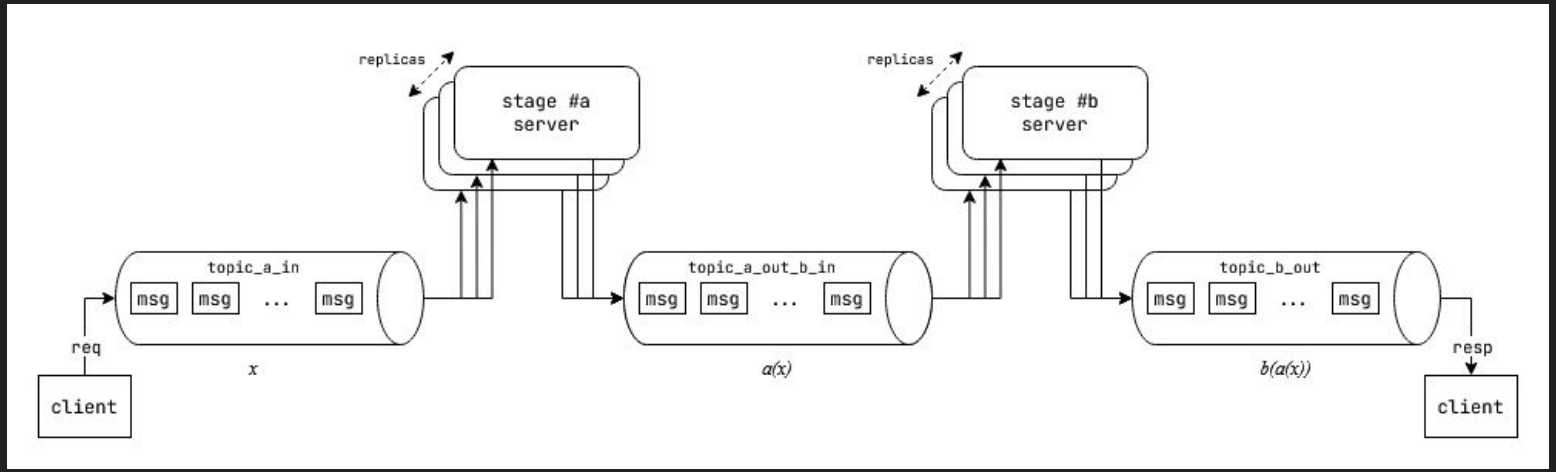
Message Queues

Function and Role in ML Inference

What is this talk about?

- Multi-stage horizontally scalable inference pipelines with Message Queues
- An introduction to message queues
- A guide on designing services with message queues



A bit about me.

```
$ sudo lifectl status arindas.service
● arindas.service - MLOps Engineer: Arindam Das
         Company: Claritas Healthtech
        Industry: Medical Image enhancement, Disease prediction from reports
            Role: Machine Learning Models → Production scale Web Services
     Open Source: Low latency Distributed systems, end user utilities
        Building: laminarmq, a resource efficient alternative to Apache Kafka

Jun 18 12:00:00 body life[1]: Starting Talk: “Message Queue: Function and”...
```

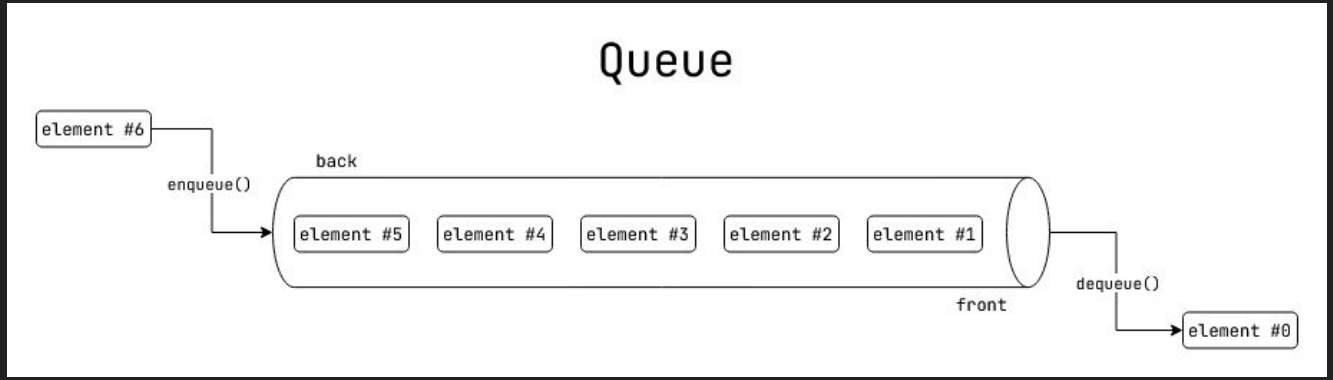
Message Queues

An introduction.

A brief description of the requirements for message queues.

What are message queues?

Well, we see the word “queue” in it.

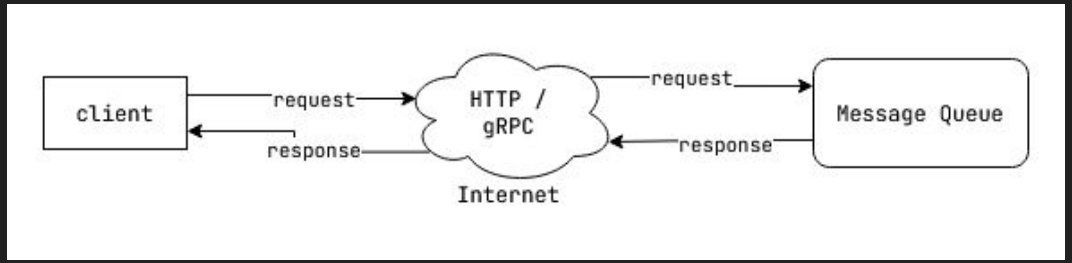


Queue: A FIFO data structure of elements.

However, from this definition we don’t know: Storage? Access?

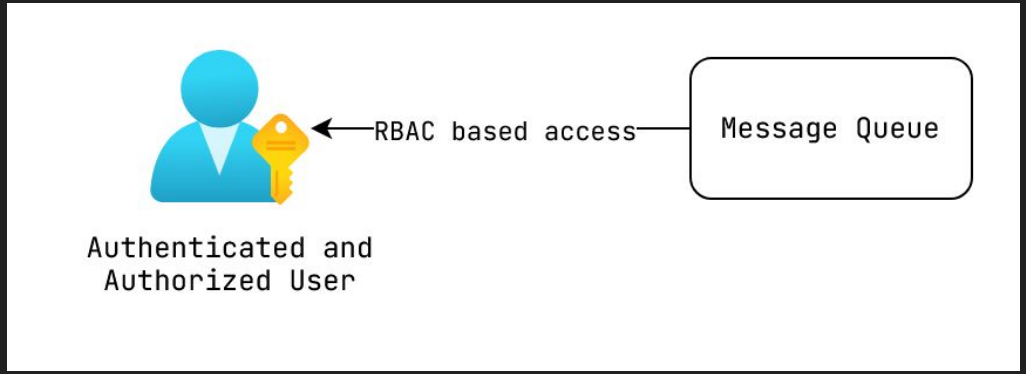
Additional req. in 202X

- Internet accessible (HTTP / gRPC)



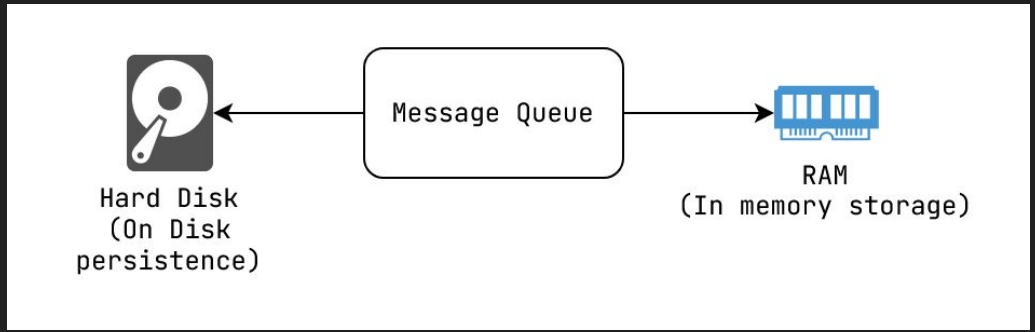
Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)



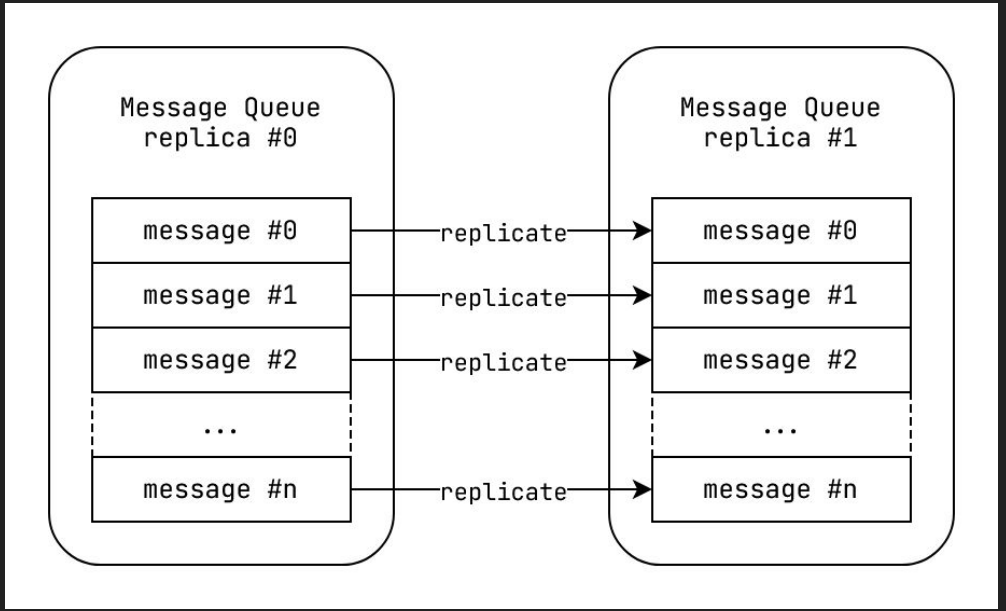
Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)



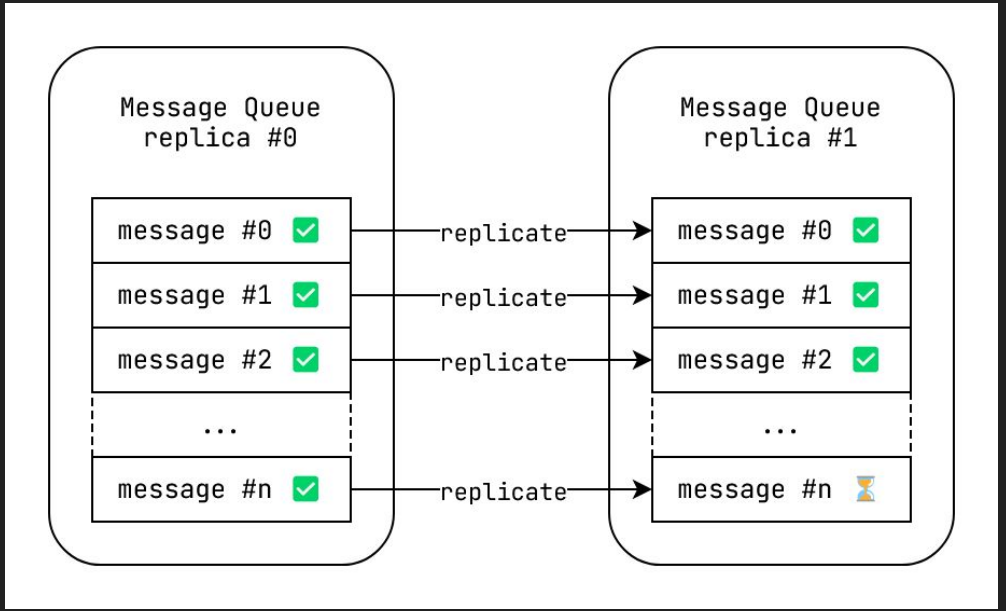
Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication



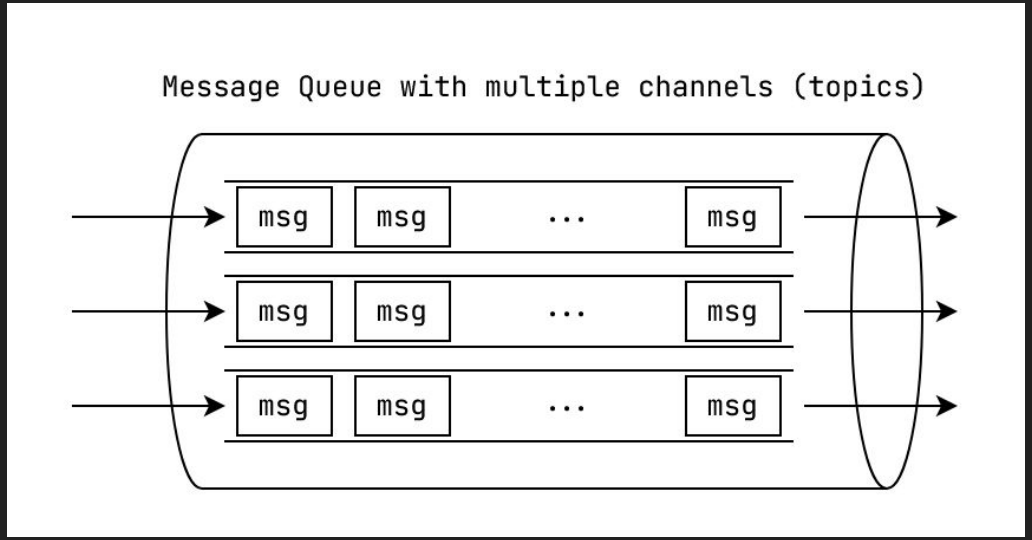
Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication
- Consistency among Replicas



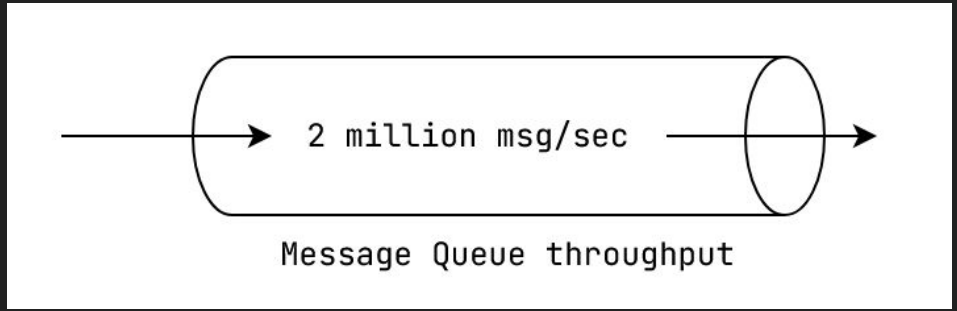
Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication
- Consistency among Replicas
- Support multiple channels



Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication
- Consistency among Replicas
- Support multiple channels
- Reasonably Fast



Additional req. in 202X

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication
- Consistency among Replicas
- Support multiple channels
- Reasonably Fast
- Effortless

What is a message queue (in 202X) ?

- Internet accessible (HTTP / gRPC)
- Access control (RBAC)
- Persistent (on Disk, In mem)
- Availability with Replication
- Consistency among Replicas
- Support multiple channels
- Reasonably Fast
- Effortless

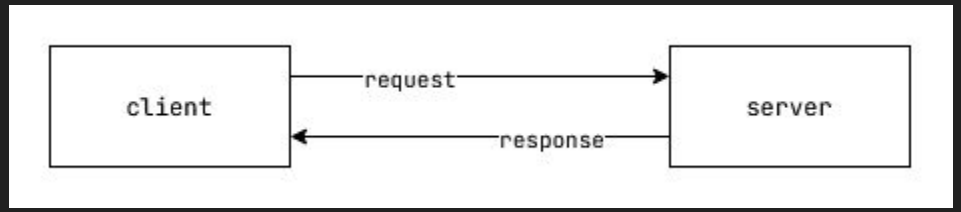
A Message Queue is a queue of “message” elements which fulfils all these requirements.

Message Queue

Usage model

Enabling horizontally scalable asynchronous processing

Background: Client Server model



client -request⟶ server

client ⟵response- server

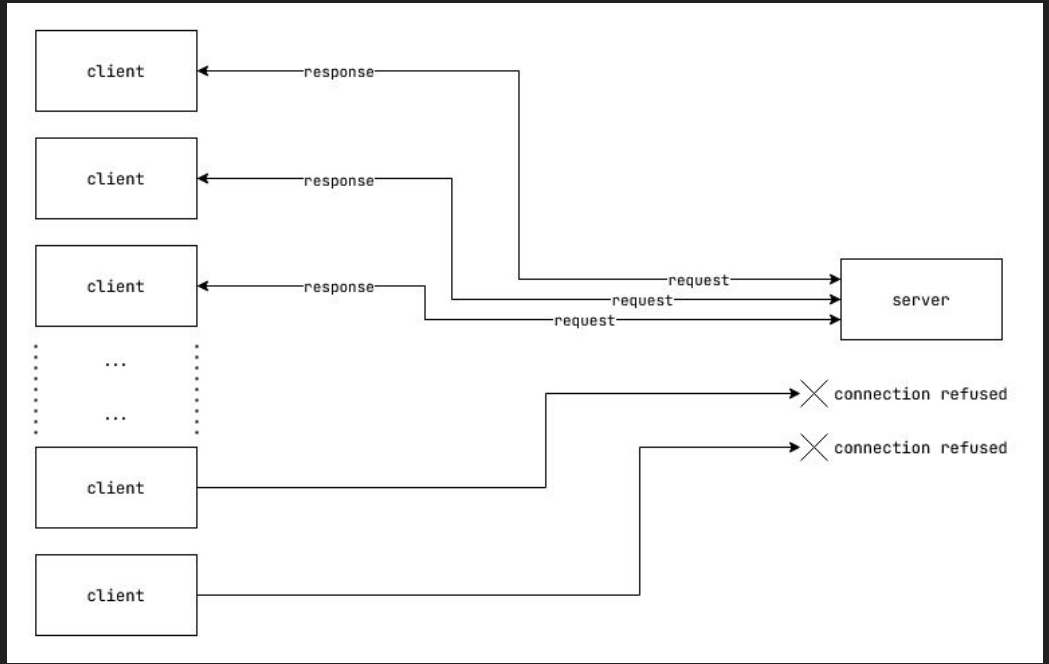
(Synchronous request handling, blocking in nature)

Background: Client Server model (contd.)

Server overwhelmed. What now?

- Delay requests
- Refuse connections past `MAX_CONNECTIONS` limit
- Rate limit requests

Some requests are lost!



Message Queue: Asynchronous Processing

`client → req_msg`

`req_msg → req_q`

`server ← req_msg ← req_q`

`server ::process(req_msg)`

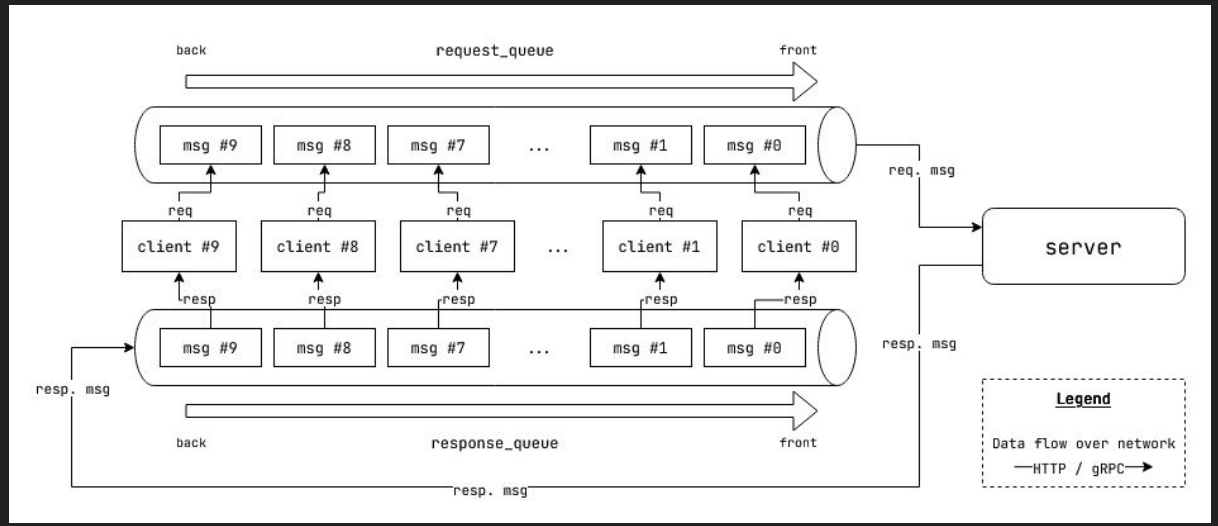
`server → resp_msg → resp_q`

`client ← resp_msg ← resp_q`

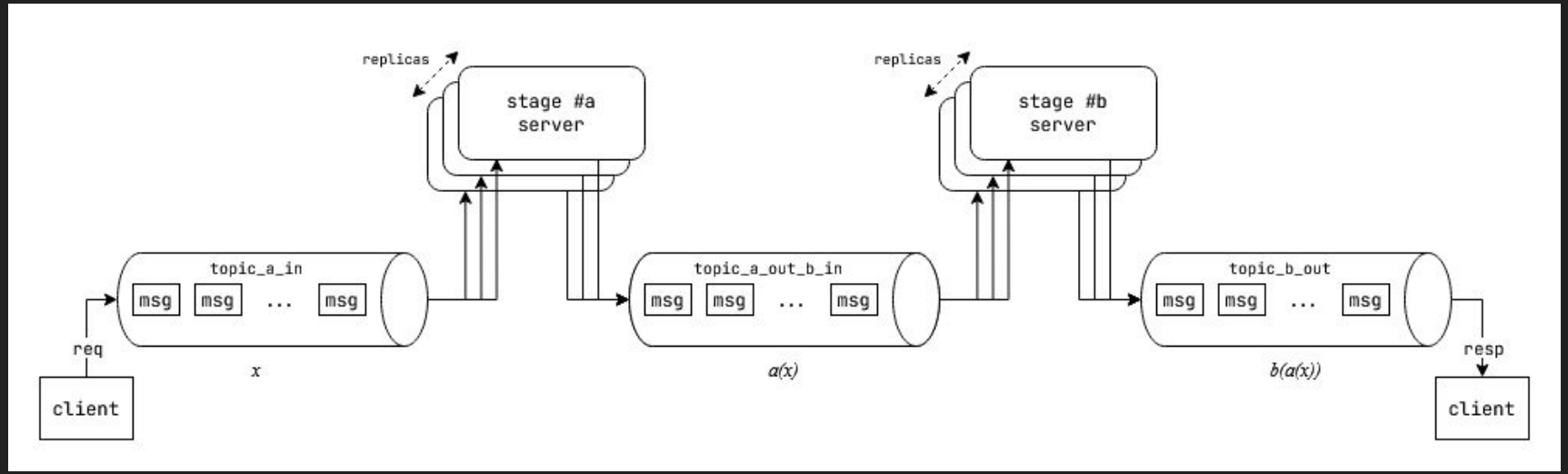
Server pull: when resources available

Queues buffer messages.

Non blocking! **No requests are lost!**

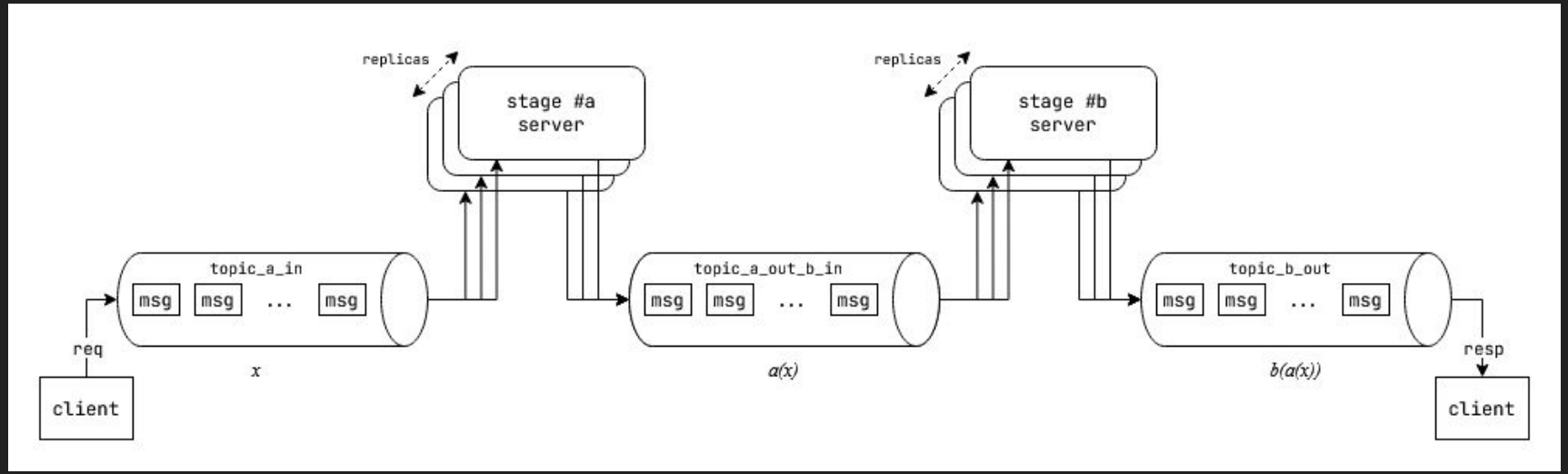


Message Queue: Multi-stage horizontally scalable asynchronous processing



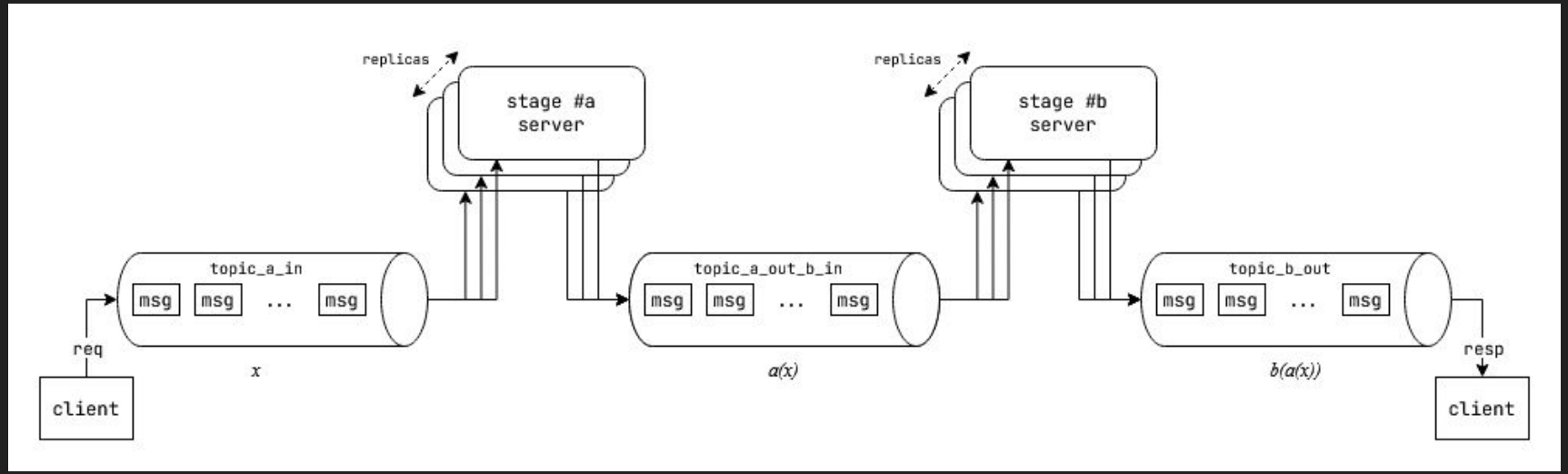

```
client -(req_msg)→ topic_a_in
```

Message Queue: Multi-stage horizontally scalable asynchronous processing



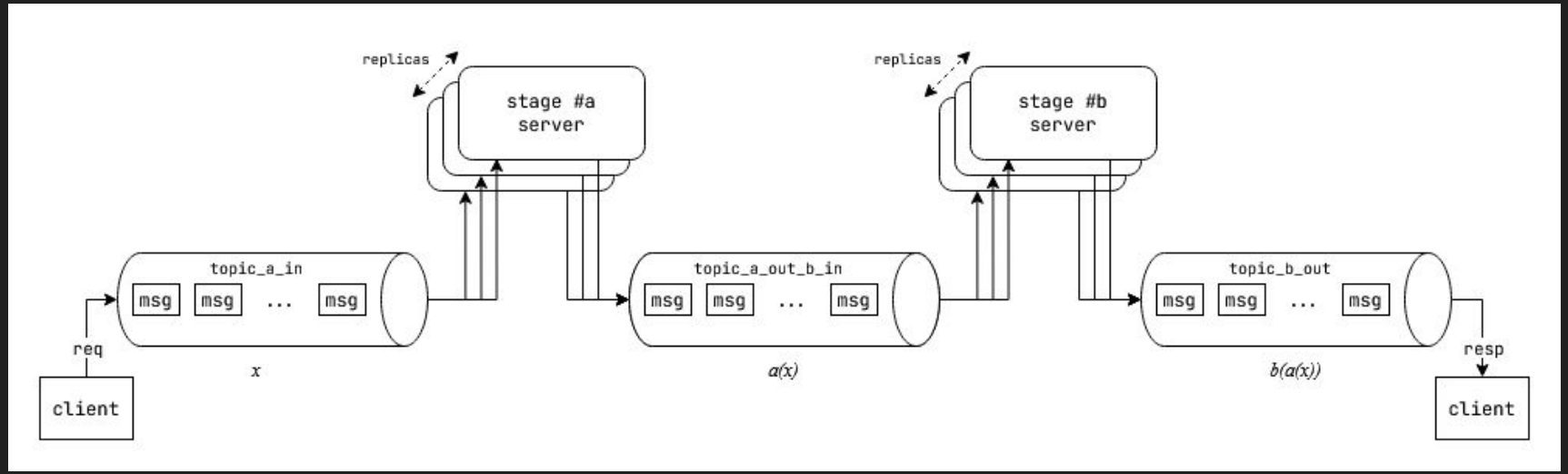
[stage #a] server ←(req_msg)- topic_a_in

Message Queue: Multi-stage horizontally scalable asynchronous processing



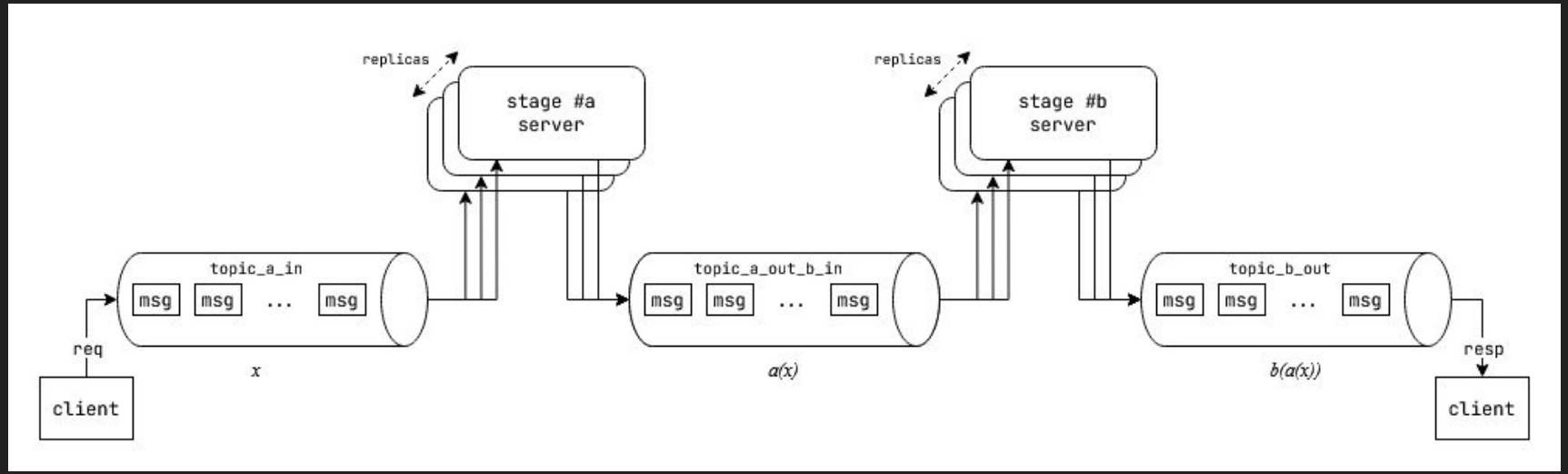

```
[stage #a] server ::process(req_msg)
```

Message Queue: Multi-stage horizontally scalable asynchronous processing



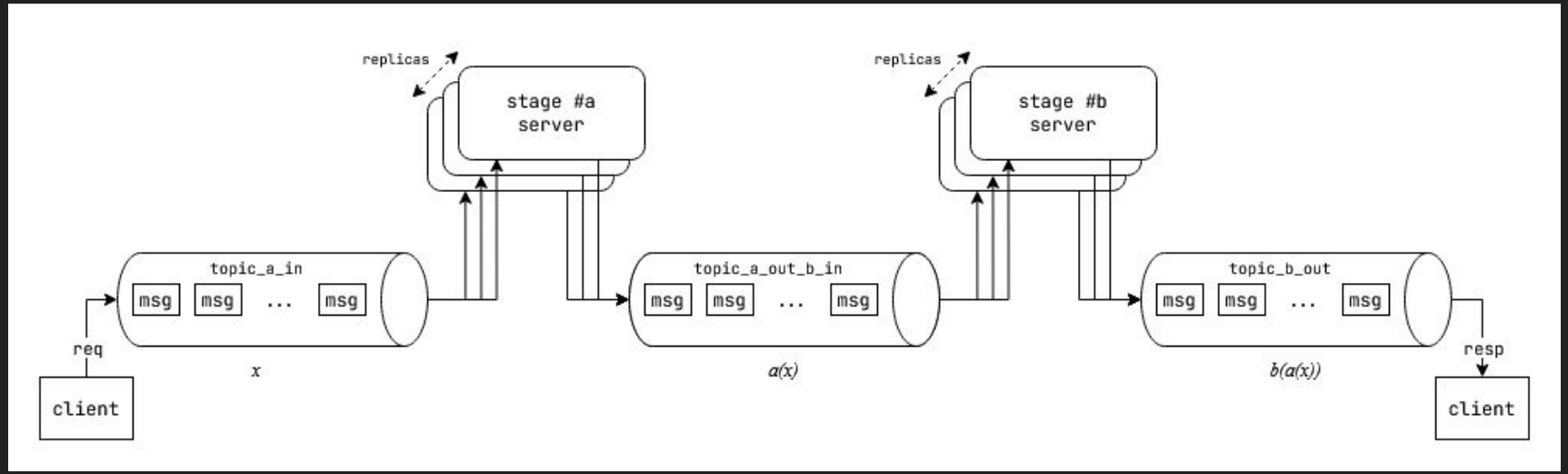
[stage #a] server -(a_out_msg)→ topic_a_out_b_in

Message Queue: Multi-stage horizontally scalable asynchronous processing



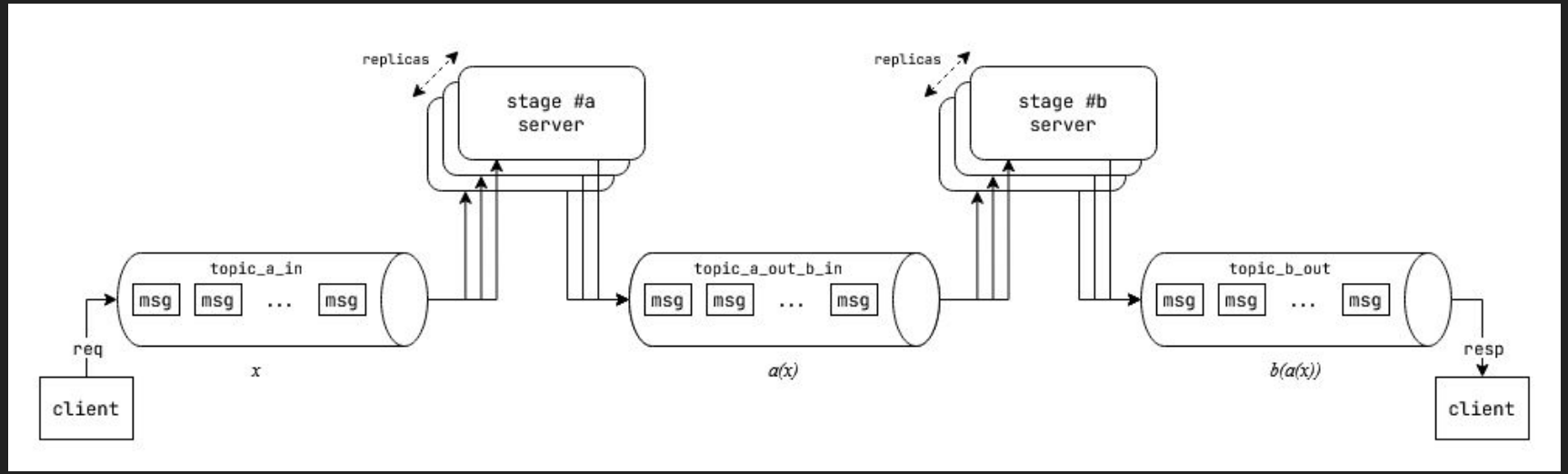
[stage #b] server ←(a_out_msg)- topic_a_out_b_in

Message Queue: Multi-stage horizontally scalable asynchronous processing



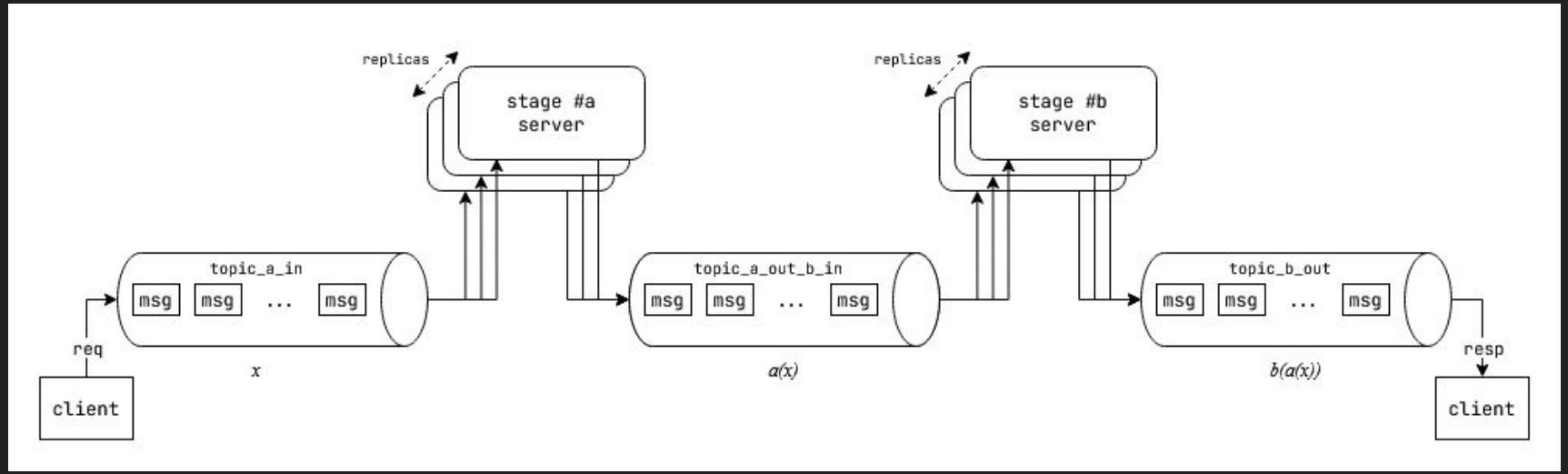

```
[stage #b] server ::process(a_out_msg)
```

Message Queue: Multi-stage horizontally scalable asynchronous processing



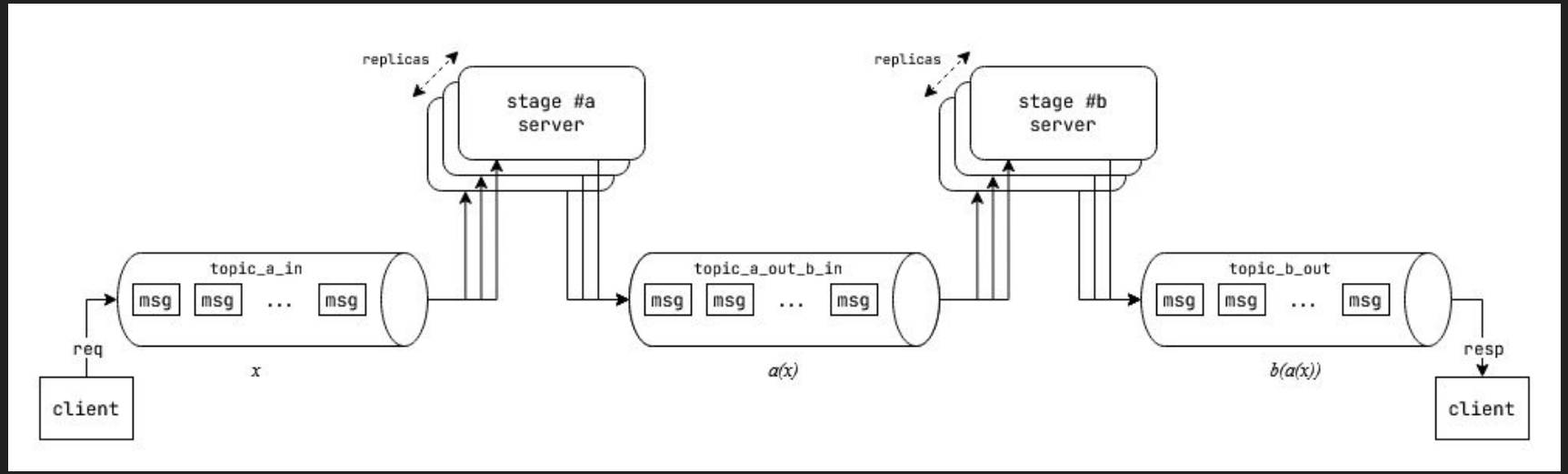
[stage #b] server -(b_out_msg)→ topic_b_out

Message Queue: Multi-stage horizontally scalable asynchronous processing



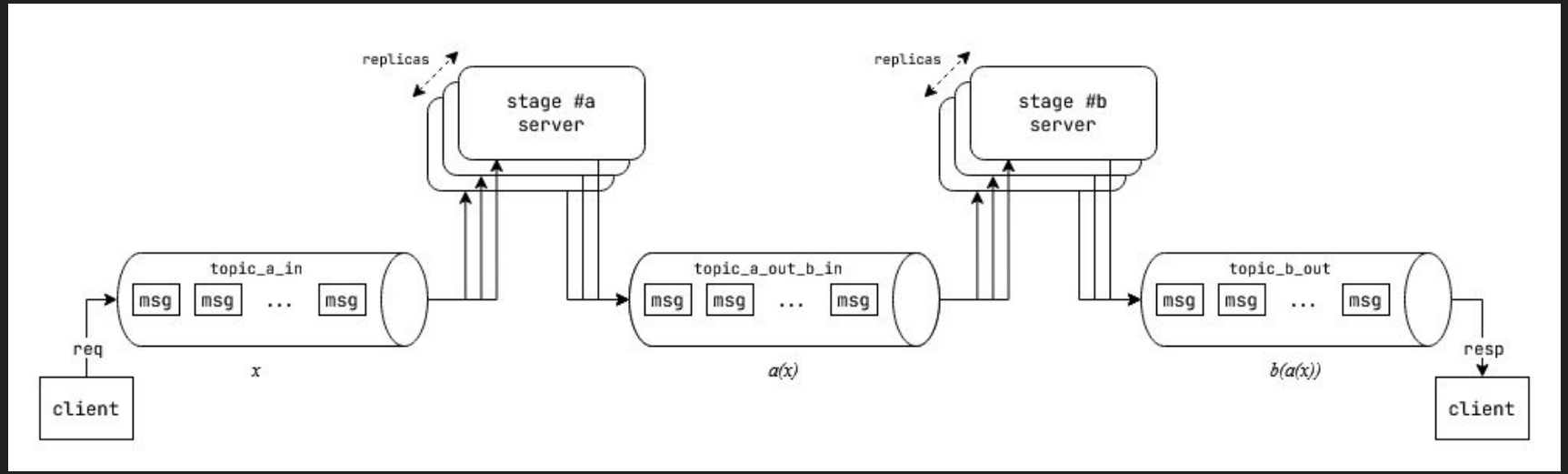
client ←(b_out_msg)- topic_b_out

Message Queue: Multi-stage horizontally scalable asynchronous processing



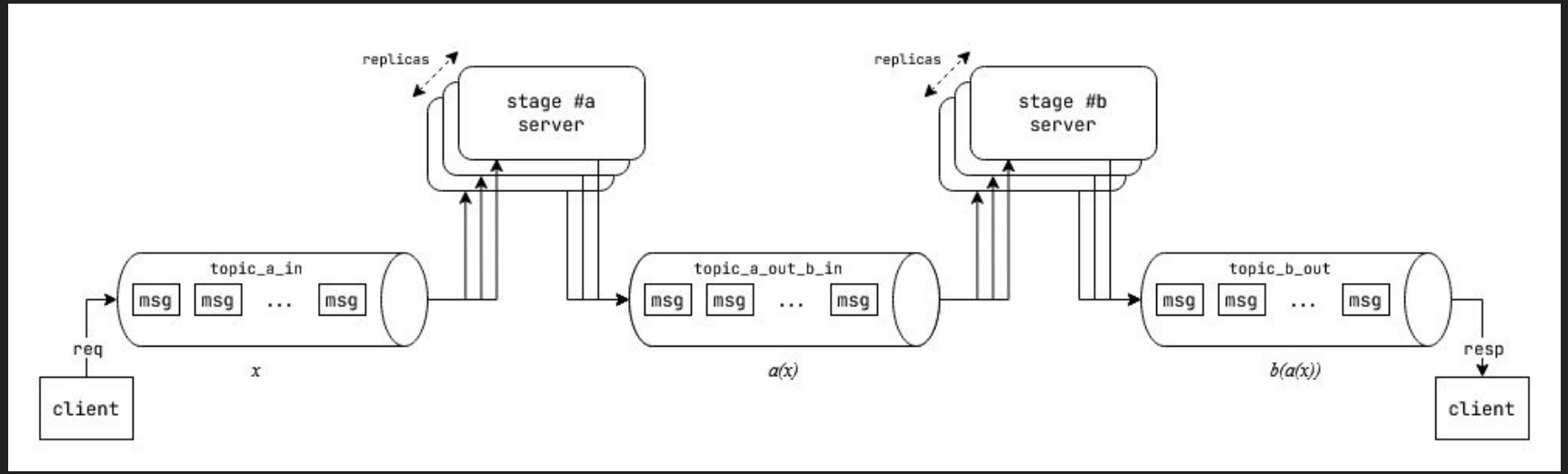
`[stage #a] :=` $a(x)$`; [stage #b] :=` $b(x)$

Message Queue: Multi-stage horizontally scalable asynchronous processing



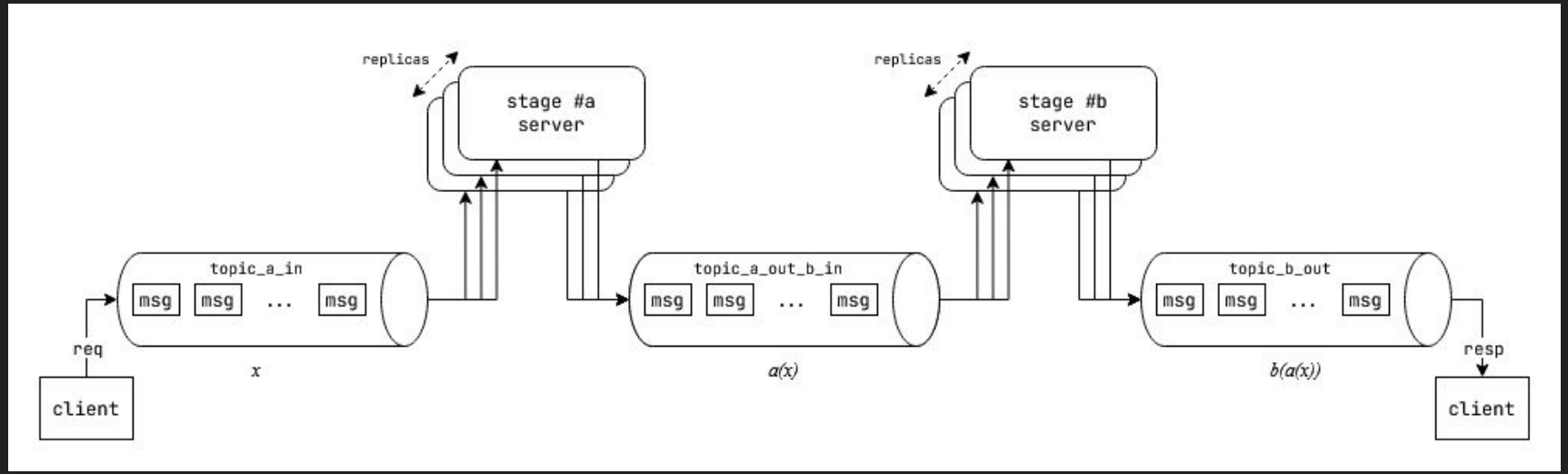
$x \rightarrow a(x) \rightarrow b(a(x))$ | x : client input message

Message Queue: Multi-stage horizontally scalable asynchronous processing



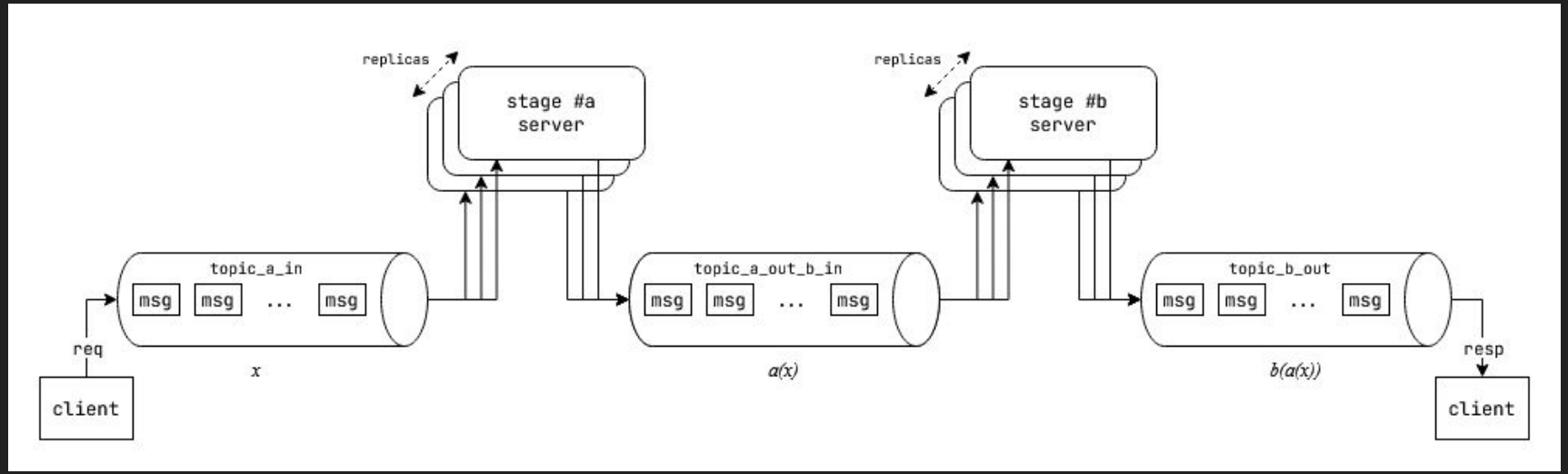
Difference: Queue of input elements $X = \{x_0, x_1, ...\}$

Message Queue: Multi-stage horizontally scalable asynchronous processing



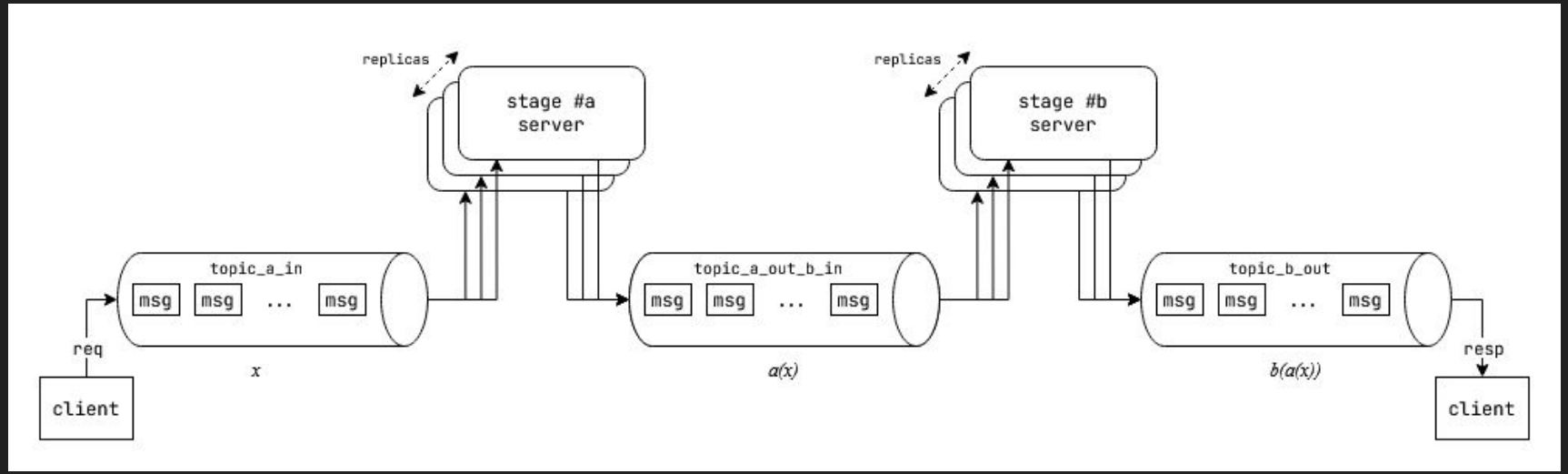
Difference: Parameters are passed to “function”(s) over the internet

Message Queue: Multi-stage horizontally scalable asynchronous processing



Difference: Parallelism at every stage with stage server replicas

Message Queue: Multi-stage horizontally scalable asynchronous processing

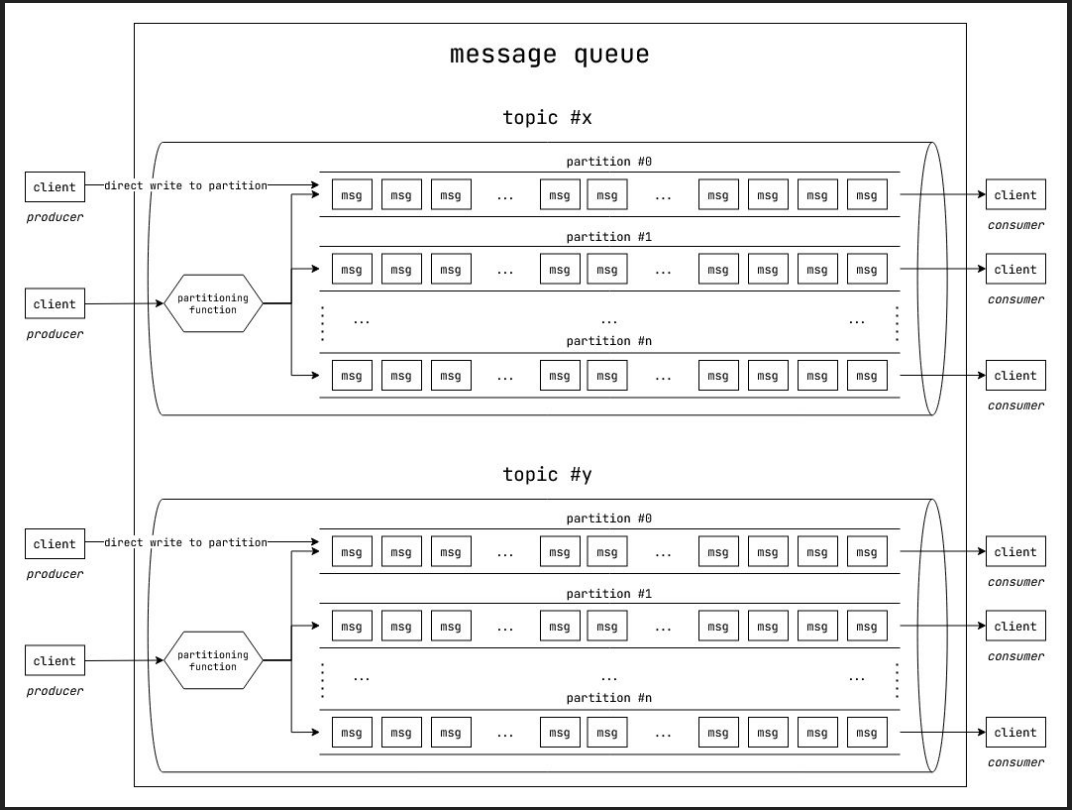


Difference: Load balancing at every stage by message queue

MQ: Load balancing / Horizontal scaling in practice

Apache Kafka: Topics and partitions

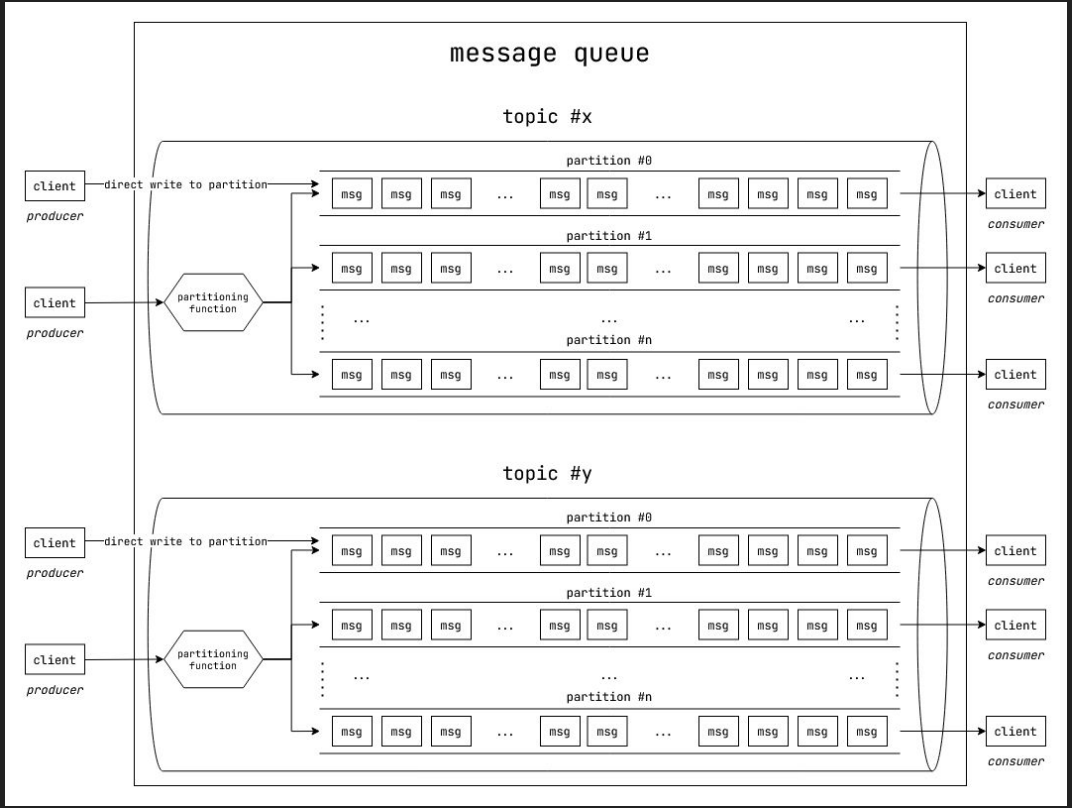
```
[message_queue]
├── topic_x
│   ├── partition_0
│   ├── partition_1
│   └── partition_2
├── topic_y
│   └── partition_1
└── topic_z
    └── ...
```



Apache Kafka: Topics and partitions

Two types of clients:

- Producers: produces / writes to partitions.
  - Direct Write → partition
  - Load balanced write with topic partitioner (Round robin, Hash based etc.)
- Consumers: consumes / reads from partitions.



Apache Kafka: Consumer Group

In Apache Kafka, consumers are part of consumer groups.

When multiple consumers subscribe to the same topic and are part of the same consumer group, each consumer receives messages from a different subset of partitions.

Different consumer groups read messages independently of each other.

RULE: A single partition can be read only by a single consumer in particular consumer group.

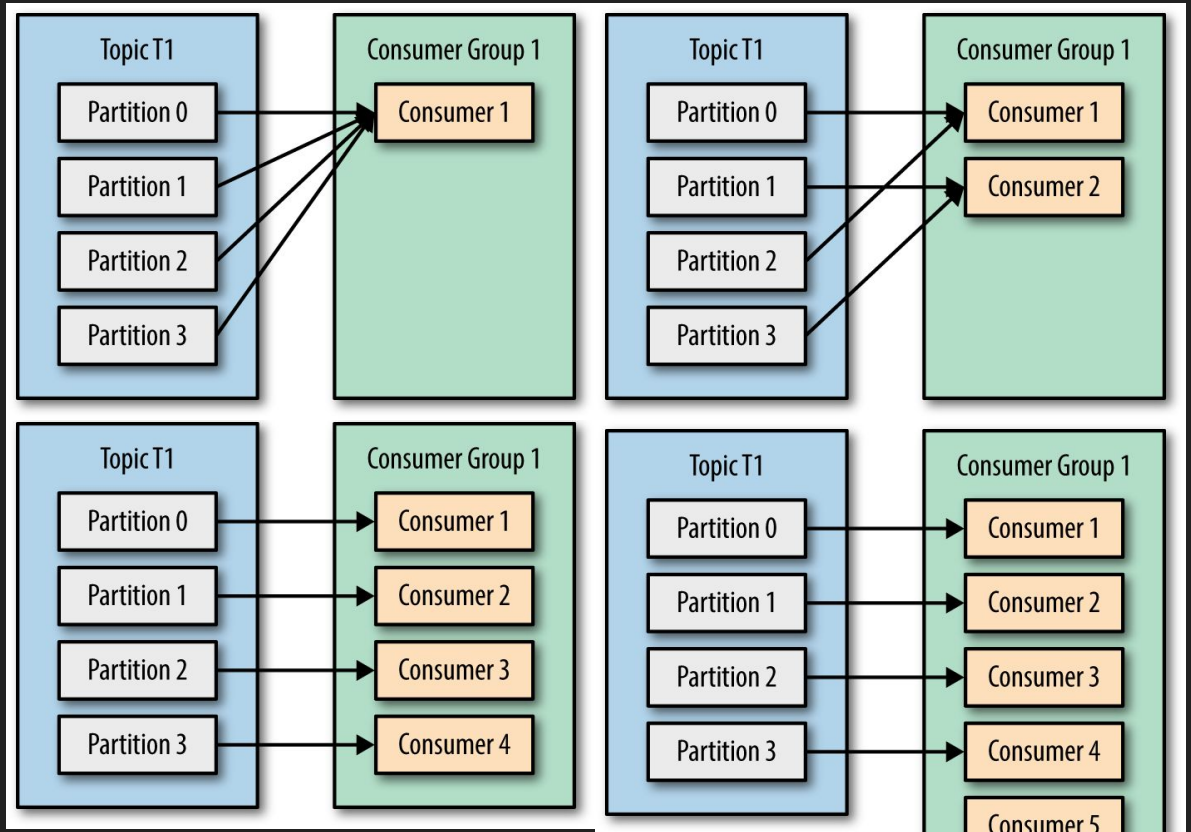


Fig Credit: [Kafka The Definitive Guide]

Apache Kafka: Consumer Group

In Apache Kafka, consumers are part of consumer groups.

When multiple consumers subscribe to the same topic and are part of the same consumer group, each consumer receives messages from a different subset of partitions.

Different consumer groups read messages independently of each other.

RULE: A single partition can be read only by a single consumer in particular consumer group.

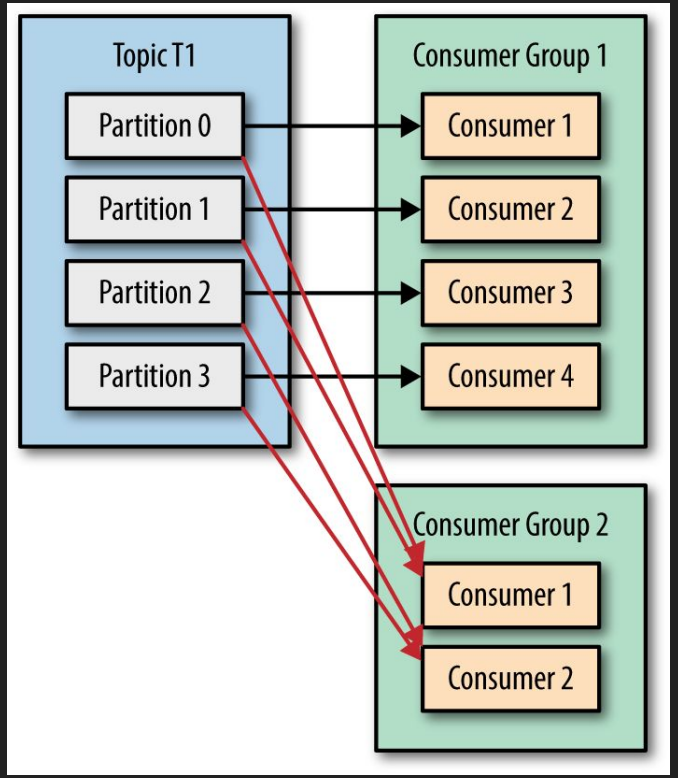


Fig Credit: Kafka The Definitive Guide

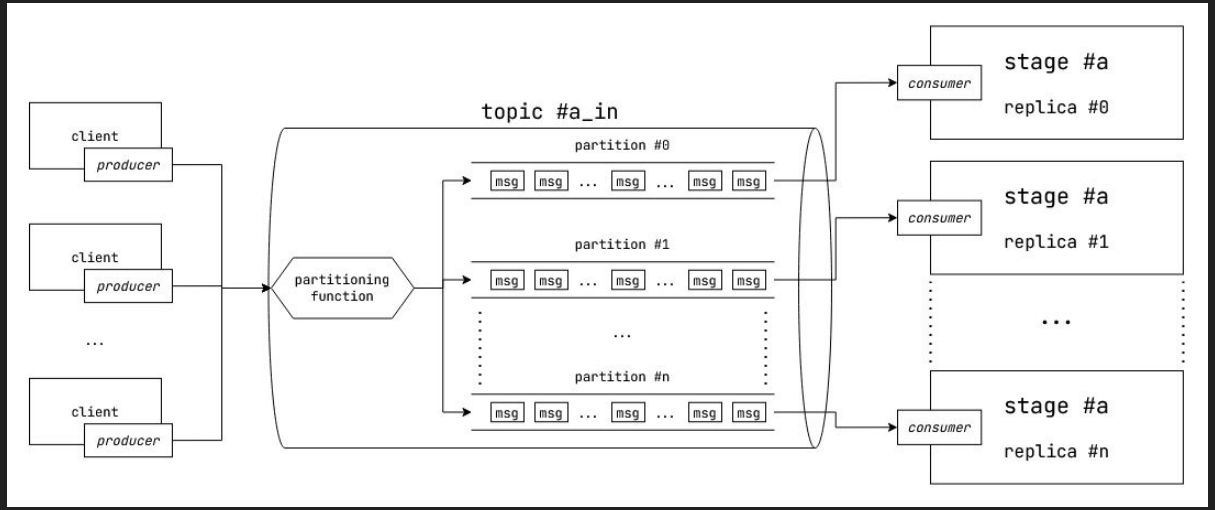
Apache Kafka: Horizontal Scaling

Multiple producer clients produce to the topic. The messages are load balanced to different partitions with the topic partitioner.

A stage server has multiple replicas.

Each stage replica has a consumer. All the stage replica consumers subscribe to same topic and are part of the same consumer group.

Each replica receives messages from a subset of partitions.



Primary distinction between synchronous and asynchronous processing

Synchronous Processing: Requests are delivered to server as soon as they arrive. Enough resources? Handle. No? Delay or drop.

Asynchronous Processing: Server pulls in a request when it has capacity to handle request. Until then requests are buffered in queue.

(Asynchronous architectures like this are also called event driven architectures where individual messages are treated as events.)

Message Queue

Advantages over Database

Reasons to choose message queues for asynchronous processing over databases.

Why use message queues at all? Databases seem fine!

- Topics can be represented with tables
- Rows can be messages
- Partitioning with partitioning function over row primary key

This is indeed possible. Used in production in some places.

However, message queues have some fundamental advantages:

- Speed
- Resilience
- Ease of use

… but how? Let’s find out!

Advantages due to different storage data-structures

Message queues rely on `segmented_log(s)` while conventional relational databases rely on data-structures from the B-Tree family.

Let’s briefly discuss the `segmented_log` data structure to understand its performance characteristics.

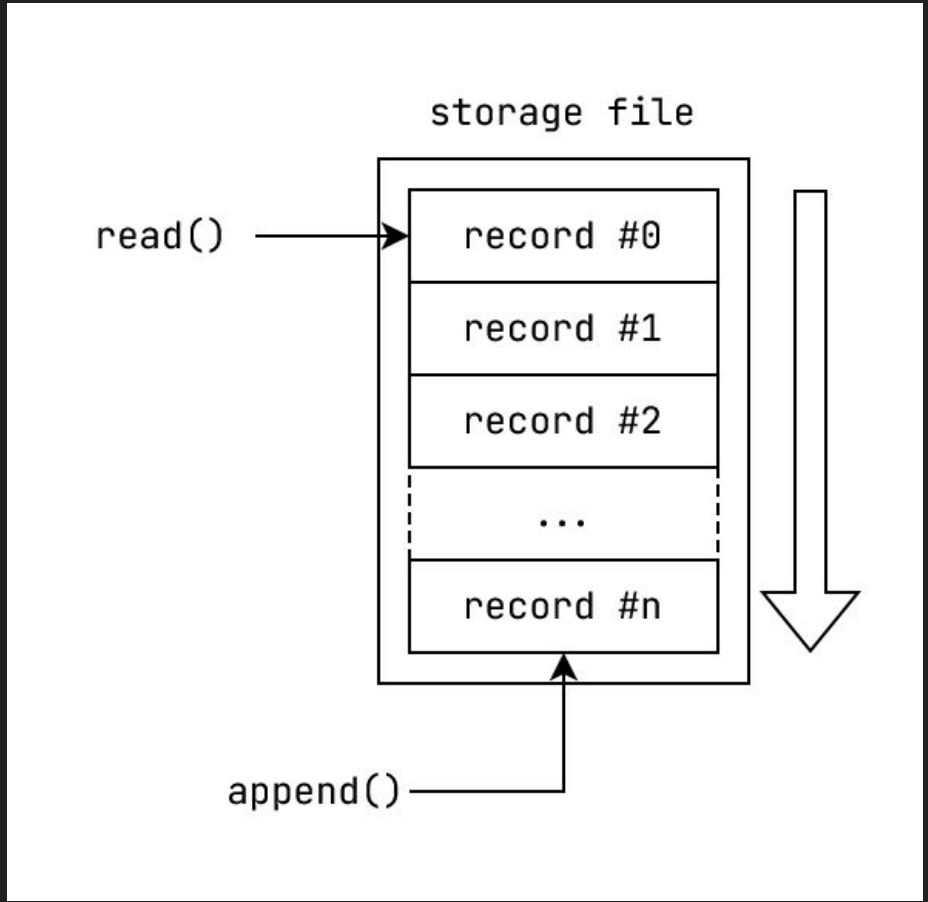
`segmented_log`: An introduction

What is the simplest data structure for representing a sequence of elements?

An array.

However, we need persistence. So let's use a file instead. We sequentially append to the end of the file, and read from the start to implement a queue.

(You can quite literally map a file to a contiguous region of memory in the process address space with the `mmap()` system call on Linux)



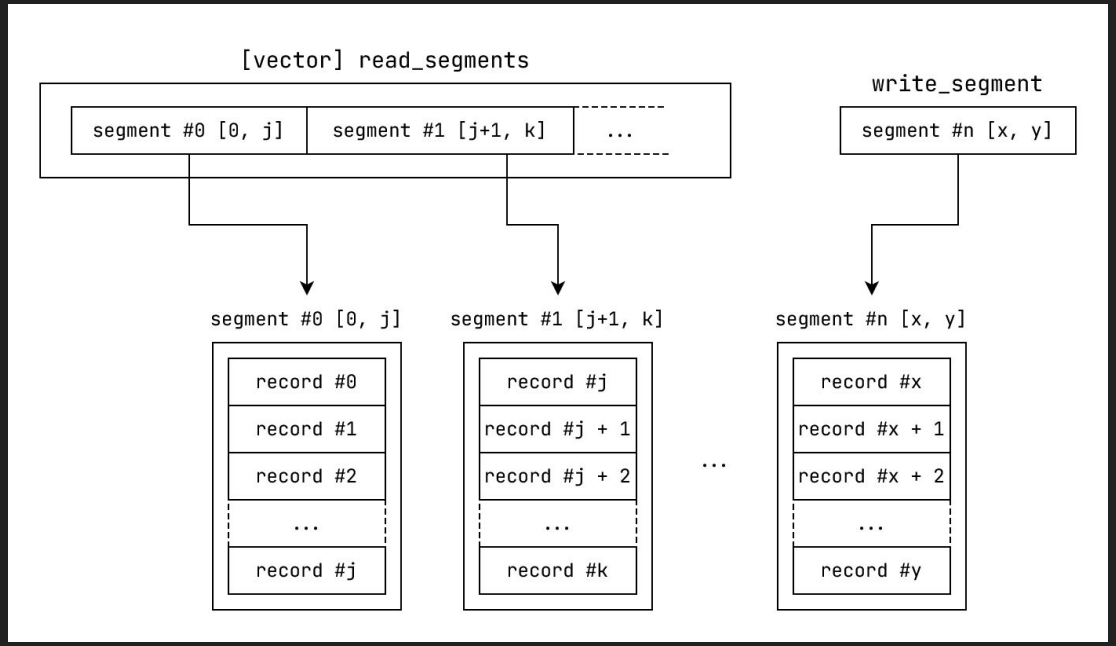
`segmented_log`: An introduction (con.)

However problems with a single large file:

- Difficult to move and copy
- Few bad disc sectors → whole file unrecoverable

Solution:

- Split index range into smaller files called `segment(s)`
- Maintain an entry for each segment with index range and file handle
- Maintain the segment entries sorted by index
- Last segment entry: `write_segment`
- Remaining segment entities: `[vector] read_segments`



`segmented_log`: Reads and Writes

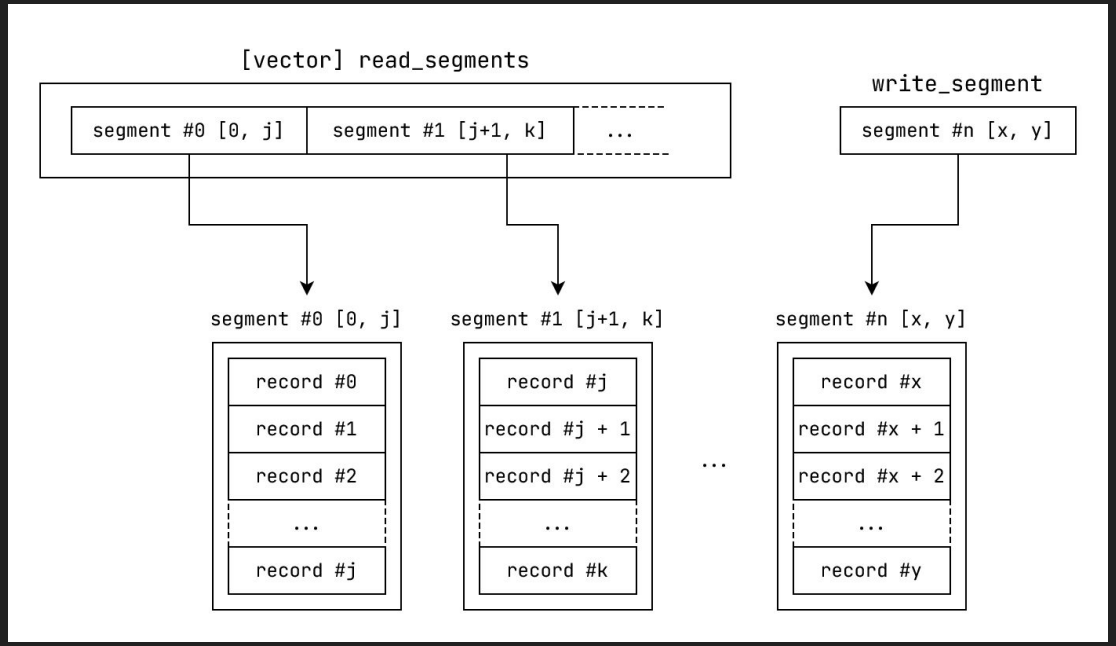
All writes are appended to the `write_segment`.

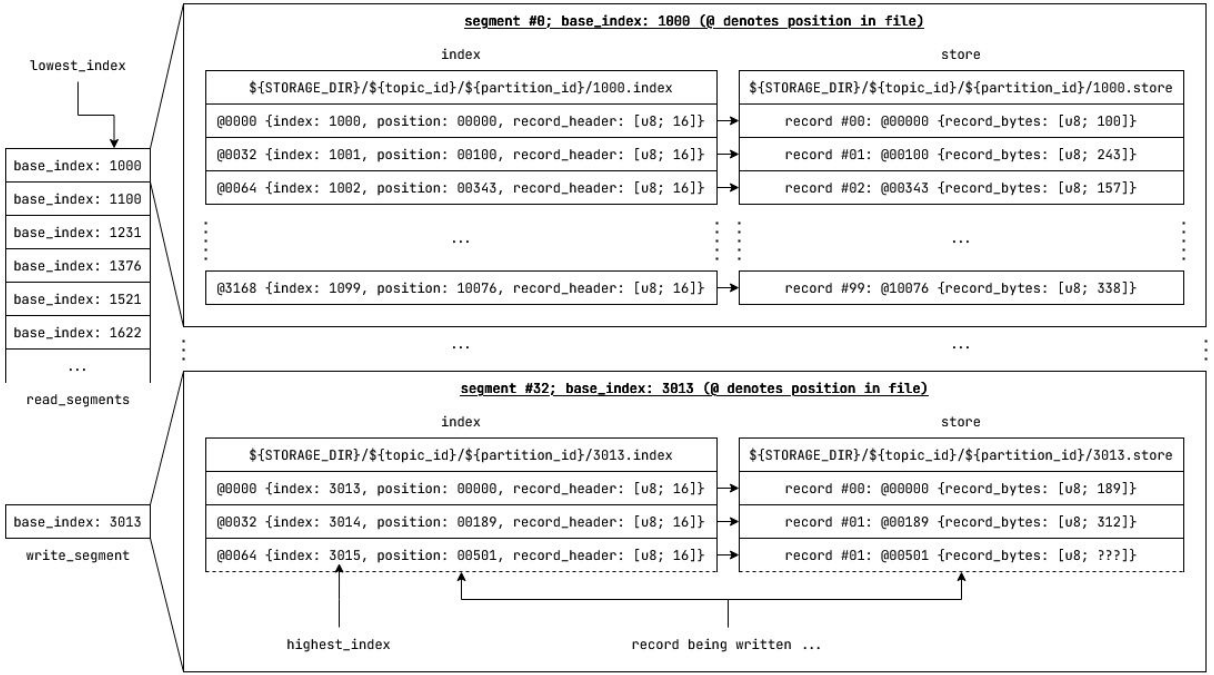
If `write_segment` size crosses a threshold:

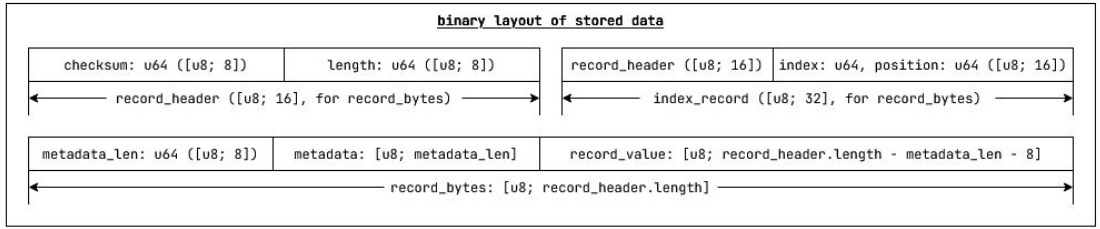
- Close the old `write_segment`
- Reopen it as a read segment
- Push it back to the `[vector] read_segments`
- Create a new `write_segment`

For reading a record with a particular index:

- First, locate the segment containing the record from the vector of read_segment or if not found the write_segment
- Read the record at the given index from the segment




segment #0; base_index: 1000 (@ denotes position in file)
index
store
lowest_index
base_index: 1000
base_index: 1100
base_index: 1231
base_index: 1376
base_index: 1521
base_index: 1622
...
read_segments
${STORAGE_DIR}/${topic_id}/${partition_id}/1000.index
${STORAGE_DIR}/${topic_id}/${partition_id}/1000.store
@0000 {index: 1000, position: 00000, record_header: [u8; 16]}
record #00: @00000 {record_bytes: [u8; 100]}
@0032 {index: 1001, position: 00100, record_header: [u8; 16]}
record #01: @00100 {record_bytes: [u8; 243]}
@0064 {index: 1002, position: 00343, record_header: [u8; 16]}
record #02: @00343 {record_bytes: [u8; 157]}
@3168 {index: 1099, position: 10076, record_header: [u8; 16]}
record #99: @10076 {record_bytes: [u8; 338]}
segment #32; base_index: 3013 (@ denotes position in file)
index
store
base_index: 3013
write_segment
${STORAGE_DIR}/${topic_id}/${partition_id}/3013.index
${STORAGE_DIR}/${topic_id}/${partition_id}/3013.store
@0000 {index: 3013, position: 00000, record_header: [u8; 16]}
record #00: @00000 {record_bytes: [u8; 189]}
@0032 {index: 3014, position: 00189, record_header: [u8; 16]}
record #01: @00189 {record_bytes: [u8; 312]}
@0064 {index: 3015, position: 00501, record_header: [u8; 16]}
record #01: @00501 {record_bytes: [u8; ???]}
highest_index
record being written ...


binary layout of stored data
checksum: u64 ([u8; 8])
length: u64 ([u8; 8])
record_header ([u8; 16])
index: u64, position: u64 ([u8; 16])
record_header ([u8; 16], for record_bytes)
index_record ([u8; 32], for record_bytes)
metadata_len: u64 ([u8; 8])
metadata: [u8; metadata_len]
record_value: [u8; record_header.length - metadata_len - 8]
record_bytes: [u8; record_header.length]

`segmented_log` performance compared to `B-Tree`

- `segmented_log` writes on average: $O(1)$, `B-Tree` insertion: $O(\log n)$
  - Appending to `write_segment`: $O(1)$
  - Rotating `write_segment` to `read_segments` complexity = `vec::push_back()` complexity
    - `vec::push_back()`: $O(1)$ on average
    - Worst case (very rare): $O(len(vec))$
      - Mitigated with exponential memory reservation
      - Multiple disjoint memory region allocation, new elements pushed to last region. Reduces worst case to $O(regions) \sim O(1)$
  - Average write complexity: write $O(1)$ for append + $O(1)$ for `write_segment` rotation = $O(1)$

`segmented_log` performance compared to `B-Tree`

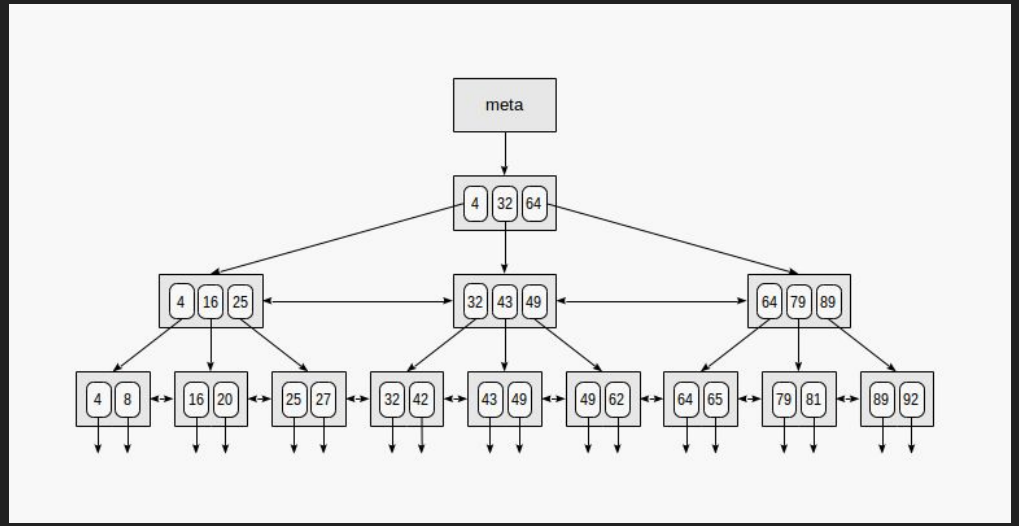
- `segmented_log`: guaranteed: *O(log len(segments))*, `B-Tree`: *O(log n)*
  - `n / len(segments) ⩾ 1000`

    `⟹ log(n) - log(len(segments)) ⩾ 3 * log(10)`

  - Reading from a single segment: *O(1)*
  - Locating segment for reading: binary search (possible because segments sorted by indices) *O(log len(segments))*
  - Average read complexity = *O(1) + O(log len(segments)) = O(log len(segments))*
  - If we can guarantee
    - constant number of records per segment,
    - constant record size with padding,
    - Identifying segment for record in *O(1)* is possible with integer arithmetic
    - *O(1)* reads possible

`segmented_log` performance compared to `B-Tree`

- `segmented_log`: Lesser number of memory indirections for sequential iteration than `B-Tree`
  - Uses a vector, contiguous memory allocation, very cache friendly
- B-Tree has a tree like structure with pointers
  - More memory indirections
  - Increased memory page faults
  - Decreased cache locality



`segmented_log` performance compared to `B-Tree`

- `segmented_log`: Higher disk page cache locality due to sequential appends
  - Same disk page referenced many times, very disk page cache local

- `segmented_log`: optimized on SSDs due to append only nature

- **In conclusion: `segmented_log`: faster reads and faster writes than `B-Tree` on average**

Message Queues and Database Comparison (contd.)

- Modern databases using `LSM Tree` (Cassandra, DynamoDb), similar performance characteristics to `segmented_log`
- Message Queues don’t need query parsing and execution engine
- Databases rely on transactions for consistency, message queues rely on consensus algorithms
  - Distributed databases → distributed transactions
  - Two Phase commit, sophisticated transaction algorithm, blocks on failure
  - Message Queues → consensus alg. (Raft, Paxos etc.), non blocking, resilient to failure
- Explicitly designed to provide queue like semantics, easier to use

Gaps are decreasing…

- Apache Kafka provides SQL interface for streaming analytics
- Redis, In Memory Key Value store comes with message queue semantics out of the box

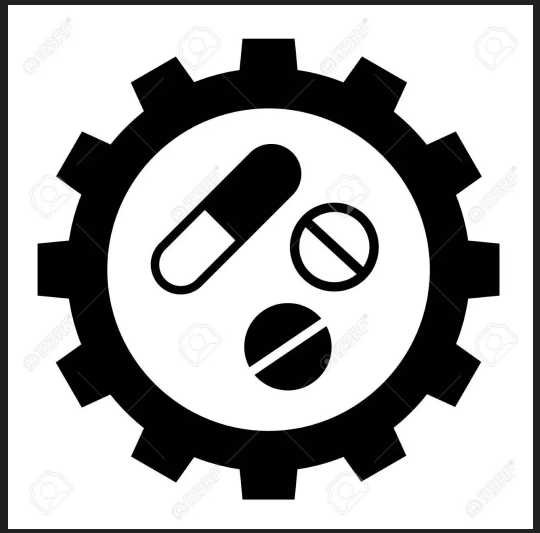
… but, where’s ML?

Plant medicine effectiveness

Case Study: Analysis of plant medicine effectiveness on crops.

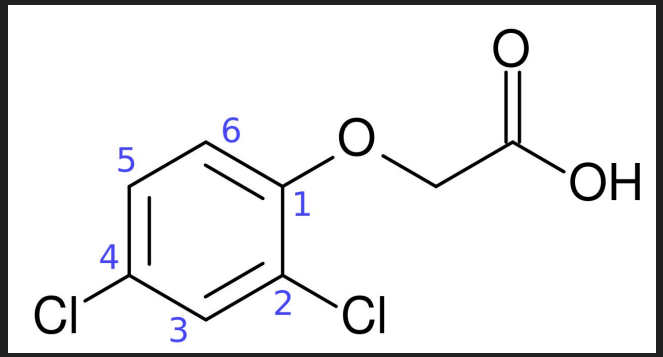
A case study depicting horizontally scalable asynchronous ML inference in action.

Case Study: Plant medicine effectiveness on Crops



Plant Pharma Company

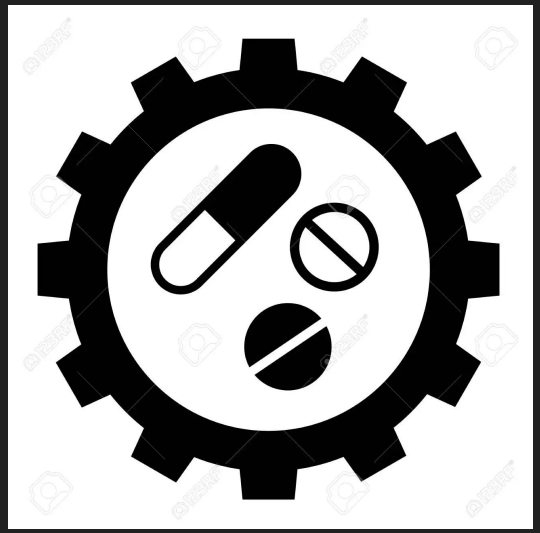
→



The new medicine

Don’t @ me, I don’t know chemistry

Case Study: Plant medicine effectiveness on Crops



Plant Pharma Company

→

Very cool and all, but… How effective is it?

We need quantitative analysis.

Case Study: Plant medicine effectiveness on Crops

Company partners with a farm to test out medicine.

Uniform medicine distribution, say: 120 ml / sqft

Farm is divided into experiment and control areas.

Medicine administered only in experiment area

Case Study: Plant medicine effectiveness on Crops

Company employs a fleet of drones to monitor plant health.

The farm is marked at various positions with GPS. Each drone has a subset of positions to go to.

Drone takes photograph of each position at some time interval (e.g Daily)

Each image has at max 20 leaves.

Case Study: Plant medicine effectiveness on Crops

… btw, the plants are in a greenhouse.

They don’t move around a lot.

The Machine Learning engineers are to analyze the photographs taken by the drones and measure the effect of the medicine on the leaves.

How do we go about solving this?

Take a look at this fungus affected plant.

Intuitively:

- Disease progresses → disease area increase
- Plant heals → disease area decreases

Area? Hmm…

Use semantic segmentation!

From a single image with multiple leaves, multiple disease types can be detected along with the area covered by each type.

… but no semantic segmentation dataset! 😭

(At least, in this scenario.)

That would have been an ideal case. However, life is seldom ideal.

The ML Engineers only have the following datasets:

- Leaf object detection dataset
- Disease grade classification for 5 diseases (the grades correspond to severity)

Solution Architecture

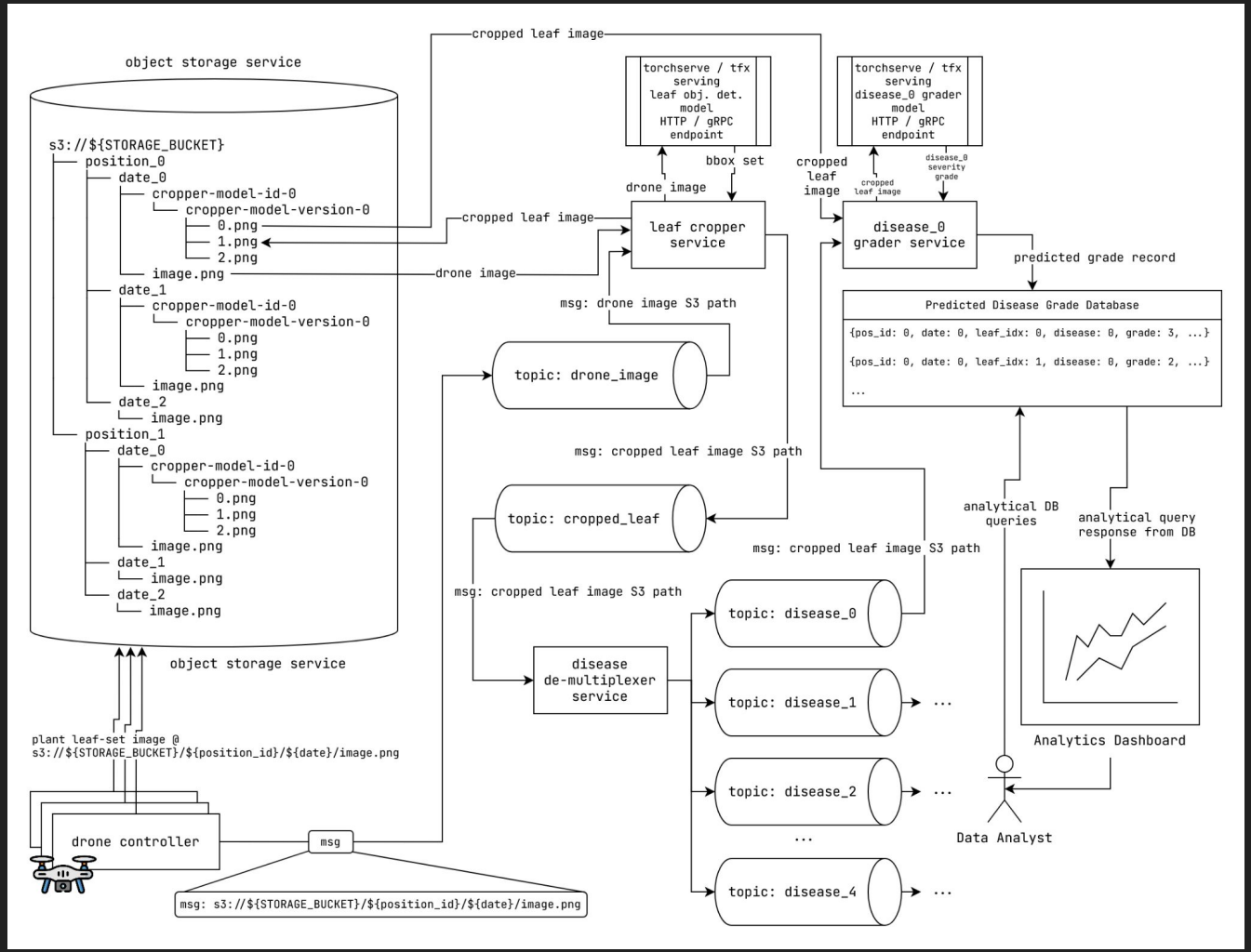
The machine learning engineers come up the following models:

- A leaf object detector
- 5 disease grade classifier models. Each disease grade classifier models has 4 classes. (0 - healthy, 4 - most severe)

Plan of action

Plant image → Leaf obj detection → Leaf bounding boxes → Leaf image Crops → Disease grade classification on individual leaf images → Store prediction in DB → Aggregate Queries for insights

The software engineers design a solution using a suite of ML inference services and drone controllers.

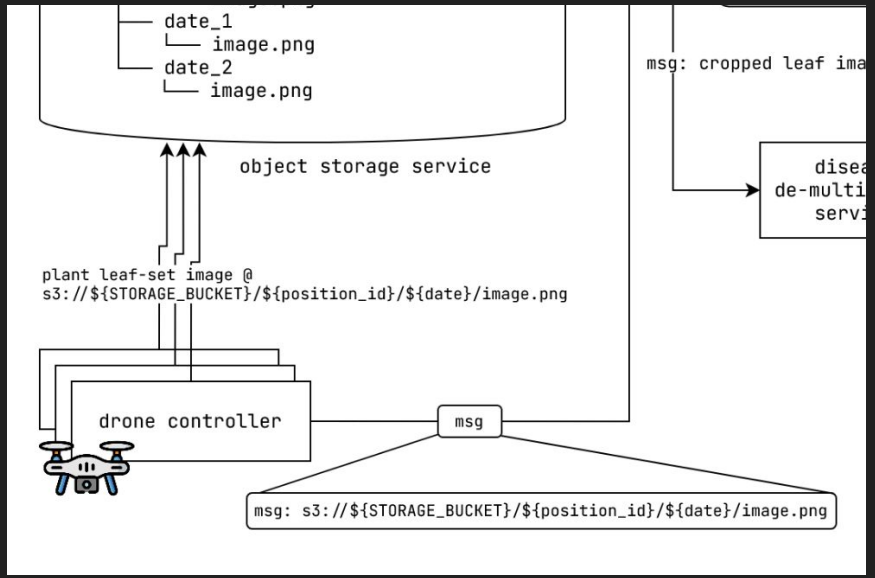

object storage service
s3://${STORAGE_BUCKET}
position_0
date_0
cropper-model-id-0
cropper-model-version-0
0.png
1.png
2.png
image.png
date_1
cropper-model-id-0
cropper-model-version-0
0.png
1.png
2.png
image.png
date_2
image.png
position_1
date_0
cropper-model-id-0
cropper-model-version-0
0.png
1.png
2.png
image.png
date_1
image.png
date_2
image.png
object storage service
plant leaf-set image @
s3://${STORAGE_BUCKET}/${position_id}/${date}/image.png
drone controller
msg
msg: s3://${STORAGE_BUCKET}/${position_id}/${date}/image.png
cropped leaf image
torchserve / tfx serving leaf obj. det. model HTTP / gRPC endpoint
bbox set
drone image
cropped leaf image
leaf cropper service
drone image
msg: drone image S3 path
topic: drone_image
cropped leaf image
torchserve / tfx serving disease_0 grader model HTTP / gRPC endpoint
cropped leaf image
disease_0 severity grade
disease_0 grader service
predicted grade record
Predicted Disease Grade Database
{pos_id: 0, date: 0, leaf_idx: 0, disease: 0, grade: 3, ...}
{pos_id: 0, date: 0, leaf_idx: 1, disease: 0, grade: 2, ...}
...
msg: cropped leaf image S3 path
topic: cropped_leaf
msg: cropped leaf image S3 path
msg: cropped leaf image S3 path
disease de-multiplexer service
topic: disease_0
topic: disease_1
topic: disease_2
topic: disease_4
analytical DB queries
analytical query response from DB
Analytics Dashboard
Data Analyst

Drone Image Collector

- Moves to specific position
- Clicks plant image
- Upload plant image to S3 bucket at specific path

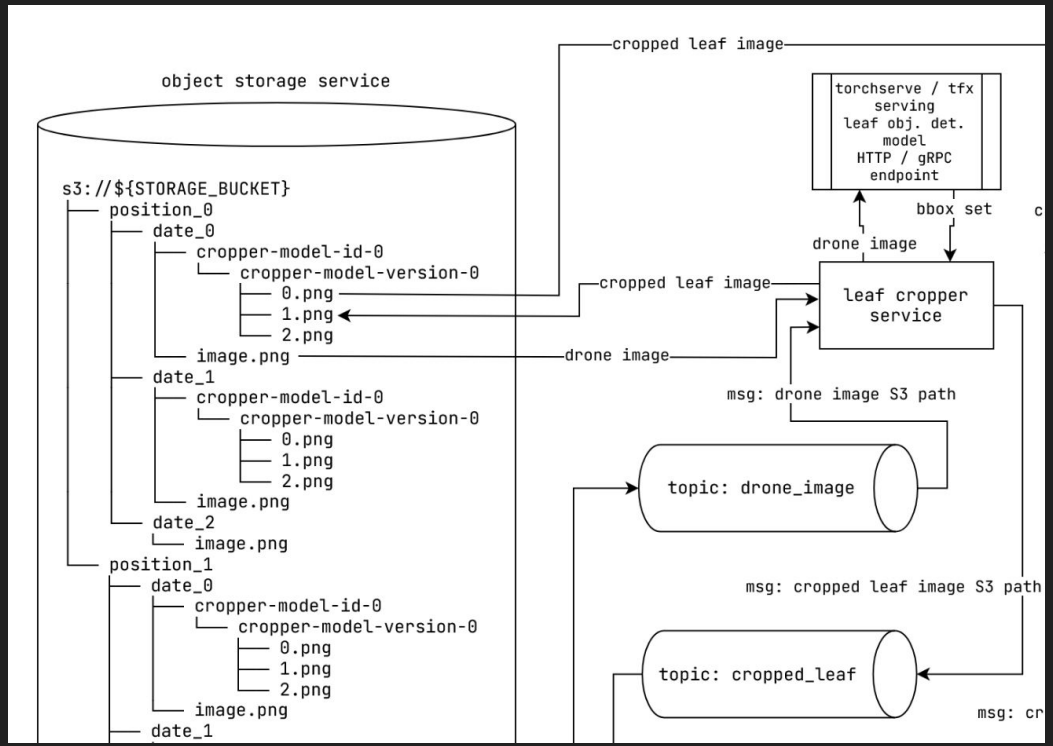
  `s3://${STORAGE_BUCKET}/${position_id}/${date}/image.png`

- Produces a message on “`drone_image`” topic, containing the S3 image path



Leaf image cropper

- Consumes a message from “drone_image” topic
- Downloads image from S3 using the S3 path in the message
- Obtains predicted bbox set with corresponding torchserve / tfx serving HTTP / gRPC endpoint for the drone image
- Uses predicted bbox set to crop out leaves
- . . .

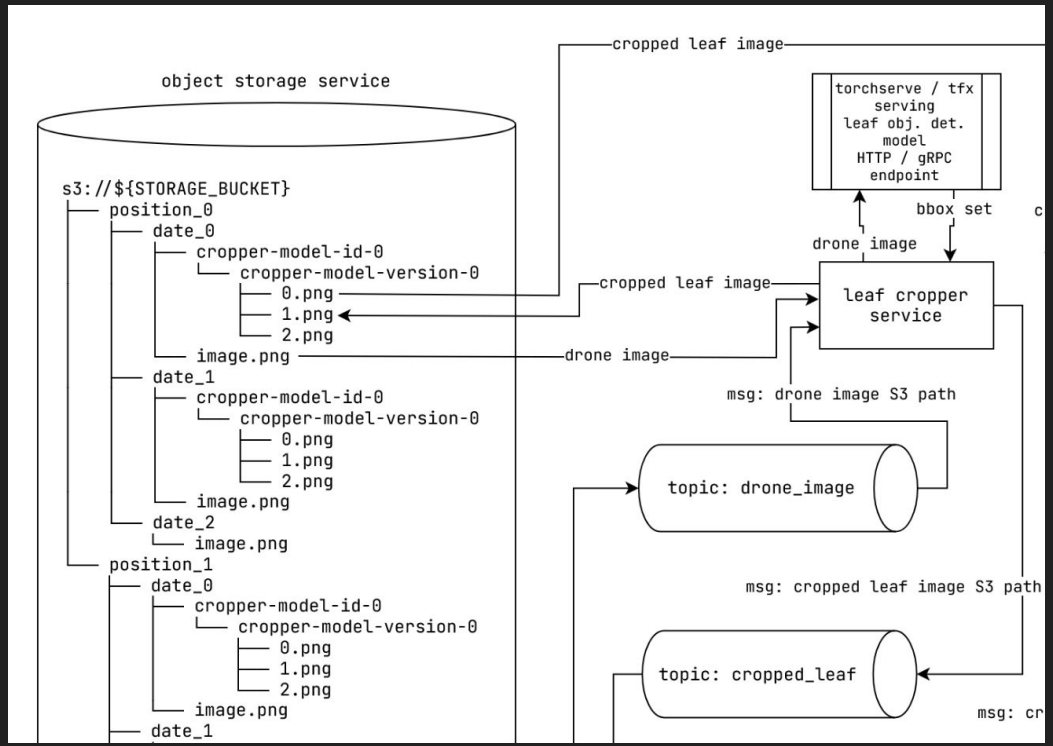


Leaf image cropper (contd.)

- ...
- For every cropped out leaf, it uploads it to S3 at a specific path:

  `s3://${STORAGE_BUCKET}/${position_id}/${date}/${cropper-model-id}/${cropper-model-version}/${leaf_crop_idx}.png`

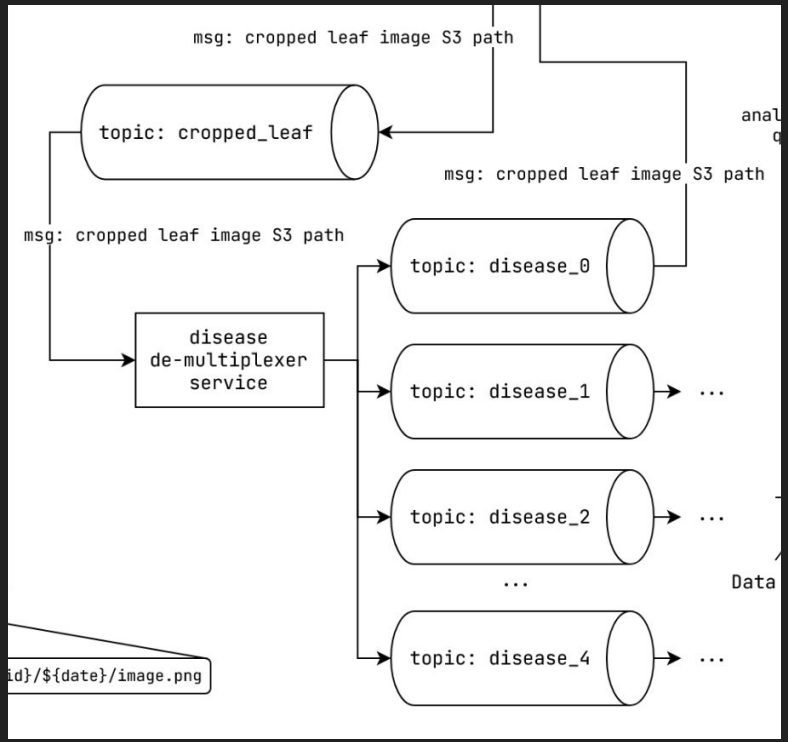
- For every cropped leaf, it produces a message on the "`cropped_leaf`" topic containing the S3 path.



Disease De-Multiplexer service

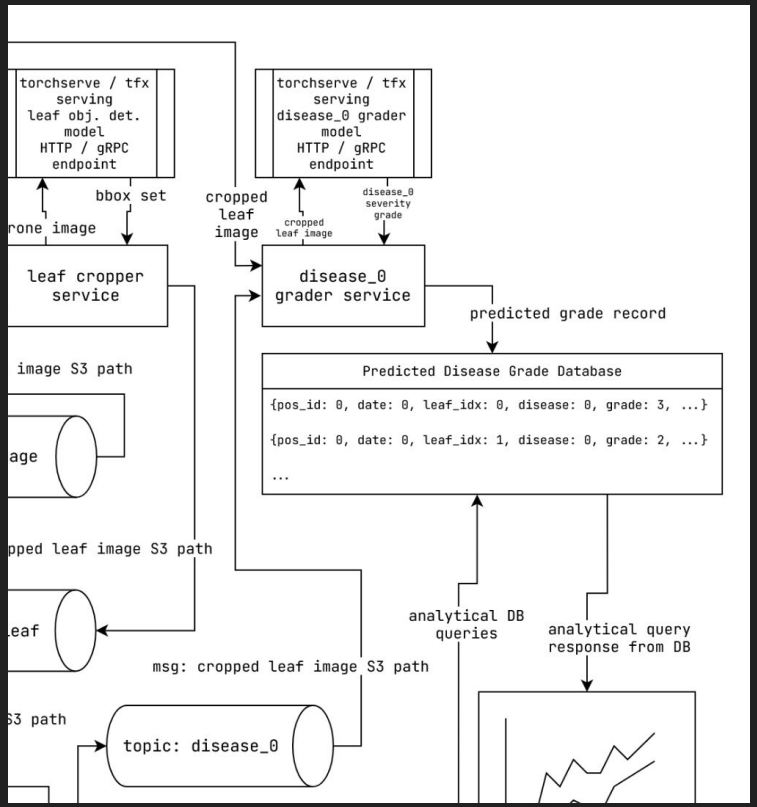
It broadcasts all the messages in the `cropped_leaf` topic to all the disease input topics.

This service might not be necessary in some message queues where they provide utilities to do this automatically.



Individual disease grader service

- Each disease has 4 grades. 0 - healthy, 4 - most severe
- Consumes message from it's dedicated input "`disease_#n`" topic
- Downloads cropped leaf image from S3 with the path in the message
- Predicts grade for cropped leaf image using tfx serving / torchserve HTTP / gRPC endpoint
- …



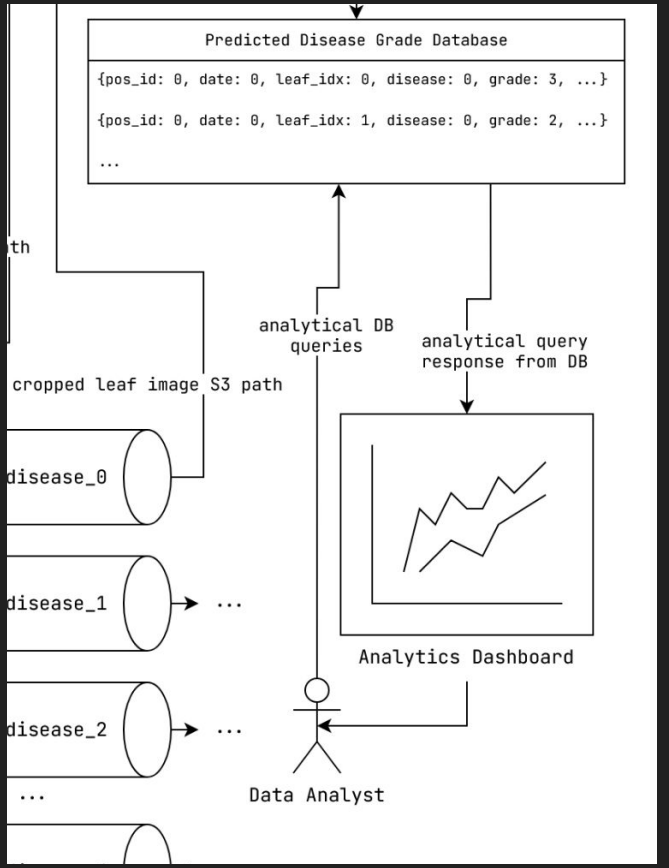
Individual disease grader service

- …
- Writes a record of prediction details to DB with the following schema:

```
{
  position_id: …, // drone position on field
  date: …, // date on which image was taken
  leaf_crop_idx: …, // leaf crop index from leaf crops with obj. det. model
  disease_type: …, // disease grader disease type
  predicted_grade: …, // predicted disease severity grade
  cropper_model_id: …, // model used for cropping
  cropper_model_version: … // version of cropper model used
  disease_model_id: …, // model used for predicting disease
  disease_model_version: …, // version of the disease predictor model used
}
```

Data Analysis

- Aggregate analytical queries on prediction data to obtain insights
- Determine whether disease severity grade decreases on average for all leaves when using medicine
- See if some diseases are unaffected
- See if there are any side effects where severity of some disease increases with medicine
- Compare with control farm area leaf images to verify effect



Audits and Reproducibility

- We keep model information with predictions to enable prediction with different models with different versions.
- To regenerate results, we simply need to regenerate messages.
- We can regenerate messages by dir walking S3 bucket. But we can do better!
- We can directly replay the messages in any topic.
  - New disease_3 model?
  - Replay messages in disease_3 input topic.
  - Only disease_3 service runs!
  - The remaining services are unaffected!

Scaling up horizontally

- We can have multiple replicas of the leaf cropper service or the individual disease grader service.
- Each replica has a consumer → consumer group → receives messages from partition subset → load balanced
- Web services and TFX serving / Torchserve endpoints can be scaled independently!
- Multiple replicas can use same TFX serving / Torchserve deployment!
- Possible because comm. over network!

Support for multiple environments

- We don’t specify whether service instances are:
  - Containers
  - K8S pods
  - Linux processes
- As long as they communicate over a message queue, it doesn’t matter!
- High Flexibility → Multi - Cloud architecture possible!
- Can even run the entire system on a beefy laptop! (with *some* loss of throughput)
- All of this possible because the design is correct.

Conclusion

Message Queues act as a general communication layer between different cooperating web services. They enable asynchronous processing of requests, horizontal scaling at each layer and higher capacity for handling requests. These properties make them an excellent communication layer between different Machine Learning inference services.

Thank you!

Slides and transcript at: https://github.com/arindas/tfug-ml-mq-infer