Introduction to Redis Streams

The Stream is a new data type introduced with Redis 5.0, which models a *log data structure* in a more abstract way. However the essence of a log is still intact: like a log file, often implemented as a file open in append only mode, Redis Streams are primarily an append only data structure. At least conceptually, because being an abstract data type represented in memory, Redis Streams implement powerful operations to overcome the limitations of a log file.

What makes Redis streams the most complex type of Redis, despite the data structure itself being quite simple, is the fact that it implements additional, non mandatory features: a set of blocking operations allowing consumers to wait for new data added to a stream by producers, and in addition to that a concept called **Consumer Groups**.

Consumer groups were initially introduced by the popular messaging system Kafka (TM). Redis reimplements a similar idea in completely different terms, but the goal is the same: to allow a group of clients to cooperate consuming a different portion of the same stream of messages.

Streams basics

For the goal of understanding what Redis Streams are and how to use them, we will ignore all the advanced features, and instead focus on the data structure itself, in terms of commands used to manipulate and access it. This is, basically, the part which is common to most of the other Redis data types, like Lists, Sets, Sorted Sets and so forth. However, note that lists also have an optional more complex blocking API, exported by commands like **BLPOP** and similar. So streams are not much different than lists in this regard, it's just that the additional API is more complex and more powerful.

Because Streams are an append only data structure, the fundamental write command, called **XADD**, appends a new entry into the specified stream. A stream entry is not just a string, but is instead composed of one or multiple field-value pairs. This way, each entry of a stream is already structured, like an append only file written in CSV format where multiple separated fields are present in each line.

```
> XADD mystream * sensor-id 1234 temperature 19.8
1518951480106-0
```

The above call to the **XADD** command adds an entry `sensor-id: 1234, temperature: 19.8` to the stream at key `mystream`, using an auto-generated entry ID, which is the one returned by the command, specifically `1518951480106-0`. It gets as its first argument the key name `mystream`, the second argument is the entry ID that identifies every entry inside a stream. However, in this case, we passed `*` because we want the server to generate a new ID for us. Every new ID will be monotonically increasing, so in more simple terms, every new entry added will have a higher ID compared to all the past entries. Auto-generation of IDs by the server is almost always what you want, and the reasons for specifying an ID explicitly are very rare. We'll talk more about this later. The fact that each Stream entry has an ID is another similarity with log files, where line numbers, or the byte offset inside the file, can be used in order to identify a given entry. Returning back at our **XADD** example, after the key name and ID, the next arguments are the field-value pairs composing our stream entry.

It is possible to get the number of items inside a Stream just using the **XLEN** command:

```
> XLEN mystream
(integer) 1
```

Entry IDs

The entry ID returned by the **XADD** command, and identifying univocally each entry inside a given stream, is composed of two parts:

```
<millisecondsTime>-<sequenceNumber>
```

The milliseconds time part is actually the local time in the local Redis node generating the stream ID, however if the current milliseconds time happens to be smaller than the previous entry time, then the previous entry time is used instead, so if a clock jumps backward the monotonically incrementing ID property still holds. The sequence number is used for entries created in the same millisecond. Since the sequence number is 64 bit wide, in practical terms there are no limits to the number of entries that can be generated within the same millisecond.

The format of such IDs may look strange at first, and the gentle reader may wonder why the time is part of the ID. The reason is that Redis streams support range queries by ID. Because the ID is related to the time the entry is generated, this gives the ability to query for time ranges basically for free. We will see this soon while covering the **XRANGE** command.

If for some reason the user needs incremental IDs that are not related to time but are actually associated to another external system ID, as previously mentioned, the **XADD** command can take an explicit ID instead of the `*` wildcard ID that triggers auto-generation, like in the following examples:

```
> XADD somestream 0-1 field value
0-1
> XADD somestream 0-2 foo bar
0-2
```

Note that in this case, the minimum ID is 0-1 and that the command will not accept an ID equal or smaller than a previous one:

```
> XADD somestream 0-1 foo bar
(error) ERR The ID specified in XADD is equal or smaller than the tar
```

Getting data from Streams

Now we are finally able to append entries in our stream via **XADD**. However, while appending data to a stream is quite obvious, the way streams can be queried in order to extract data is not so obvious. If we continue with the analogy of the log file, one obvious way is to mimic what we normally do with the Unix command `tail -f`, that is, we may start to listen in order to get the new messages that are appended to the stream. Note that unlike the blocking list operations of Redis, where a given element will reach a single client which is blocking in a *pop style* operation like **BLPOP**, with streams we want multiple consumers to see the new messages appended to the stream (the same way many `tail -f` processes can see what is added to a log). Using the traditional terminology we want the streams to be able to *fan out* messages to multiple clients.

However, this is just one potential access mode. We could also see a stream in quite a different way: not as a messaging system, but as a *time series store*. In this case, maybe it's also useful to get the new messages appended, but another natural query mode is to get messages by ranges of time, or alternatively to iterate the messages using a cursor to incrementally check all the history. This is definitely another useful access mode.

Finally, if we see a stream from the point of view of consumers, we may want to access the stream in yet another way, that is, as a stream of messages that can be partitioned to multiple consumers that are processing such messages, so that groups of consumers can only see a subset of the messages arriving in a single stream. In this way, it is possible to scale the message processing across different consumers, without single consumers having to process all the messages: each consumer will just get different messages to process. This is basically what Kafka (TM) does with consumer groups. Reading messages via consumer groups is yet another interesting mode of reading from a Redis Stream.

Redis Streams support all the three query modes described above via different commands. The next sections will show them all, starting from the simplest and more direct to use:

range queries.

Querying by range: XRANGE and XREVRANGE

To query the stream by range we are only required to specify two IDs, *start* and *end*. The range returned will include the elements having start or end as ID, so the range is inclusive. The two special IDs - and + respectively mean the smallest and the greatest ID possible.

```
> XRANGE mystream - +
1) 1) 1518951480106-0
   2) 1) "sensor-id"
      2) "1234"
      3) "temperature"
      4) "19.8"
2) 1) 1518951482479-0
   2) 1) "sensor-id"
      2) "9999"
      3) "temperature"
      4) "18.2"
```

Each entry returned is an array of two items: the ID and the list of field-value pairs. We already said that the entry IDs have a relation with the time, because the part at the left of the - character is the Unix time in milliseconds of the local node that created the stream entry, at the moment the entry was created (however note that streams are replicated with fully specified **XADD** commands, so the replicas will have identical IDs to the master). This means that I could query a range of time using **XRANGE**. In order to do so, however, I may want to omit the sequence part of the ID: if omitted, in the start of the range it will be assumed to be 0, while in the end part it will be assumed to be the maximum sequence number available. This way, querying using just two milliseconds Unix times, we get all the entries that were generated in that range of time, in an inclusive way. For instance, if I want to query a two milliseconds period I could use:

```
> XRANGE mystream 1518951480106 1518951480107
1) 1) 1518951480106-0
   2) 1) "sensor-id"
      2) "1234"
      3) "temperature"
      4) "19.8"
```

I have only a single entry in this range, however in real data sets, I could query for ranges of hours, or there could be many items in just two milliseconds, and the result returned could be huge. For this reason, **XRANGE** supports an optional **COUNT** option at the end. By specifying a count, I can just get the first *N* items. If I want more, I can get the last ID returned, increment the sequence part by one, and query again. Let's see this in the following example. We start adding 10 items with **XADD** (I won't show that, lets assume that the stream `mystream` was populated with 10 items). To start my iteration, getting 2 items per command, I start with the full range, but with a count of 2.

```
> XRANGE mystream - + COUNT 2
1) 1) 1519073278252-0
   2) 1) "foo"
      2) "value_1"
2) 1) 1519073279157-0
   2) 1) "foo"
      2) "value_2"
```

In order to continue the iteration with the next two items, I have to pick the last ID returned, that is `1519073279157-0` and add the prefix `(` to it. The resulting exclusive range interval, that is `(1519073279157-0` in this case, can now be used as the new *start* argument for the next **XRANGE** call:

```
> XRANGE mystream (1519073279157-0 + COUNT 2
1) 1) 1519073280281-0
   2) 1) "foo"
      2) "value_3"
2) 1) 1519073281432-0
   2) 1) "foo"
      2) "value_4"
```

And so forth. Since **XRANGE** complexity is *O(log(N))* to seek, and then *O(M)* to return M elements, with a small count the command has a logarithmic time complexity, which means that each step of the iteration is fast. So **XRANGE** is also the de facto *streams iterator* and does not require an **XSCAN** command.

The command **XREVRANGE** is the equivalent of **XRANGE** but returning the elements in inverted order, so a practical use for **XREVRANGE** is to check what is the last item in a Stream:

```
> XREVRANGE mystream + - COUNT 1
1) 1) 1519073287312-0
   2) 1) "foo"
      2) "value_10"
```

Note that the **XREVRANGE** command takes the *start* and *stop* arguments in reverse order.

Listening for new items with XREAD

When we do not want to access items by a range in a stream, usually what we want instead is to *subscribe* to new items arriving to the stream. This concept may appear related to Redis Pub/Sub, where you subscribe to a channel, or to Redis blocking lists, where you wait for a key to get new elements to fetch, but there are fundamental differences in the way you consume a stream:

1. A stream can have multiple clients (consumers) waiting for data. Every new item, by default, will be delivered to *every consumer* that is waiting for data in a given stream. This behavior is different than blocking lists, where each consumer will get a different element. However, the ability to *fan out* to multiple consumers is similar to Pub/Sub.
2. While in Pub/Sub messages are *fire and forget* and are never stored anyway, and while when using blocking lists, when a message is received by the client it is *popped* (effectively removed) from the list, streams work in a fundamentally different way. All the messages are appended in the stream indefinitely (unless the user explicitly asks to delete entries): different consumers will know what is a new message from its point of view by remembering the ID of the last message received.
3. Streams Consumer Groups provide a level of control that Pub/Sub or blocking lists cannot achieve, with different groups for the same stream, explicit acknowledgment of processed items, ability to inspect the pending items, claiming of unprocessed messages, and coherent history visibility for each single client, that is only able to see its private past history of messages.

The command that provides the ability to listen for new messages arriving into a stream is called **XREAD**. It's a bit more complex than **XRANGE**, so we'll start showing simple forms, and later the whole command layout will be provided.

```
> XREAD COUNT 2 STREAMS mystream 0
1) 1) "mystream"
   2) 1) 1) 1519073278252-0
         2) 1) "foo"
            2) "value_1"
      2) 1) 1519073279157-0
         2) 1) "foo"
            2) "value_2"
```

The above is the non-blocking form of **XREAD**. Note that the **COUNT** option is not mandatory, in fact the only mandatory option of the command is the **STREAMS** option, that specifies a list of keys together with the corresponding maximum ID already seen for each stream by the calling consumer, so that the command will provide the client only with messages with an ID greater than the one we specified.

In the above command we wrote `STREAMS mystream 0` so we want all the messages in the Stream `mystream` having an ID greater than `0-0`. As you can see in the example above, the command returns the key name, because actually it is possible to call this command with more than one key to read from different streams at the same time. I could write, for instance: `STREAMS mystream otherstream 0 0`. Note how after the **STREAMS** option we need to provide the key names, and later the IDs. For this reason, the **STREAMS** option must always be the last one.

Apart from the fact that **XREAD** can access multiple streams at once, and that we are able to specify the last ID we own to just get newer messages, in this simple form the command is not doing something so different compared to **XRANGE**. However, the interesting part is that we can turn **XREAD** into a *blocking command* easily, by specifying the **BLOCK** argument:

```
> XREAD BLOCK 0 STREAMS mystream $
```

Note that in the example above, other than removing **COUNT**, I specified the new **BLOCK** option with a timeout of 0 milliseconds (that means to never timeout). Moreover, instead of passing a normal ID for the stream `mystream` I passed the special ID `$`. This special ID means that **XREAD** should use as last ID the maximum ID already stored in the stream `mystream`, so that we will receive only *new* messages, starting from the time we started listening. This is similar to the `tail -f` Unix command in some way.

Note that when the **BLOCK** option is used, we do not have to use the special ID `$`. We can use any valid ID. If the command is able to serve our request immediately without blocking, it will do so, otherwise it will block. Normally if we want to consume the stream starting from new entries, we start with the ID $, and after that we continue using the ID of the last message received to make the next call, and so forth.

The blocking form of **XREAD** is also able to listen to multiple Streams, just by specifying multiple key names. If the request can be served synchronously because there is at least one stream with elements greater than the corresponding ID we specified, it returns with the results. Otherwise, the command will block and will return the items of the first stream which gets new data (according to the specified ID).

Similarly to blocking list operations, blocking stream reads are *fair* from the point of view of clients waiting for data, since the semantics is FIFO style. The first client that blocked for a given stream will be the first to be unblocked when new items are available.

XREAD has no other options than **COUNT** and **BLOCK**, so it's a pretty basic command with a specific purpose to attach consumers to one or multiple streams. More powerful features to consume streams are available using the consumer groups API, however reading via consumer groups is implemented by a different command called **XREADGROUP**, covered in the next section of this guide.

Consumer groups

When the task at hand is to consume the same stream from different clients, then **XREAD** already offers a way to *fan-out* to N clients, potentially also using replicas in order to provide more read scalability. However in certain problems what we want to do is not to provide the same stream of messages to many clients, but to provide a *different subset* of messages from the same stream to many clients. An obvious case where this is useful is that of messages which are slow to process: the ability to have N different workers that will receive different parts of the stream allows us to scale message processing, by routing different messages to different workers that are ready to do more work.

In practical terms, if we imagine having three consumers C1, C2, C3, and a stream that contains the messages 1, 2, 3, 4, 5, 6, 7 then what we want is to serve the messages according to the following diagram:

```
1 -> C1
2 -> C2
3 -> C3
4 -> C1
5 -> C2
6 -> C3
7 -> C1
```

In order to achieve this, Redis uses a concept called *consumer groups*. It is very important to understand that Redis consumer groups have nothing to do, from an implementation standpoint, with Kafka (TM) consumer groups. Yet they are similar in functionality, so I decided to keep Kafka's (TM) terminology, as it originaly popularized this idea.

A consumer group is like a *pseudo consumer* that gets data from a stream, and actually serves multiple consumers, providing certain guarantees:

1. Each message is served to a different consumer so that it is not possible that the same message will be delivered to multiple consumers.
2. Consumers are identified, within a consumer group, by a name, which is a case-sensitive string that the clients implementing consumers must choose. This means that even after a disconnect, the stream consumer group retains all the state, since the client will claim again to be the same consumer. However, this also means that it is up to the client to provide a unique identifier.
3. Each consumer group has the concept of the *first ID never consumed* so that, when a consumer asks for new messages, it can provide just messages that were not previously delivered.
4. Consuming a message, however, requires an explicit acknowledgment using a specific command. Redis interperts the acknowledgment as: this message was correctly processed so it can be evicted from the consumer group.
5. A consumer group tracks all the messages that are currently pending, that is, messages that were delivered to some consumer of the consumer group, but are yet to be acknowledged as processed. Thanks to this feature, when accessing the message history of a stream, each consumer *will only see messages that were delivered to it*.

In a way, a consumer group can be imagined as some *amount of state* about a stream:

```
+----------------------------------------+
| consumer_group_name: mygroup           |
| consumer_group_stream: somekey         |
| last_delivered_id: 1292309234234-92    |
|                                        |
| consumers:                             |
|    "consumer-1" with pending messages  |
|       1292309234234-4                  |
|       1292309234232-8                  |
|    "consumer-42" with pending messages |
|       ... (and so forth)               |
+----------------------------------------+
```

If you see this from this point of view, it is very simple to understand what a consumer group can do, how it is able to just provide consumers with their history of pending messages, and how consumers asking for new messages will just be served with message IDs greater than `last_delivered_id`. At the same time, if you look at the consumer group as an auxiliary data structure for Redis streams, it is obvious that a single stream can have multiple consumer groups, that have a different set of consumers. Actually, it is even possible for the same stream to have clients reading without consumer groups via **XREAD**, and clients reading via **XREADGROUP** in different consumer groups.

Now it's time to zoom in to see the fundamental consumer group commands. They are the following:

- **XGROUP** is used in order to create, destroy and manage consumer groups.
- **XREADGROUP** is used to read from a stream via a consumer group.
- **XACK** is the command that allows a consumer to mark a pending message as correctly processed.

Creating a consumer group

Assuming I have a key `mystream` of type stream already existing, in order to create a consumer group I just need to do the following:

```
> XGROUP CREATE mystream mygroup $
OK
```

As you can see in the command above when creating the consumer group we have to specify an ID, which in the example is just `$`. This is needed because the consumer group, among the other states, must have an idea about what message to serve next at the first consumer connecting, that is, what was the *last message ID* when the group was just created. If we provide `$` as we did, then only new messages arriving in the stream from now on will be provided to the consumers in the group. If we specify `0` instead the consumer group will consume *all* the messages in the stream history to start with. Of course, you can specify any other valid ID. What you know is that the consumer group will start delivering messages that are greater than the ID you specify. Because `$` means the current greatest ID in the stream, specifying `$` will have the effect of consuming only new messages.

`XGROUP CREATE` also supports creating the stream automatically, if it doesn't exist, using the optional `MKSTREAM` subcommand as the last argument:

```
> XGROUP CREATE newstream mygroup $ MKSTREAM
OK
```

Now that the consumer group is created we can immediately try to read messages via the consumer group using the **XREADGROUP** command. We'll read from consumers, that we will call Alice and Bob, to see how the system will return different messages to Alice or Bob.

XREADGROUP is very similar to **XREAD** and provides the same **BLOCK** option, otherwise it is a synchronous command. However there is a *mandatory* option that must be always specified, which is **GROUP** and has two arguments: the name of the consumer group, and the name of the consumer that is attempting to read. The option **COUNT** is also supported and is identical to the one in **XREAD**.

Before reading from the stream, let's put some messages inside:

```
> XADD mystream * message apple
1526569495631-0
> XADD mystream * message orange
1526569498055-0
> XADD mystream * message strawberry
1526569506935-0
> XADD mystream * message apricot
1526569535168-0
> XADD mystream * message banana
1526569544280-0
```

Note: *here message is the field name, and the fruit is the associated value, remember that stream items are small dictionaries.*

It is time to try reading something using the consumer group:

```
> XREADGROUP GROUP mygroup Alice COUNT 1 STREAMS mystream >
1) 1) "mystream"
   2) 1) 1) 1526569495631-0
         2) 1) "message"
            2) "apple"
```

XREADGROUP replies are just like **XREAD** replies. Note however the `GROUP <group-name> <consumer-name>` provided above. It states that I want to read from the stream using the consumer group `mygroup` and I'm the consumer `Alice`. Every time a consumer performs an operation with a consumer group, it must specify its name, uniquely identifying this consumer inside the group.

There is another very important detail in the command line above, after the mandatory **STREAMS** option the ID requested for the key `mystream` is the special ID >. This special ID is only valid in the context of consumer groups, and it means: **messages never delivered to other consumers so far**.

This is almost always what you want, however it is also possible to specify a real ID, such as `0` or any other valid ID, in this case, however, what happens is that we request from **XREADGROUP** to just provide us with the **history of pending messages**, and in such case, will never see new messages in the group. So basically **XREADGROUP** has the following behavior based on the ID we specify:

- If the ID is the special ID > then the command will return only new messages never delivered to other consumers so far, and as a side effect, will update the consumer group's *last ID*.
- If the ID is any other valid numerical ID, then the command will let us access our *history of pending messages*. That is, the set of messages that were delivered to this specified consumer (identified by the provided name), and never acknowledged so far with **XACK**.

We can test this behavior immediately specifying an ID of 0, without any **COUNT** option: we'll just see the only pending message, that is, the one about apples:

```
> XREADGROUP GROUP mygroup Alice STREAMS mystream 0
1) 1) "mystream"
   2) 1) 1) 1526569495631-0
         2) 1) "message"
            2) "apple"
```

However, if we acknowledge the message as processed, it will no longer be part of the pending messages history, so the system will no longer report anything:

```
> XACK mystream mygroup 1526569495631-0
(integer) 1
> XREADGROUP GROUP mygroup Alice STREAMS mystream 0
1) 1) "mystream"
   2) (empty list or set)
```

Don't worry if you yet don't know how **XACK** works, the idea is just that processed messages are no longer part of the history that we can access.

Now it's Bob's turn to read something:

```
> XREADGROUP GROUP mygroup Bob COUNT 2 STREAMS mystream >
1) 1) "mystream"
   2) 1) 1) 1526569498055-0
         2) 1) "message"
            2) "orange"
      2) 1) 1526569506935-0
         2) 1) "message"
            2) "strawberry"
```

Bob asked for a maximum of two messages and is reading via the same group `mygroup`. So what happens is that Redis reports just *new* messages. As you can see the "apple" message is not delivered, since it was already delivered to Alice, so Bob gets orange and strawberry, and so forth.

This way Alice, Bob, and any other consumer in the group, are able to read different messages from the same stream, to read their history of yet to process messages, or to mark messages as processed. This allows creating different topologies and semantics for consuming messages from a stream.

There are a few things to keep in mind:

- Consumers are auto-created the first time they are mentioned, no need for explicit creation.
- Even with **XREADGROUP** you can read from multiple keys at the same time, however for this to work, you need to create a consumer group with the same name in every stream. This is not a common need, but it is worth mentioning that the feature is technically available.
- **XREADGROUP** is a *write command* because even if it reads from the stream, the consumer group is modified as a side effect of reading, so it can only be called on master instances.

An example of a consumer implementation, using consumer groups, written in the Ruby language could be the following. The Ruby code is aimed to be readable by virtually any experienced programmer, even if they do not know Ruby:

```
require 'redis'

if ARGV.length == 0
    puts "Please specify a consumer name"
    exit 1
end

ConsumerName = ARGV[0]
GroupName = "mygroup"
r = Redis.new

def process_message(id,msg)
    puts "[#{ConsumerName}] #{id} = #{msg.inspect}"
end

$lastid = '0-0'

puts "Consumer #{ConsumerName} starting..."
check_backlog = true
while true
    # Pick the ID based on the iteration: the first time we want to
    # read our pending messages, in case we crashed and are recoverin
    # Once we consumed our history, we can start getting new messages
    if check_backlog
        myid = $lastid
    else
        myid = '>'
    end

    items = r.xreadgroup('GROUP',GroupName,ConsumerName,'BLOCK','2000

    if items == nil
        puts "Timeout!"
        next
    end

    # If we receive an empty reply, it means we were consuming our hi
    # and that the history is now empty. Let's start to consume new m
    check_backlog = false if items[0][1].length == 0

    items[0][1].each{|i|
        id,fields = i

        # Process the message
        process_message(id,fields)
        # Acknowledge the message as processed
        r.xack(:my_stream_key,GroupName,id)

        $lastid = id
    }
end
```

As you can see the idea here is to start by consuming the history, that is, our list of pending messages. This is useful because the consumer may have crashed before, so in the event of a restart we want to re-read messages that were delivered to us without getting acknowledged. Note that we might process a message multiple times or one time (at least in the case of consumer failures, but there are also the limits of Redis persistence and replication involved, see the specific section about this topic).

Once the history was consumed, and we get an empty list of messages, we can switch to use the > special ID in order to consume new messages.

Recovering from permanent failures

The example above allows us to write consumers that participate in the same consumer group, each taking a subset of messages to process, and when recovering from failures re-reading the pending messages that were delivered just to them. However in the real world consumers may permanently fail and never recover. What happens to the pending messages of the consumer that never recovers after stopping for any reason?

Redis consumer groups offer a feature that is used in these situations in order to *claim* the pending messages of a given consumer so that such messages will change ownership and will be re-assigned to a different consumer. The feature is very explicit. A consumer has to inspect the list of pending messages, and will have to claim specific messages using a special command, otherwise the server will leave the messages pending forever and assigned to the old consumer. In this way different applications can choose if to use such a feature or not, and exactly how to use it.

The first step of this process is just a command that provides observability of pending entries in the consumer group and is called **XPENDING**. This is a read-only command which is always safe to call and will not change ownership of any message. In its simplest form, the command is called with two arguments, which are the name of the stream and the name of the consumer group.

```
> XPENDING mystream mygroup
1) (integer) 2
2) 1526569498055-0
3) 1526569506935-0
4) 1) 1) "Bob"
      2) "2"
```

When called in this way the command outputs the total number of pending messages in the consumer group, only two messages in this case, the lower and higher message ID among the pending messages, and finally a list of consumers and the number of pending messages they have. We have only Bob with two pending messages because the single message that Alice requested was acknowledged using **XACK**.

We can ask for more information by giving more arguments to **XPENDING**, because the full command signature is the following:

```
XPENDING <key> <groupname> [<start-id> <end-id> <count> [<consumer-na
```

By providing a start and end ID (that can be just - and + as in **XRANGE**) and a count to control the amount of information returned by the command, we are able to know more about the pending messages. The optional final argument, the consumer name, is used if we want to limit the output to just messages pending for a given consumer, but won't use this feature in the following example.

```
> XPENDING mystream mygroup - + 10
1) 1) 1526569498055-0
   2) "Bob"
   3) (integer) 74170458
   4) (integer) 1
2) 1) 1526569506935-0
   2) "Bob"
   3) (integer) 74170458
   4) (integer) 1
```

Now we have the details for each message: the ID, the consumer name, the *idle time* in milliseconds, which is how much milliseconds have passed since the last time the message was delivered to some consumer, and finally the number of times that a given message was delivered. We have two messages from Bob, and they are idle for 74170458 milliseconds, about 20 hours.

Note that nobody prevents us from checking what the first message content was by just using **XRANGE**.

```
> XRANGE mystream 1526569498055-0 1526569498055-0
1) 1) 1526569498055-0
   2) 1) "message"
      2) "orange"
```

We have just to repeat the same ID twice in the arguments. Now that we have some idea, Alice may decide that after 20 hours of not processing messages, Bob will probably not recover in time, and it's time to *claim* such messages and resume the processing in place of Bob. To do so, we use the **XCLAIM** command.

This command is very complex and full of options in its full form, since it is used for replication of consumer groups changes, but we'll use just the arguments that we need normally. In this case it is as simple as:

```
XCLAIM <key> <group> <consumer> <min-idle-time> <ID-1> <ID-2> ... <ID
```

Basically we say, for this specific key and group, I want that the message IDs specified will change ownership, and will be assigned to the specified consumer name `<consumer>`. However, we also provide a minimum idle time, so that the operation will only work if the idle time of the mentioned messages is greater than the specified idle time. This is useful because maybe two clients are retrying to claim a message at the same time:

```
Client 1: XCLAIM mystream mygroup Alice 3600000 1526569498055-0
Client 2: XCLAIM mystream mygroup Lora 3600000 1526569498055-0
```

However claiming a message, as a side effect will reset its idle time! And will increment its number of deliveries counter, so the second client will fail claiming it. In this way we avoid trivial re-processing of messages (even if in the general case you cannot obtain exactly once processing).

This is the result of the command execution:

```
> XCLAIM mystream mygroup Alice 3600000 1526569498055-0
1) 1) 1526569498055-0
   2) 1) "message"
      2) "orange"
```

The message was successfully claimed by Alice, that can now process the message and acknowledge it, and move things forward even if the original consumer is not recovering.

It is clear from the example above that as a side effect of successfully claiming a given message, the **XCLAIM** command also returns it. However this is not mandatory. The **JUSTID** option can be used in order to return just the IDs of the message successfully claimed. This is useful if you want to reduce the bandwidth used between the client and the server (and also the performance of the command) and you are not interested in the message because your consumer is implemented in a way that it will rescan the history of pending messages from time to time.

Claiming may also be implemented by a separate process: one that just checks the list of pending messages, and assigns idle messages to consumers that appear to be active. Active consumers can be obtained using one of the observability features of Redis streams. This is the topic of the next section.

Automatic claiming

The XAUTOCLAIM command, added in Redis 6.2, implements the claiming process that we've described above. XPENDING and XCLAIM provide the basic building blocks for different types of recovery mechanisms. This command optimizes the generic process by having Redis manage it and offers a simple solution for most recovery needs.

XAUTOCLAIM identifies idle pending messages and transfers ownership of them to a consumer. The command's signature looks like this:

```
XAUTOCLAIM <key> <group> <consumer> <min-idle-time> <start> [COUNT co
```

So, in the example above, I could have used automatic claiming to claim a single message like this:

```
> XAUTOCLAIM mystream mygroup Alice 3600000 0-0 COUNT 1
1) 1526569498055-0
2) 1) 1526569498055-0
   2) 1) "message"
      2) "orange"
```

Like XCLAIM, the command replies with an array of the claimed messages, but it also returns a stream ID that allows iterating the pending entries. The stream ID is a cursor, and I can use it in my next call to continue in claiming idle pending messages:

```
> XAUTOCLAIM mystream mygroup Lora 3600000 1526569498055-0 COUNT 1
1) 0-0
2) 1) 1526569506935-0
   2) 1) "message"
      2) "strawberry"
```

When XAUTOCLAIM returns the "0-0" stream ID as a cursor, that means that it reached the end of the consumer group pending entries list. That doesn't mean that there are no new idle pending messages, so the process continues by calling XAUTOCLAIM from the beginning of the stream.

Claiming and the delivery counter

The counter that you observe in the **XPENDING** output is the number of deliveries of each message. The counter is incremented in two ways: when a message is successfully claimed via **XCLAIM** or when an **XREADGROUP** call is used in order to access the history of pending messages.

When there are failures, it is normal that messages will be delivered multiple times, but eventually they usually get processed and acknowledged. However there might be a problem processing some specific message, because it is corrupted or crafted in a way that triggers a bug in the processing code. In such a case what happens is that consumers will continuously fail to process this particular message. Because we have the counter of the delivery attempts, we can use that counter to detect messages that for some reason are not processable. So once the deliveries counter reaches a given large number that you chose, it is probably wiser to put such messages in another stream and send a notification to the system administrator. This is basically the way that Redis Streams implements the *dead letter* concept.

Streams observability

Messaging systems that lack observability are very hard to work with. Not knowing who is consuming messages, what messages are pending, the set of consumer groups active in a given stream, makes everything opaque. For this reason, Redis Streams and consumer groups have different ways to observe what is happening. We already covered **XPENDING**, which allows us to inspect the list of messages that are under processing at a given moment, together with their idle time and number of deliveries.

However we may want to do more than that, and the **XINFO** command is an observability interface that can be used with sub-commands in order to get information about streams or consumer groups.

This command uses subcommands in order to show different information about the status of the stream and its consumer groups. For instance **XINFO STREAM** reports information about the stream itself.

```
> XINFO STREAM mystream
 1) length
 2) (integer) 13
 3) radix-tree-keys
 4) (integer) 1
 5) radix-tree-nodes
 6) (integer) 2
 7) groups
 8) (integer) 2
 9) first-entry
10) 1) 1526569495631-0
    2) 1) "message"
       2) "apple"
11) last-entry
12) 1) 1526569544280-0
    2) 1) "message"
       2) "banana"
```

The output shows information about how the stream is encoded internally, and also shows the first and last message in the stream. Another piece of information available is the number of consumer groups associated with this stream. We can dig further asking for more information about the consumer groups.

```
> XINFO GROUPS mystream
1) 1) name
   2) "mygroup"
   3) consumers
   4) (integer) 2
   5) pending
   6) (integer) 2
   7) last-delivered-id
   8) "1588152489012-0"
2) 1) name
   2) "some-other-group"
   3) consumers
   4) (integer) 1
   5) pending
   6) (integer) 0
   7) last-delivered-id
   8) "1588152498034-0"
```

As you can see in this and in the previous output, the **XINFO** command outputs a sequence of field-value items. Because it is an observability command this allows the human user to immediately understand what information is reported, and allows the command to report more information in the future by adding more fields without breaking compatibility with older clients. Other commands that must be more bandwidth efficient, like **XPENDING**, just report the information without the field names.

The output of the example above, where the **GROUPS** subcommand is used, should be clear observing the field names. We can check in more detail the state of a specific consumer group by checking the consumers that are registered in the group.

```
> XINFO CONSUMERS mystream mygroup
1) 1) name
   2) "Alice"
   3) pending
   4) (integer) 1
   5) idle
   6) (integer) 9104628
2) 1) name
   2) "Bob"
   3) pending
   4) (integer) 1
   5) idle
   6) (integer) 83841983
```

In case you do not remember the syntax of the command, just ask the command itself for help:

```
> XINFO HELP
1) XINFO <subcommand> arg arg ... arg. Subcommands are:
2) CONSUMERS <key> <groupname>  -- Show consumer groups of group <gro
3) GROUPS <key>                 -- Show the stream consumer groups.
4) STREAM <key>                 -- Show information about the stream.
5) HELP                         -- Print this help.
```

Differences with Kafka (TM) partitions

Consumer groups in Redis streams may resemble in some way Kafka (TM) partitioning-based consumer groups, however note that Redis streams are, in practical terms, very different. The partitions are only *logical* and the messages are just put into a single Redis key, so the way the different clients are served is based on who is ready to process new messages, and not from which partition clients are reading. For instance, if the consumer

C3 at some point fails permanently, Redis will continue to serve C1 and C2 all the new messages arriving, as if now there are only two *logical* partitions.

Similarly, if a given consumer is much faster at processing messages than the other consumers, this consumer will receive proportionally more messages in the same unit of time. This is possible since Redis tracks all the unacknowledged messages explicitly, and remembers who received which message and the ID of the first message never delivered to any consumer.

However, this also means that in Redis if you really want to partition messages in the same stream into multiple Redis instances, you have to use multiple keys and some sharding system such as Redis Cluster or some other application-specific sharding system. A single Redis stream is not automatically partitioned to multiple instances.

We could say that schematically the following is true:

- If you use 1 stream -> 1 consumer, you are processing messages in order.
- If you use N streams with N consumers, so that only a given consumer hits a subset of the N streams, you can scale the above model of 1 stream -> 1 consumer.
- If you use 1 stream -> N consumers, you are load balancing to N consumers, however in that case, messages about the same logical item may be consumed out of order, because a given consumer may process message 3 faster than another consumer is processing message 4.

So basically Kafka partitions are more similar to using N different Redis keys, while Redis consumer groups are a server-side load balancing system of messages from a given stream to N different consumers.

Capped Streams

Many applications do not want to collect data into a stream forever. Sometimes it is useful to have at maximum a given number of items inside a stream, other times once a given size is reached, it is useful to move data from Redis to a storage which is not in memory and not as fast but suited to store the history for, potentially, decades to come. Redis streams have some support for this. One is the **MAXLEN** option of the **XADD** command. This option is very simple to use:

```
> XADD mystream MAXLEN 2 * value 1
1526654998691-0
> XADD mystream MAXLEN 2 * value 2
1526654999635-0
> XADD mystream MAXLEN 2 * value 3
1526655000369-0
> XLEN mystream
(integer) 2
> XRANGE mystream - +
1) 1) 1526654999635-0
   2) 1) "value"
      2) "2"
2) 1) 1526655000369-0
   2) 1) "value"
      2) "3"
```

Using **MAXLEN** the old entries are automatically evicted when the specified length is reached, so that the stream is left at a constant size. There is currently no option to tell the stream to just retain items that are not older than a given period, because such command, in order to run consistently, would potentially block for a long time in order to evict items. Imagine for example what happens if there is an insertion spike, then a long pause, and another insertion, all with the same maximum time. The stream would block to evict the data that became too old during the pause. So it is up to the user to do some planning and understand what is the maximum stream length desired. Moreover, while the length of the stream is proportional to the memory used, trimming by time is less simple to control and anticipate: it depends on the insertion rate which often changes over time (and when it does not change, then to just trim by size is trivial).

However trimming with **MAXLEN** can be expensive: streams are represented by macro nodes into a radix tree, in order to be very memory efficient. Altering the single macro node, consisting of a few tens of elements, is not optimal. So it's possible to use the command in the following special form:

```
XADD mystream MAXLEN ~ 1000 * ... entry fields here ...
```

The ~ argument between the **MAXLEN** option and the actual count means, I don't really need this to be exactly 1000 items. It can be 1000 or 1010 or 1030, just make sure to save at least 1000 items. With this argument, the trimming is performed only when we can remove a whole node. This makes it much more efficient, and it is usually what you want.

There is also the **XTRIM** command, which performs something very similar to what the **MAXLEN** option does above, except that it can be run by itself:

```
> XTRIM mystream MAXLEN 10
```

Or, as for the **XADD** option:

```
> XTRIM mystream MAXLEN ~ 10
```

However, **XTRIM** is designed to accept different trimming strategies. Another trimming strategy is **MINID**, that evicts entries with IDs lower than the one specified.

As **XTRIM** is an explicit command, the user is expected to know about the possible shortcomings of different trimming strategies.

Another useful eviction strategy that may be added to **XTRIM** in the future, is to remove by a range of IDs to ease use of **XRANGE** and **XTRIM** to move data from Redis to other storage systems if needed.

Special IDs in the streams API

You may have noticed that there are several special IDs that can be used in the Redis API. Here is a short recap, so that they can make more sense in the future.

The first two special IDs are - and +, and are used in range queries with the XRANGE command. Those two IDs respectively mean the smallest ID possible (that is basically `0-1`) and the greatest ID possible (that is `18446744073709551615-18446744073709551615`). As you can see it is a lot cleaner to write - and + instead of those numbers.

Then there are APIs where we want to say, the ID of the item with the greatest ID inside the stream. This is what $ means. So for instance if I want only new entries with XREADGROUP I use this ID to signify I already have all the existing entries, but not the new ones that will be inserted in the future. Similarly when I create or set the ID of a consumer group, I can set the last delivered item to $ in order to just deliver new entries to the consumers in the group.

As you can see $ does not mean +, they are two different things, as + is the greatest ID possible in every possible stream, while $ is the greatest ID in a given stream containing given entries. Moreover APIs will usually only understand + or $, yet it was useful to avoid loading a given symbol with multiple meanings.

Another special ID is >, that is a special meaning only related to consumer groups and only when the XREADGROUP command is used. This special ID means that we want only entries that were never delivered to other consumers so far. So basically the > ID is the *last delivered ID* of a consumer group.

Finally the special ID ∗, that can be used only with the XADD command, means to auto select an ID for us for the new entry.

So we have −, +, $, > and ∗, and all have a different meaning, and most of the times, can be used in different contexts.

Persistence, replication and message safety

A Stream, like any other Redis data structure, is asynchronously replicated to replicas and persisted into AOF and RDB files. However what may not be so obvious is that also the consumer groups full state is propagated to AOF, RDB and replicas, so if a message is pending in the master, also the replica will have the same information. Similarly, after a restart, the AOF will restore the consumer groups' state.

However note that Redis streams and consumer groups are persisted and replicated using the Redis default replication, so:

- AOF must be used with a strong fsync policy if persistence of messages is important in your application.
- By default the asynchronous replication will not guarantee that **XADD** commands or consumer groups state changes are replicated: after a failover something can be missing depending on the ability of replicas to receive the data from the master.
- The **WAIT** command may be used in order to force the propagation of the changes to a set of replicas. However note that while this makes it very unlikely that data is lost, the Redis failover process as operated by Sentinel or Redis Cluster performs only a *best effort* check to failover to the replica which is the most updated, and under certain specific failure conditions may promote a replica that lacks some data.

So when designing an application using Redis streams and consumer groups, make sure to understand the semantical properties your application should have during failures, and configure things accordingly, evaluating whether it is safe enough for your use case.

Removing single items from a stream

Streams also have a special command for removing items from the middle of a stream, just by ID. Normally for an append only data structure this may look like an odd feature, but it is actually useful for applications involving, for instance, privacy regulations. The command is called **XDEL** and receives the name of the stream followed by the IDs to delete:

```
> XRANGE mystream - + COUNT 2
1) 1) 1526654999635-0
   2) 1) "value"
      2) "2"
2) 1) 1526655000369-0
   2) 1) "value"
      2) "3"
> XDEL mystream 1526654999635-0
(integer) 1
> XRANGE mystream - + COUNT 2
1) 1) 1526655000369-0
   2) 1) "value"
      2) "3"
```

However in the current implementation, memory is not really reclaimed until a macro node is completely empty, so you should not abuse this feature.

Zero length streams

A difference between streams and other Redis data structures is that when the other data structures no longer have any elements, as a side effect of calling commands that remove elements, the key itself will be removed. So for instance, a sorted set will be completely removed when a call to **ZREM** will remove the last element in the sorted set. Streams, on the other hand, are allowed to stay at zero elements, both as a result of using a **MAXLEN** option with a count of zero (**XADD** and **XTRIM** commands), or because **XDEL** was called.

The reason why such an asymmetry exists is because Streams may have associated consumer groups, and we do not want to lose the state that the consumer groups defined just because there are no longer any items in the stream. Currently the stream is not deleted even when it has no associated consumer groups, but this may change in the future.

Total latency of consuming a message

Non blocking stream commands like XRANGE and XREAD or XREADGROUP without the BLOCK option are served synchronously like any other Redis command, so to discuss latency of such commands is meaningless: it is more interesting to check the time complexity of the commands in the Redis documentation. It should be enough to say that stream commands are at least as fast as sorted set commands when extracting ranges, and that XADD is very fast and can easily insert from half a million to one million items per second in an average machine if pipelining is used.

However latency becomes an interesting parameter if we want to understand the delay of processing a message, in the context of blocking consumers in a consumer group, from the moment the message is produced via XADD, to the moment the message is obtained by the consumer because XREADGROUP returned with the message.

How serving blocked consumers work

Before providing the results of performed tests, it is interesting to understand what model Redis uses in order to route stream messages (and in general actually how any blocking operation waiting for data is managed).

- The blocked client is referenced in an hash table that maps keys for which there is at least one blocking consumer, to a list of consumers that are waiting for such key. This way, given a key that received data, we can resolve all the clients that are waiting for such data.
- When a write happens, in this case when the XADD command is called, it calls the `signalKeyAsReady()` function. This function will put the key into a list of keys that need to be processed, because such keys may have new data for blocked consumers. Note that such *ready keys* will be processed later, so in the course of the same event loop cycle, it is possible that the key will receive other writes.
- Finally, before returning into the event loop, the *ready keys* are finally processed. For each key the list of clients waiting for data is scanned, and if applicable, such clients will receive the new data that arrived. In the case of streams the data is the messages in the applicable range requested by the consumer.

As you can see, basically, before returning to the event loop both the client calling XADD and the clients blocked to consume messages, will have their reply in the output buffers, so the caller of XADD should receive the reply from Redis about at the same time the consumers will receive the new messages.

This model is *push based*, since adding data to the consumers buffers will be performed directly by the action of calling XADD, so the latency tends to be quite predictable.

Latency tests results

In order to check this latency characteristics a test was performed using multiple instances of Ruby programs pushing messages having as an additional field the computer millisecond time, and Ruby programs reading the messages from the consumer group and processing them. The message processing step consisted in comparing the current computer time with the message timestamp, in order to understand the total latency.

Such programs were not optimized and were executed in a small two core instance also running Redis, in order to try to provide the latency figures you could expect in non optimal conditions. Messages were produced at a rate of 10k per second, with ten simultaneous consumers consuming and acknowledging the messages from the same Redis stream and consumer group.

Results obtained:

```
Processed between 0 and 1 ms -> 74.11%
Processed between 1 and 2 ms -> 25.80%
Processed between 2 and 3 ms -> 0.06%
Processed between 3 and 4 ms -> 0.01%
Processed between 4 and 5 ms -> 0.02%
```

So 99.9% of requests have a latency <= 2 milliseconds, with the outliers that remain still very close to the average.

Adding a few million unacknowledged messages to the stream does not change the gist of the benchmark, with most queries still processed with very short latency.

A few remarks:

- Here we processed up to 10k messages per iteration, this means that the `COUNT` parameter of XREADGROUP was set to 10000. This adds a lot of latency but is needed in order to allow the slow consumers to be able to keep with the message flow. So you can expect a real world latency that is a lot smaller.
- The system used for this benchmark is very slow compared to today's standards.