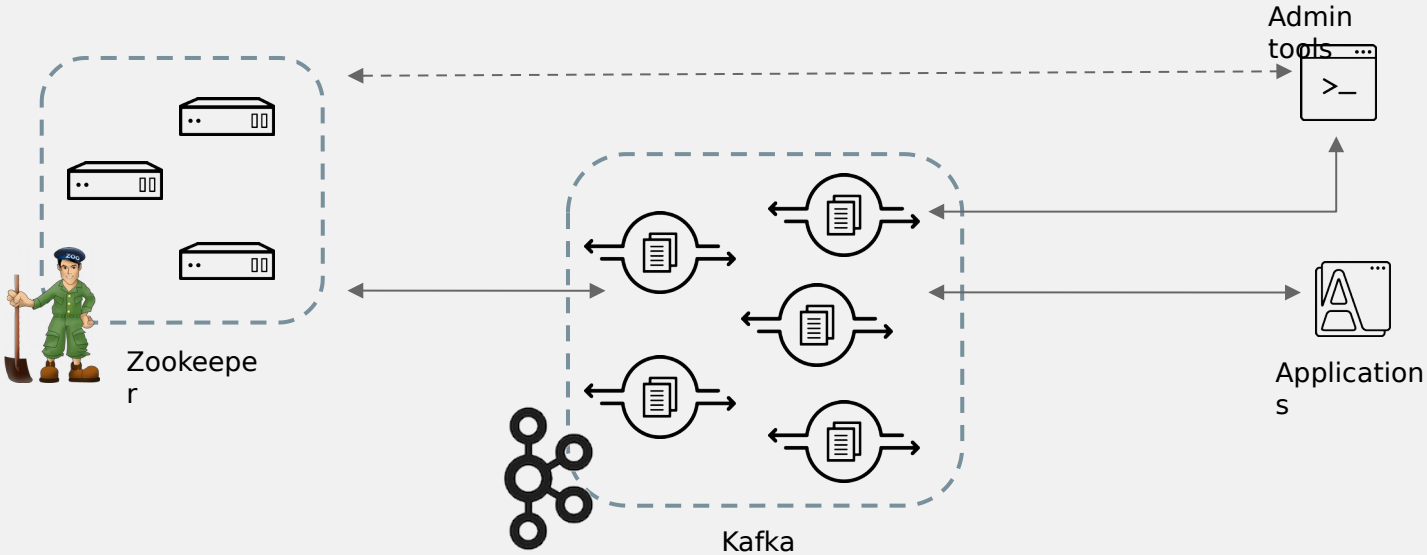
- Apache Kafka has an ecosystem consisting of many components / tools
  - Kafka Core
    - Broker
    - Clients library (Producer, Consumer, Admin)
    - Management tools
  - Kafka Connect
  - Kafka Streams
  - Mirror Maker

Apache Kafka ecosystem

Apache Kafka components

- **Kafka Broker**
  - Central component responsible for hosting topics and delivering messages
  - One or more brokers run in a cluster alongside with a Zookeeper ensemble
- **Kafka Producers and Consumers**
  - Java-based clients for sending and receiving messages
- **Kafka Admin tools**
  - Java- and Scala- based tools for managing Kafka brokers
  - Managing topics, ACLs, monitoring etc.

Kafka & Zookeeper

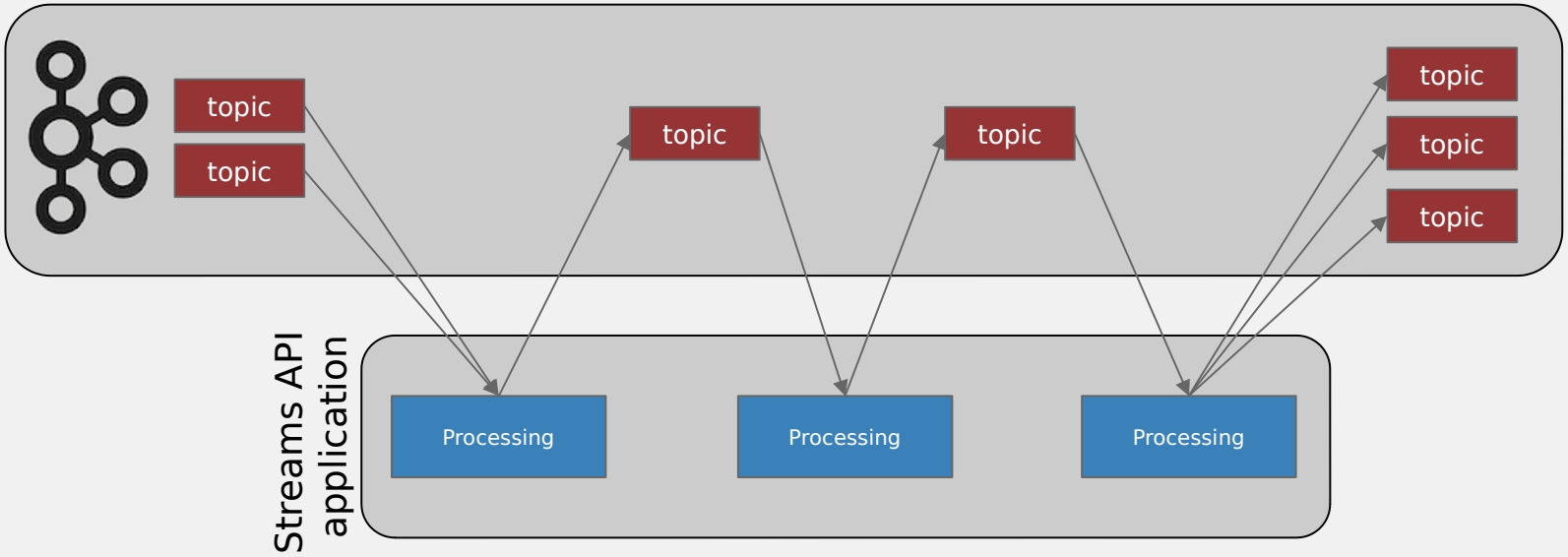


Kafka ecosystem

Apache Kafka components

- **Kafka Streams**
  - Stream processing framework
  - Streams are Kafka topics (as input and output)
  - It’s really just a Java library to include in your application
  - Scaling the stream application horizontally
  - Creates a topology of processing nodes (filter, map, join etc) acting on a stream
    - Low level processor API
    - High level DSL
    - Using “internal” topics (when re-partitioning is needed or for “stateful” transformations)
  - Scala wrapper (New in Kafka 2.0)

Kafka Streams API



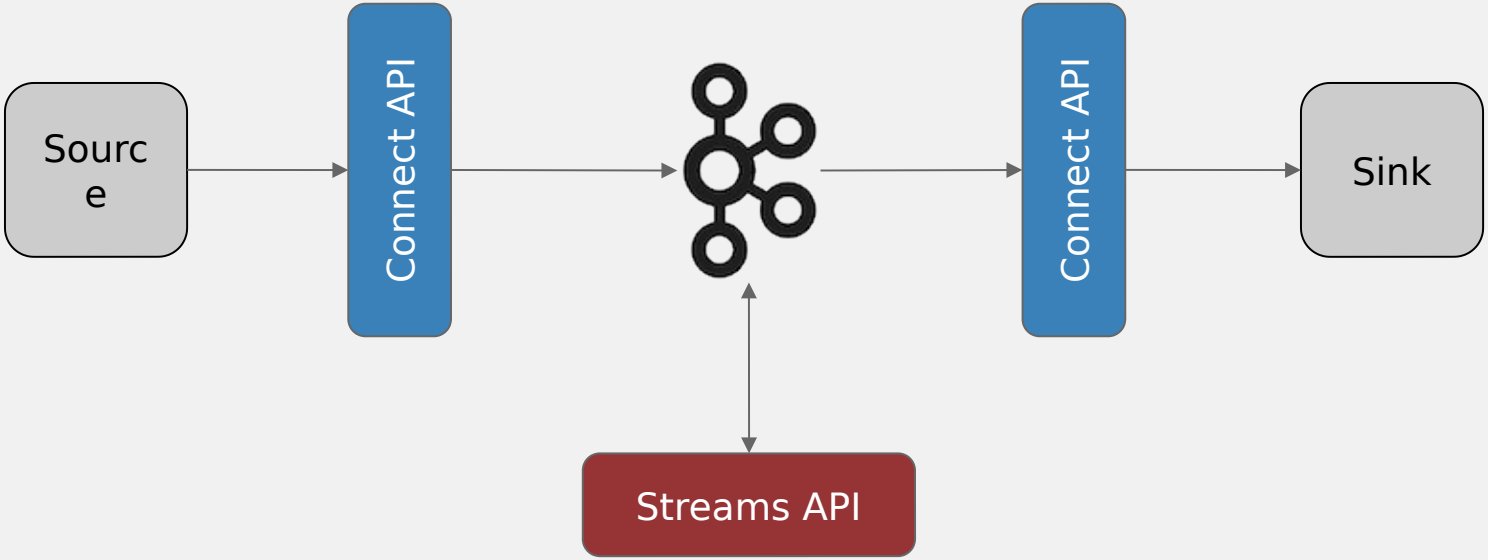
Kafka ecosystem

Apache Kafka components

- **Kafka Connect**
  - Framework for transferring data between Kafka and other data systems
  - Facilitate data conversion, scaling, load balancing, fault tolerance, ...
  - Connector plugins are deployed into Kafka connect cluster
    - Well defined API for creating new connectors (with Sink/Source)
    - Apache Kafka itself includes only FileSink and FileSource plugins (reading records from file and posting them as Kafka messages / writing Kafka messages to files)
    - Many additional plugins are available outside of Apache Kafka

Kafka ecosystem

Extract ... Transform ... Load



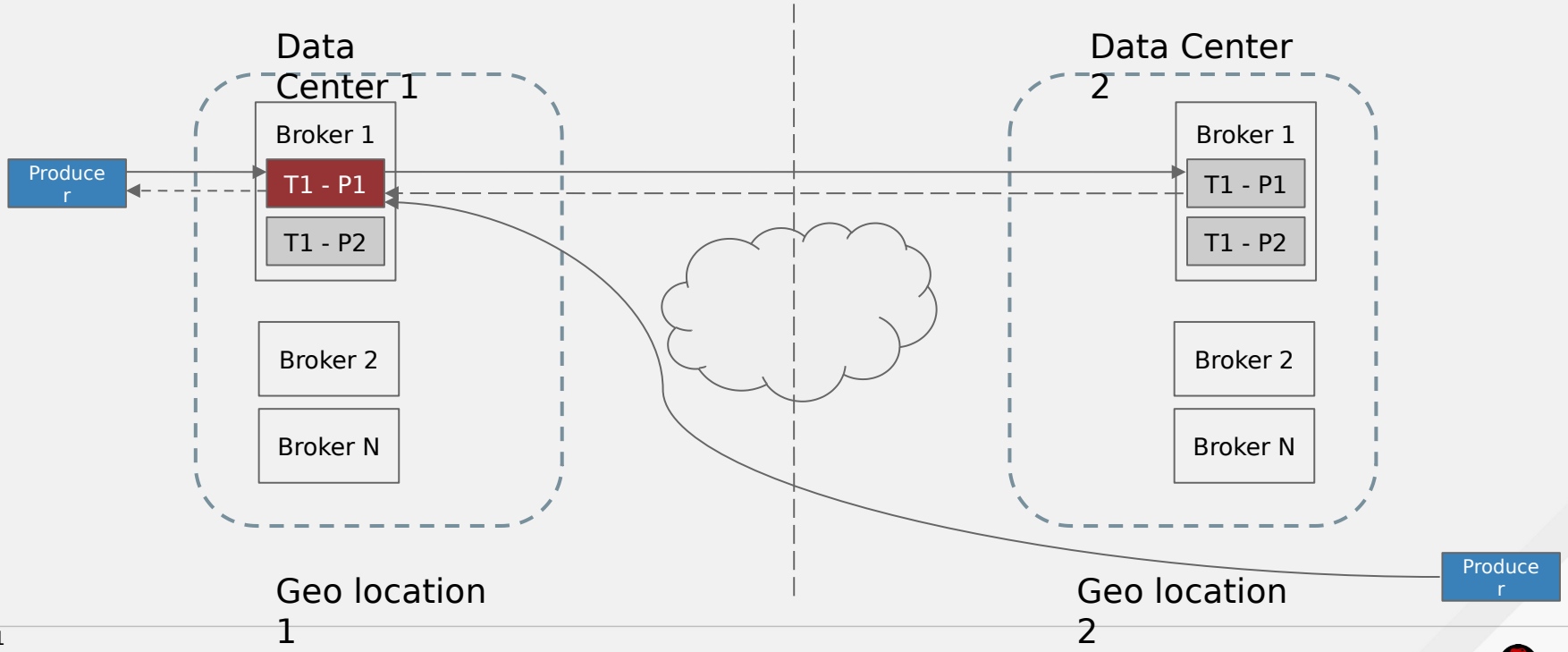
Kafka ecosystem

Apache Kafka components

- **Mirror Maker**
  - Kafka clusters do not work well when split across multiple datacenters
    - Low bandwidth, High latency
    - For use within multiple datacenters it is recommended to setup independent cluster in each data center and mirror the data
  - Tool for replication of topics between different clusters

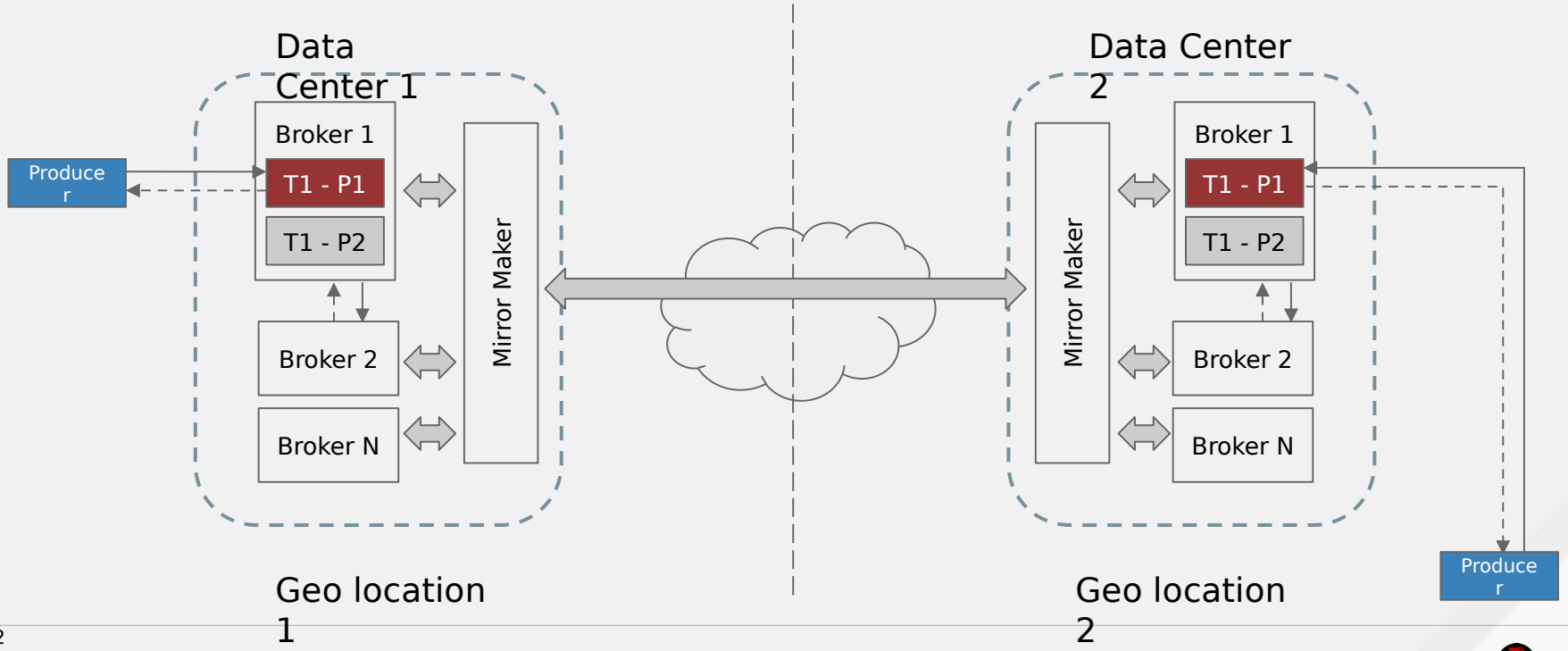
Across Data Centers

Problems



Across Data Centers

Mirror Maker



Kafka ecosystem

Outside of Apache Kafka project

- Clients for other languages
- REST Proxy for bridging between HTTP and Kafka
- Schema Registry
- Cluster Balancers
- Management and Monitoring consoles
- Kafka Connect plugins

\+ Kafka can be used with many other projects (e.g. Apache Spark, Apache Flink, Apache Storm)

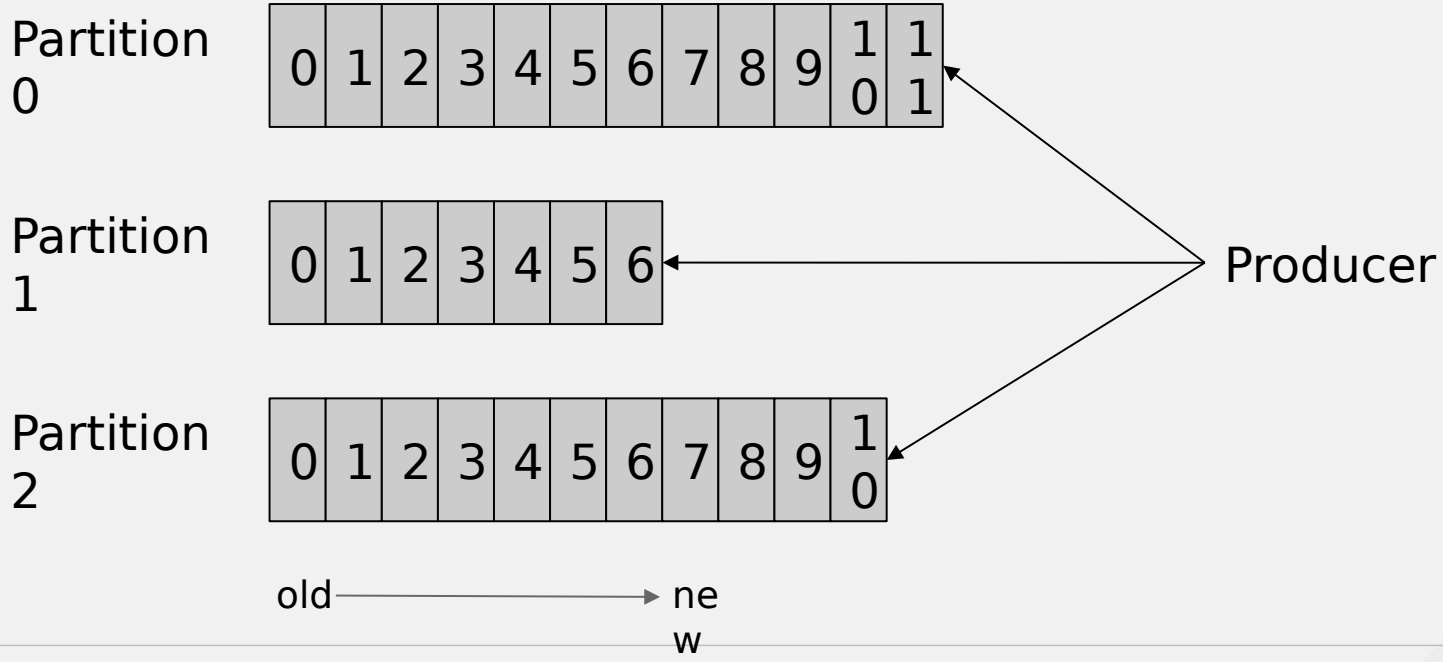
Concepts

Topic & Partitions

- Messages / records are sent to / received from topic
  - Topics are split into one or more partitions
  - Partition = Shard
  - All actual work is done on partition level, topic is just a virtual object
- Each message is written only into a one selected partition
  - Partitioning is usually done based on the message key
  - Message ordering within the partition is fixed
- Clean-up policies
  - Based on size / message age
  - Compacted based on message key

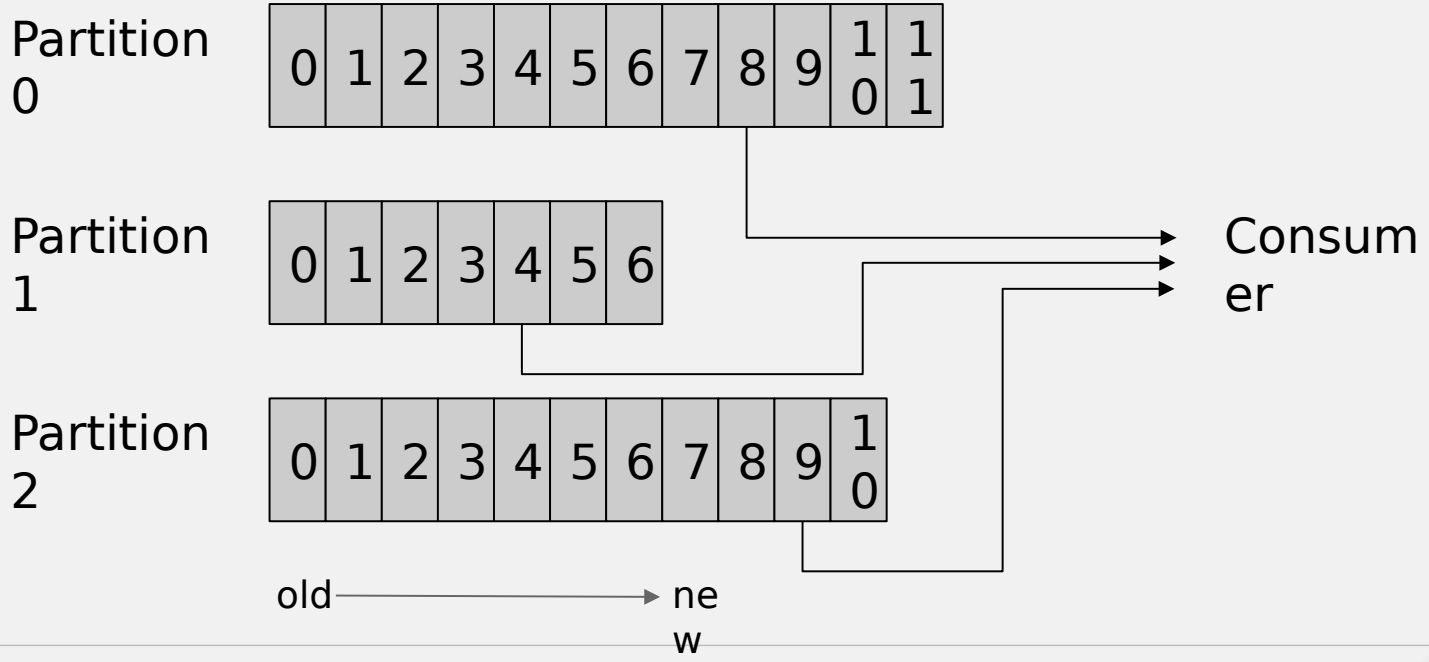
Topic & Partitions

Producing messages



Topic & Partitions

Consuming messages



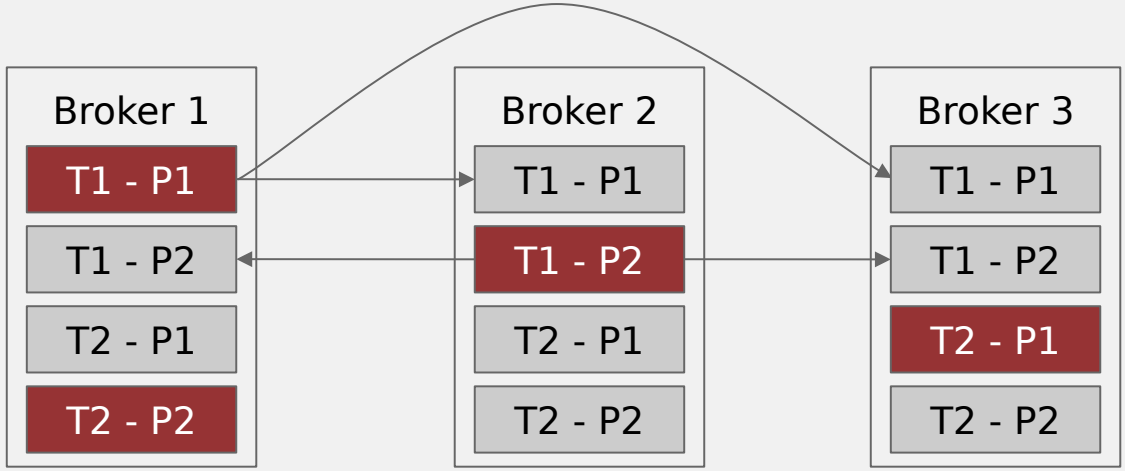
Replication

Leaders & Followers

- They are “backup” for a partition
  - Provide redundancy
- It’s the way Kafka guarantees availability and durability in case of node failures
- Two roles :
  - Leader : a replica used by producers/consumers for exchanging messages
  - Followers : all the other replicas
    - They don’t serve client requests
    - They replicate messages from the leader to be “in-sync” (ISR)
  - A replica changes its role as brokers come and go

Partitions Distribution

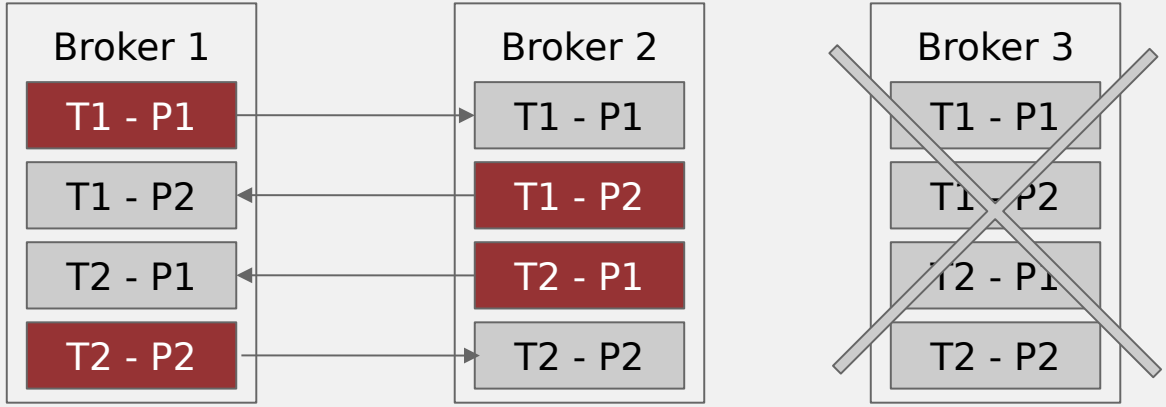
Leaders & Followers



- Leaders and followers spread across the cluster
  - producers/consumers connect to leaders
  - multiple connections needed for reading different partitions

Partitions Distribution

Leader election



- A broker with leader partition goes down
- New leader partition is elected on different node

Clients

- They are really “smart” (unlike “traditional” messaging)
- Configured with a “bootstrap servers” list for fetching first metadata
  - Where are interested topics ? Connect to broker which holds partition leaders
  - Producer specifies destination partition
  - Consumer handles messages offsets to read
  - If error happens, refresh metadata (something is changed in the cluster)
- Batching on producing/consuming

Producers

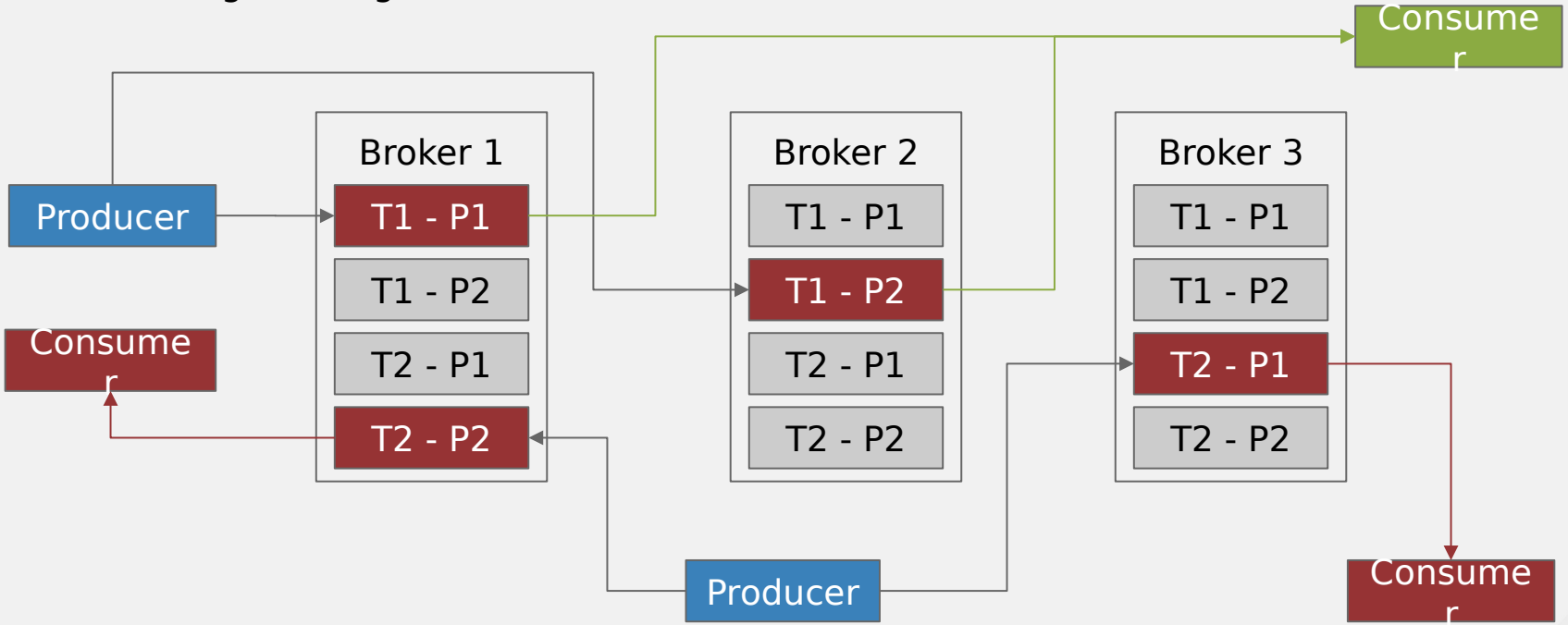
- Destination partition computed on client
  - Round robin
  - Specified by hashing the “key” in the message
  - Custom partitioning
- Writes messages to “leader” for a partition
- Acknowledge :
  - No ack
  - Ack on message written to “leader”
  - Ack on message also replicated to “in-sync” replicas

Consumers

- Read from one (or more) partition(s)
- Track (commit) the offset for given partition
  - A partitioned topic “__consumer_offsets” is used for that
  - Key → [group, topic, partition], Value → [offset]
  - Offset is shared inside the consumer group
- QoS
  - At most once : read message, commit offset, process message
  - At least once : read message, process message, commit offset
  - Exactly once : read message, commit message output and offset to a transactional system
- Gets only “committed” messages (depends on producer “ack” level)

Producers & Consumers

Writing/Reading to/from leaders



Consumer: partitions assignment

Available approaches

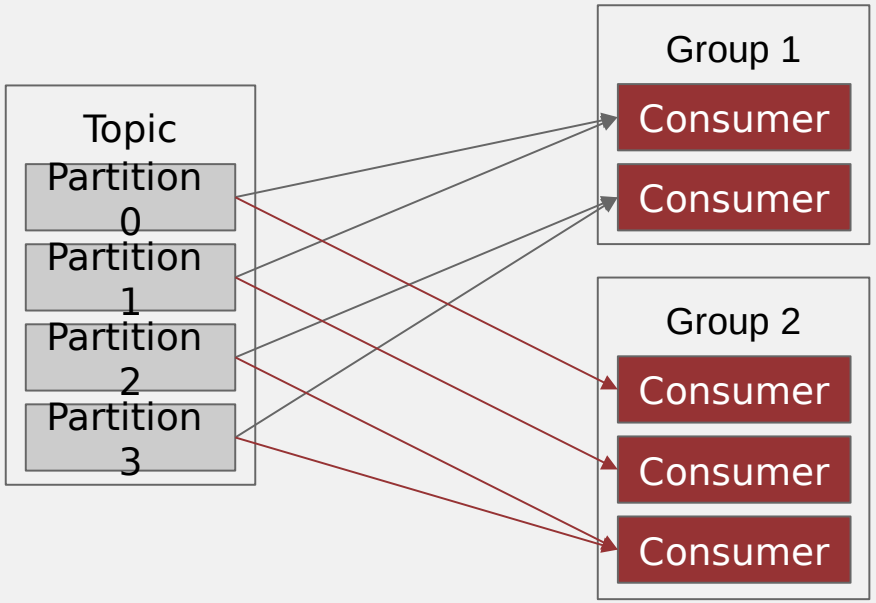
- The consumer asks for a specific partition (assign)
  - An application using one or more consumers has to handle such assignment on its own, the scaling as well
- The consumer is part of a “consumer group”
  - Consumer groups are an easier way to scale up consumption
  - One of the consumers, as “group lead”, applies a strategy to assign partitions to consumers in the group
  - When new consumers join or leave, a rebalancing happens to reassign partitions
  - This allows pluggable strategies for partition assignment (e.g. stickiness)

Consumer Groups

- Consumer Group
  - Grouping multiple consumers
  - Each consumer reads from a “unique” subset of partition → max consumers = num partitions
  - They are “competing” consumers on the topic, each message delivered to one consumer
  - Messages with same “key” delivered to same consumer
- More consumer groups
  - Allows publish/subscribe
  - Same messages delivered to different consumers in different consumer groups

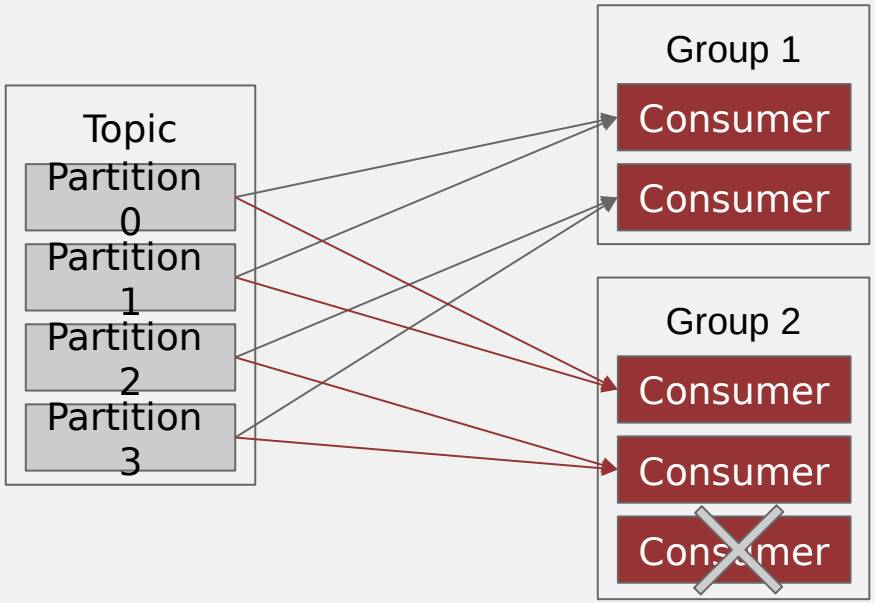
Consumer Groups

Partitions assignment



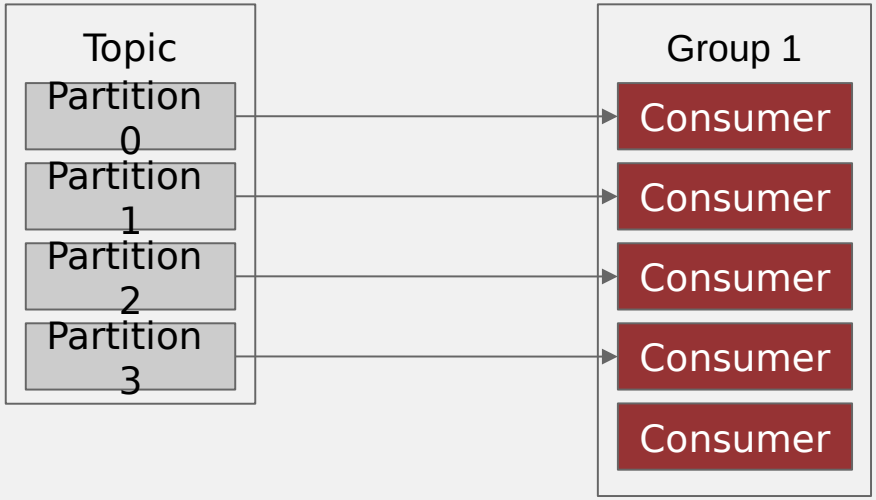
Consumer Groups

Rebalancing



Consumer Groups

Max parallelism & Idle consumer



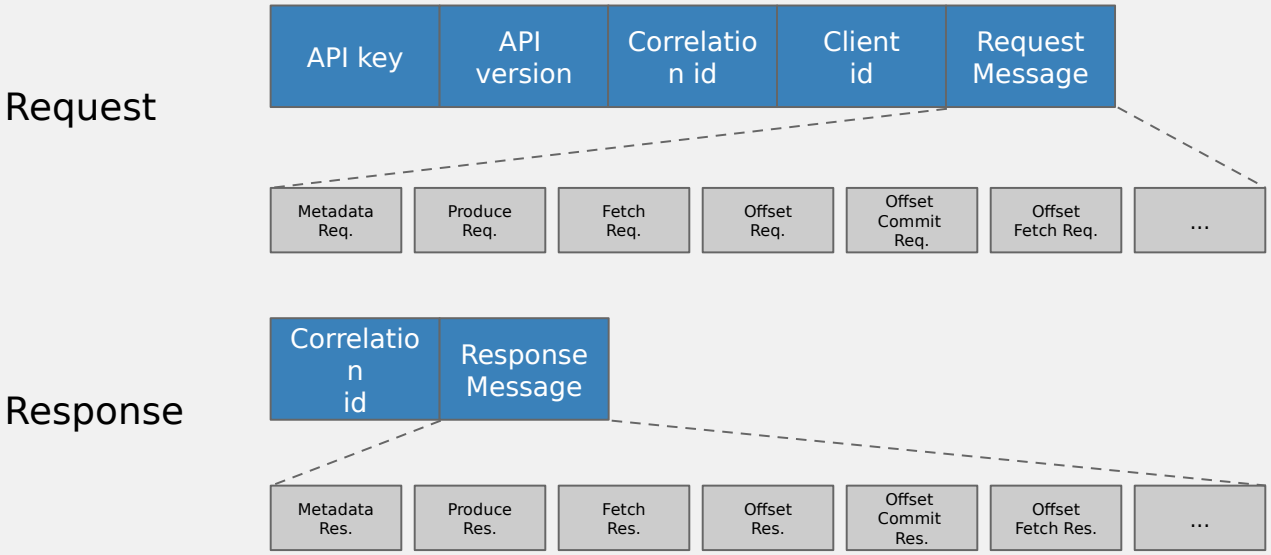
Protocol

Kafka Protocol

- It’s not standard
- It’s a binary protocol over TCP
- Request/Response message pairs
- All requests are initiated by the client
  - A client could be a broker as well, for inter-broker communication (i.e. followers)
- APIs
  - Core client requests (metadata, send, fetch, offsets management)
  - Group membership (join, sync, leave) and heartbeat
  - Administrative (describe, list, ...)
- Evolves with every Kafka version

Kafka Protocol

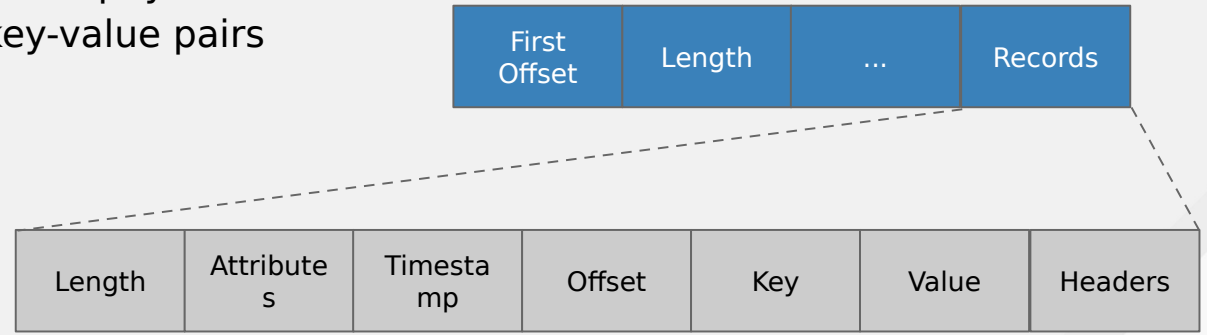
At 20000 feet



Messages

RecordBatch & Record

- Messages are “batched”
- Messages are simply byte arrays
  - Timestamp : create or log append
  - Key : used for specifying the destination partition
  - Value : with custom payload
  - Header : with key-value pairs



Internals

Controller

- The controller is one of the brokers and it is responsible for maintaining the leader/follower relationship for all the partitions
- When a node shuts down, it is the controller that tells other replicas to become partition leaders to replace the partition leaders on the node that is going away
- Elected through Zookeeper

Fast

- Zero-copy
  - it calls the OS kernel direct rather than at the application layer to move data fast
  - It works if clients speak same protocol version, otherwise in-memory copy
- Batching data
  - batching the data into chunks. This minimises cross-machine latency with all the buffering/copying that accompanies this
- Avoids random disk access
  - an immutable commit log it does not need to rewind the disk and do many random I/O operations and can just access the disk in a sequential manner
- Horizontal scaling
  - ability to have thousands of partitions for a single topic spread among

Zookeeper

- Electing controller
  - Zookeeper is used to elect a controller, make sure there is only one and elect a new one it if it crashes
- Cluster membership
  - Which brokers are alive and part of the cluster? This is also managed through ZooKeeper
- Topic configuration
  - Which topics exist, how many partitions each has, where are the replicas, who is the preferred leader, what configuration overrides are set for each topic
- Quotas and ACLs
  - How much data is each client allowed to read and write and who is allowed to do that to which topic

Security

Security

- Encryption between clients and brokers and between brokers
  - Using SSL
- Authentication of clients (and brokers) connecting to brokers
  - Using SSL (mutual authentication)
  - Using SASL (with PLAIN, Kerberos or SCRAM-SHA as mechanisms)
- Authorization of read/writes operation by clients
  - ACLs on resources such as topics
  - Authenticated “principal” for configuring ACLs
  - Pluggable
- It’s possible to mix encryption/no-encryption and authentication/no-authentication

Security

- Zookeeper
  - No encryption support in latest stable version
  - Support for SASL authentication
    - Using Kerberos or locally stored credentials
- Kafka treats information in public
  - Every user can see it
  - ACL protection can be enabled for changing the data

Why should you use AMQ Streams

Why should you use AMQ Streams

- Scalability and Performance
  - Designed for horizontal scalability
  - Cluster sizes from few brokers up to 1000s of brokers
    - 3 nodes usually seen as minimum for production (HA, message durability)
    - Most clusters are under 50 nodes
    - Different approaches: One big cluster vs. several small clusters
  - Scaling has minimal impact on throughput and latency
  - Adding nodes to running cluster is easy

Why should you use AMQ Streams

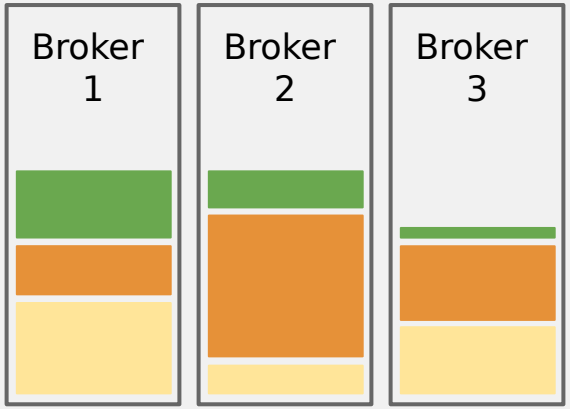
- Message ordering guarantee
  - Messages are written to disk in the same order as received by the broker
  - Messages are read from disk from the requested offset
  - Kafka protocol makes sure that one consumer can read only one partition
  - Note ...
    - Order is guaranteed only within a single partition!
    - No synchronization between producers!

Why should you use AMQ Streams

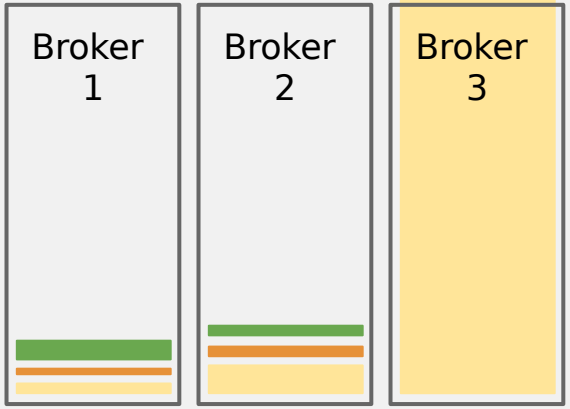
- Message rewind / replay
  - Limited only by available disk space
    - Amount of stored messages has no impact on performance
  - Topic / Partition size has no direct impact on performance
  - Allows to reconstruct application state by replaying the messages
  - Combined with compacted topics allows to use Kafka as key-value store
  - Event sourcing (https://martinfowler.com/eaaDev/EventSourcing.html)
  - Parallel running (https://en.wikipedia.org/wiki/Parallel_running)

What’s the catch?

- Partitioning has its limits
  - 80:20 principle (https://en.wikipedia.org/wiki/Pareto_princ



Ideal scenario



Worst case scenario

What’s the catch?

- Proxying Kafka protocol can be difficult
  - Exposing clusters to the “outside world” might be complicated
  - Clients need access to all brokers in the cluster
  - Producers / consumers might need to maintain large number of TCP connections
  - Can be solved by proxying to another protocol
    - HTTP REST proxy
    - AMQP-Kafka bridge
- Kafka protocol cannot be load-balanced
  - To balance the cluster, the topics / partitions have to be reassigned between nodes

What’s the catch?

- Dumb broker, smart clients
  - Architecture has to be carefully thought through
  - Carefully decide what would be the number of partitions for each topic
    - Too many partitions => Too many brokers / Too much load per broker
    - Too few partitions => Not enough clients running in parallel
  - Adding partitions from topic which is already used might be tricky
    - Messages with the same key might end up in several topics
  - Removing partitions is not possible
  - How should the partitioning be done
    - What should be the key?
    - Balancing between the ordering guarantee and scalability

Use cases

Use cases

- Messaging & Data Integration
  - Comparable to traditional messaging systems
  - Kafka can achieve high throughput and low-latency thanks to partitioning
  - Kafka Connect can be used to easily integrate with existing applications
    - Other messaging systems
    - Databases
    - Data stores

Use cases

- Metrics and Log Aggregation
  - Thanks to its scale, Kafka can be used for delivery
  - Kafka Streams can be used for aggregation and alerting
  - Kafka Connect can be used to store the data
- Website activity tracking
  - Original use case at LinkedIn
  - High volume of data requires scalability
  - Ordering is important for analyzing the events

Use cases

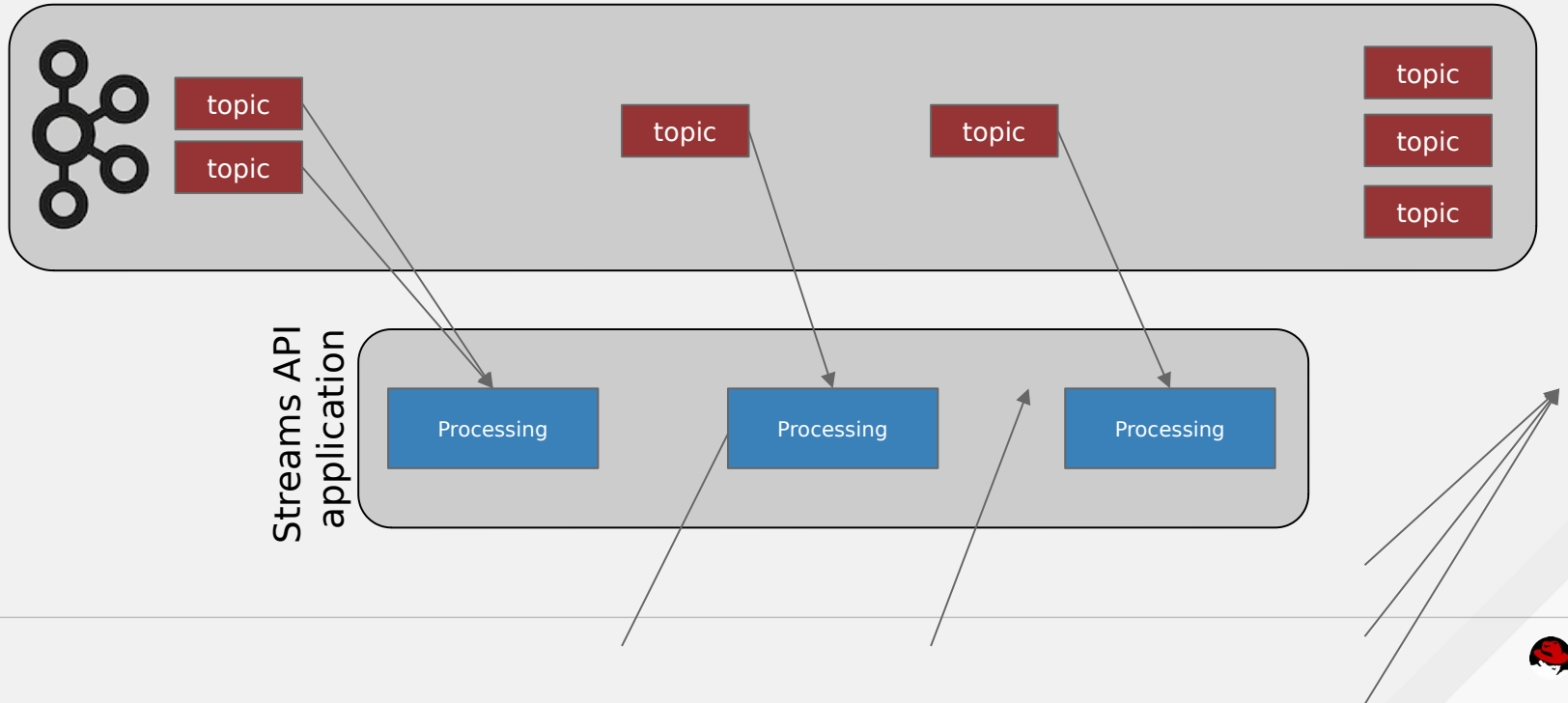
- Event sourcing
  - State changes are logged as a time-ordered sequence of records
  - Can be used to reconstruct the state at any point in time
  - Audit logs, capturing database changes (see Debezium for more details)
- Commit log
  - Shared and distributed data storage for distributed applications
  - Applications can keep in-memory database and persist the data into compacted Kafka topic

Use cases

- Stream processing
  - Read, process and write streams for real-time analysis
  - Data processing can be done using the Kafka Streams library
  - Kafka can be used not only to deliver the stream but also to store the state for stateful operations (aggregations, windowing etc.)
  - Alternatively can use ot tools
    - Apache Storm, Apache Samza or Apache Spark

Use cases

Stream processing



Comparing with other AMQ components

AMQ Streams

vs. AMQ Interconnect

- Scalable fault tolerant messaging network
- Works well across data-centres and geographies
- Advanced message routing
  - Configure how messages are distributed across the messaging network
- Native support for standard protocols (AMQP 1.0)
- Well suited for request/response or RPC patterns
- Doesn’t store messages

AMQ Streams

vs. AMQ Broker

- Supports JMS 1.1 and 2.0 specification
- Native support for standard protocols (AMQP 1.0, MQTT)
- Message routing build around queues and topics
- Advanced queueing
  - TTL, DLQ, Selectors / Message filters, Competing consumers
- Full integration with Red Hat JBoss EAP
- Additional persistence options such as JDBC

AMQ Streams

vs. AMQ Broker

- No long-time persistence
  - Messages are usually deleted once they are consumed
  - Not designed to store messages for a long time
- Not easy to rewind / replay messages

AMQ Streams

vs. AMQ Broker

- Use AMQ Broker when
  - You care about individual messages
  - You use request/response patterns
  - You want to use standard clients and protocols
  - You need TTL or DLQ semantics
  - You need more sophisticated routing patterns

AMQ Streams

vs. AMQ Broker

- Use AMQ Streams when
  - You need scalability to achieve highest possible throughput
  - You need large number of parallel consumers
  - Your messages are cattle and not pets
  - You need message ordering or replay features
  - You can reuse some of the components available in Kafka ecosystem instead (e.g. Kafka Connect plugins)