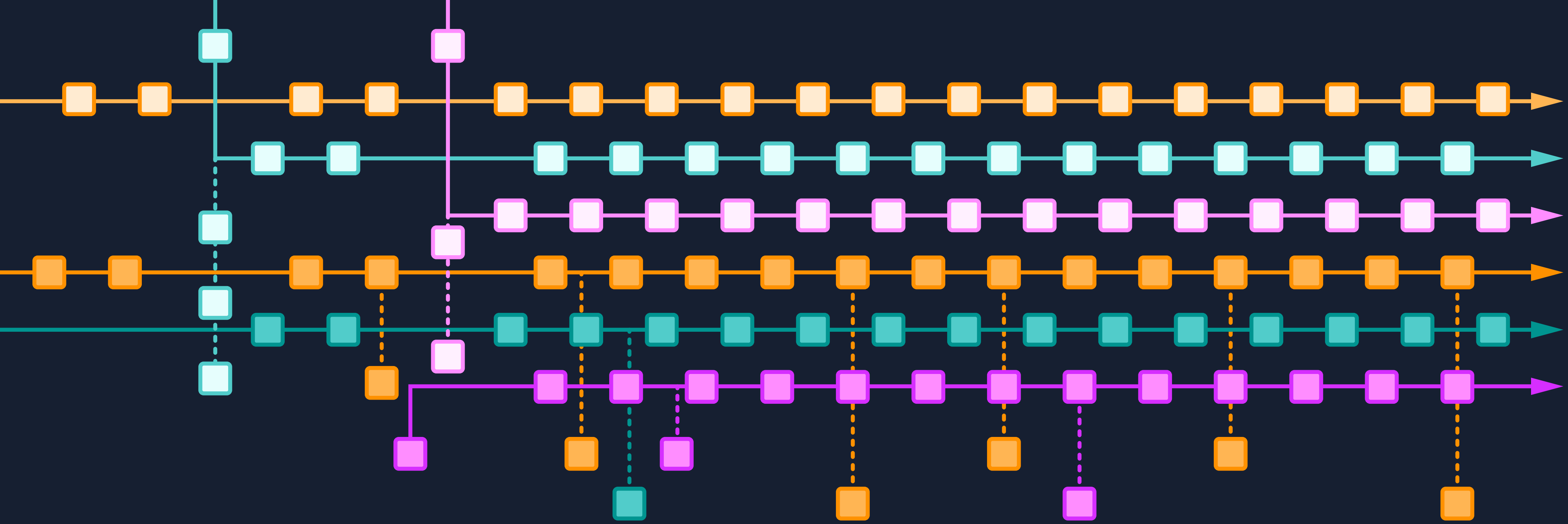


Understanding Streams in Redis and Kafka

A Visual Guide

redis

© 2022 Redis

Table of Contents

© 2022 Redis

Part 1

Introducing the concept of streams

Streams are incredibly useful but can be a little confusing to describe. Part of this is due to the fact that they can be explained in at least two distinct ways and that each description provides only half of the picture. Backend developers (NodeJS and Java) see streams (in contrast to buffers) as a memory-efficient way of processing large amounts of data. The big data folks take a different perspective. They view streams as a way of dealing with data that arrives over time or as a means of decoupling the producers and consumers of that data.

It's like that old story where blind people are asked to describe an elephant. The first one touches the elephant's leg and says the elephant feels like a tree. The second one touches the elephant's trunk and concludes that the elephant feels like a snake and so on.

Figure 1. Image credit: https://stock.adobe.com/search?k=blind+men+and+elephant&asset_id=460307967

© 2022 Redis

Even when you think you have a firm understanding of it, stream processing can still be a very complex topic. In fact, it's difficult to maintain a good mental model of streaming unless you really understand some stream processing systems.

The goal of this e-book is to help you build that mental model. I'll use text, code snippets, and more than 50 illustrations to explain

1. How to think about streams and connect the dots between different perspectives so you get a bigger picture
2. Some of the challenges of handling streams
3. How stream processing systems such as Redis Streams and Kafka work. We are using these two systems as examples in the hope that you'll gain a more thorough understanding as opposed to learning just how one system handles the processing.

Even though we'll be covering deep and complex topics, thanks to the format and the illustrations, it should be an easy and fun read overall.

By the end of this, you should have

- An expert-level theoretical understanding of streams, the challenges of stream processing, and how two stream processing systems (Kafka and Redis streams) work
- Enough knowledge to do a proof-of-concept of Redis Streams or Kafka and to determine which one is best suited for you
- Enough theoretical knowledge to get a head start on certification for either Redis or Kafka

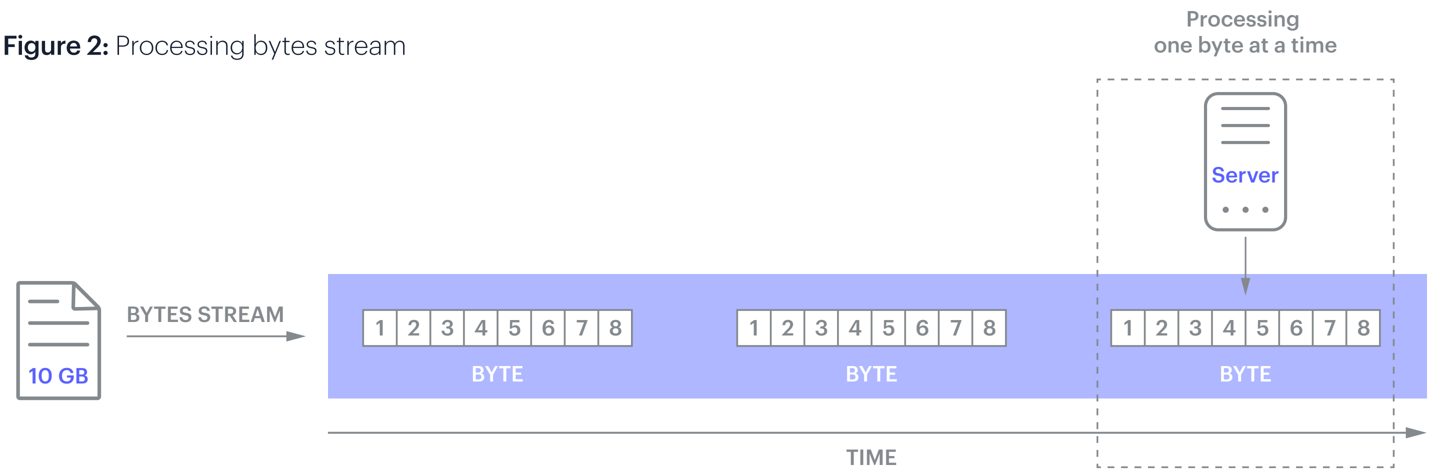
OK, let's get started.

© 2022 Redis

What are streams?

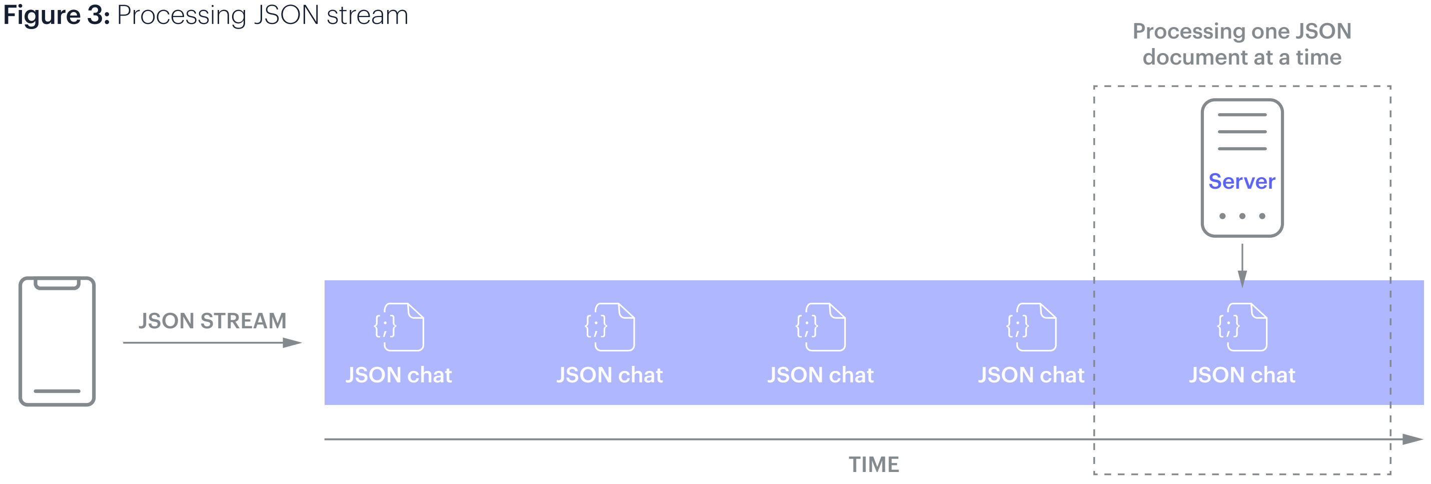
In computer science, a stream is a sequence of data elements (i.e., series of strings, JSON, binary, raw bytes) that are made available for processing in small chunks over time. As with the contents of a text file, this data may be finite, but even then it will be processed in pieces, one word, or one line at a time in a sequence (word after word, or line after line) until all that data has been processed.

Figure 2: Processing bytes stream



In other cases, the data might be infinite and might never end. For example, say you are processing data in a chat messenger server. You'll only get a chat message to process when someone writes one. And it can happen any time and may continue for as long as people keep chatting.

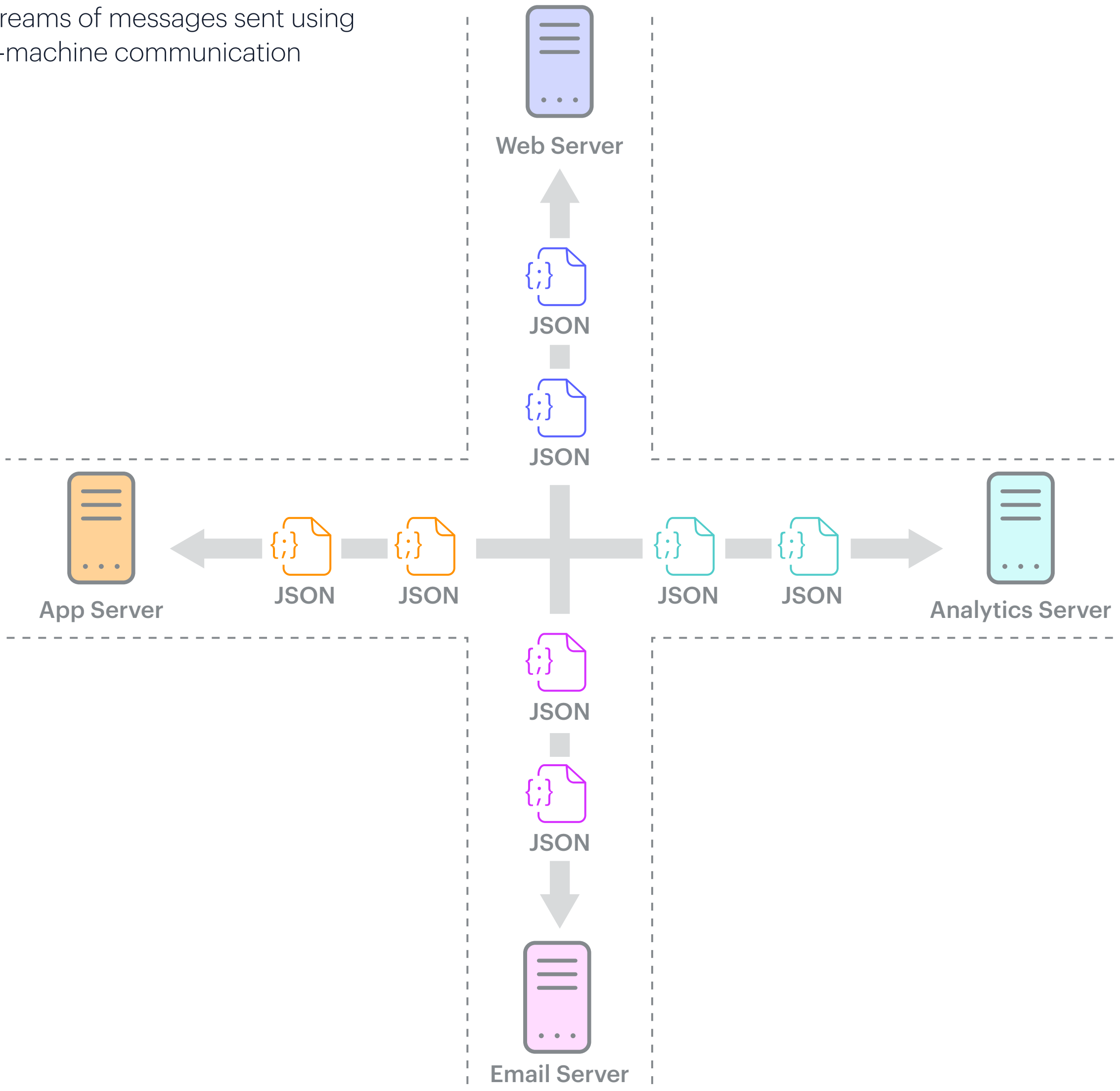
Figure 3: Processing JSON stream



© 2022 Redis

This data can be internal as well as external. It doesn't have to come from the outside world. It could originate from different systems sending messages to each other. For example, a webserver, after receiving payment information, might use a JSON message to tell an email server to send an email via a JSON message. That is machine-to-machine communication. You can also think of these messages as coming in the form of streams because they can come in small pieces and can come over time and at any point in time.

Figure 4: Streams of messages sent using machine-to-machine communication

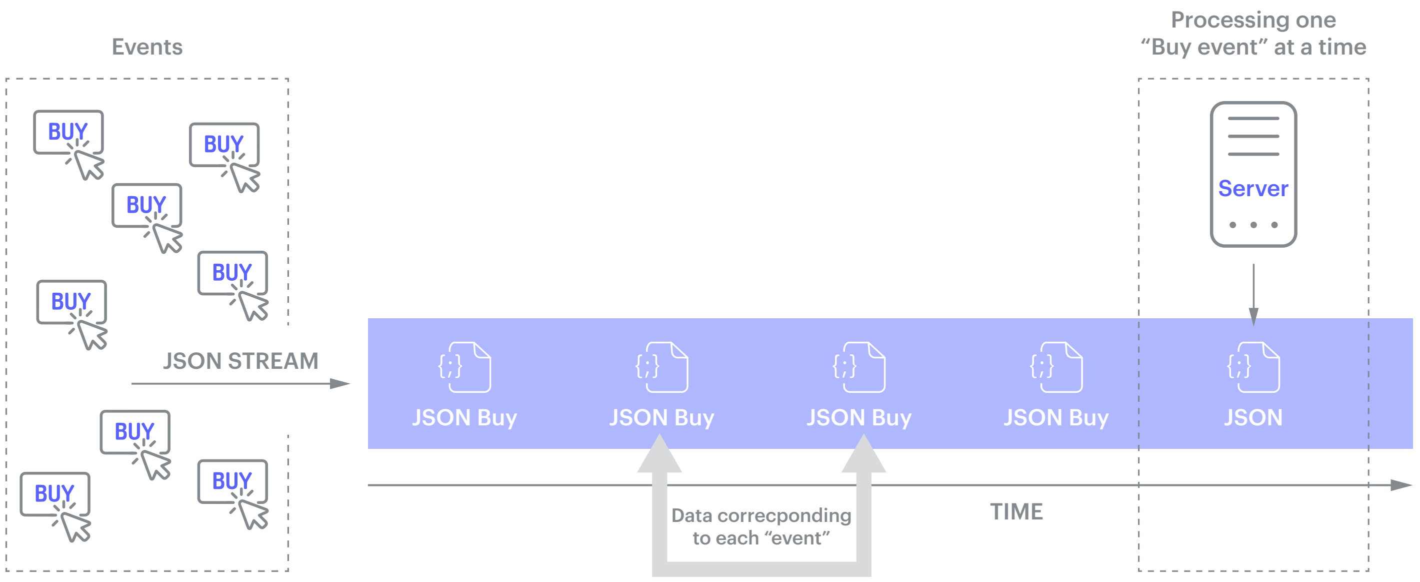


© 2022 Redis

How streams are related to events

An event is simply a mechanism, a trigger that is activated when something has occurred. For example, when someone buys a product, that triggers an event that leads to the creation of a JSON message that contains the person's information, payment amount, product info, and so on. This usually originates at the browser or mobile app, and then the message is sent to the server. Here, the event is the act of buying the product, indicating something occurred. And since the buying event can happen at any time, the resulting data (typically JSON) representing that event flows into the system as a stream.

Figure 5: How events generate streams of data


Events
BUY
JSON STREAM
Processing one "Buy event" at a time
Server
JSON Buy
JSON Buy
JSON Buy
JSON Buy
JSON
Data correcponding to each "event"
TIME

© 2022 Redis

How streams compare to buffering

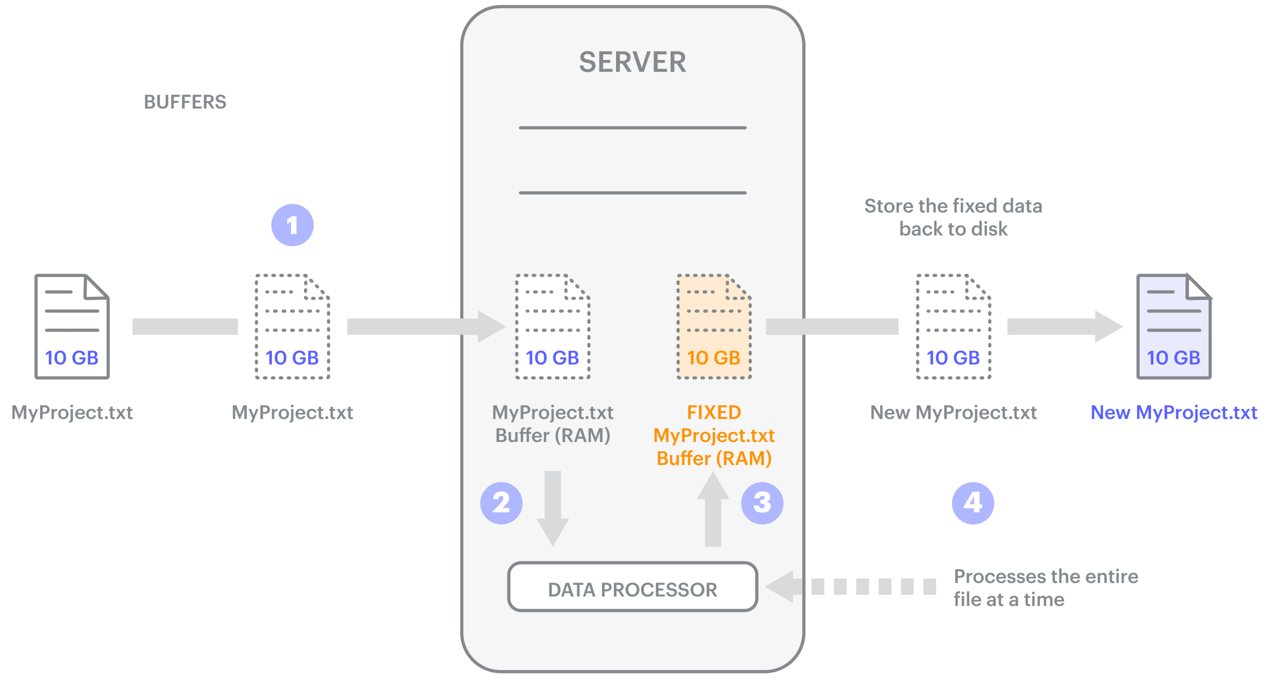
If you ask backend engineers who work in Java, NodeJS, and other programming languages, they'll tell you that streams are more efficient than buffers for processing chunks of data. They come from the perspective of processing large data inside an application server.

An example should help us to understand their perspective a little better.

Say you have a 10 GB file containing hundreds of typos that say "breams" instead of "streams." Let's look at how to use buffers to replace "breams" with "streams," and then we'll see how the process would work using streams.

Processing using just buffers

Figure 6: How buffers are used to process data


SERVER
BUFFERS
1
10 GB
MyProject.txt
10 GB
MyProject.txt
10 GB
MyProject.txt Buffer (RAM)
10 GB
FIXED MyProject.txt Buffer (RAM)
Store the fixed data back to disk
10 GB
New MyProject.txt
10 GB
New MyProject.txt
2
3
4
DATA PROCESSOR
Processes the entire file at a time

Here is how it works:

1. You first read the entire 10 GB file into RAM (can be slow to load all that data).
2. You then send this data to a data processor that will fix all the typos from "breams" to "streams."
3. Once the data processor finishes processing, the new data will be stored back in the RAM (so you may need an additional 10 GB of memory).
4. After all the processing is done, you write the entire file into a new file.

As you can see, this process not only tends to be slow, but it can also take up a lot of memory.

© 2022 Redis

Processing using streams

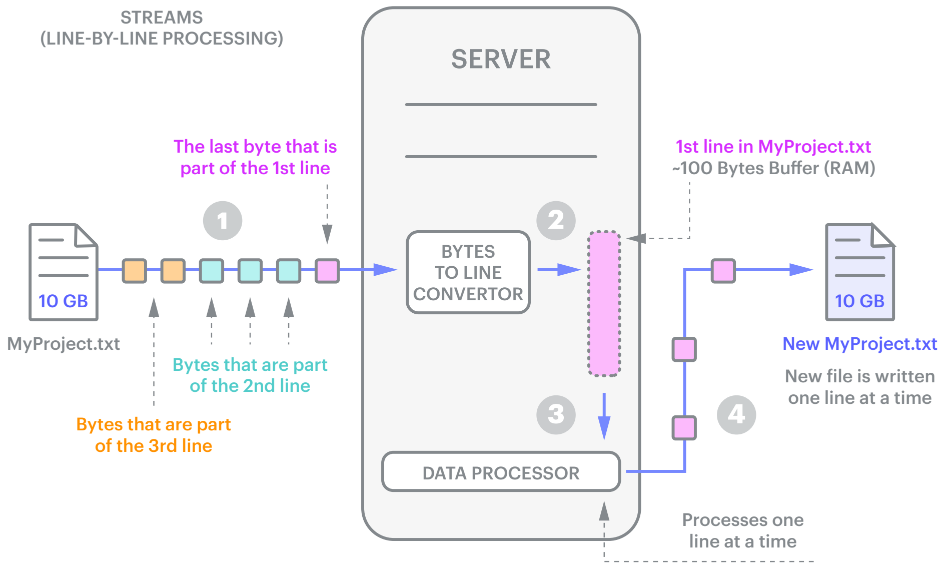
A better approach is to read the data as a stream. Here the data is transferred in bytes, and you figure out how to group these bytes into tiny chunks, then process each chunk.

Here is how it works:

1. Data comes in bytes; that is, one byte at a time.
2. The producer assembles those bytes into a chunk that you've specified. For example, if you've decided to process the file a line at a time, it keeps appending bytes until it spots a newline character that signals the end of this particular chunk. This chunk is now ready to pass on to the consumer.
3. The consumer processes the line, looks for the existence of the typo and, if it finds one, replaces "breams" with "streams."
4. The processed chunk is then written as a stream to the new file.
5. The whole process is repeated until the end-of-file character is detected. At that point, the process is complete, and the stream is closed.

As you can probably see, compared to buffering, streaming has some clear benefits. It's faster, more efficient, and places significantly less of a burden on memory. Although both streaming and buffering require a buffer, in the case of buffering, that buffer must be large enough to contain the entire file or message. With streaming, the buffer only needs to be large enough to accommodate the size of a specified chunk. Moreover, once the current chunk has been processed, the buffer can be cleared and then used to accommodate the next chunk. As a result, regardless of the size of the file, the buffer consumes only 50-100 bytes of memory at a time. Second, because the entire file doesn't need to be loaded into RAM first, the process can begin right away.

Figure 7: How streams are used to process data (Basic "stream processing")


STREAMS
(LINE-BY-LINE PROCESSING)
SERVER
The last byte that is part of the 1st line
1st line in MyProject.txt
~100 Bytes Buffer (RAM)
1
2
10 GB
MyProject.txt
BYTES TO LINE CONVERTOR
10 GB
New MyProject.txt
Bytes that are part of the 2nd line
New file is written one line at a time
Bytes that are part of the 3rd line
3
4
DATA PROCESSOR
Processes one line at a time

Now that you've seen how backend engineers view streams, let's look at streams through the eyes of big data engineers. But first, in order to do so, we need to better understand some of the challenges of stream processing.

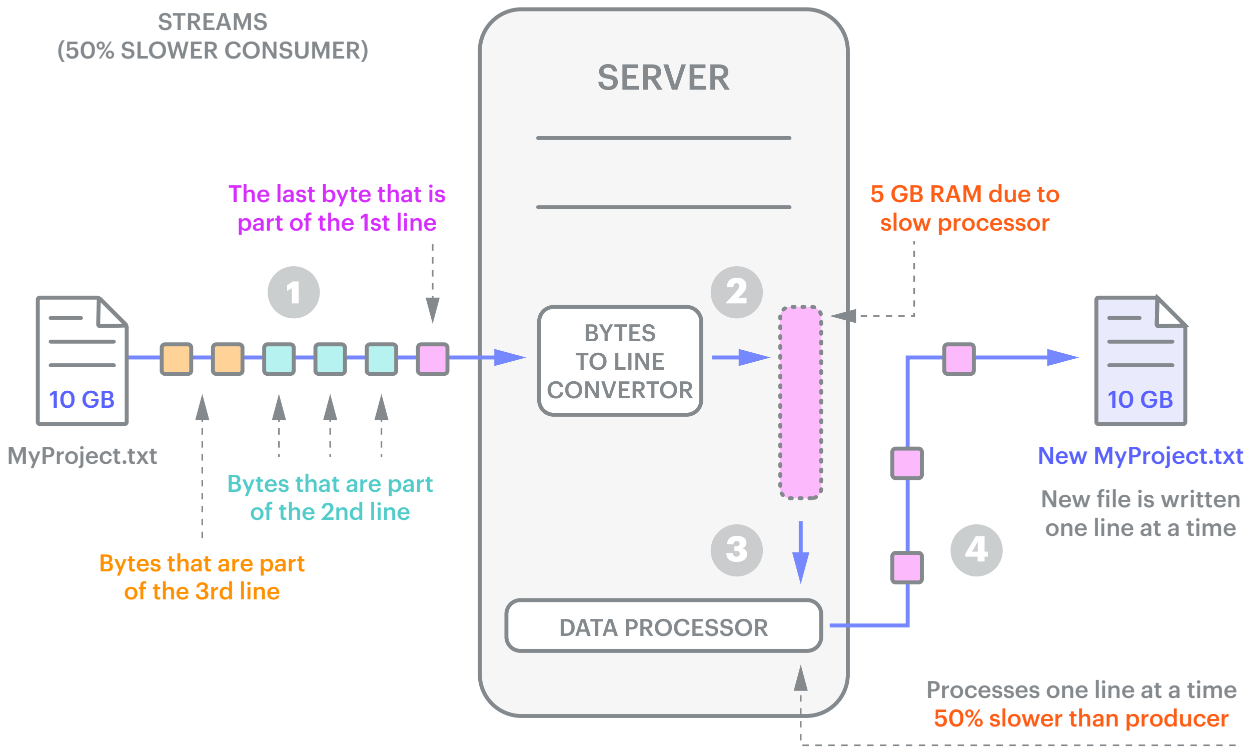
© 2022 Redis

The challenges of stream processing

Although streams can be a very efficient way of processing huge volumes of data, they come with their own set of challenges. Let's take a look at a few of them.

1. What happens if the consumer is unable to process the chunks as quickly as the producer creates them? Taking our current example, what if the consumer is 50"% slower then the producer? If we're starting out with a 10 GB file, that means by the time the producer has processed all 10 GBs, the consumer would only have processed 5 GB. What happens to the remaining 5 GB while it's waiting to be processed? Suddenly, that 50-100 bytes allocated for data that still needs to processed would have to be expanded to 5 GB.

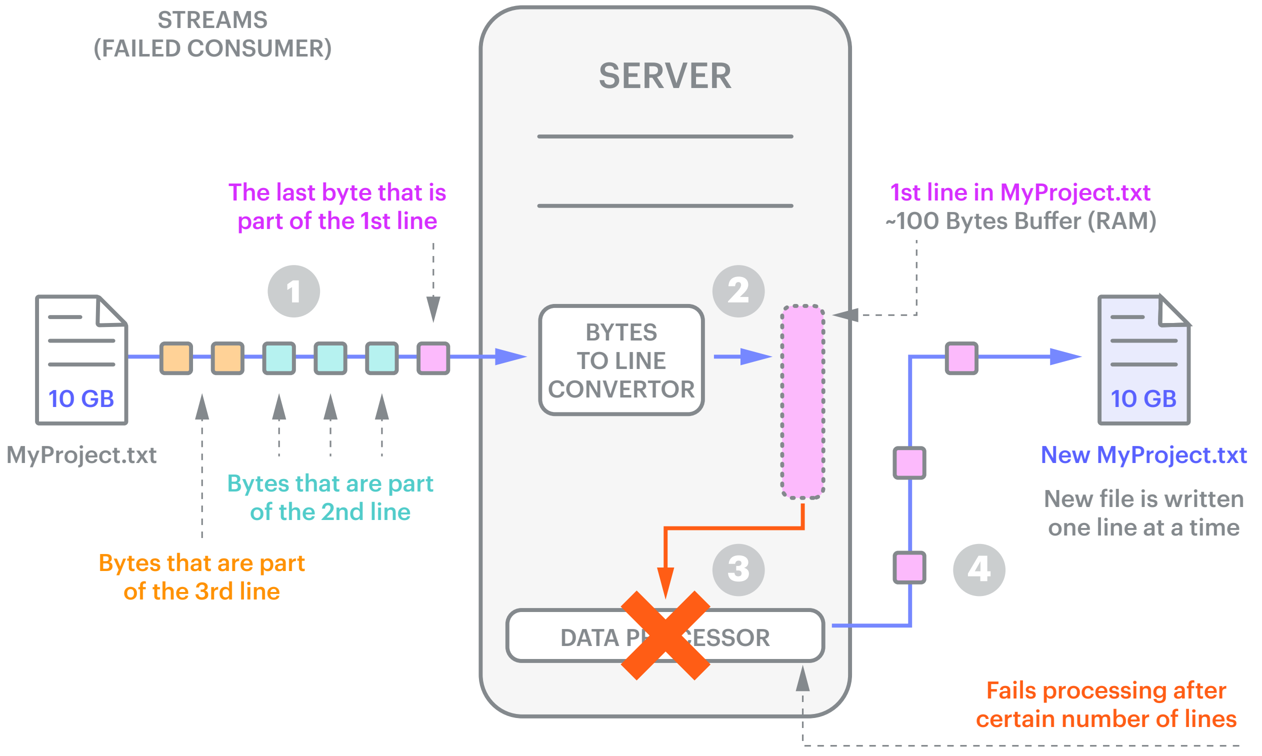
Figure 8: If the consumer is slower than the producer, you'll need additional memory.


STREAMS
(50% SLOWER CONSUMER)
SERVER
The last byte that is part of the 1st line
5 GB RAM due to slow processor
1
2
3
4
10 GB
MyProject.txt
BYTES TO LINE CONVERTOR
10 GB
New MyProject.txt
New file is written one line at a time
Bytes that are part of the 2nd line
Bytes that are part of the 3rd line
DATA PROCESSOR
Processes one line at a time
50% slower than producer

© 2022 Redis

2. And that's just one nightmare scenario. There are others. For example, what happens if the consumer suddenly dies while it's processing a line? You'd need a way of keeping track of the line that was being processed and a mechanism that would allow you to reread that line and all the lines that follow.

Figure 9: When the consumer fails


STREAMS
(FAILED CONSUMER)
SERVER
The last byte that is part of the 1st line
1st line in MyProject.txt
~100 Bytes Buffer (RAM)
1
2
3
4
10 GB
MyProject.txt
BYTES TO LINE CONVERTOR
10 GB
New MyProject.txt
New file is written one line at a time
Bytes that are part of the 2nd line
Bytes that are part of the 3rd line
DATA PROCESSOR
Fails processing after certain number of lines

© 2022 Redis

3. Finally, what happens if you need to be able to process different events and send them to different consumers? And, to add an extra level of complexity, what if you have interdependent processing, when the process of one consumer depends on the actions of another? There's a real risk that you'll wind up with a complex, tightly coupled, monolithic system that's very hard to manage. This is because these requirements will keep changing as you keep adding and removing different producers and consumers.

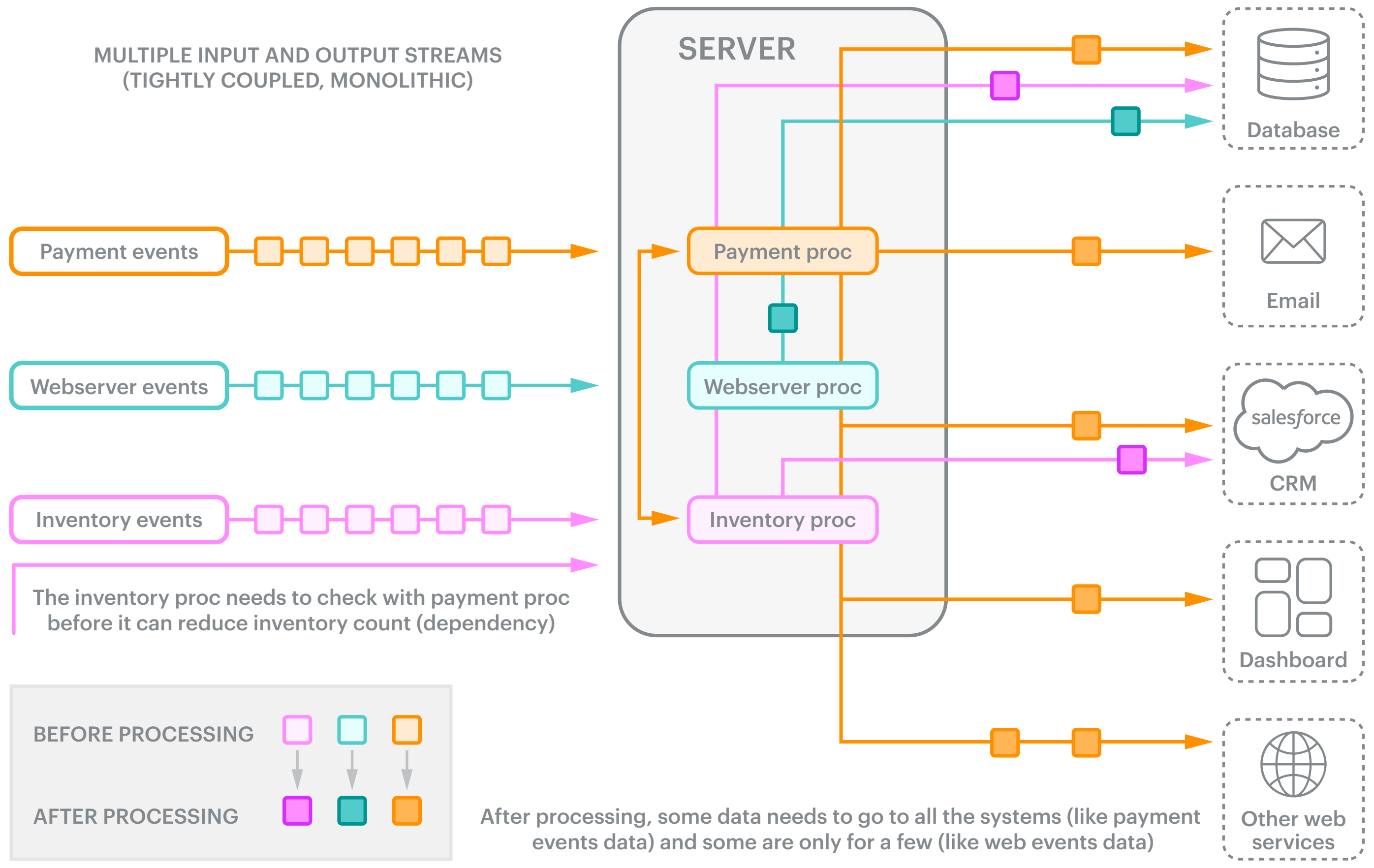
For example (Figure 10), let's assume we have a large retail shop with thousands of servers that support shopping through web apps and mobile apps.

Imagine that we are processing three types of data related to payments, inventory, and webserver logs and that each has a corresponding consumer: a "payment processor," an "inventory processor," and a "webserver events processor." In addition, there is an important interdependency between two of the consumers. Before you can process the inventory, you need to verify payment first. Finally, each type of data has different destinations. If it's a payment event, you send the output to all the systems, such as the database, email system, CRM, and so on. If it's a webserver event, then you send it just to the database. If it's an Inventory event, you send it to the database and the CRM.

As you can imagine, this can quickly become quite complicated and messy. And that's not even including the slow consumers and fault-tolerance issues that we'll need to deal with for each consumer.

© 2022 Redis

Figure 10: The challenge of tight coupling because of multiple producers and consumers


MULTIPLE INPUT AND OUTPUT STREAMS
(TIGHTLY COUPLED, MONOLITHIC)
SERVER
Payment events
Webserver events
Inventory events
Payment proc
Webserver proc
Inventory proc
The inventory proc needs to check with payment proc before it can reduce inventory count (dependency)
Database
Email
salesforce
CRM
Dashboard
Other web services
BEFORE PROCESSING
AFTER PROCESSING
After processing, some data needs to go to all the systems (like payment events data) and some are only for a few (like web events data)

© 2022 Redis

Of course, all of this assumes that you're dealing with a monolithic architecture, that you have a single server receiving and processing all the events. How would you deal with a "microservices architecture"? In this case, numerous small servers (that is, microservices) would be processing the events, and they would all need to be able to talk to each other. Suddenly, you don't just have multiple producers and consumers. You have them spread out over multiple servers.

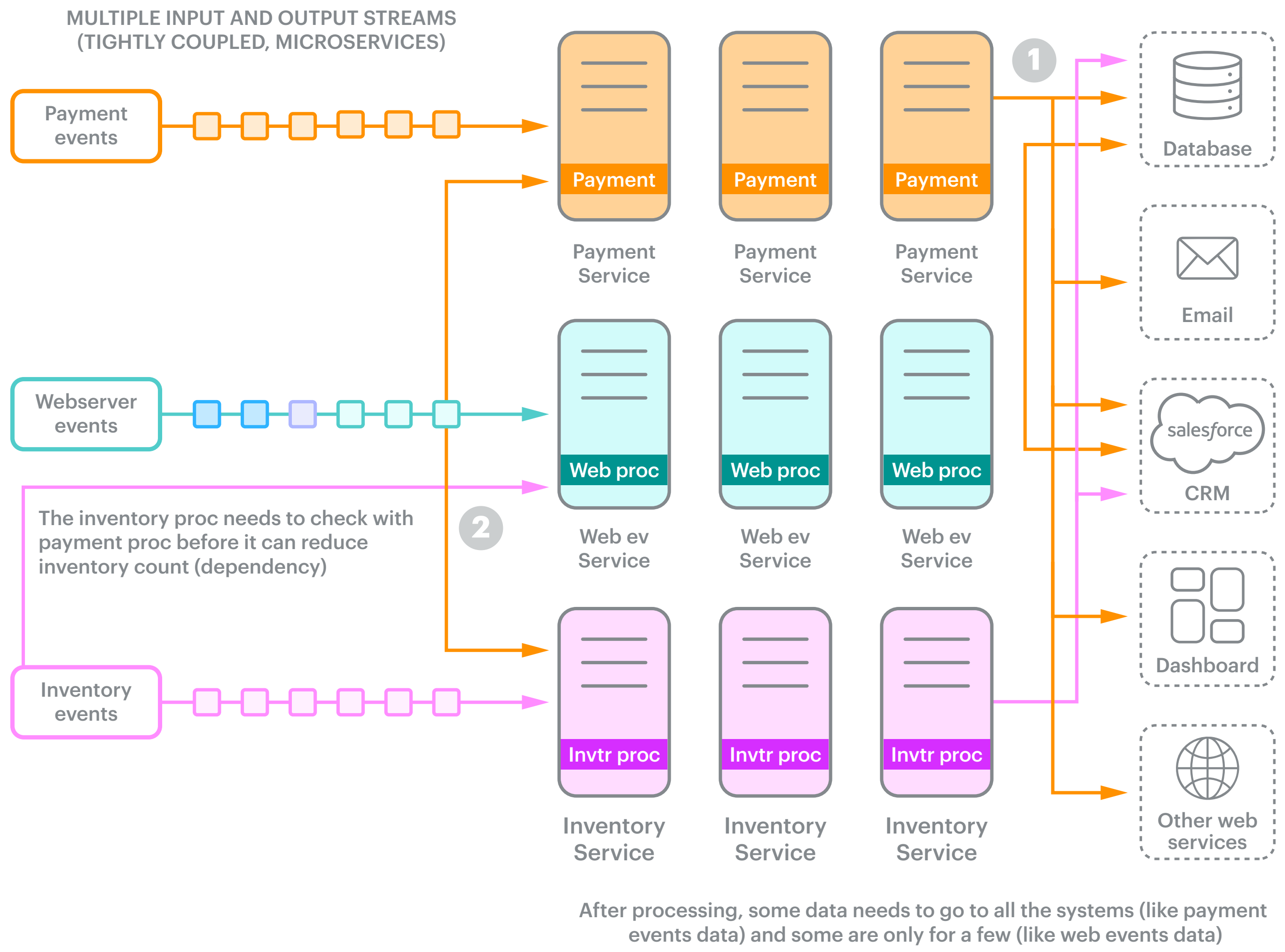
A key benefit of microservices is that they solve the problem of scaling specific services depending on changing needs. Unfortunately, although microservices solve some problems, they leave others unaddressed. We still have tight coupling between our producers and consumers, and we retain the dependency between the inventory microservices and the payment ones. Finally, the problems we pinpointed in our original streaming example remain problems.

1. We haven't figured out what to do when a consumer crashes.
2. We haven't come up with a method for managing slow consumers that doesn't force us to vastly inflate the size of the buffer.
3. We don't yet have a way to ensure that our data isn't lost.

These are just some of the main challenges. Let's take a look at how to address them.

© 2022 Redis

Figure 11: The challenges of tight coupling in the microservices world



© 2022 Redis

Specialized stream processing systems

As we've seen, streams can be great for processing large amounts of data but also introduce a set of challenges. New specialized systems such as Apache Kafka and Redis Streams were introduced to solve these challenges. In the world of Kafka and Redis Streams, servers no longer lie at the center, the streams do, and everything else revolves around them.

Data engineers and data architects frequently share this stream-centered worldview. Perhaps it's not surprising that when streams become the center of the world, everything is streamlined.

Figure 12 illustrates a direct mapping of the tightly coupled example you saw earlier. Let's see how it works at a high level.

Note: We'll go into the details later in the context of Redis Streams and Kafka to give you an in-depth understanding of the following:

1. Here the streams and the data(events) are first-class citizens as opposed to systems that are processing them.
2. Any system that is interested in sending data (producer), receiving data (consumer), or both sending and receiving data (producer + consumer) connects to the stream processing system.
3. Because producers and consumers are decoupled, you can add additional consumers or producers at will. You can listen to any event you want. This makes it perfect for microservices architectures.
4. If the consumer is slow, you can increase consumption by adding more consumers.
5. If one consumer is dependent on another, you can simply listen to the output stream of that consumer and then do your processing. For example, in Figure 11, the inventory service is receiving events from both the inventory stream (purple) and also the output of the payment processing stream (orange) before it processes the inventory event. This is how you solve the interdependency problems.
6. The data in the streams are persistent (as in a database). Any system can access any data at any time. If for some reason data wasn't processed, you can reprocess it.

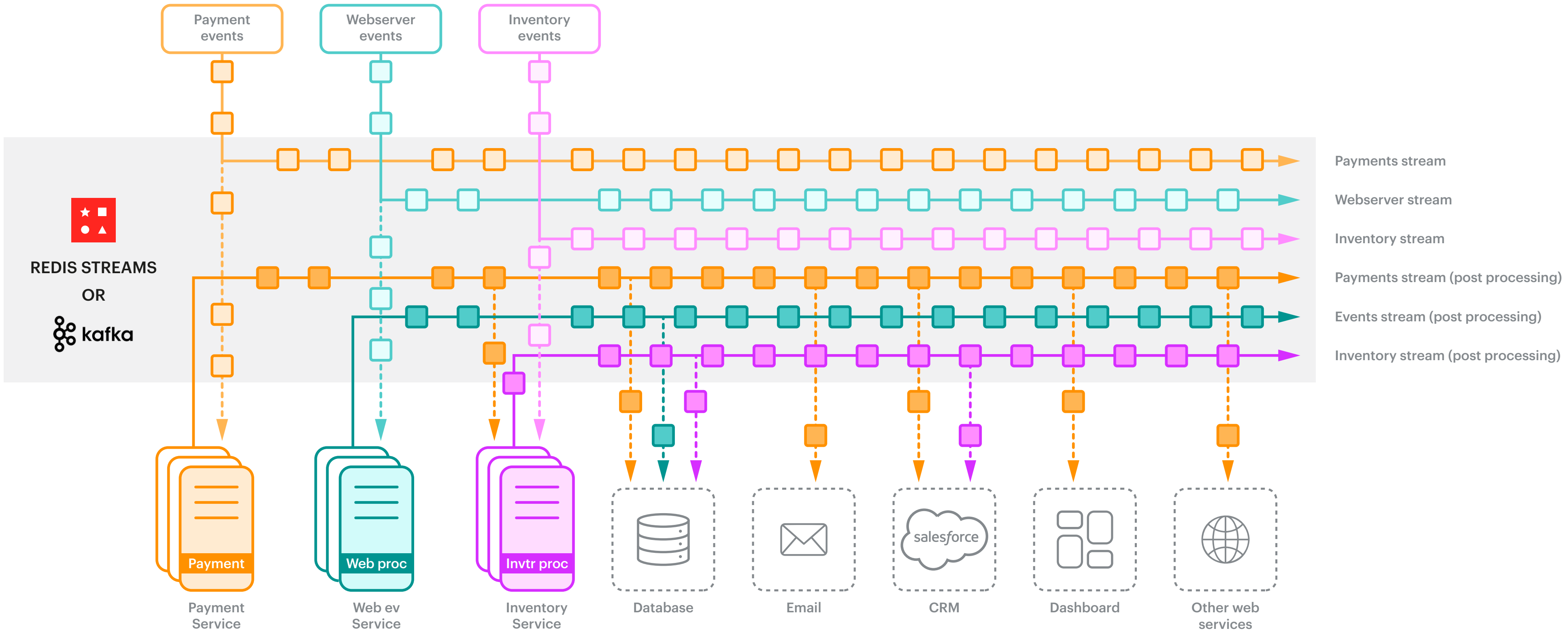
A number of streaming challenges that once seemed formidable, even insurmountable, can readily be solved just by putting streams at the center of the world. This is why more and more people are using Kafka and Redis Streams in their data layer.

This is also why data engineers view streams as the center of the world.

Now that we understand what streams, events, and stream processing systems are, let's take a look at Redis Streams and Kafka to understand stream processing and how they solve various challenges. By the end of this, you should be an expert, at least in the theoretical aspects of stream processing to the extent that you can readily do a proof-of-concept for each system or easily earn Kafka or Redis certification.

© 2022 Redis

Figure 12: When we make streams the center of the world, everything becomes streamlined.



© 2022 Redis

Part 2

Comparing the approaches of Kafka and Redis to handling streams

Apache Kafka is open source (Apache License 2.0, written in Scala) and a leading distributed streaming platform. It's a very feature-rich stream processing system. Kafka also comes with additional ecosystem services such as KsqlDB and Kafka Connect to provide for more comprehensive capabilities.

Redis is an open-source (BSD3, written in C), in-memory database, considered to be the fastest and most loved database. It's also the leading database on AWS. Redis Streams is just one of the capabilities of Redis. With Redis, you'll get a multi-model, multi-data structure database with 6 modules and more than 10 data structures.

So the key thing to remember is that, when you are thinking about Kafka and Redis Streams, you should really think of Kafka and Redis (not just Redis Streams).

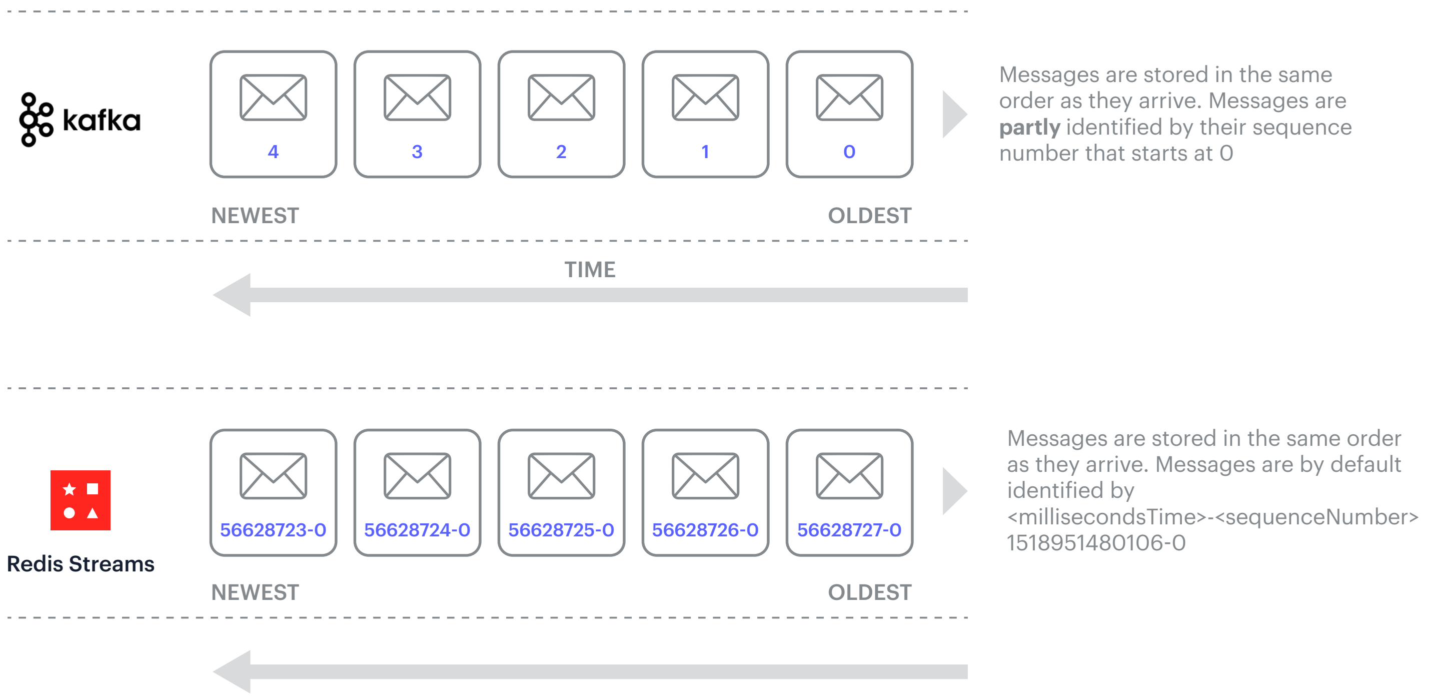
© 2022 Redis

How messages (event data) are stored

Although their storage is similar, Kafka and Redis Streams have different ways of identifying each message. In Kafka, each message is given a sequence number that starts with 0. But each message can only be partly identified by its sequence number. That's because of another concept called a "partition" that we'll get into later.

In Redis Streams, each message by default gets a timestamp as well as a sequence number. The sequence number is provided to accommodate messages that arrive at the exact same millisecond. So if two messages arrived at the exact same millisecond (1518951480106), their ids would look like 1518951480106-0 and 1518951480106-1.

Figure 13: How messages look in Kafka and Redis Streams


kafka
4
3
2
1
0
NEWEST
OLDEST
TIME
Messages are stored in the same order as they arrive. Messages are **partly** identified by their sequence number that starts at 0
Redis Streams
56628723-0
56628724-0
56628725-0
56628726-0
56628727-0
NEWEST
OLDEST
Messages are stored in the same order as they arrive. Messages are by default identified by <millisecondsTime>-<sequenceNumber> 1518951480106-0

© 2022 Redis

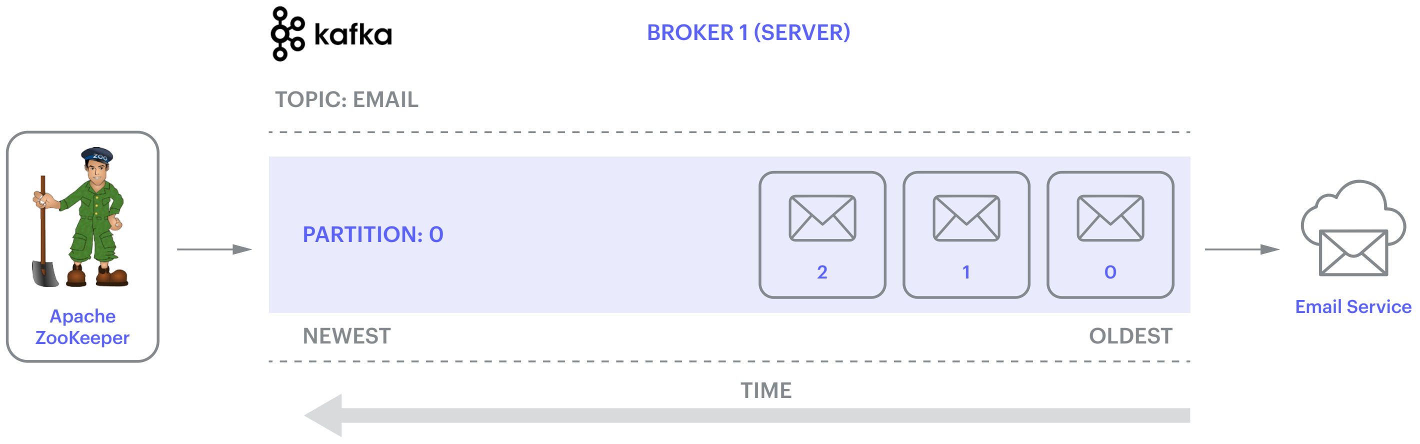
Creating streams

In Kafka, you create what's called a "topic." You can think of this as the name of the stream. However, in Kafka, you also need to understand four key concepts.

1. **Partition:** You can think of it as a file on the disk.
2. **Broker:** You can think of the actual server.
3. **Replication Factor:** The number of duplicate copies of the messages you want to keep.
4. **ZooKeeper:** This is an additional system that you need to use in order to manage Kafka.

We'll get into all these in a bit but for now let's assume you have one partition, one broker, and one replication factor.

Figure 14: How messages look in Kafka for topic Email with one broker, one partition, and one replication factor



Note the command to create a Kafka topic with one partition and one replication factor would look like this:

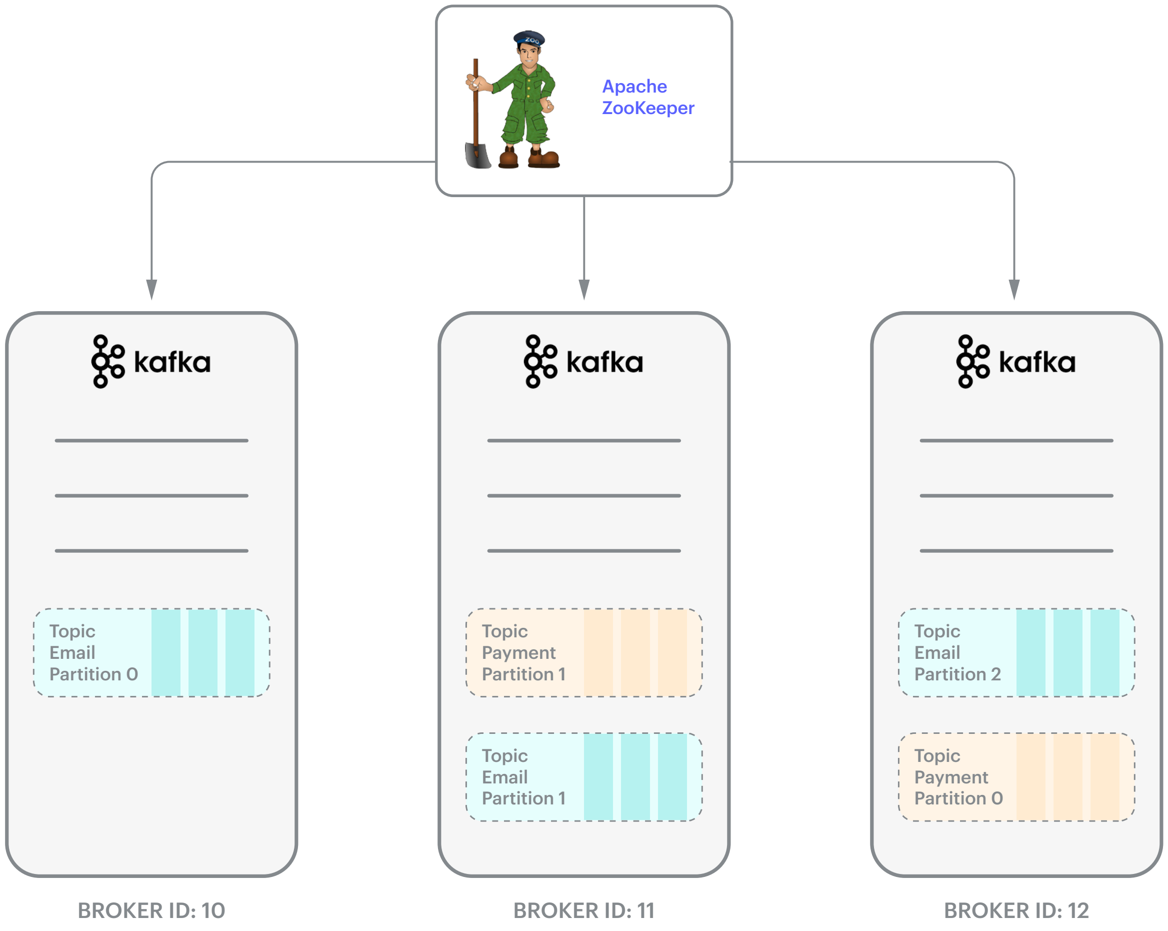
```
> kafka-topics --zookeeper 127.0.0.1:2181 --topic Email --create --partitions 1
--replication-factor 1
```

Important: Kafka no longer requires Zookeeper as of Kafka 3.3.1. Since 3.3.1, Kafka uses Kafka Raft (KRaft), which is built-in. Instead of "--zookeeper" you now use "--bootstrap-server".

© 2022 Redis

Note: The example below (Figure 14a) shows how these would look in a Kafka cluster. We'll discuss that later, but for now just imagine that there is only one broker.

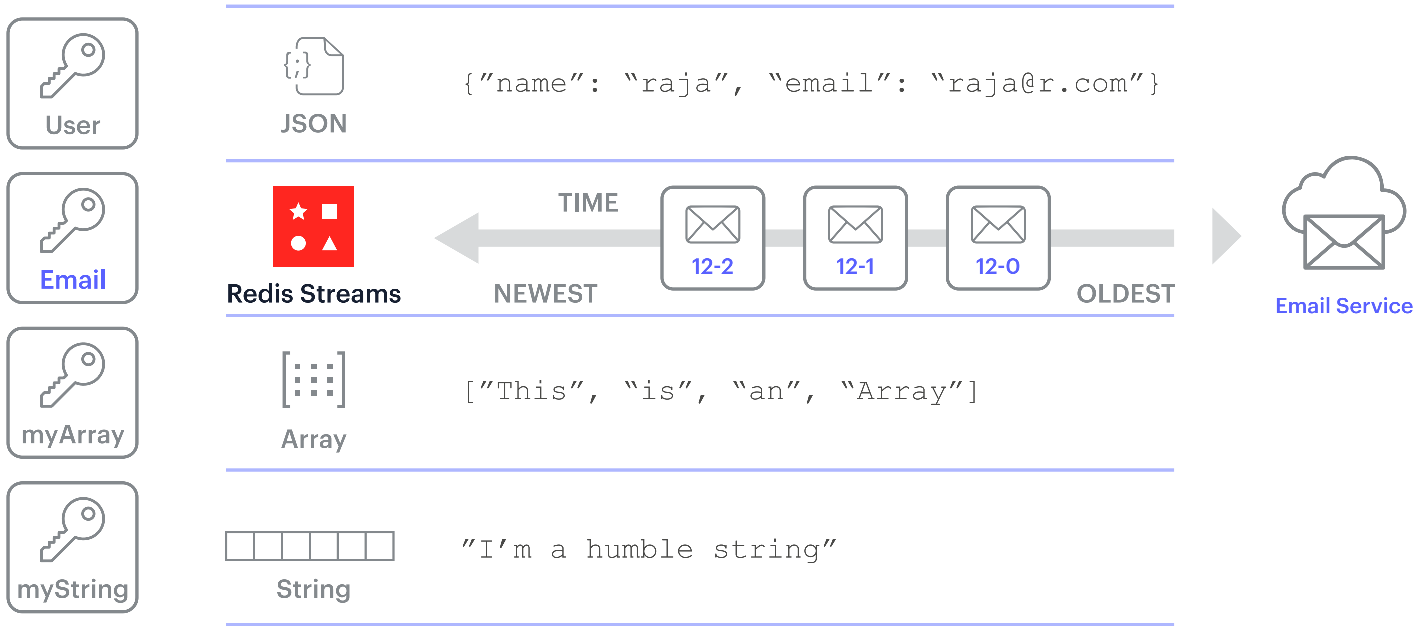
Figure 14a: A Kafka cluster with three brokers (servers), two topics (Email and Payment), where Email-topic has three partitions that are spread across three brokers (10, 11, and 12) and Payment topic has two partitions that are spread across two brokers (11 and 12).



© 2022 Redis

In Redis, you simply create a stream and give it a key. Note that all the data within this stream is part of this single key ("Email"). Note also that this key and its stream just resides along with other keys and data structures. Redis allows for a number of data structures. A stream is just one of them. (See Figure 15.)

Figure 15: How messages look in Redis for an Email stream



The command to create a Redis stream would look like this:

```
XADD Email * email_subject "1st email" email_body "hello world"
```

If the Email stream already exists, it will append the message. If it doesn't exist, Redis will automatically create a stream (using "Email" as its key) and then append the first message. The asterisk will auto-generate the message id (timestamp-sequence) for this message.

© 2022 Redis

Adding messages

Kafka has a concept called "producers." These are responsible for sending messages. They can also send messages with some options such as acknowledgments, serialization format, and so on.

In the following command, you are using a Kafka producer CLI tool to send three messages to the Email topic.

```
$ kafka-console-producer --brokers-list 127.0.0.1:9092 --topic Email
> my first email
> my second email
> my third email
```

In Redis Streams, use the XADD command to send the data in a hash to the Email key.

```
XADD Email * subject "my first email"
XADD Email * subject "my second email"
XADD Email * subject "my third email"
```

In Redis Streams, you can set up acknowledgments and many other things as part of the Redis server or Redis cluster settings. Remember that these settings will get applied to the entire Redis and not just for the Redis Streams data structure.

© 2022 Redis

Consuming messages

Both Kafka and Redis Streams have the concepts of consumers and consumer groups. We'll cover just the basics first.

With Kafka

In Kafka, the following command reads all the messages in the Email topic. The "bootstrap-server" is the main Kafka server. The "--from-beginning" flag tells Kafka to send all the data from the beginning. If we don't provide this flag, the consumer will only retrieve messages that arrive after it has connected to Kafka and started to listen.

```
$ kafka-console-consumer --bootstrap-server 127.0.0.1:9092 --topic Email --from-
beginning

Response:
> my first email
> my second email
> my third email
```

Note: The above consumer client will continue to wait for new messages in a blocking fashion and will display them when they arrive.

© 2022 Redis

With Redis Streams

In Redis Streams, you have two main options:

1. Consume messages by using XREAD (equivalent to Kafka's command). In the command below, "BLOCK 0" tells the Redis CLI to maintain the connection forever (0) in a blocking manner. "Email 0" after the keyword "STREAMS" means to get messages from the "Email" stream and from the beginning of time.

```
XREAD BLOCK 0 STREAMS Email 0
```

Response:

```
1) 1) 1518951480106-0
    2) 1) "subject"
      2) "my first email"
2) 1) 1518951482479-0
    2) 1) "subject"
      2) "my second email"
3) 1) 1518951482480-0
    2) 1) "subject"
      2) "my third email"
```

Notes:

- *If you use "Email $", then it would get only new messages from the "Email" stream. That is, "*`XREAD BLOCK 0 STREAMS Email $`*"*
- *You can use any other timestamp id after the stream name to get messages after that timestamp id. That is, "*`XREAD BLOCK 0 STREAMS Email 1518951482479-0`*"*

© 2022 Redis

2. Consume messages by giving a range of ids or by using some special commands.

c. You can use the command XRANGE and get everything from the smallest ("-") timestamp to the latest one ("+").

```
> XRANGE Email - +
```

Response:

```
1) 1) 1518951480106-0
    2) 1) "subject"
      2) "my first email"
2) 1) 1518951482479-0
    2) 1) "subject"
      2) "my second email"
3) 1) 1518951482480-0
    2) 1) "subject"
      2) "my third email"
```

d. You can also provide timestamps directly.

```
> XRANGE Email 1518951482479  1518951482480
```

Response:

```
1) 1) 1518951482479-0
   2) 1) "subject"
       2) "my second email"
2) 1) 1518951482480-0
   2) 1) "subject"
      2) "my third email"
```

© 2022 Redis

c. You can limit the result by specifying a count.

```
> XRANGE Email - + COUNT 1
```

Response:

```
1) 1) 1518951480106-0
   2) 1) "subject"
       2) "my first email"
```

d. By prefixing the last id with a "(", you can pick up where you left off, starting with the messages that immediately followed the one with that id and keeping the "+" for the ending point. In the example below, we are retrieving two messages that come after a message with a "1518951480106-0" id.

```
> XRANGE Email (1518951480106-0 + COUNT 2
```

Response:

```
1) 1) 1518951482479-0
    2) 1) "subject"
      2) "my second email"
2) 1) 1518951482480-0
    2) 1) "subject"
      2) "my third email"
```

© 2022 Redis

Approaches to scaling consumption

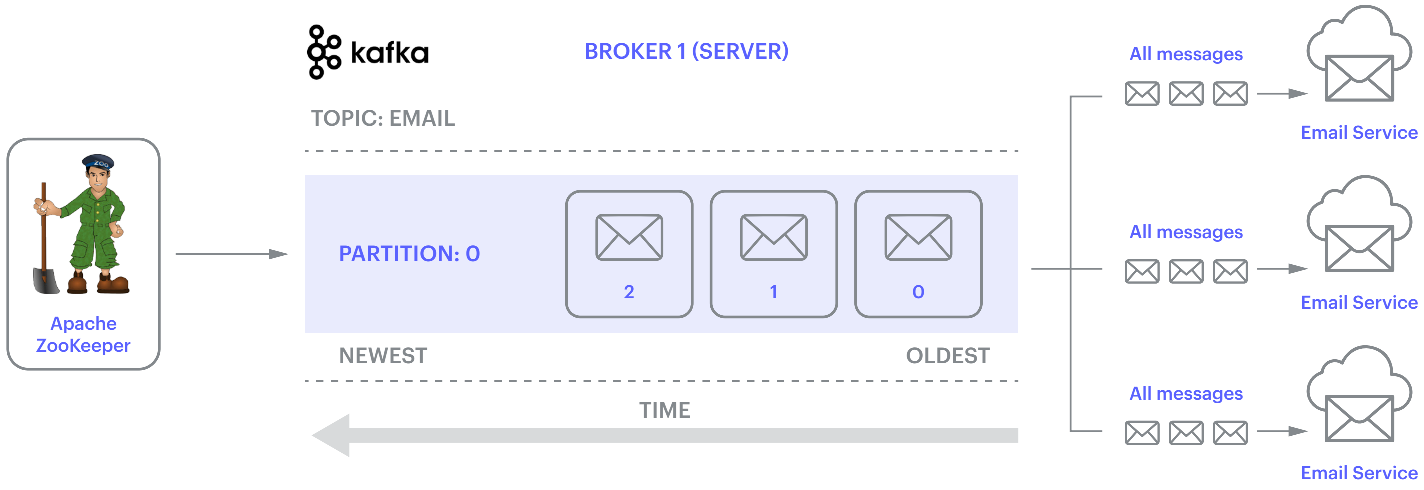
You just saw the basics of producers and consumers in both Kafka and Redis. Now, let's dig in and see how these streaming services scale consumption.

Single partition and multiple consumers

Scenario: Let's imagine you have three emails that need to be processed in no particular order by three email processors (consumers) so you can get the job done in one third the time.

In Kafka, let's say you connected all three consumers to the Email topic. Then all three messages are sent to all three consumers. So you end up processing duplicate messages. This is called a "fan out."

Figure 16: A "fan out" in Kafka when multiple consumers connect to a single topic

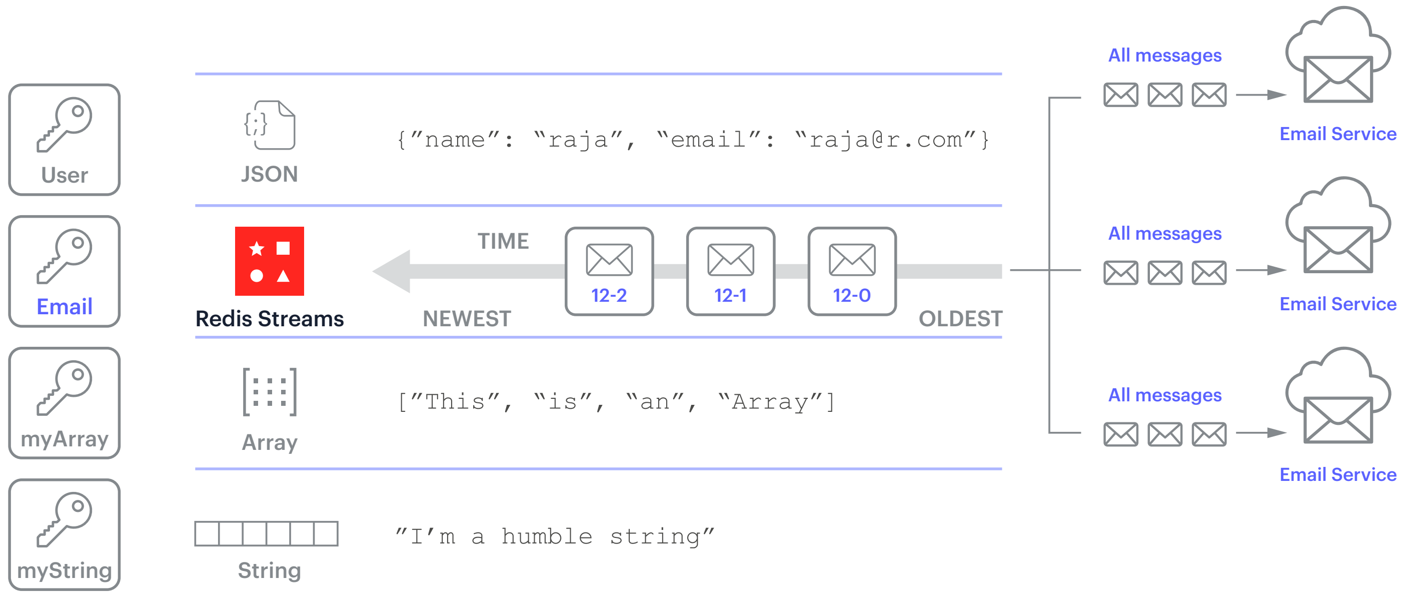


Note: Although it doesn't work for this scenario, it works fine in the chat messenger clients where you can connect multiple users to the same topic and they all receive all chat messages.

© 2022 Redis

It works exactly like that in Redis Streams as well.

Picture 17: A “fan out” in Redis Streams



© 2022 Redis

Multiple partitions and multiple consumers

In Kafka, there is a concept called a partition. You can think of a partition as a physical file on the disk. Partitions are used for scaling purposes. However you should use them carefully, either with "keys" or with "consumer groups." We'll talk about both of them in a bit. But just know that consumers generally don't care and are not aware of the partitions. They just subscribe to a "Topic" (the higher-level abstraction) and consume whatever Kafka sends them.

We are going to cover multiple cases of just using multiple partitions and multiple consumers, and it may look odd at first.

Case 1: An equal number of partitions and consumers (three each)

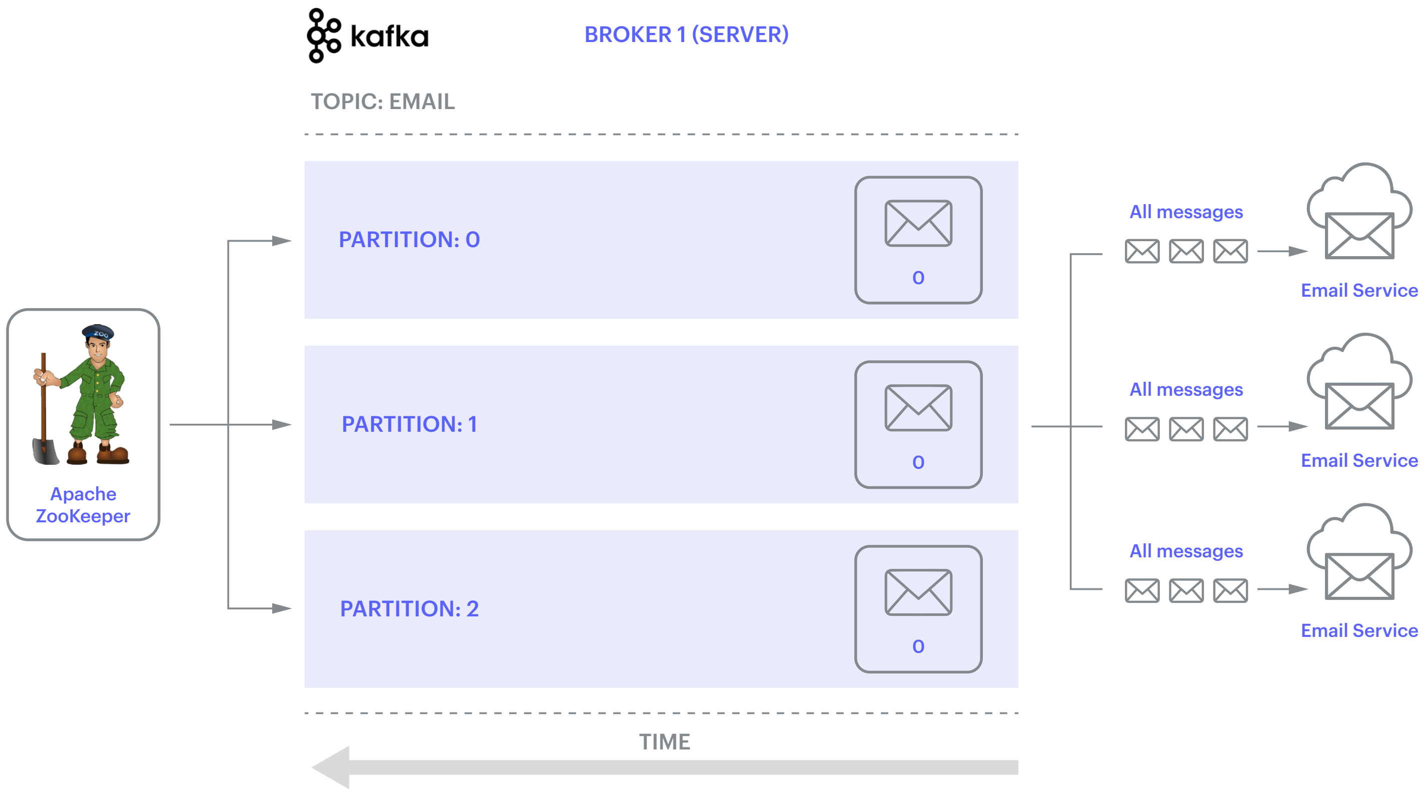
In the example below, we have created three partitions for the "Email" topic using the following command:

```
> kafka-topics --zookeeper 127.0.0.1:2181 --topic Email --create --partitions 3
--replication-factor 1
```

Now when we add three messages to the topic, they are automatically distributed to each of the partitions using a hashing algorithm. So each partition gets just one message each in our example. But when consumers connect to this topic (they are not aware of the partitions), all the messages that are in each partition are sent to each consumer in a fan-out fashion.

© 2022 Redis

Figure 18: A "fan out" when there are multiple partitions (default behavior)



Notes:

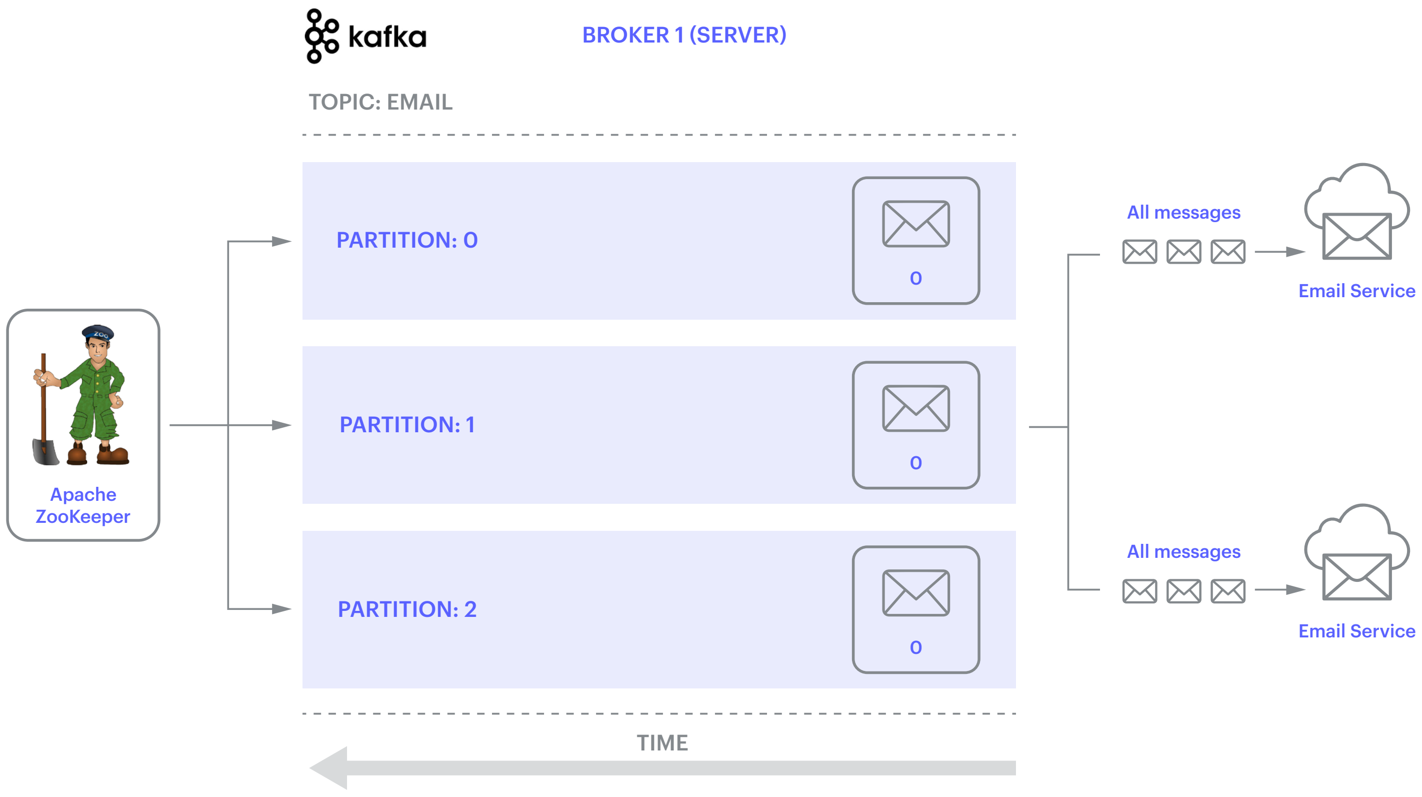
- *Message order: The order in which consumers receive messages is not guaranteed. For example, "Email Service 1" might receive "message 1", "message 3" and finally "message 2". Whereas "Email Service 2" might get them in the following order: "message 3", "message 1" and "message 2". This is because message order is only maintained within a single partition.*
- *Later, we'll learn more about the ordering and how to use keys and consumer groups to alter this default behavior.*

© 2022 Redis

Case 2: More partitions (three) and fewer consumers (two)

It still works the same. Each consumer gets all the messages irrespective of partitions. Message order is not guaranteed.

Figure 19: A “fan out” when there are more partitions than consumers

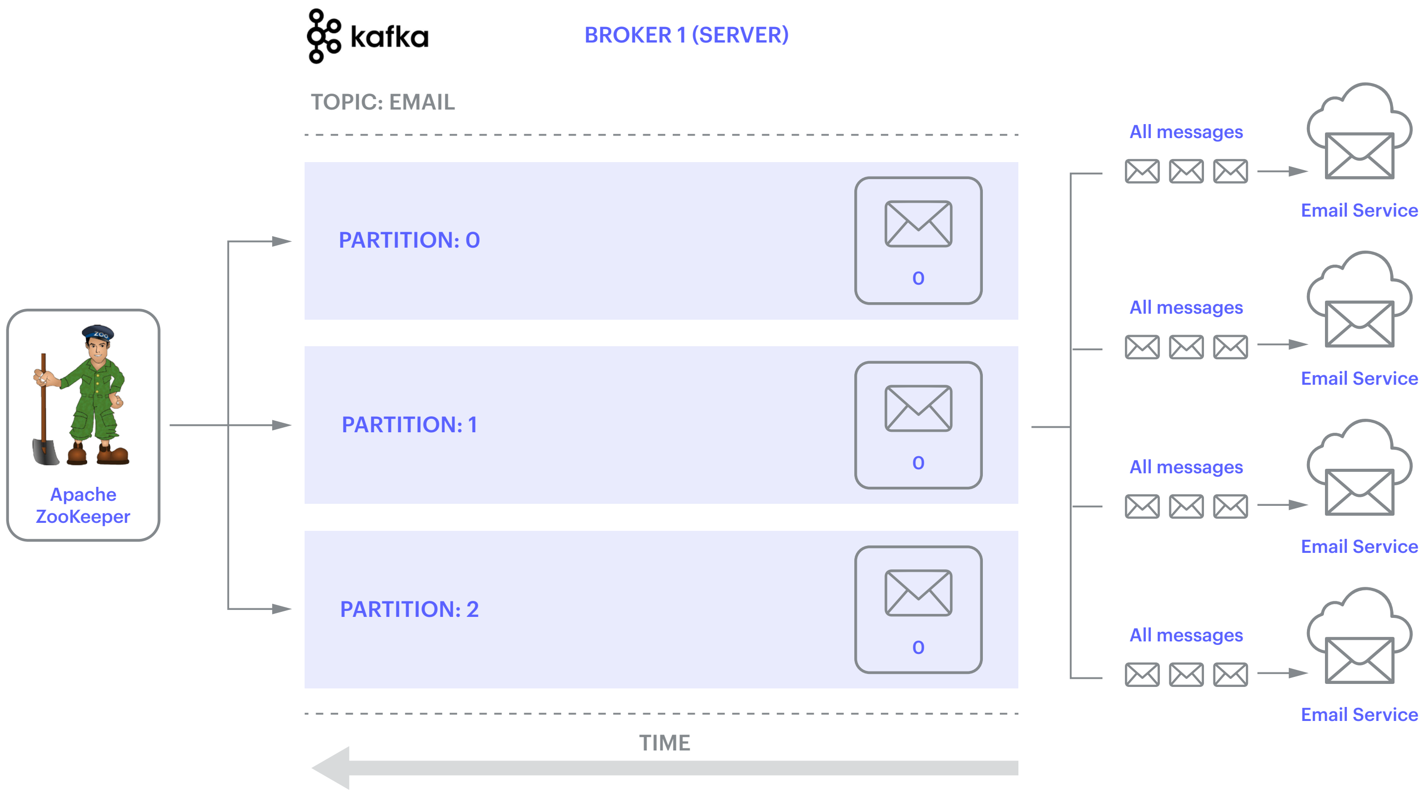


© 2022 Redis

Case 3: Multiple but fewer partitions (three) and more consumers (four)

It still works the same. Each consumer receives all the messages irrespective of partitions. Message order is still random.

Figure 20: A “fan out” when there are fewer partitions than consumers



© 2022 Redis

In Redis Streams, there is no such concept as partitions. If you are using a standalong Redis Server, you don't need to worry about partitioning. If you do want to distribute messages in the same stream across several servers, then you should use a combination of multiple stream keys and a sharding system like Redis Cluster, or some other applicaiton-specific sharding system.

Let's look at how you might implement something resembling partitions in Redis.

You can create "partitions" by creating multiple streams and then distributing data yourself. And on the consumer side, unlike Kafka, since you have direct access to each of these streams, you can consume the data in a fan-out fashion by connecting to all the streams, or by using a key or keys to connect to specific streams.

Say you created three streams: "Email:P0", "Email:P1", and "Email:P2". And say you want to distribute the incoming messages in a round-robin fashion. And finally you want to consume data in a "fan-out" fashion and also in a "per-stream" fashion.

© 2022 Redis

Consuming from Redis Stream "partitions" in a "fan out" fashion (Figure 21)

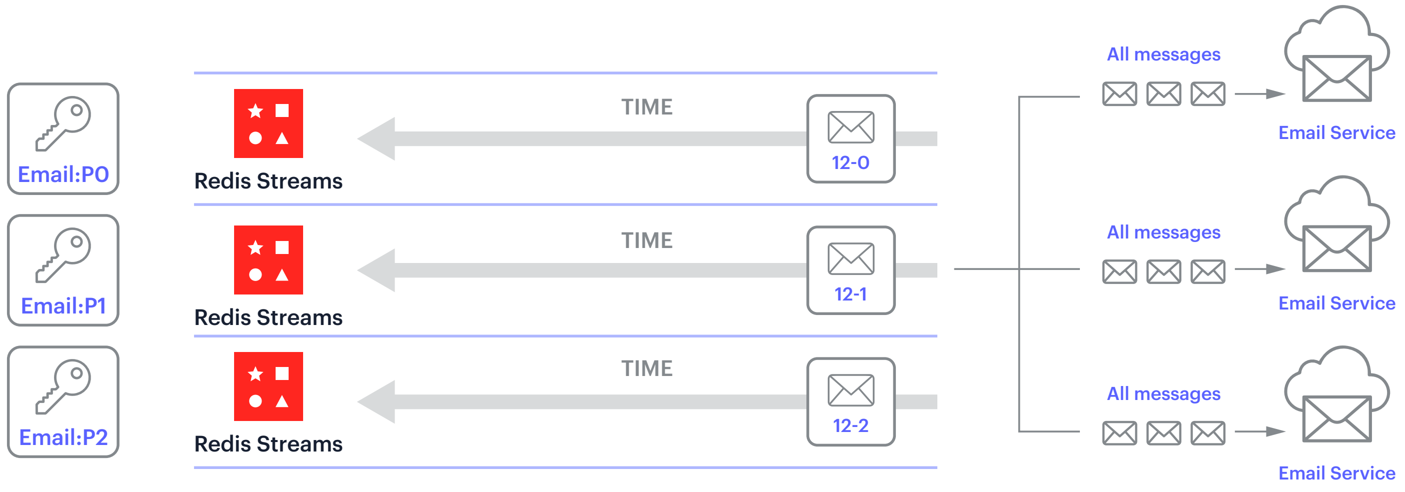
To consume the data in a "fan out" fashion, simply listen to all the streams (Figure 21).

```
//Consumer 1
XREAD BLOCK 0 STREAMS Email:P0 0 Email:P1 0 Email:P2 0
//Consumer 2
XREAD BLOCK 0 STREAMS Email:P0 0 Email:P1 0 Email:P2 0
//Consumer 3
XREAD BLOCK 0 STREAMS Email:P0 0 Email:P1 0 Email:P2 0
```

Notes:

- *BLOCK 0 = Wait indefinitely for messages.*
- *"Email:P0 0" = Read all messages from the beginning (0-0).*
- *By providing multiple stream names and "0"s each consumer can receive all messages.*

Figure 21: How to implement "partitions" in Redis streams and consume messages in a "fan out" manner



© 2022 Redis

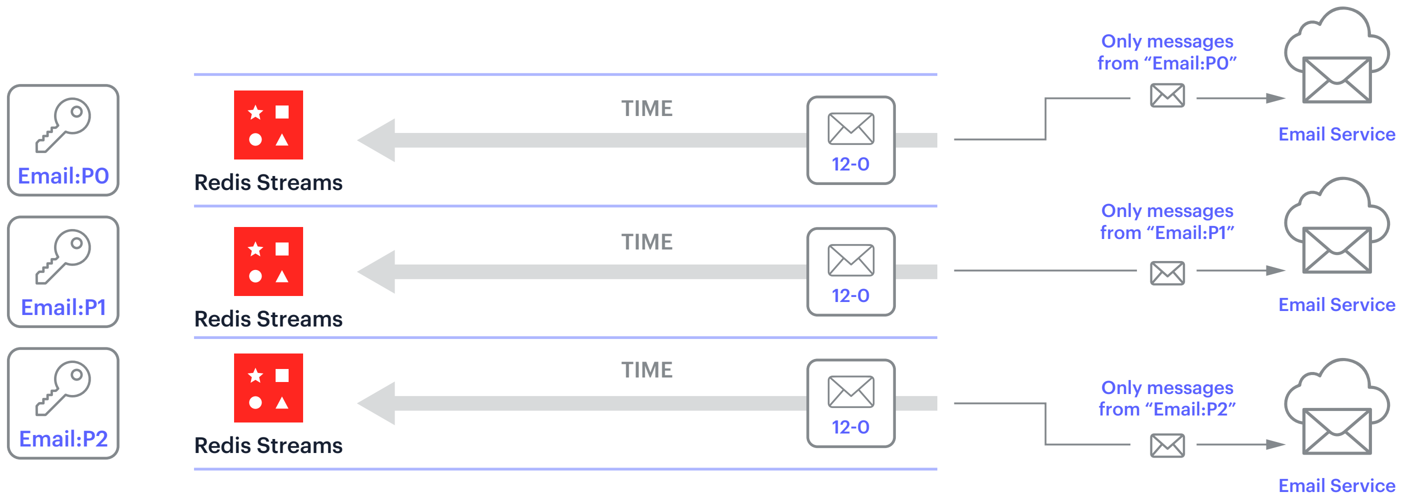
Consuming from Redis Stream "partitions" in a per-stream fashion (Figure 22)

To consume the data in a "per-stream" fashion, simply listen to the stream of your choice. Here message orders are preserved (Figure 22).
//Consumer 1
```
XREAD BLOCK 0 STREAMS Email:P0 0
```
//Consumer 2
```
XREAD BLOCK 0 STREAMS Email:P1 0
```
//Consumer 3
```
XREAD BLOCK 0 STREAMS Email:P2 0
```

To implement a round-robin, you can keep a counter in Redis, say **checkout_counter**, and increment it (INCR checkout_counter) every time you send a new message to a stream. Then use a modulus of the checkout_counter (checkout_counter% number of streams) to determine which stream you should send the next message to.

The following command creates a "check_out:p0" Redis stream.
```
XADD check_out:p0 * message 0 cartId 1
items 5 cost $100
INCR checkout_counter //Use this for
round-robin
```

Figure 22: How to implement "partitions" in Redis Streams and consume them in a "per-stream" manner


Email:PO
Email:P1
Email:P2
Redis Streams
TIME
12-0
Only messages from "Email:PO"
Only messages from "Email:P1"
Only messages from "Email:P2"
Email Service

© 2022 Redis

In-order and in-parallel message processing

Tasks can be handled in order, in parallel, or with a combination of both. As the name implies, with in-order processing, tasks are handled in a specific order. For example, when processing a credit card, we need to first check for validity of the card, then do fraud detection, and then check for a balance. With in-parallel processing, tasks are handled simultaneously so they can be completed more quickly. With in-order and in-parallel processing, the system splits the tasks into groups of tasks that need to be handled in order and then assigns those to different consumers that can perform those ordered tasks in parallel. Kafka and Redis Streams handle this process a little differently. How they differ will become clearer when you look at each system's implementation.

How Kafka handles it

In Kafka, you can send metadata called a "key" (aka "message key") along with the message. When you do that, those messages with the same key will end up in the same partition. This helps in message ordering. Message keys are also useful in other things such as log compaction, but we'll not cover that here.

Secondly, Kafka uses the concept of "consumer groups," where you define a bunch of individual consumers as part of the same consumer group. Kafka will then ensure that messages are distributed across different consumers that are part of that group. This helps in scaling consumption and also avoids "fan out," so each message is read by only one consumer. Another key aspect of consumer groups is that, assuming the number of consumers is greater than or equal to the number of partitions, each consumer in a group is tied to a single partition and is allowed to read messages from just that partition. It cannot read messages from multiple partitions. This way when you combine message keys and consumer groups you'll wind up with highly distributed consumption, although order is still not guaranteed in the event of a consumer failure.

Let's look at an example to make it clear.

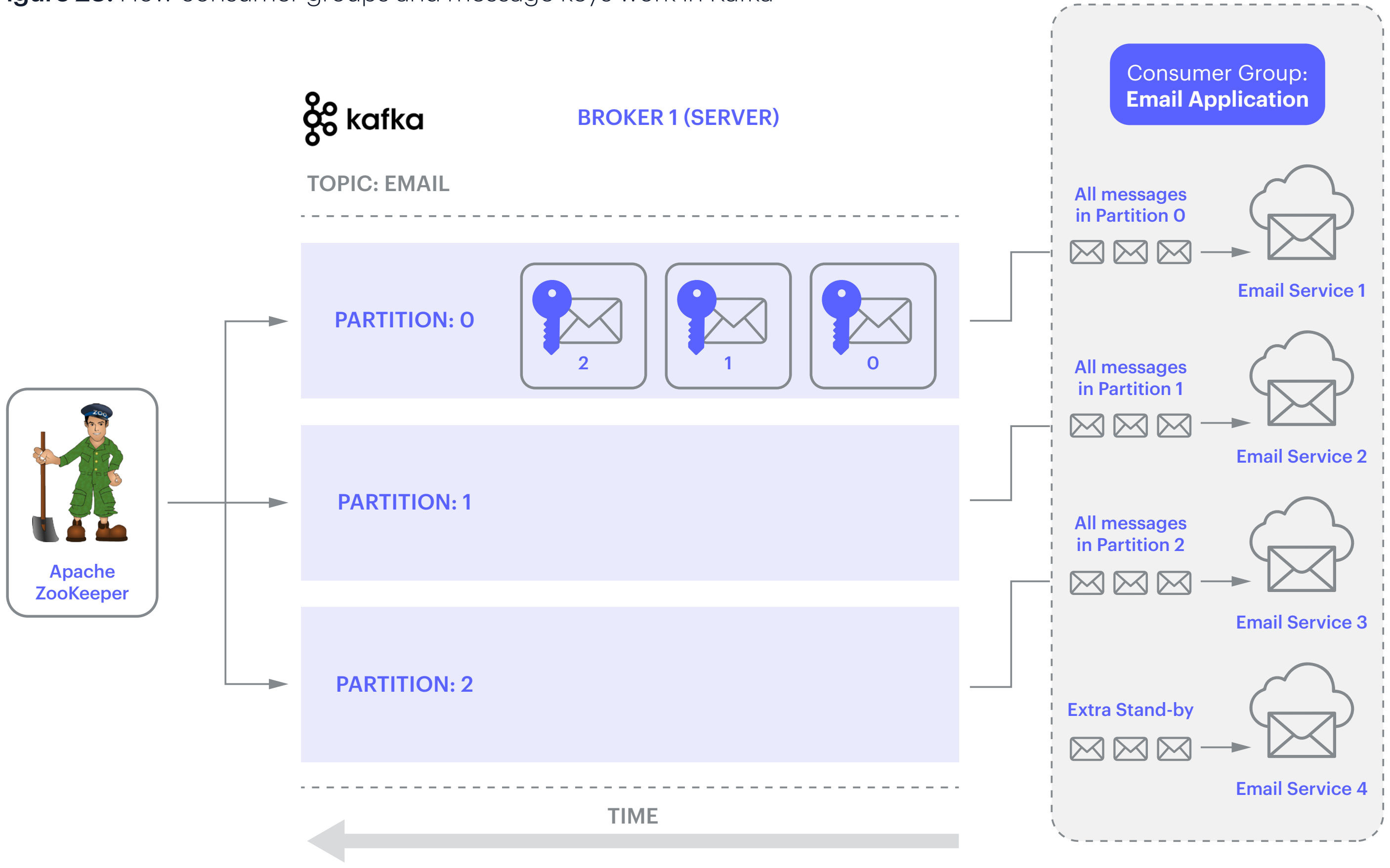
© 2022 Redis

Referring to Figure 23, let's say you are processing emails for an e-commerce store. You need to send the following emails and in the following order:

1. Payment Received
2. Product Shipped
3. Product Delivered

In this case, to make sure they are sent in that order, we can use the order id ("order1234") as the key we send to Kafka to ensure that all the messages end up in the same partition.

Figure 23: How consumer groups and message keys work in Kafka



© 2022 Redis

And secondly, we can define a consumer group "Email Application" and designate four consumers as part of that group. Kafka will then internally connect each consumer to one partition (1:1 mapping).

- If there are more consumers than partitions, the additional consumers will be kept idle and won't receive any messages. So, in our example, the fourth consumer will be left idle and won't receive any messages.
- If there are fewer consumers than partitions, then some of the consumers will receive data from multiple partitions. However, there will still be only one consumer (that's part of the group) per partition.

Let's see how this actually looks in CLI.

1. Create a producer to send messages to "Email" topic with a key. Use the "=" sign to separate the "key" (e.g., orderid1234) and the "value" (the actual message content).

```
$  kafka-console-producer.sh --broker-list localhost:9092 --topic Email 1
--property
"parse.key=true" --property "key.separator=="
```

2. Then, send three messages with different statuses "payment_received", "product_shipped" and "product_delivered".

```
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"payment_received"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"product_shipped"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"product_delivered"}
```

3. Create four consumers within the consumer group "EmailApp" and connect them to the "Email" topic (in four different CLI windows).

```
kafka-console-consumer --bootstrap-server 127.0.0.1:9092
--group EmailApp --topic Email
kafka-console-consumer --bootstrap-server 127.0.0.1:9092
--group EmailApp --topic Email
kafka-console-consumer --bootstrap-server 127.0.0.1:9092
--group EmailApp --topic Email
kafka-console-consumer --bootstrap-server 127.0.0.1:9092
--group EmailApp --topic Email
```

4. Here's how just one of the consumers will receive all three messages.

```
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"payment_received"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"product_shipped"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"product_delivered"}
```

© 2022 Redis

How Redis Streams handle it

Although, like Kafka, Redis Streams has a concept of "consumer groups," it operates differently. In fact, you don't need it for this specific use case. We'll learn in the next section how Redis uses consumer groups, but for now let's see how in-order and in-parallel message processing works in Redis.

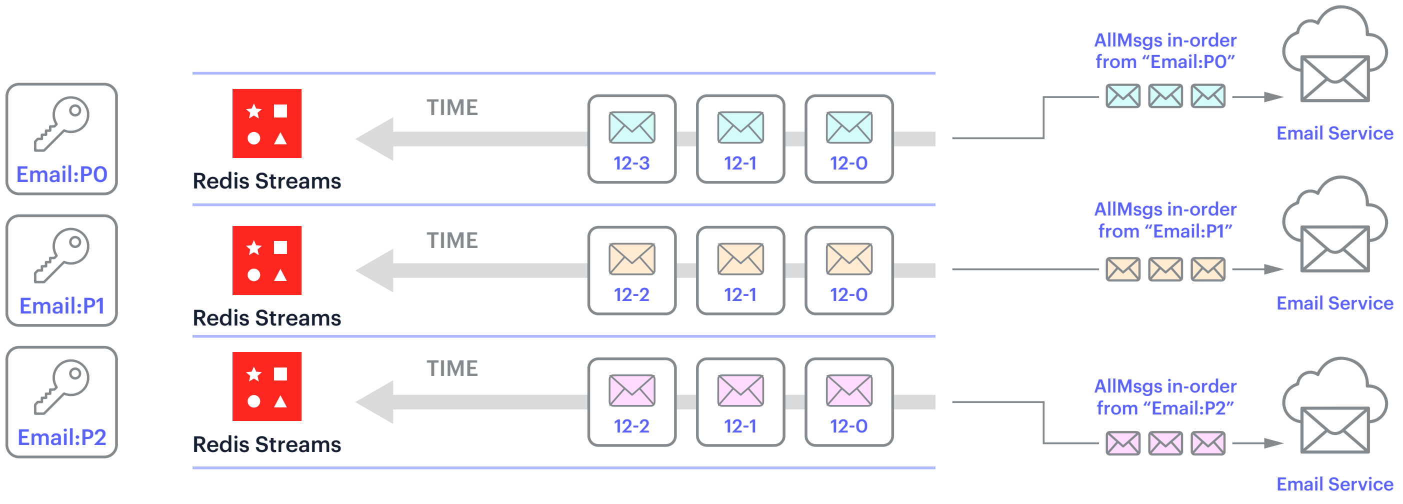
Creating streams in Redis is cheap. You simply define multiple streams (essentially simulating "partitions") based on some simple hash algorithm and then send the messages to those different streams. Once you do, you will be able to use different consumers to consume those messages in order and in parallel.

Let's look at an example.

Let's say someone purchases three different products. For each product, you have "payment_received", "product_shipped", and "product_delivered" messages (for a total of nine), and you want to process them in order but also in parallel.

In the example (Figure 24) below, yellow, purple, and pink represent three products. Each product has three messages representing its different states. As you can see, if you want to process three messages at a time, simply create three streams and send each product's data into a specific stream based on the product id or some unique identifier. This is similar to "keys" in Kafka. After that, connect each consumer to each stream (i.e., 1:1 mapping). This is also similar to Kafka. Then you'll get both parallel processing and in-order message processing at the same time. As we already mentioned, unlike with Kafka, with Redis you don't really need consumer groups in this case.

Figure 24: Using Redis Streams to process multiple messages in parallel and in order



© 2022 Redis

Let’s see how it looks in the CLI.

1. Use a key to keep track of the number of emails so we can use it for round-robin.

   ```
   INCR email_counter
   ```

2. Hash the unique id like the order id and take a modulus of the number of streams to determine which stream (Email:P0 or Email:P1 or Email:P2) to send the data to.

   ```
   var streamName = “Email:P” + ((murmurHash(“order1234”) % 3)
   ```

   All we need to do is to convert a string like orderId into a number using a popular hash called “murmur” hash and then take mod of the number of streams.

   Note: Kafka also uses “murmur” hash for converting string into a number. There are “murmur” libraries in every language, such as this one in Nodejs. A “murmur” hash, while not strictly necessary, is fast and sufficient given you do not require cryptographic secutiry.

3. Send the messages to the appropriate stream. Notice that because of the hash we employed in the above steps, we’ll have a 1:1 mapping between the order id and the stream name. So, for example, all the messages with order id order1234 will go to “Email:P0” stream.

```
XADD Email:P0 * id order1234 name “light bulb” price “$1:00” status “payment_received”
XADD Email:P0 * id order1234 name “light bulb” price “$1:00” status “order_shipped”
XADD Email:P0 * id order1234 name “light bulb” price “$1:00” status “order_delivered”

XADD Email:P1 * id order2222  name “chair” price “$100:00” status “payment_received”
XADD Email:P1 * id order2222 name “chair” price “$100:00” status “order_shipped”
XADD Email:P1 * id order2222 name “chair” price “$100:00” status “order_delivered”

XADD Email:P2 * id order5555  name “Yoga book” price “$31:00” status “payment_received”
XADD Email:P2 * id order5555 name “Yoga book” price “$31:00” status “order_shipped”
XADD Email:P2 * id order5555 name “Yoga book” price “$31:00” status “order_delivered”
```

© 2022 Redis

4. Here's how just one of the consumers will receive all three messages.

```
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"payment_received"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"order_shipped"}
> orderId1234={id:"orderid1234", name: "light bulb", price:" $1.00", status:
"order_delivered"}
```

© 2022 Redis

The role of consumer groups in Redis Streams

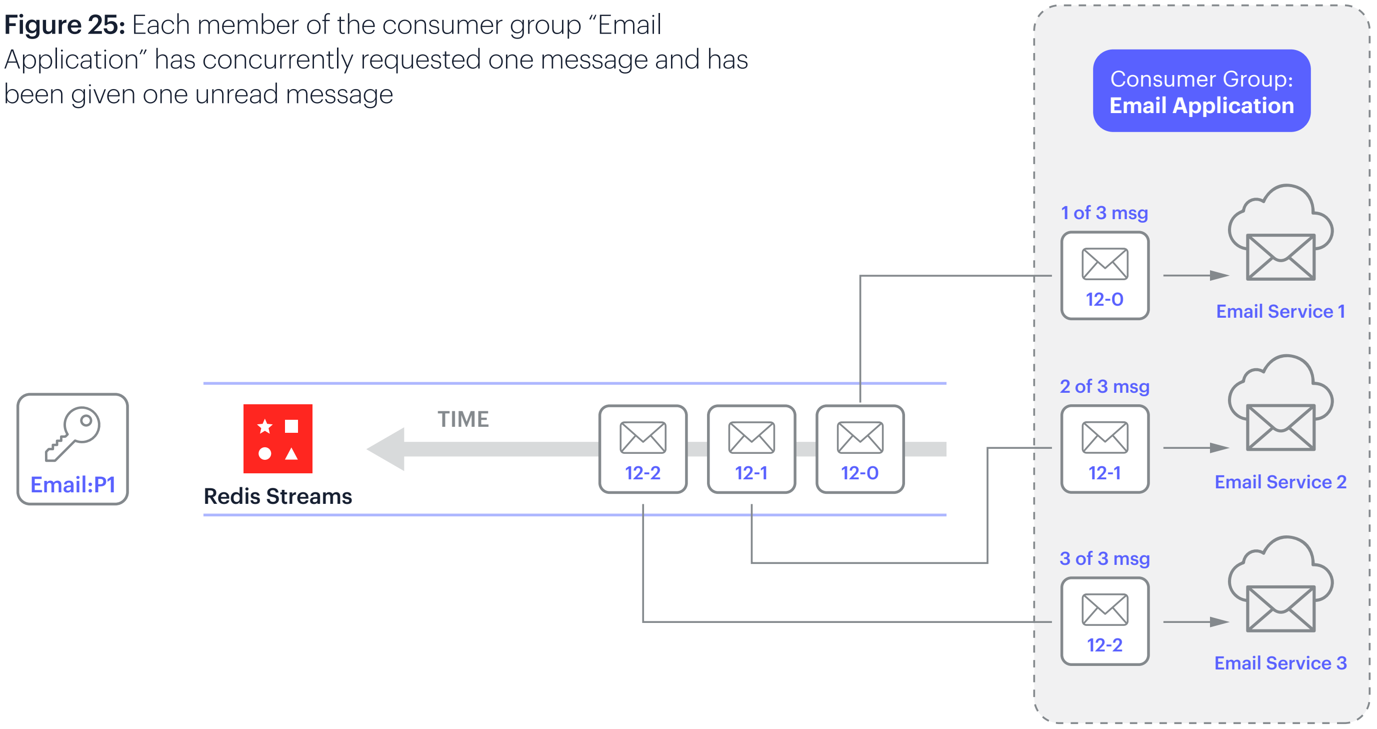
In Redis Streams, although there is no concept of "message keys" like in Kafka, you can still get message ordering without it. However, it does have a concept of "consumer groups" but again it works differently from Kafka.

First, let's understand how consumer groups work in Redis Streams and then later we'll see how Redis Streams handles message ordering.

In Redis Streams you can connect multiple consumers that are part of the same consumer group to a single stream and do parallel processing without the need for partitions.

In the example below (Figure 25), we have created a consumer group called "Email Application" and have made three consumers part of that group. Each consumer is asking for one message each at the same time for concurrent processing. In this case, Redis Streams simply distributes unread (unconsumed) messages to each consumer.

Figure 25: Each member of the consumer group "Email Application" has concurrently requested one message and has been given one unread message



Note: Each consumer within the consumer group needs to identify itself with a name. In Figure 25, we have named the services that are part of the "Email Application" group as "emailService1", "emailService2", and "emailService3".

© 2022 Redis

Let's see how it looks in the CLI.

Notes:

- *The ">" means, send me new messages; that is, ones that have not been read by anyone (including me).*
- *Caution: If you use "0" or a timestamp instead of ">", then Redis Streams will only return messages that have already been read (but not acknowledged) by the current consumer.*
- *Note that It doesn't return all the messages from the mainstream. This is because, when a consumer is part of a group, an additional list called a "pending list" is created and treated as a micro-stream that holds messages for that particular consumer. This is a little tricky to understand, we'll discuss this in a bit*
- *"COUNT 1" means give me just one message.*

1. Create a stream (Email), a consumer group (Email Application - "EmailApplnGroup",) and set it to read all messages from the beginning ("0"). Note: If you use "$" instead of "0", then it will send only new messages. Also, if you provide any other id, then it will start reading from that id. Note: MKSTREAM is used to make a new stream if the stream doesn't already exist.

```
XGROUP CREATE Email EmailApplnGroup 0 MKSTREAM
```

2. Add three messages to the stream.

```
XADD Email * subject "1st email" body "Hello world"
XADD Email * subject "2nd email" body "Hello world"
XADD Email * subject "3rd email" body "Hello world"
```

3. Let's consume messages using the three consumers, each asking concurrently for one email.

```
XREADGROUP GROUP EmailApplnGroup emailService1 COUNT 1 STREAMS Email >
XREADGROUP GROUP EmailApplnGroup emailService2 COUNT 1 STREAMS Email >
XREADGROUP GROUP EmailApplnGroup emailService3 COUNT 1 STREAMS Email >
```

4. In this case, each consumer will receive one message.

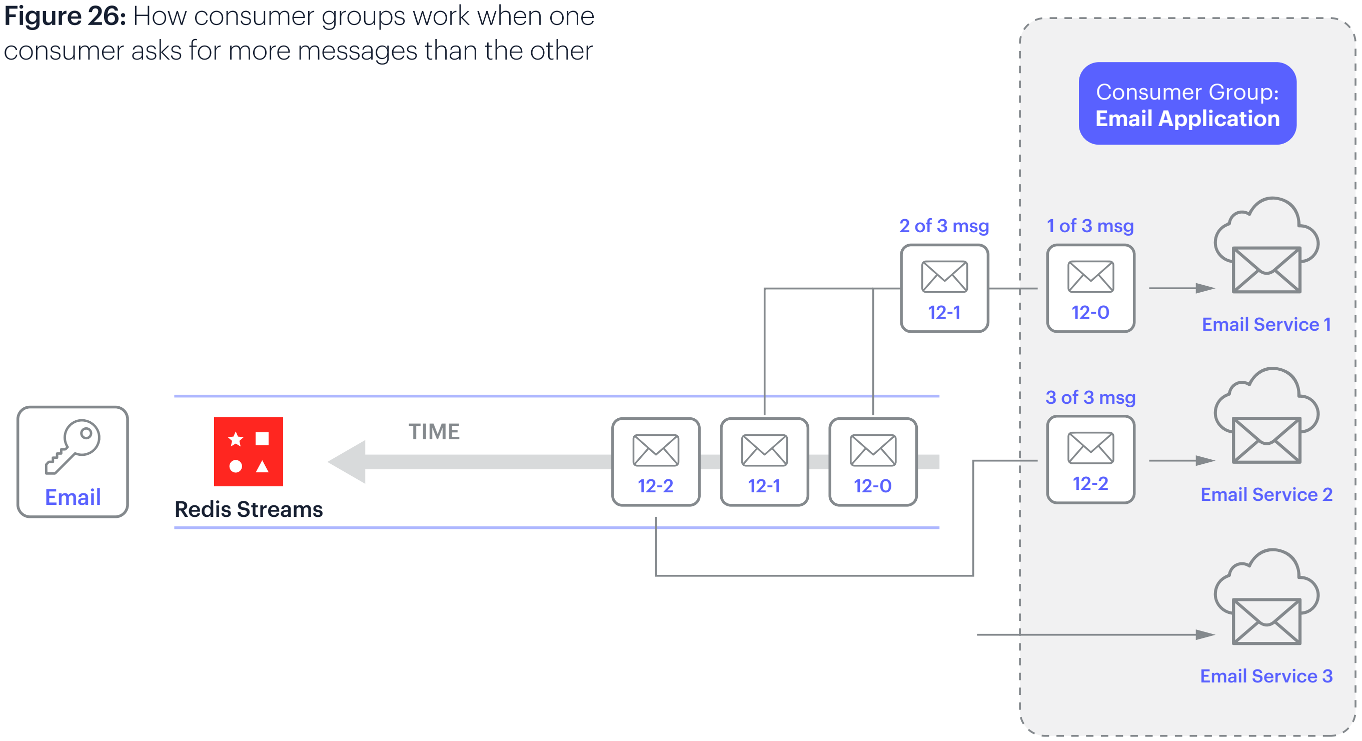
```
1) 1) "Email"
   2) 1) 1) 1526569495631-0
         2) 1) "subject"
            2) "1st email"
            3) "body"
            4) "Hello world"
```

© 2022 Redis

As mentioned earlier, unlike Kafka, each consumer within the group can ask for as many messages as it wants.

In Figure 26, we have "emailService1" asking for two messages instead of one, while at the same time "emailService2" is asking for one. Finally, a little bit later, "emailService3" asks for one message. In this case, emailService1 gets to process two messages, emailService2 gets to process one, but emailService3 doesn't wind up with any because there are no more unclaimed messages available.

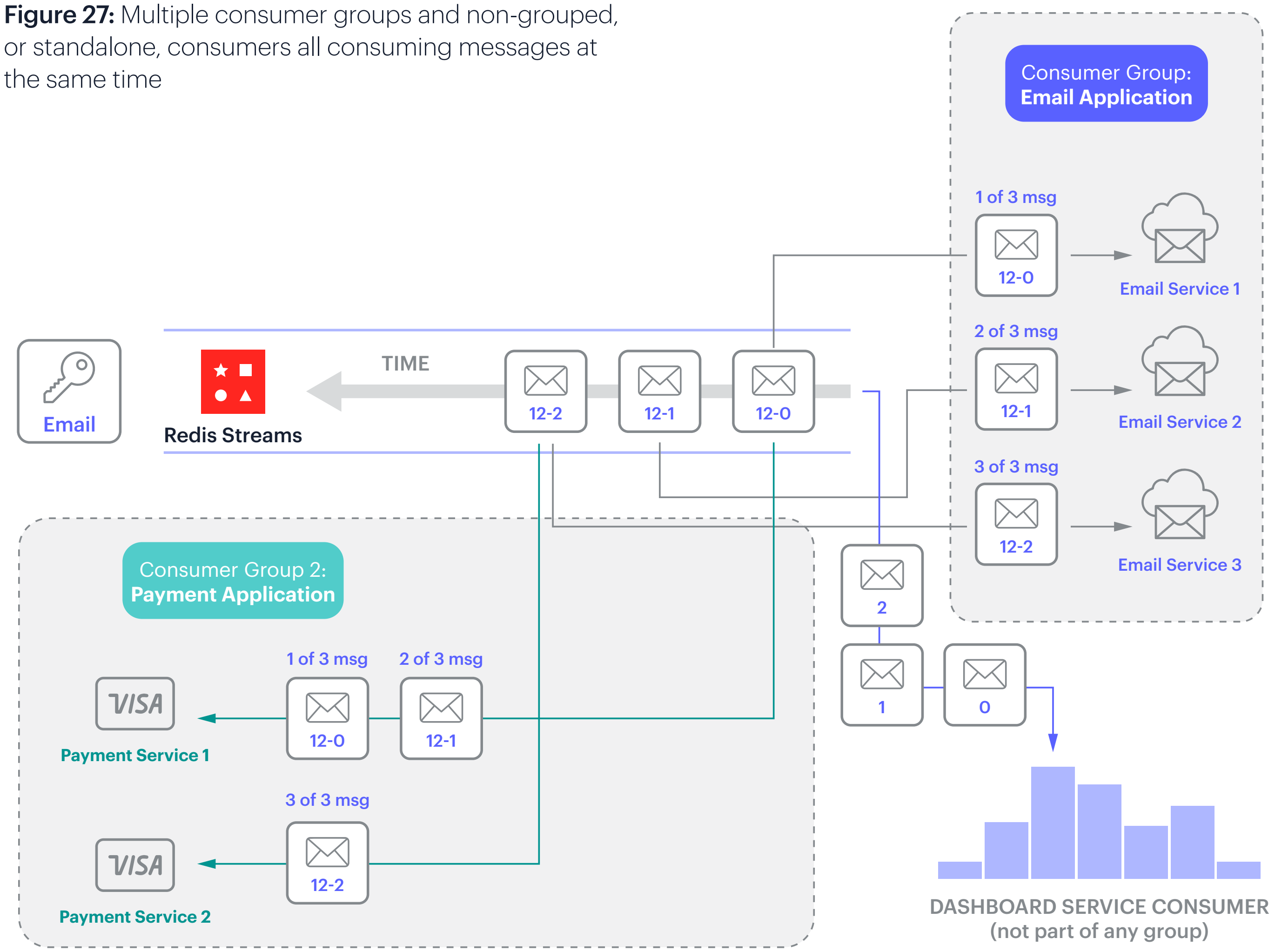
Figure 26: How consumer groups work when one consumer asks for more messages than the other



© 2022 Redis

This scenario doesn't have to be limited to a single consumer group. It's possible to have multiple consumer groups as well as regular consumers that are not part of any group, all consuming messages at the same time. In Figure 27, there are two different consumer groups ("Email Application" and "Payment Application") as well as a regular consumer (Dashboard Service) that are all consuming the messages.

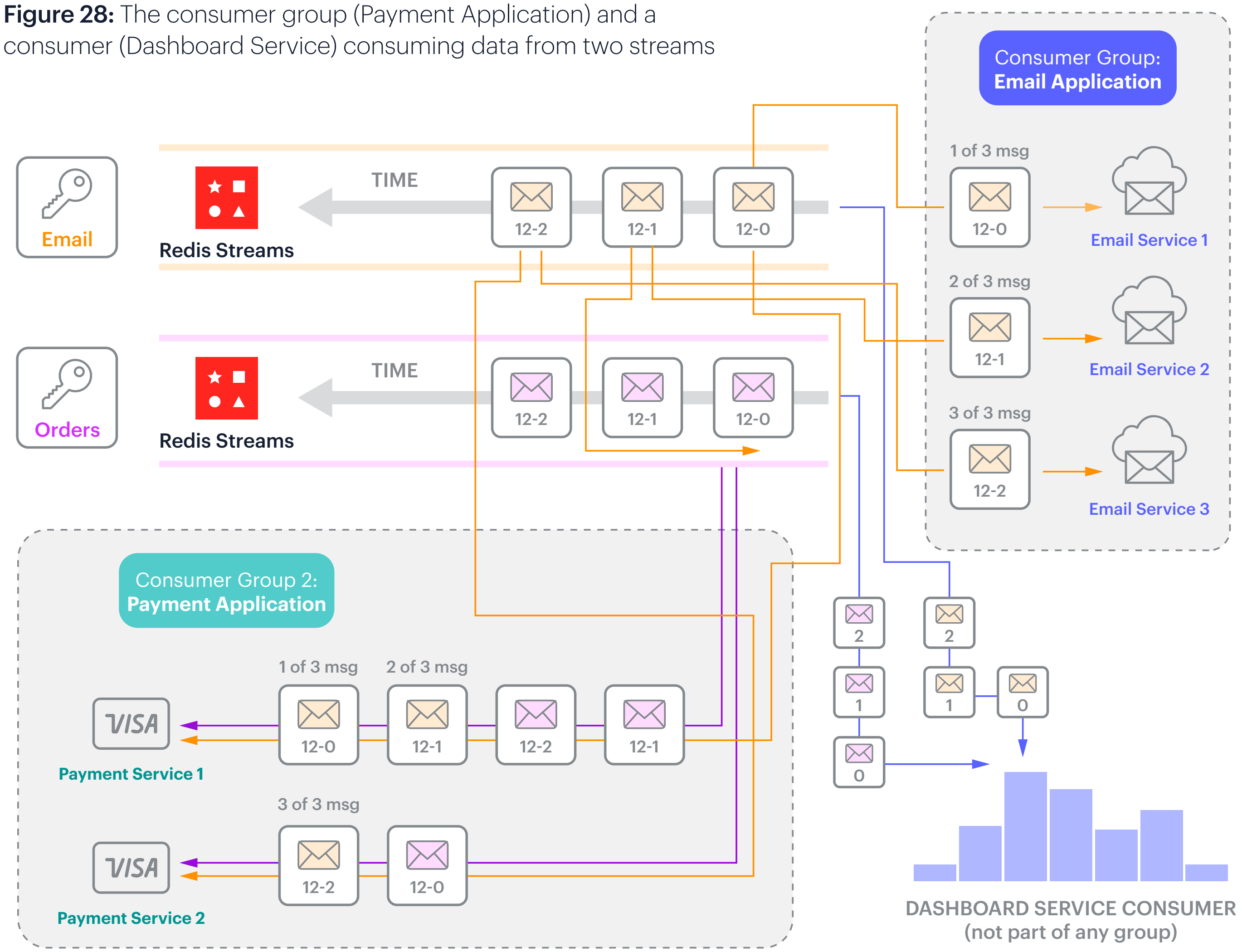
Figure 27: Multiple consumer groups and non-grouped, or standalone, consumers all consuming messages at the same time



© 2022 Redis

And finally, a consumer or a consumer group can consume data from multiple streams (see Figure 28).

Figure 28: The consumer group (Payment Application) and a consumer (Dashboard Service) consuming data from two streams



© 2022 Redis

To consume from multiple streams, you simply need to list them. This is how it looks like in the CLI.

1. Making a consumer group's (Payment Application Consumer Group 2 - "paymentApplnGroup") consumer (paymentService1) get two (Count 2) unread messages from Email stream (Email >) and also from the Orders stream (Orders >)

   ```
   XREADGROUP GROUP paymentApplnGroup paymentService1 COUNT 2
   STREAMS Email > Orders >
   ```

2. Making the dashboard service get messages from the beginning from both Email (Email 0) and the Order (Orders 0) streams and also waiting for any new streams in a blocking fashion (BLOCK 0).

   ```
   XREAD BLOCK 0 STREAMS Email 0 Orders 0
   ```

Now that you have seen the basics of how stream processing works, let's look at how some of the challenges are addressed. One of the most effective ways to handle them is via "message acknowledgements."

Let's dig in.

© 2022 Redis

How messages are acknowledged

In the context of stream processing, acknowledgement is simply a way for one system to confirm to another system that it has received a message or that it has processed that message.

Message acknowledgements can be used to solve the following four stream processing challenges:

1. Providing message delivery guarantees for producers
2. Providing message consumption guarantees for consumers
3. Enabling retries after temporary outages
4. Permitting reassignment following a permanent outage

1. **Providing message delivery guarantees for producers.** Once a message has been sent, how can we be sure that it has been received? We need the streaming system to acknowledge that it has in fact safely stored the incoming message.

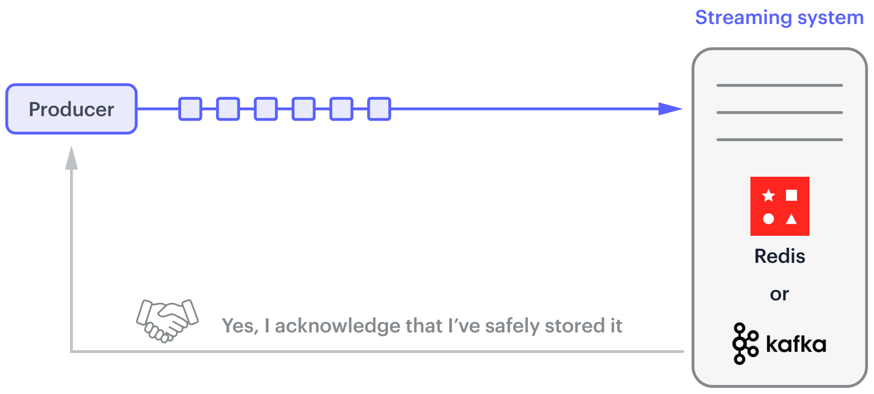


Figure 29: How stream processing systems acknowledge message reception to producers

2. **Providing message consumption guarantees for consumers.** There needs to be a way for the consumer to acknowledge back to the system that it has successfully processed the message.

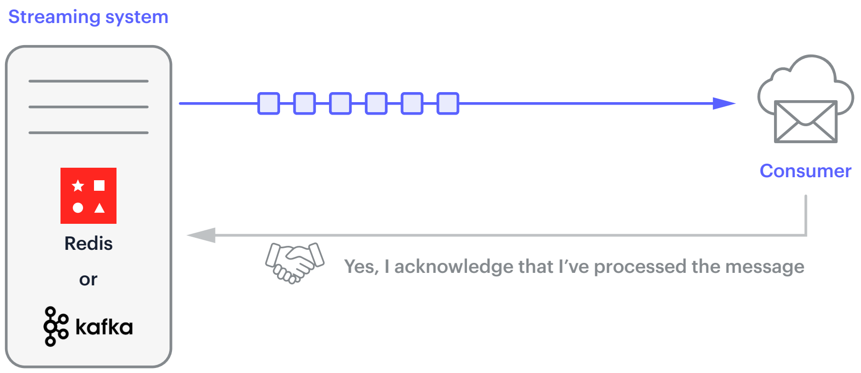


Figure 30: A consumer acknowledgement to the streaming system after processing the message

© 2022 Redis

3. **Enabling retries after temporary outages.** We need to be able to reprocess messages in the event that a consumer dies while processing them. For this we need to have a mechanism that enables the consumer to acknowledge to the system that it has processed a message. And if there is an issue, the processing system needs to provide a way to re-process that message in case of a temporary failure (Figure 31).

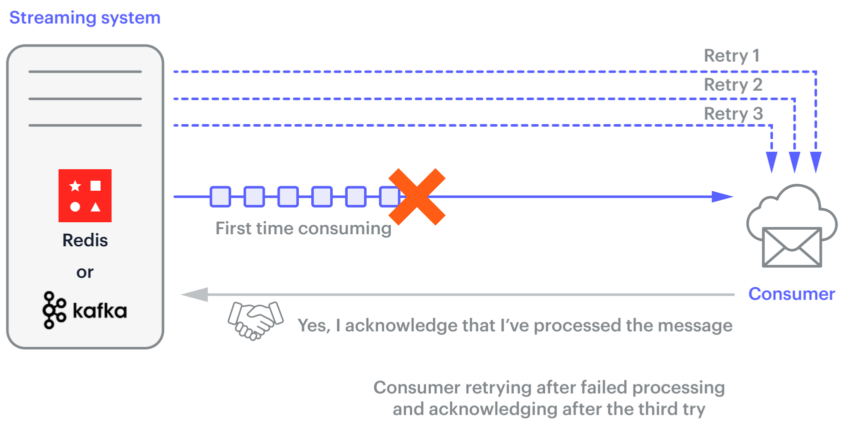

Streaming system
Retry 1
Retry 2
Retry 3
Redis
or
kafka
First time consuming
Consumer
Yes, I acknowledge that I've processed the message
Consumer retrying after failed processing and acknowledging after the third try

Figure 31: If the consumer fails to process the message on the first try, a mechanism is needed that enables it to retry processing the message until it has been successfully processed and that enables it to acknowledge when the message has finally been processed.

4. **Permitting reassignment following a permanent outage.** And lastly, if the consumer permanently fails (say, crashes), we need a way to either assign the job to a different consumer or allow different consumers to find out about the failure and take over the job (Figure 32).

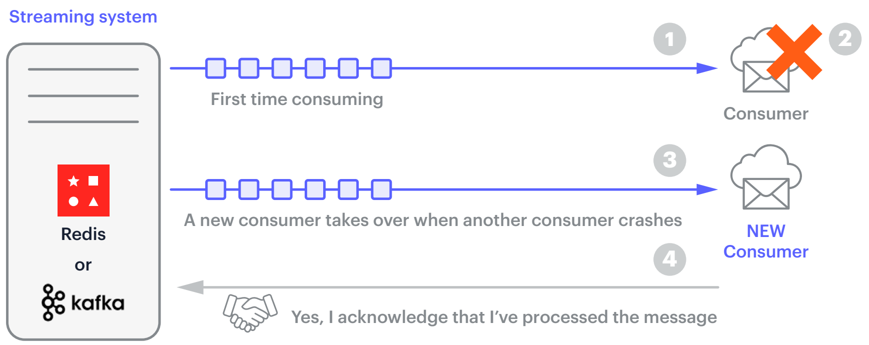

Streaming system
1
2
First time consuming
Consumer
3
A new consumer takes over when another consumer crashes
NEW Consumer
4
Redis
or
kafka
Yes, I acknowledge that I've processed the message

Figure 32: When a consumer, while attempting to read new messages (1) permanently crashes (2), a new consumer takes over (3), successfully processes the messages, and then sends back an acknowledgment to the streaming system (4).

Now let's look at how Kafka and Redis Streams handle each one of these.

© 2022 Redis

Letting the producer know that the message has been delivered

With Kafka

In Kafka, you can have the following configurations

Ack = 0: Send the message but don't wait for any confirmation (you may lose data, but it will be extremely fast).
Ack = 1: At least one of the nodes in the cluster must acknowledge receipt.
Ack = All: All the leader and replicas must acknowledge that they have received the messages. This can be slow but will ensure that the message has been stored successfully in both the leader and followers.

With Redis

In Redis Streams (especially in Redis Enterprise), you have two ways to acknowledge that a message has been delivered. You can configure Redis clusters to have a weak consistency (but more throughput) or a strong consistency (with a little less throughput).

Configuring weak consistency and durability

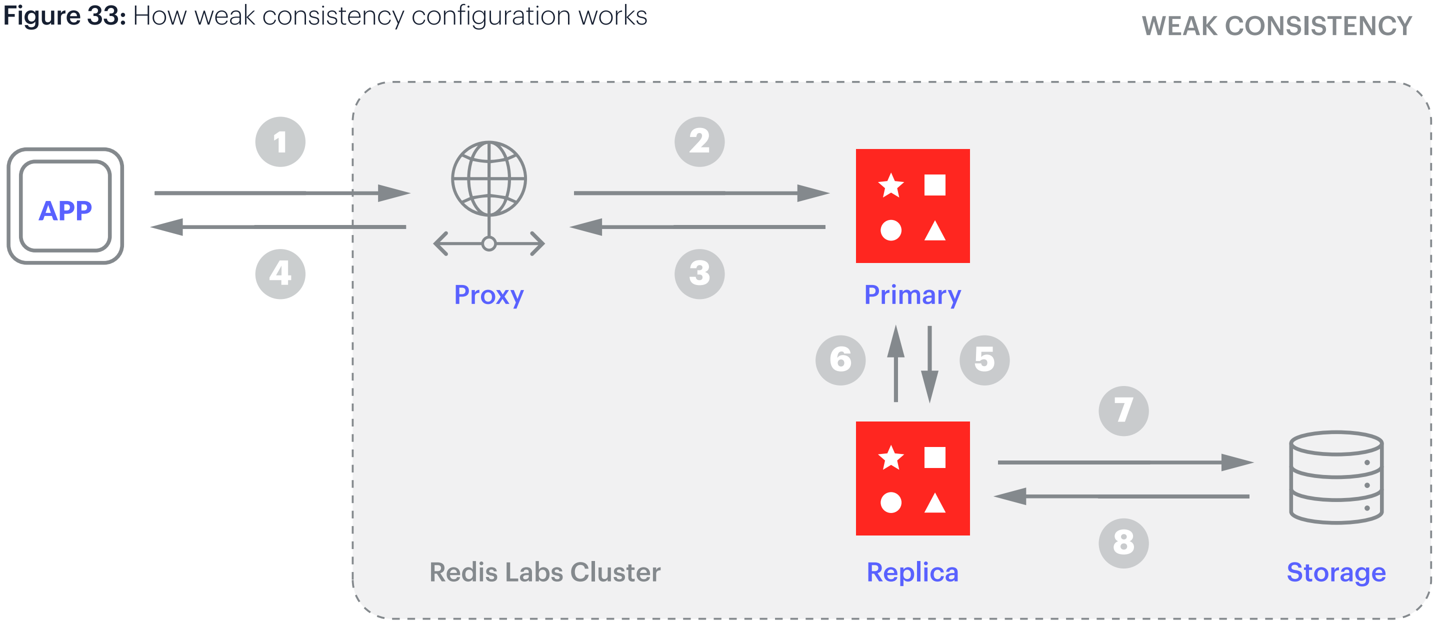
Let's see how the weak consistency and durability configuration works. And once configured, it'll work the same for all types of Redis keys including Redis Streams. This is somewhat equivalent to "aAck=1" in Kafka.

Any updates that are issued to the database are typically performed with the following flow:
1. The application issues a write to the proxy.
2. The proxy communicates with the correct primary "shard" in the system that contains the given key.
3. Once the write operation is complete, an acknowledgement is sent back to the proxy.
4. The proxy sends the acknowledgment back to the application.

Independently, the write is communicated from primary to replica and replication acknowledges the write back to the primary. These are steps 5 and 6.

Independently, the write to a replica is also persisted to disk and acknowledged within the replica. These are steps 7 and 8.

Figure 33: How weak consistency configuration works



© 2022 Redis

To read more about consistency and durability, see https://docs.redislabs.com/latest/rs/concepts/data-access/consistency-durability/

Configuring strong consistency and durability

Here's how a strong consistency and durability works in Redis Streams. This is equivalent to "ack=All" in Kafka.

Option 1

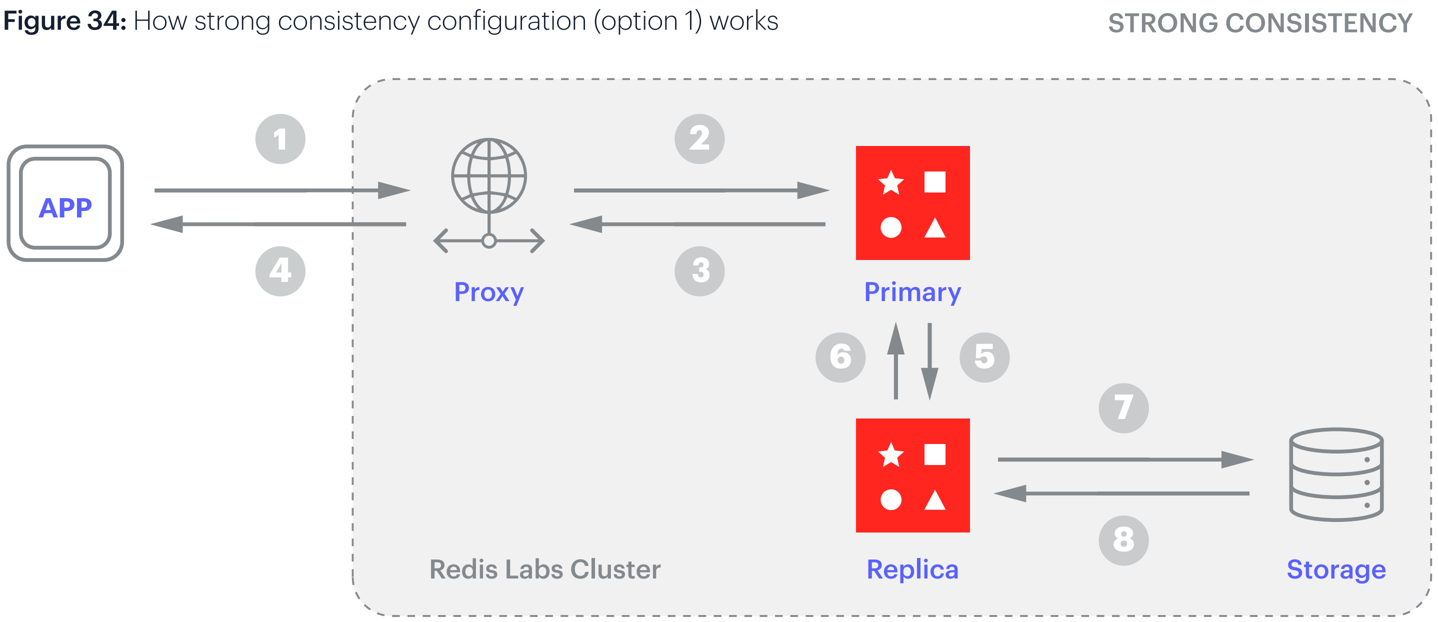
With the WAIT command, applications can ask to wait for acknowledgments only after replication or persistence is confirmed on the replica. The flow of a write operation with the WAIT command is shown below:

1. The application issues a write.
2. The proxy communicates with the correct primary "shard" in the system that contains the given key.
3. The acknowledgment is sent to the proxy once the write operation completes.
4. The proxy sends the acknowledgement back to the application.

Independently, the write is communicated from the primary to the replica and replication acknowledges the write back to the primary. These are steps 5 and 6.

Independently, the write to a replica is also persisted to disk and acknowledged within the replica. These are steps 7 and 8.

Figure 34: How strong consistency configuration (option 1) works



© 2022 Redis

With this flow, the application only gets the acknowledgment from the write after durability is achieved with replication to the replica and to the persistent storage.

With the WAIT command, Redis will make a best-effort attempt to guarantee that even under a node failure or node restart, an acknowledged write will be recorded. However there is still a possibility of failure.

See the WAIT command for details on the new durability and consistency options.

Summary: The two approaches compared

Consistency and durability	Kafka	Redis (Redis Streams)
Option 1	Ack = 0 (doesn't wait for acknowledgement)	Ignore or don't wait for acknowledgement
Option 2	Ack = 1 (wait for the leader – but not replicas – to acknowledge)	"Weak consistency configuration" where you get acknowledgement only from the leader
Option 3	Ack = All (wait for all replicas to acknowledge)	"Strong consistency config" (wait for all replicas to acknowledge). Or use "Redis Raft"

© 2022 Redis

Letting the consumer know that the message has been received

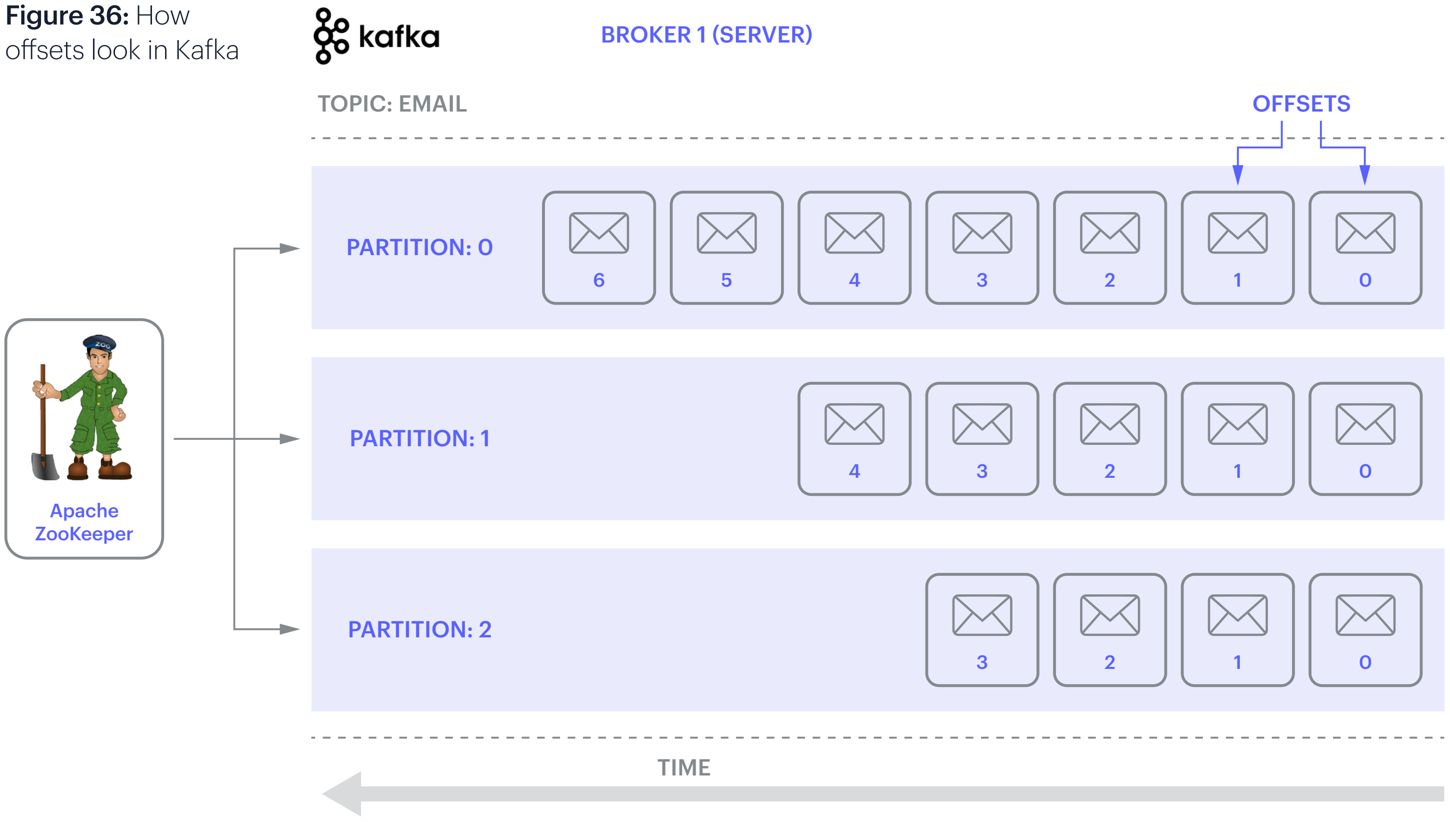
In order to understand consumption guarantees, you'll need to know a little more about some of the inner workings of Kafka and Redis Streams, more specifically, the concepts of "offsets" in Kafka and "pending lists" in Redis streams. These concepts, in conjunction with acknowledgements, will help to solve the challenge of providing consumption guarantees.

So let's take a look at "offsets" in Kafka and then "pending lists" in Redis Streams before we return to consumption guarantees.

The role of offsets in Kafka's consumption acknowledgements

Offsets are simply incremental ids that are given to each message within a given partition for a given Topic. They start from 0 for each partition for a given Topic. So there could be multiple messages with the same offset id. Therefore, the only way to uniquely identify them in the entire system is by combining the offset id with the partition id and the topic name (because there could be multiple topics with the same partition ids).

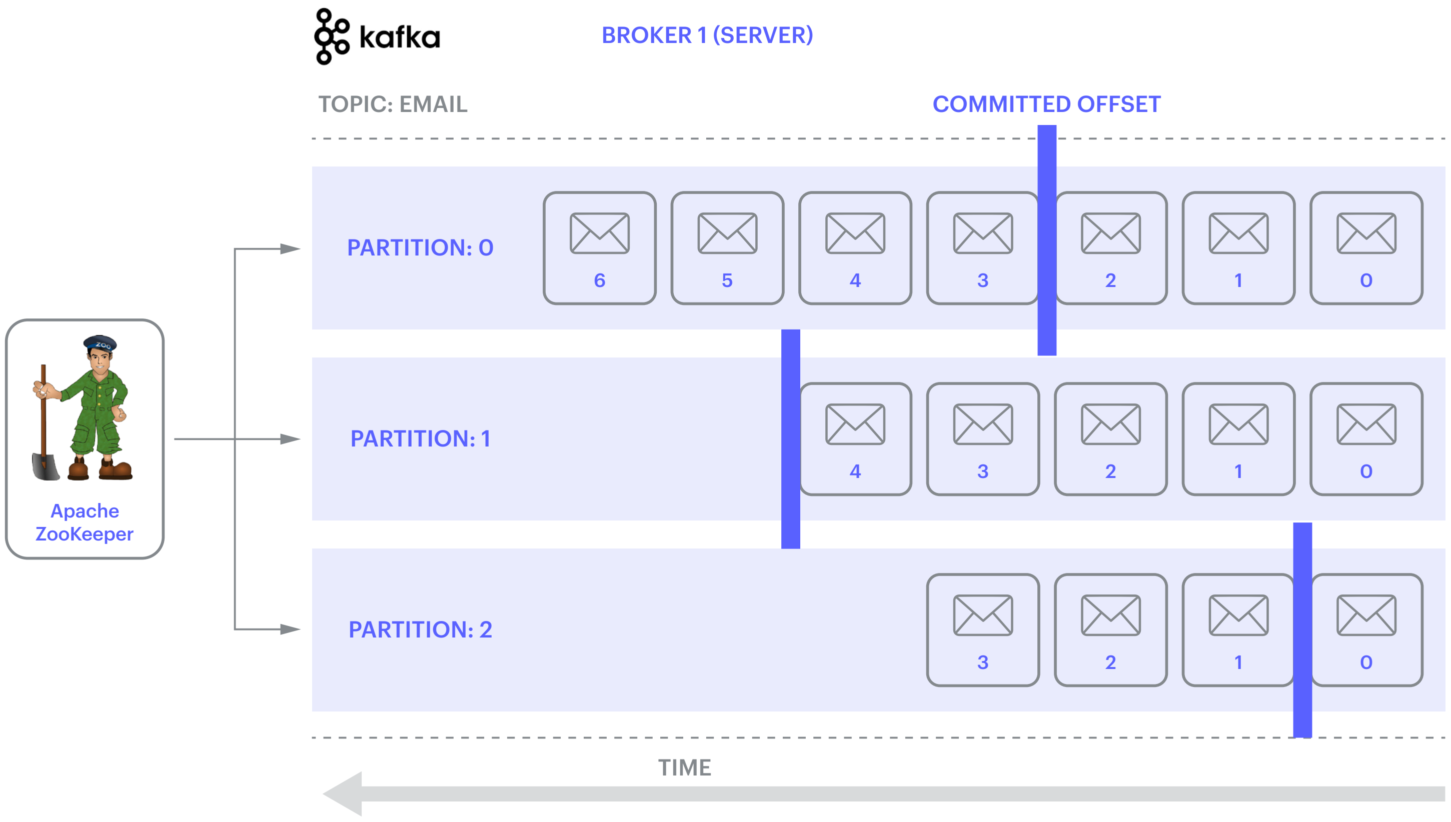
Figure 36: How offsets look in Kafka



© 2022 Redis

Committing offsets (i.e., consumer acknowledgement). When a consumer processes a message or a bunch of messages, it acknowledges this by telling Kafka the offset it has just consumed. In Kafka, this can be automatic or manual. Following the consumer's acknowledgement, this information is written to an internal topic called "__consumer_offsets", which acts like a tracking mechanism. This is how Kafka knows what message to send next to consumers.

Figure 37: Consumers have processed up to offset 2 in partition 0, up to offset 4 in partition 1, and up to offset 0 in partition 3.



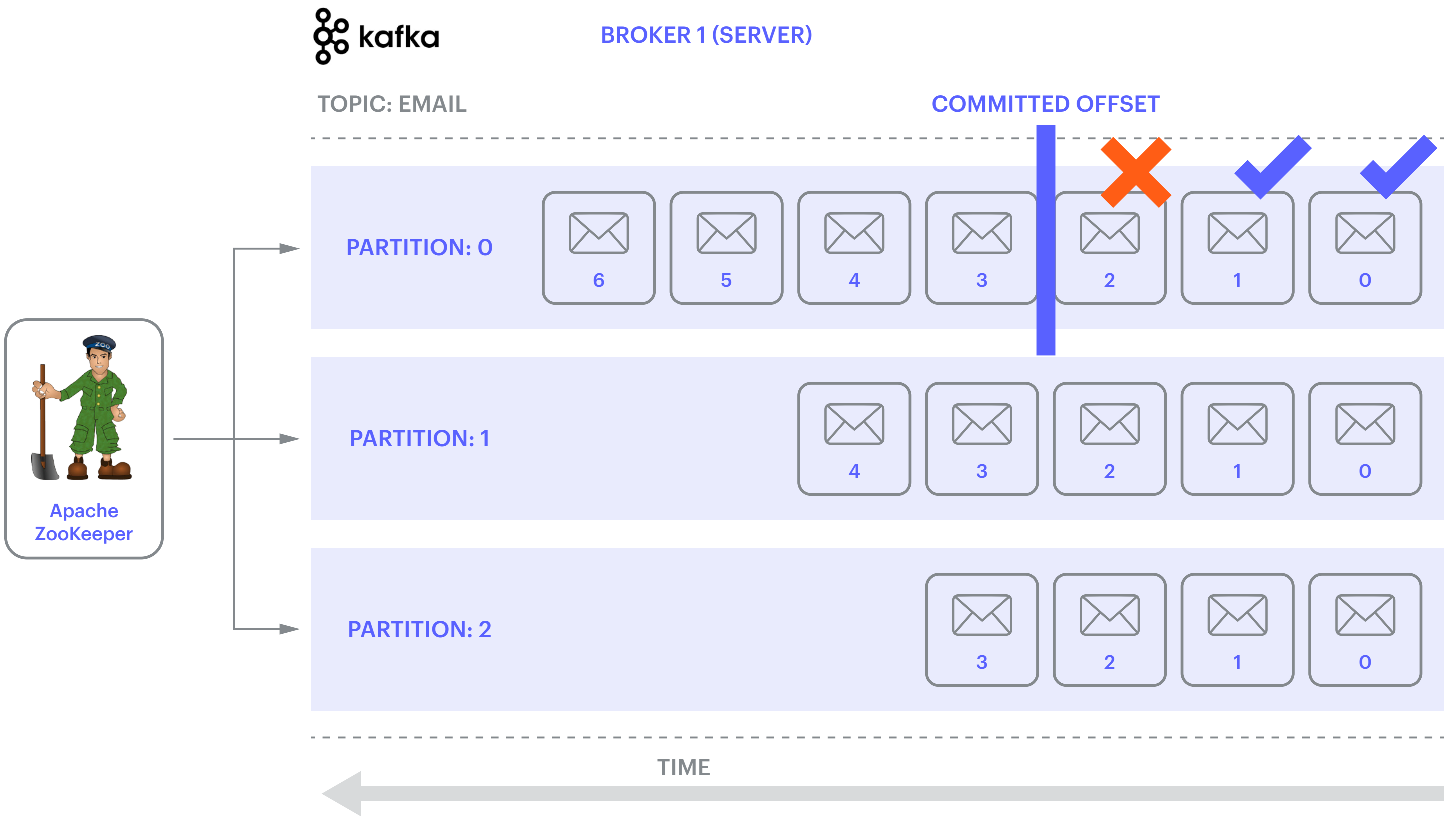
© 2022 Redis

This leads to three delivery methods, each with its own advantages:

1. At most once: In this case, the consumer provides acknowledgement as soon as it receives the message, even before it has had a chance to process it. Although this leads to higher throughput, if the consumer dies before it's able to actually process the message, that message will be lost. That's why this method is called "at most once." The consumer has only one chance to request a group of messages. Any messages it is unable to process will be lost.

For example, in Figure 38, a consumer receives three messages and acknowledges the offset for each before processing them. As it turns out, it couldn't actually process the third message (offset-2) successfully. But since the offset has already been committed, the next time it asks for new messages, Kafka will send them from offset-3 onwards. As a result, the message with offset-2 will fail to be processed.

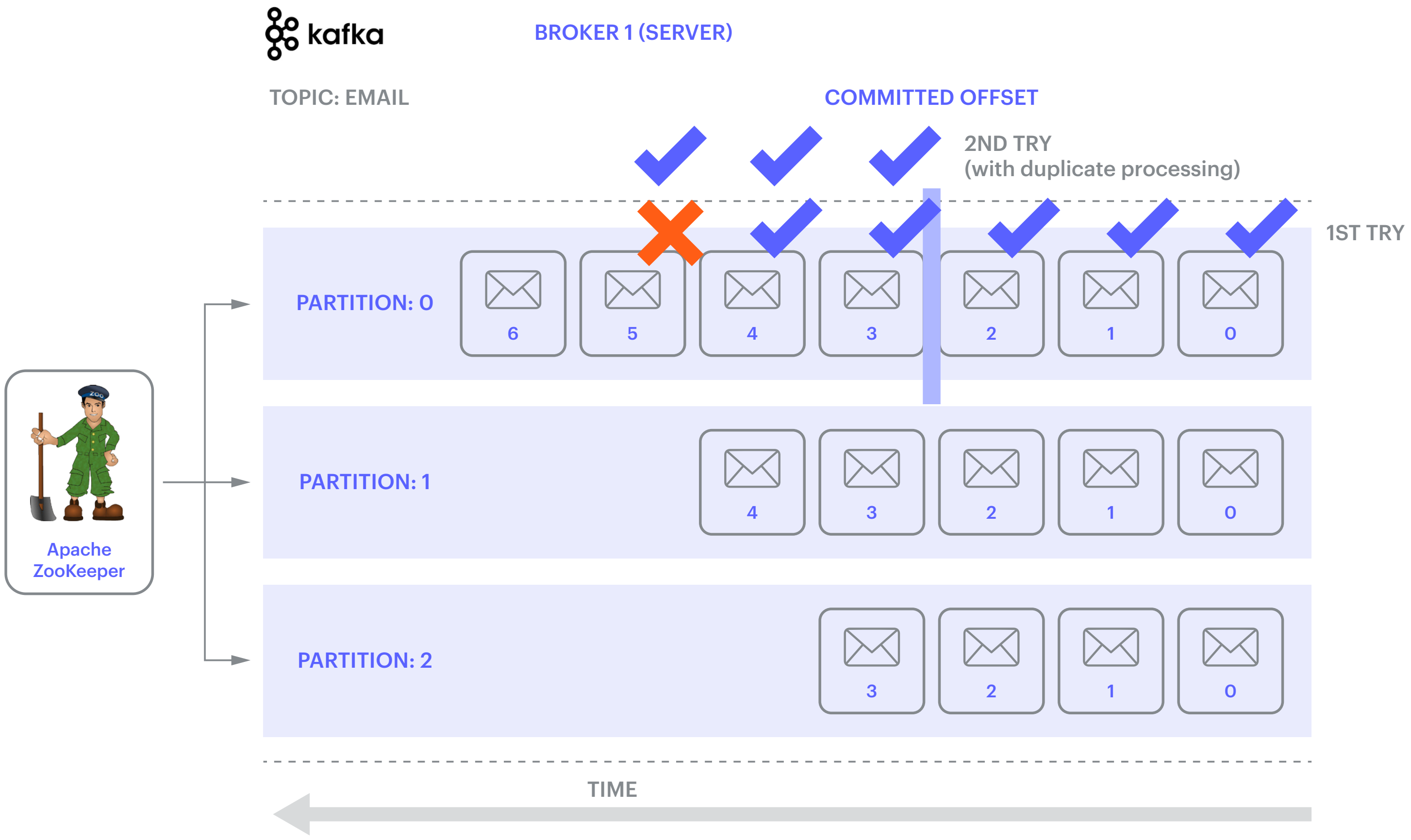
Figure 38: How at-most-once message processing works



© 2022 Redis

2. At least once: In this case, the consumer will commit only after processing. Let's imagine that for performance reasons the consumer is reading three messages at once and committing once after processing all three messages. Let's say it processed two successfully but crashed before it was able to process the third one. In this case, the consumer (or a different consumer) can come back and request these messages from Kafka again. And because the messages were never committed, Kafka will send all three messages again. As a result, the consumer will end up reprocessing messages that were already processed (i.e., duplicate processing). This approach is called "at least once" because the consumer isn't limited to a single request.

Figure 39: How at-least-once processing works



© 2022 Redis

In the illustration above, assume that a consumer is processing three messages at a time and committing an offset after it has processed all three.

Here is how it works:

1. A consumer reads messages with offsets 0, 1, and 2.
2. It processes them.
3. It commits the offset to 2.
4. The next time it asks, it gets messages with offsets 3, 4, and 5.
5. Let's say it processes offsets 3 and 4 but crashes while processing offset-5.
6. A new consumer (or the same consumer) requests messages from Kafka.
7. Kafka will again return messages with offset 3, 4, and 5.
8. Let's say this time all three are successfully processed. That's good, but it leads to duplicate processing of 3 and 4.

The way to mitigate this is to process the messages in a way that it's idempotent. This means even if you process a message multiple times, the end result won't change. For example, if you set the exact price of some product multiple times in a database, it won't matter. When building distributed applications, if you find that you cannot maintain idempotency when processing messages, you likely need to reconsider your logic to find a way to make it idempotent.

3. Exactly once: As the name suggests, it simply means that you figure out a way to ensure that a message is processed once and no more. For this you typically need extra support (programming logic) to ensure and guarantee this because there could be various reasons for duplicate processing. Kafka only provides this level of guarantee out of the box with Kafka-to-Kafka streams.

Now that we've seen how Kafka provides message consumption guarantees, let's take a look at how Redis handles them. But first, in order to do so, we need to delve into the Redis concept of the "pending list."

The role of "pending lists" in Redis' consumption acknowledgements

Remember that Redis Streams does not have a built-in mechanism for partitions. Multiple consumers that are part of the same consumer group can all connect to a single stream and yet still process messages concurrently within that stream.

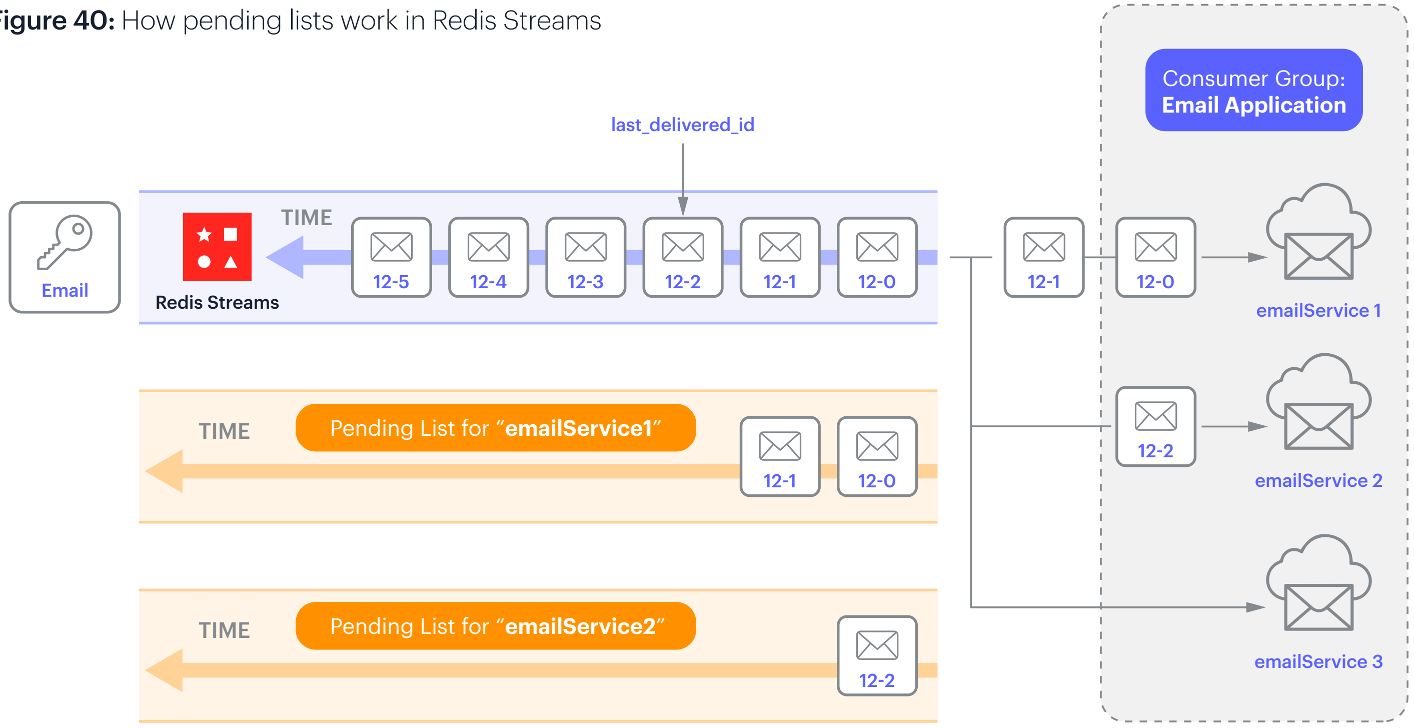
To ensure that these consumers don't process duplicate messages, Redis Streams uses an additional data structure called "pending lists" to keep track of messages that are currently being processed by one of the consumers.

© 2022 Redis

Looking at Figure 40, "emailService1" has asked for two messages and "emailService2" has asked for one. After the messages have been received, Redis Streams puts a copy (or a pointer) of them in a separate list for each consumer. So "12-0" and "12-1" are added to the list for "emailService1" and "12-2" is added to the list for "emailService2". In addition, it updates the "`last_delivered_id`" id to "12-2". This allows for three key things.

1. The "`last_delivered_id`" id ensures that only unread messages are delivered to future requests from consumers of that same group. This is kind of like the "offset commits" in Kafka.
2. The pending lists allow consumers, should they temporarily die during processing (that is, before acknowledgement), to pick up where they left off.
3. The pending lists also allow other consumers to claim pending messages (using XCLAIM) in case the death of the original consumer proves to be permanent.

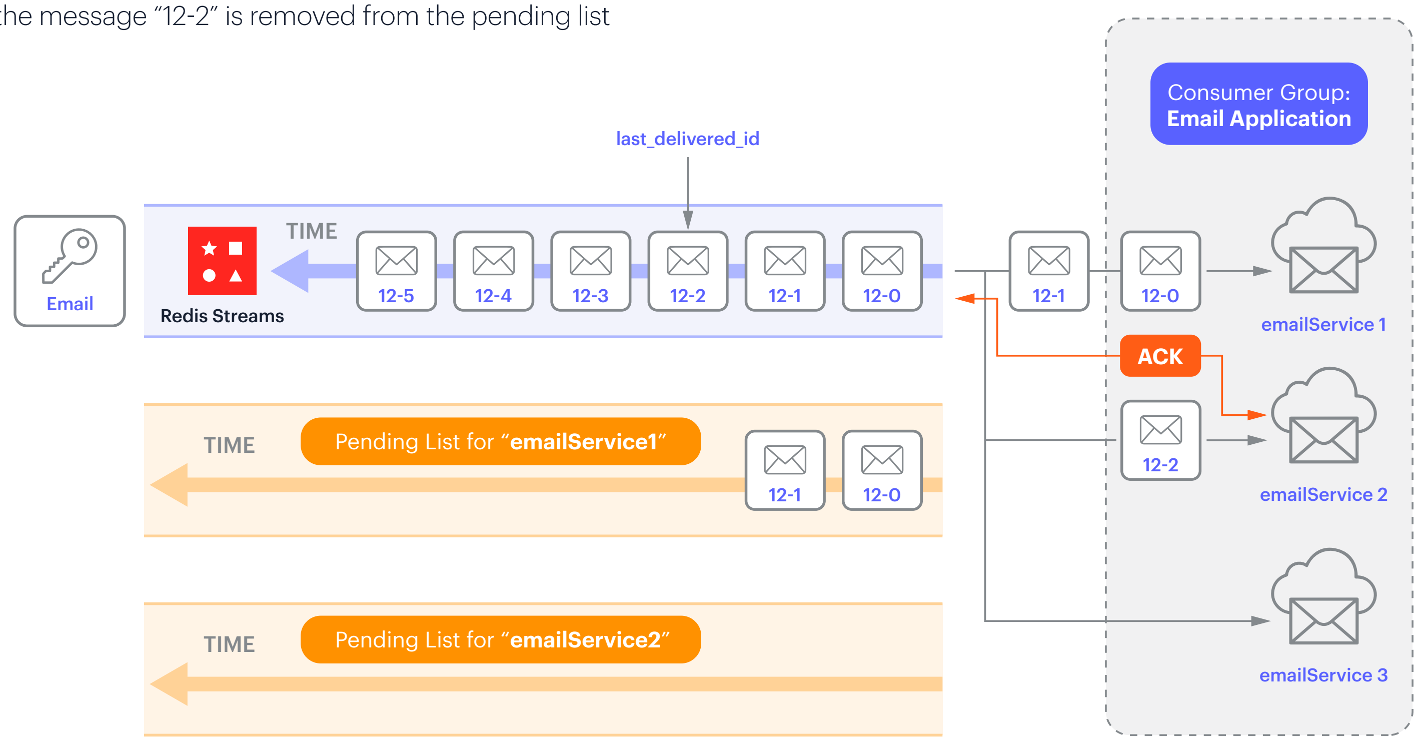
Figure 40: How pending lists work in Redis Streams



© 2022 Redis

Now, let's imagine that "emailService2" has completed its processing (Figure 41) and acknowledges this. Redis Streams responds by removing the processed items from the pending list.

Figure 41: Following acknowledgment from emailService2, the message "12-2" is removed from the pending list



© 2022 Redis

Let’s see how it looks in the CLI.

1. Create a stream (Email), a consumer group, “Email Application” (EmailApplnGroup) and set it to read all messages from the beginning (“0”). Note: If you use “$” instead of “0”, then it will send only new messages. Also, if you provide any other id, then it will start reading from that id. Note: MKSTREAM is used to make a new stream if the stream doesn’t already exist.

   ```
   XGROUP CREATE Email EmailApplnGroup 0 MKSTREAM
   ```

2. Add six messages to the stream.

```
XADD Email * subject “1st email” body “Hello world”
XADD Email * subject “2nd email” body “Hello world”
XADD Email * subject “3rd email” body “Hello world”
XADD Email * subject “4th email” body “Hello world”
XADD Email * subject “5th email” body “Hello world”
XADD Email * subject “6th email” body “Hello world”
```

3. Let’s consume a message from the “emailService2” consumer that’s part of the “EmailApplnGroup” from the “Email” stream.

```
XREADGROUP GROUP EmailApplnGroup emailService2 COUNT 1 STREAMS Email
//This will return a message that’ll look like this
1) 1) “Email”
   2) 1) 1) 1526569495632-1
         2) 1) “subject”
            2) “3rd email”
```

4. Imagine we processed that message and we acknowledged that.

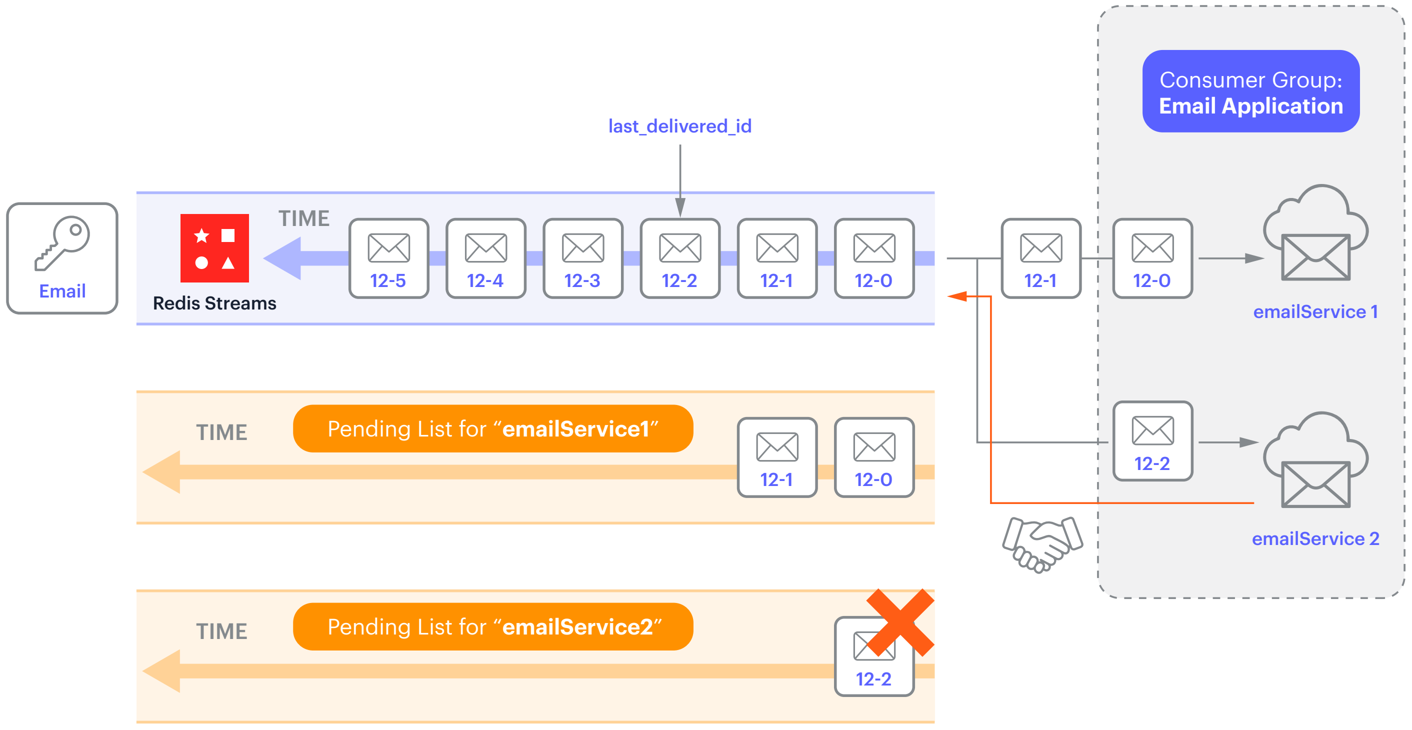
```
XACK Email emailApplGroup 1526569495632-1
```

© 2022 Redis

As you can imagine, with Redis Streams you can easily apply the same delivery approaches used with Kafka. Let's take a look.

1. At most once: In this case, you send an acknowledgement when the messages have been received but before they've been processed. Using Figure 42 for reference, let's imagine that "emailService2" acknowledges before fully processing the message in order to quickly consume more messages, and that losing some message processing doesn't matter. In this case, if the consumer crashes after acknowledgement but before processing the message, then that would be lost. Note that this message is still in the stream, so you can potentially reprocess it, although you'll never know if you'll need to or not.

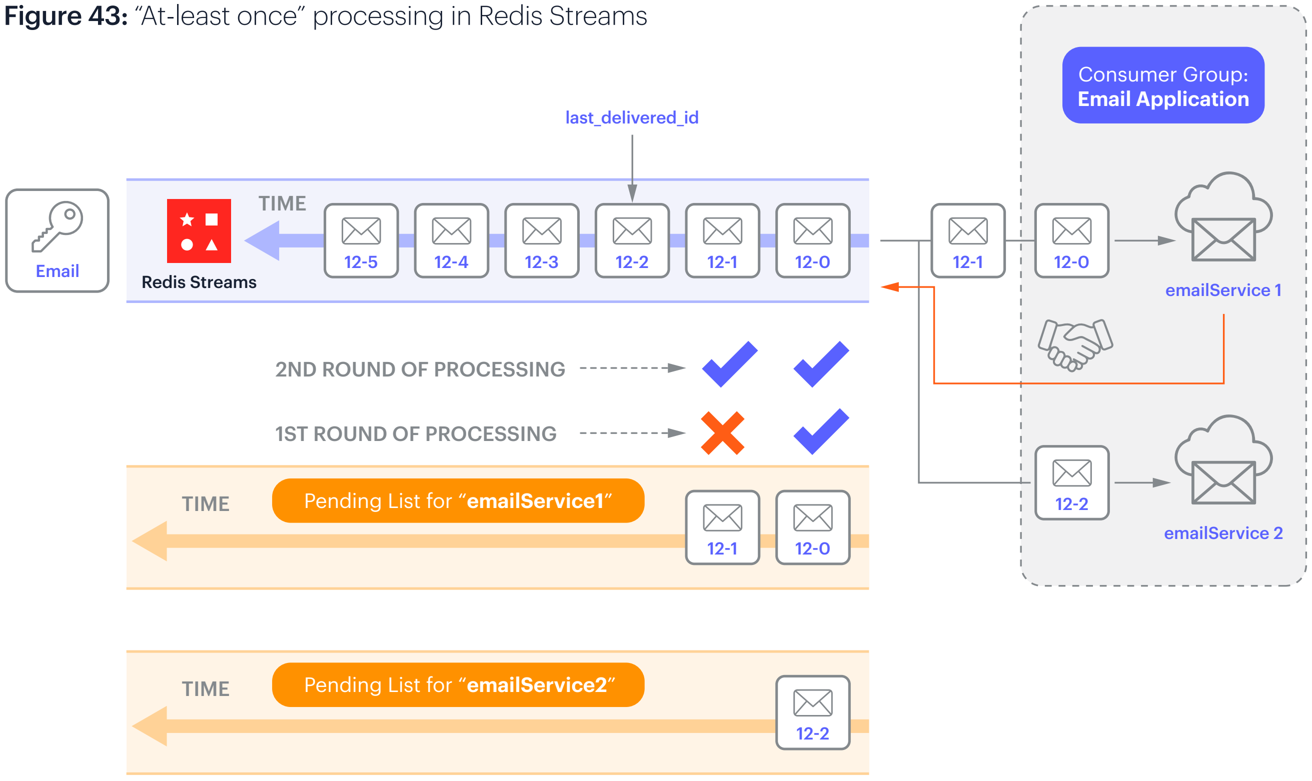
Figure 42: "At most once" processing in Redis Streams



© 2022 Redis

2. At least once: Again, this is very similar to Kafka. A message is only acknowledged after it's been processed. Here, if a consumer is acknowledged after processing multiple messages, and it crashes during the processing of one of those messages, then you'll end up reprocessing all of the messages, not just the ones that failed to be processed.

Figure 43: "At-least once" processing in Redis Streams



In the example above, we have a consumer group called "Email Application" with two consumers ("emailService1" and "emailService2").

1. "emailService1" reads the first two messages, while "emailService2" reads the third message at the same time.
2. The pending list of emailService1 stores the first two messages and similarly the pending list of emailService2 stores the third message.
3. "emailService1" starts to process both messages (and hasn't committed yet). However, let's say it temporarily crashes after processing the first message but before processing the second message.
4. When "emailService1" comes back later and reads from the pending list, it will again see both messages in that list.
5. As a result, it will process both messages.
6. Because of step 5, the consumer ends up processing the first message twice.

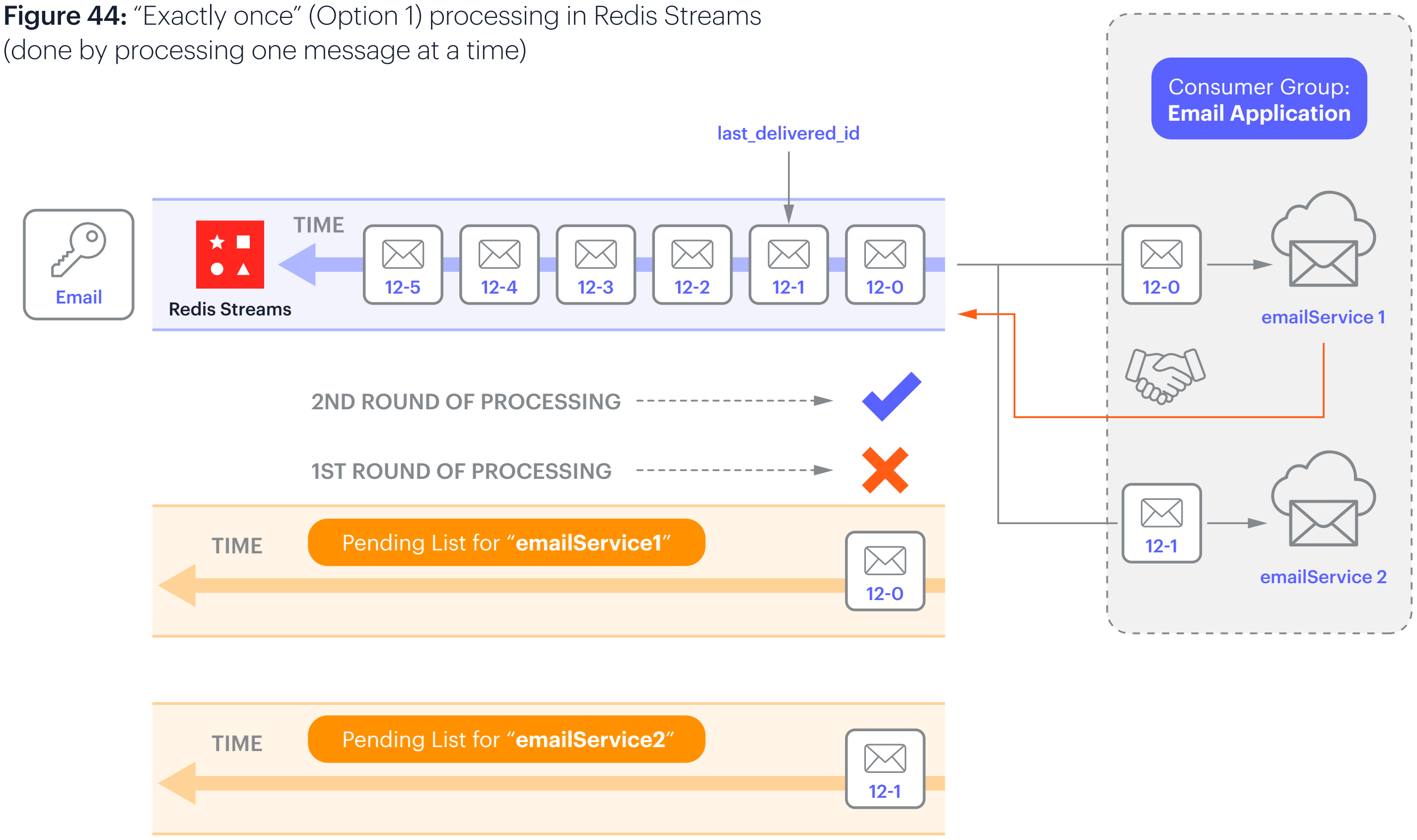
And this is why it's called "at least once." Although ideally all pending messages will be processed in one pass, it may require more.

© 2022 Redis

3. Exactly once: In Redis Streams, you have multiple ways of ensuring that each message is processed exactly one time.

Option 1: Because Redis Streams is extremely fast, you can read just one message at a time and acknowledge it after that message has been successfully processed. In this scenario, you'll always have one message in the pending list. However, even though Redis Streams is fast, consumers can still be slow to process. Consider the performance of your consumers before using this option.

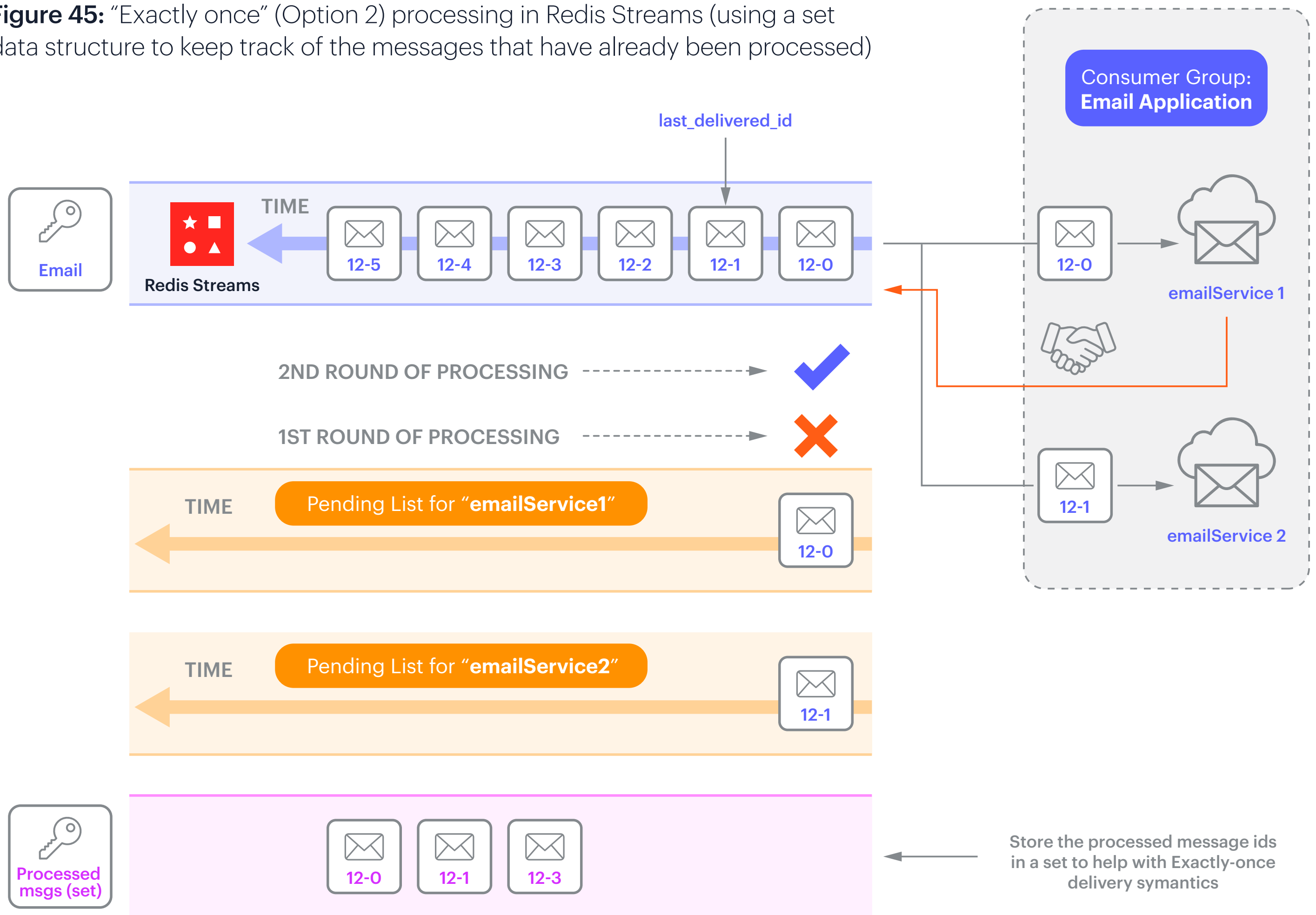
Figure 44: "Exactly once" (Option 1) processing in Redis Streams (done by processing one message at a time)



© 2022 Redis

Option 2: As an alternative to Option 1, you can also use additional data structures, such as Redis Sets, to keep track of messages that have already been processed by some consumer. This way you can check the set and make sure the message's id is not already a member of the set before you request it from the stream.

Figure 45: "Exactly once" (Option 2) processing in Redis Streams (using a set data structure to keep track of the messages that have already been processed)



© 2022 Redis

The role of clusters in Kafka and Redis

In this section we'll go over some high-level aspects of clusters. This is a very deep topic so we'll only cover the key aspects of it.

Kafka Clusters

Kafka is a distributed system. That means you typically wouldn't use it with just one server but would be more likely to use it with at least three servers. Clusters provide high availability and durability. If one of the servers goes down, the others can still keep serving the clients.

In Kafka each server is called a broker. In production, Kafka clusters might have anywhere between three brokers (minimum) to hundreds of brokers.

The example below (Figure 46) shows how a typical Kafka cluster would look. It shows a Kafka cluster with three brokers (servers), two topics (Email and Payment), where the Email topic has three partitions spread across three brokers (10, 11, and 12), and the Payment topic has two partitions that are spread across two brokers (11 and 12).

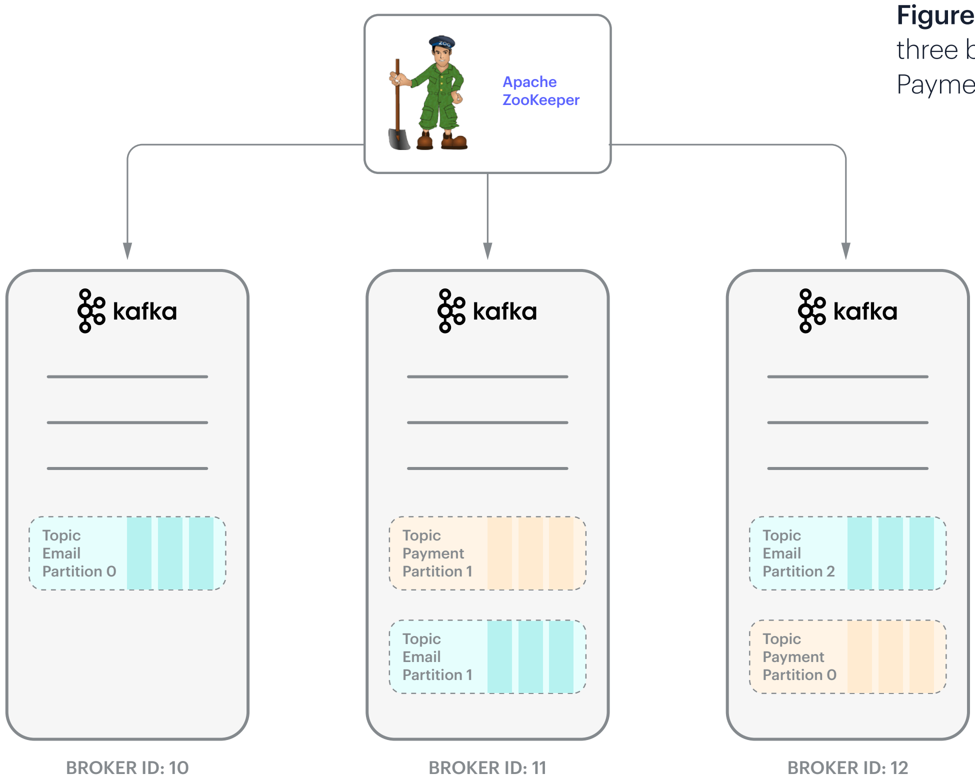


Figure 46: A Kafka cluster consisting of three brokers (servers), two topics (Email and Payment), and five partitions

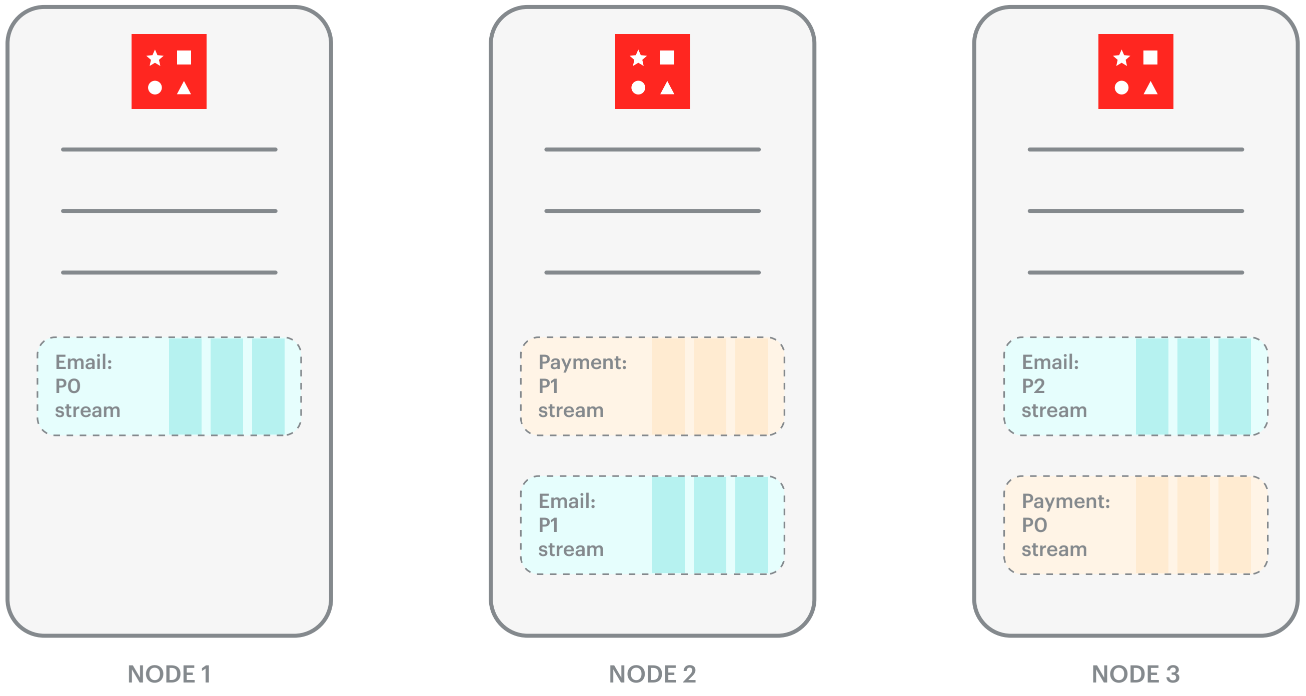
© 2022 Redis

Redis Clusters

With Redis, things work pretty much the same way. In the example below, the Redis cluster has three nodes. The messages for "Email" are sent to three different streams that are in three different nodes. The messages for "Payment" are sent to two different streams that are in two different nodes.

The only caveat if you're using the OSS cluster is that you don't have a proxy for these cluster nodes. That means your client libraries will need to manage where the data goes by directly connecting to each node within the cluster. But thankfully, the cluster APIs make it very easy and most of the Redis client libraries in all programming languages already support it.

Figure 47: A Redis OSS cluster with three brokers (servers), two topics (Email and Payment), and five streams

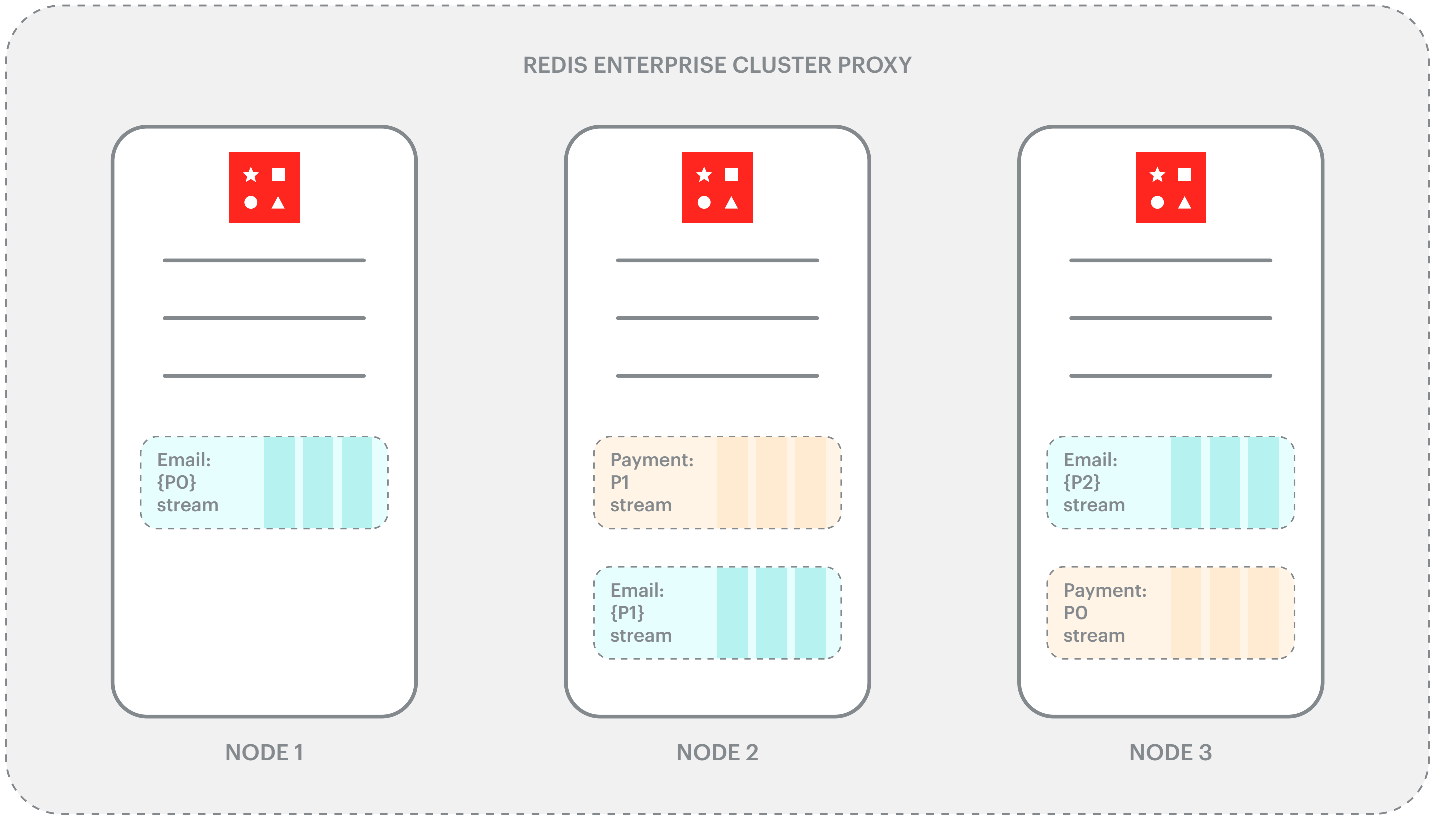


© 2022 Redis

On the other hand, Redis Enterprise provides a proxy layer on top of the clusters. This way the client can just connect to the proxy and doesn't have to worry about exactly which server the data is going to or coming from.

By the way, in Redis clusters, if the key contains curly braces ("{}"), then only the text within those curly braces will be hashed. This means you can name the keys as "Email:{P0}" and "Payment:{P1}".

Figure 47a: A Redis Enterprise cluster with three brokers (servers), two topics (Email and Payment), and five streams



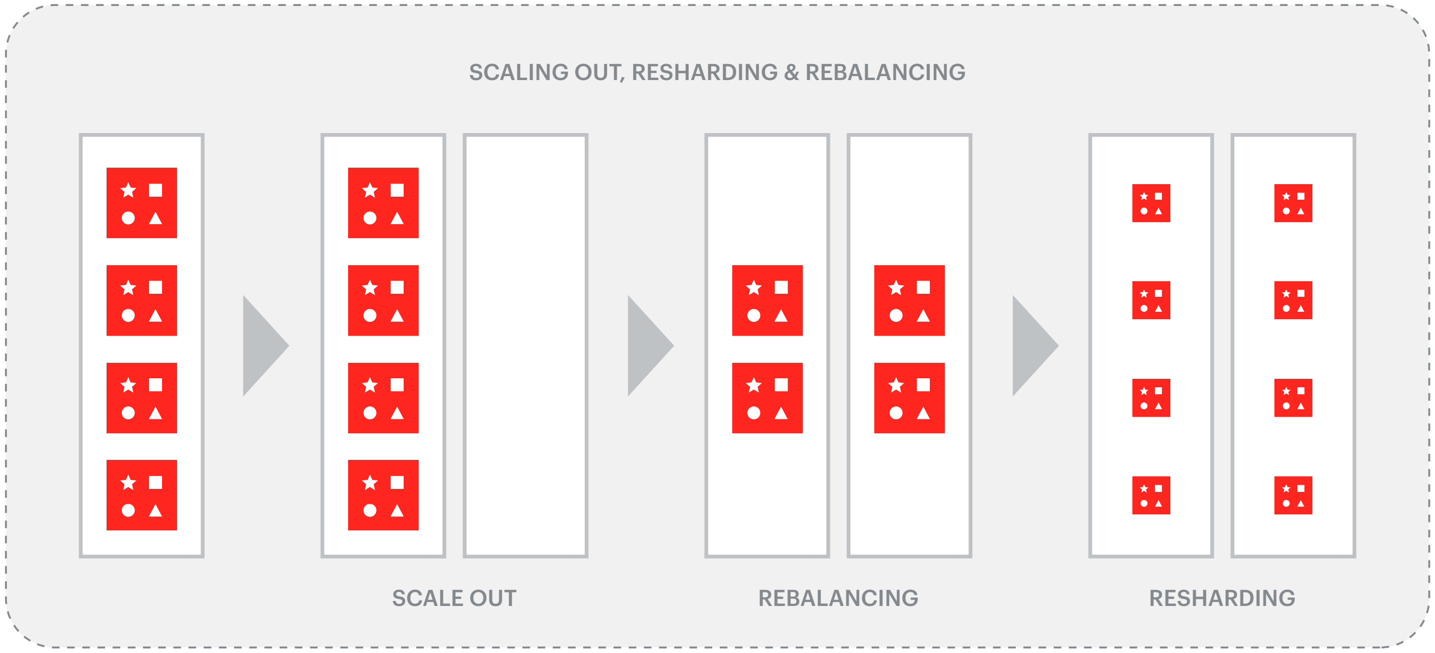
© 2022 Redis

By the way, in Redis you can run multiple Redis instances in the same node. These are called shards.

Figure 48 illustrates how the Redis cluster helps scale Redis. Here is how it works.

1. Let's say you are running four Redis instances (four shards) on a single node. And imagine you have split the data across these four instances. This is mainly for parallel processing and to fully utilize all the CPU and memory resources.
2. Now, let's say you want to move to two machines, that is, you want to "scale out". So you add a second machine. At this point you have scaled out, and this node is now part of the cluster. But this new node is empty to begin with. It contains no Redis instances or data.
3. Next, let's say you move two shards to the second node in order to better distribute the load. This is called "rebalancing".
4. Finally, in order to increase parallel processing and to fully utilize all the CPUs, you may add more Redis instances and split the data across those instances. This is called "resharding". So let's say you've added two more instances/shards in each node. Now in the beginning these new instances won't have any data. So you need to use a process called resharding to split and move some of the existing data. In the end you wind up with a total of eight shards and much higher throughput.

Figure 48: Using Redis Enterprise to increase your throughput by scaling out, rebalancing, and resharding your data



© 2022 Redis

Conclusion

Hopefully this ebook has provided you with a solid foundation for understanding both Kafka and Redis Streams. It's important to note that Kafka is a very robust streaming platform that is geared towards highly complex, distributed applications when you have very specific requirements. In contrast, Redis Streams is a great way to add streaming to an existing application that is already using Redis. Redis Streams has much lower management overhead, and if you are already using Redis for say, caching, then you can implement Redis Streams without setting up and maintaining a separate system.

If you're interested in learning more and taking this further, check out the free Redis Streams course (RU202) offered at Redis University.

© 2022 Redis