Project: Youtube Summarizer Server

The Message Queue

Arnold Doray - 5 Jan 2024

Discussion

- Why downloading is problematic in a mult-user scenario
- Message Queues solution.

Downloading YT Transcripts

- All our code now (ie, our servers based on SmojoVM) are event driven.
- Event Driven = Nothing is done except in response to an external event.
- In our case, events are browser navigation to pages on ytsum server, or
- ytsum getting/sending data from/to users or visitors services.
- No code can run outside processing these events.

Example 1

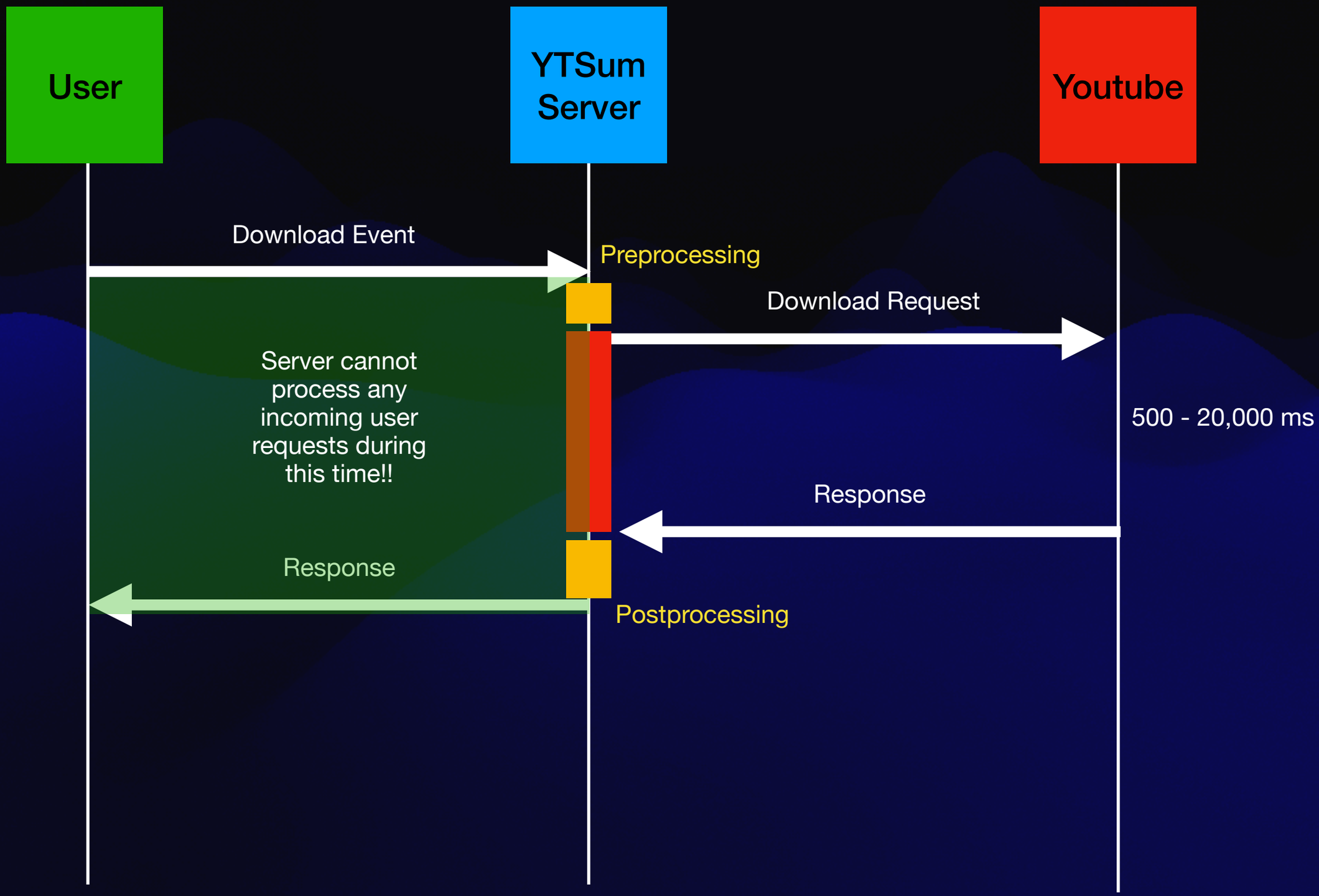
```
1  : my-processing ( -- ) ... ;
2
3
4  : main ( -- )
5      "Starting server..." . cr
6      4040 dispatch http-server
                my-processing
8  ;
```

- MY-PROCESSING is never run, since the system never exits http-server
- It is very hard/impossible to run independent code for data processing from a main that launches a server.
- For the same reason, you cannot run 2 servers in the same main

Downloading YT Transcripts Synchronously

- One way to download anything from our event-driven server is synchronously — ie, we immediately try to download when the user generates a download event.
- This is VERY BAD because it causes the server to block processing all other requests.
- Especially problematic for downloads that take time.
- Ytube may take 10-20 secs to respond to our server's download request.
- During this time, our server is blocked and will not process any other requests. BAD!

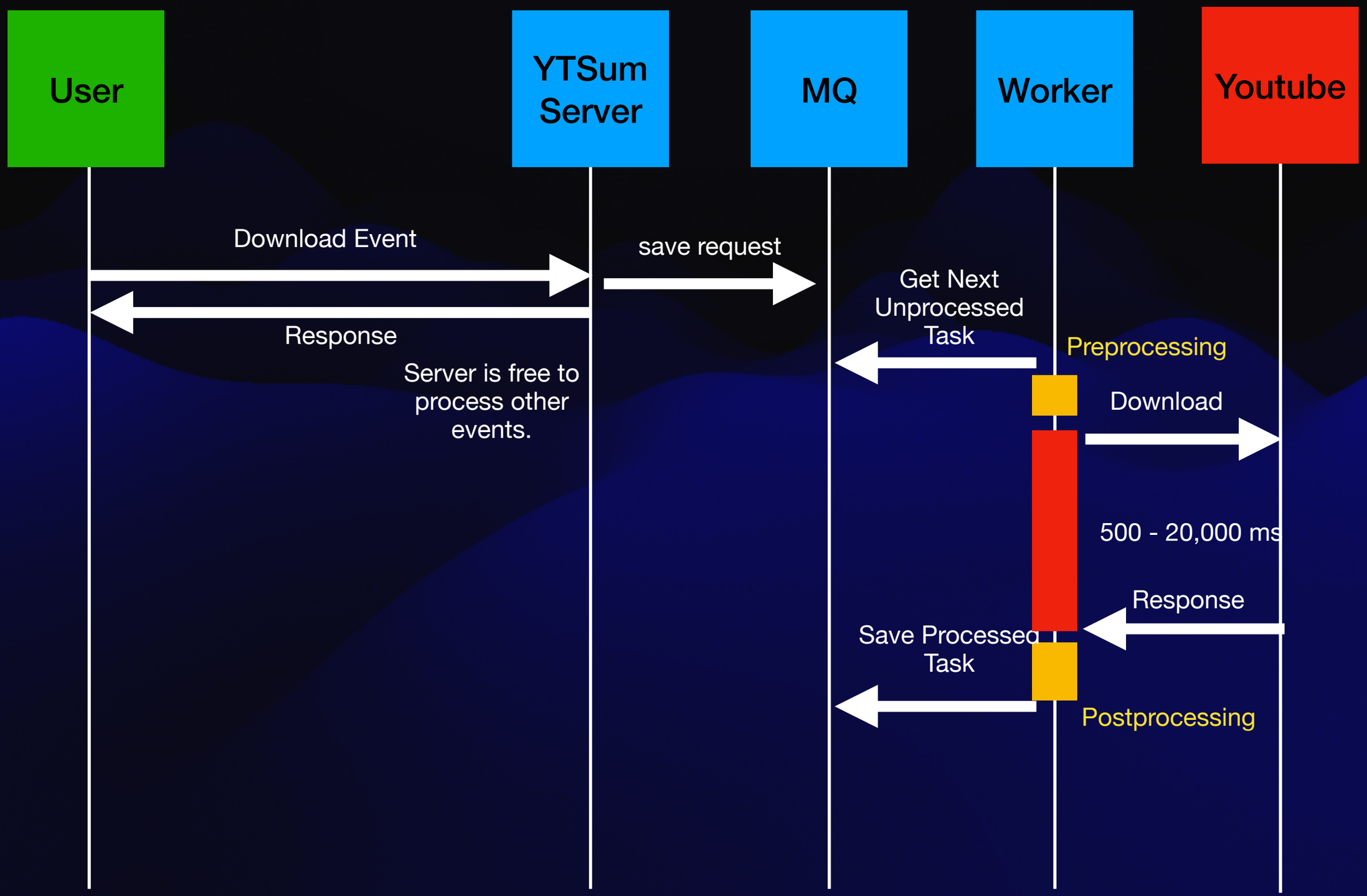
Why Synchronous Downloading is Bad



Message Queues

- A much better solution is to separate event generation from their processing.
- A Message Queue (MQ) saves events (eg requests to download)
- A Worker requests events from the MQ and processes them.
- It saves processed data back onto a different queue on the MQ, so other processes can read the processed data.
- Users can poll the system to determine status (or better, use server-side notifications to push updates to the user).

Using a Message Queue + Workers



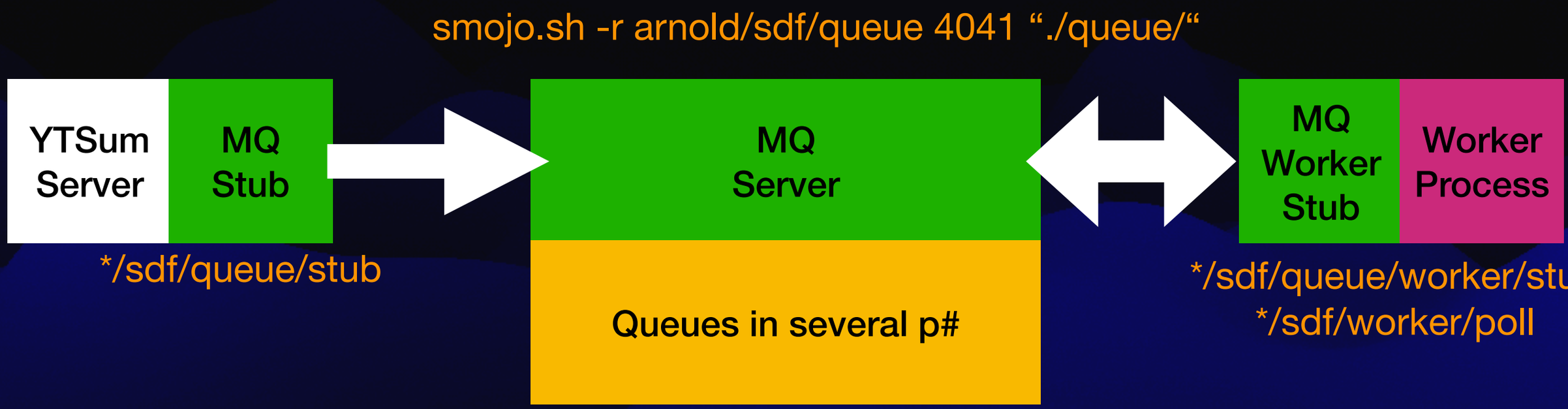
Message Queues + Workers

- We can spin up/spin down workers depending on whether there is a lot of pending tasks to process.
- We have separate queues for submitted, pending and complete tasks.
  - Submitted = just submitted by YTSum server.
  - Pending = being processed by a Worker.
  - Complete = Task is completed.
  - These queues can be implemented as a phash.

MQ Design Constraints

- MQs should be…
- data agnostic — queue operation should not require it to understand the messages being received/sent.
- We use a stub serialize and deserialize data automatically. The serialized data is a string, so MQ only handles strings.
- The MQ should allow FIFO and LIFO operations.
- MQ will operate using HTTP, which is less efficient, but easier to connect to other languages (eg curl, bash, python, etc).

MQ Design



- MQ Server:
- smojo.sh -r arnold/sdf/queue port “basepath”
- MQ Stub: require arnold/sdf/queue/stub

Message Queue Stub Words

- queue.address! (“addr” port —) sets the location of the MQ server.

- queue.store (msg “queue” cb — async)
- Stores a message. msg is any object.
- The CB must be (msg f —), where f is true if we have an error and false if no error.
- The return value into the CB (ie, msg) is the new message ID.

- queue.first (“queue” cb — async)
- Returns the first item in the queue. The item is removed from MQ. This simulates a stack (LIFO)
- queue.last (“queue” cb — async)
- Returns the last item in the queue. The item is removed from MQ. This simulates a queue (FIFO)

- queue.size (“queue” cb — async) returns the size of the queue.
- queue.move (“id” “src” “dest” —) moves an id from one queue into another.

DEMO:
Starting & Using a MQ

MQ Workers

- Workers are not servers, but an ordinary infinite loop.
- The loop polls the MQ at intervals.
- */sdf/queue/worker/stub is used to communicate with MQ
- These are similar to */sdf/queue/stub, but no need for callbacks, since workers are not servers.
- Eg,
- queue.store (msg “queue” — err true | msg false)

Poll Workers

- To write a worker, you can write an infinite loop directly (eg using BEGIN…AGAIN), or

- Use */sdf/worker/poll which contains 2 types of poll workers.
  - worker.const (xt interval-millisec —) starts a poll worker that polls at regular intervals.
  - worker.adapt (xt tmin tmax —) starts an adaptive poll worker. tmin/tmax are in millisec.

- XT needs to be written by yourself, and must be (— f) the flag is a hint to tell the poll worker process if the polling operation succeeded or not.

Example: Creating an adaptive Poll Worker

```
1  : process ( -- f )
2      "some-queue" queue.first { msg err? }
3      err? if \ error reading queue.
4          false exit
5      else
6          "Received from Queue:" . msg . cr
7          \ Process the message
8          msg null? not \ false if message was NULL
9      then
10 ;
11
12 : main ( "addr" port -- )
13     "Starting Worker..." . cr
14     int queue.address!
15     ['] process 100 20000 worker.adapt
16 ;
```

- PROCESS (— f) communicates with the MQ. Returns false if there was no data. This helps the adaptive polling algorithm to determine when to poll again.
- We started the poll worker with tmin = 100ms and tmax = 20,000ms

DEMO:
Writing an adaptive Poll worker

Homework #1

- Write a new form (called add-link) that enables the user to submit a link (any http link).
- Create a new page that displays this form. Call this page “transcripts”
- This is the main page of YTSum once the user logs in. So, ensure this page loads correctly after the user signs in.
- Use the */sdf/queue/stub and a callback to submit data entered by the user from the add-link form to a queue called ytsum.submitted.
- For now, when the form is submitted nothing is done on the UI. Ensure that this is correct.
- Write a worker process that reads the MQ and simply prints out the link.