

IIT Madras BSc Degree

Copyright and terms of use

IIT Madras is the sole owner of the content available in this portal - onlinedegree.iitm.ac.in and the content is copyrighted to IIT Madras.

- Learners may download copyrighted material for their use for the purpose of the online program only.
- Except as otherwise expressly permitted under copyright law, no use other than for the purpose of the online program is permitted.
- No copying, redistribution, retransmission, publication or exploitation, commercial or otherwise of material will be permitted without the express permission of IIT Madras.
- Learner acknowledges that he/she does not acquire any ownership rights by downloading copyrighted material.
- Learners may not modify, publish, transmit, participate in the transfer or sale, create derivative works, or in any way exploit, any of the content, in whole or in part.

Asynchronous Tasks

Web servers

- How does a web server work
- Threads and background tasks
- Long running tasks

Web server

Simplest possible HTTP server

- Open port 80 in “listen” mode - wait for incoming connections
- If incoming connection, read text, look for request
- Send back a response

Blocking connections with Flask

- Flask in “non-threaded” mode
 - <https://replit.com/@nchandra/FlaskBlocking#main.py>
- vs. Threaded mode
 - Default operation of Flask

Threaded web server

- Threaded server
 - Accept incoming request
 - Immediately start a thread to handle request
 - Go back and listen for next request
- Limitations
 - Each thread consumes resources
 - Depends on OS for handling parallel / concurrent execution
- Note: Threads are ***concurrent*** - parallelism depends on hardware

Blocking server

- Client blocks until server responds
- Can be bad for interactivity
- Need not block other clients
 - Depends on threading

Long running tasks

Example: face recognition on uploaded photos

- User uploads photos
- Server runs face detection on each photo
- Then face recognition against known database
- Alert when match found

Face recognition task problems

Blocking

- User uploads photo, but gets no response till task complete
- Cannot navigate away, do not know response

Threading

- Only one user can upload a photo at a time?
- Large photos block server for long time
- Uncontrolled thread creation drags down server performance

General problem

- Should web server directly run compute intensive tasks?
 - Or stick to handling application logic, rendering, file serving?
- Can tasks be handed off to outside servers
 - Specialized for types of compute
 - Different scaling algorithms than web
- How should web server and compute servers communicate?
 - Automatically handle scaling
 - Allow easy task distribution

Asynchronous Task Frameworks

Goals:

- User can define set of tasks
- Web server can “dispatch” tasks to be executed later
- Asynchronous completion and updating possible

When to use?

- Response to user *does not depend* on execution of task
 - Example: send email - can display a “sending” message and later update status
- Example of when NOT to use:
 - API fetch: response must be based on result of API query
 - async task will not help since you will need to block and wait for response
- Note: this is NOT the same as async on frontend!
 - Async frontend with UI reactive update is still useful
 - But the backend process should return with the correct response

Requirements

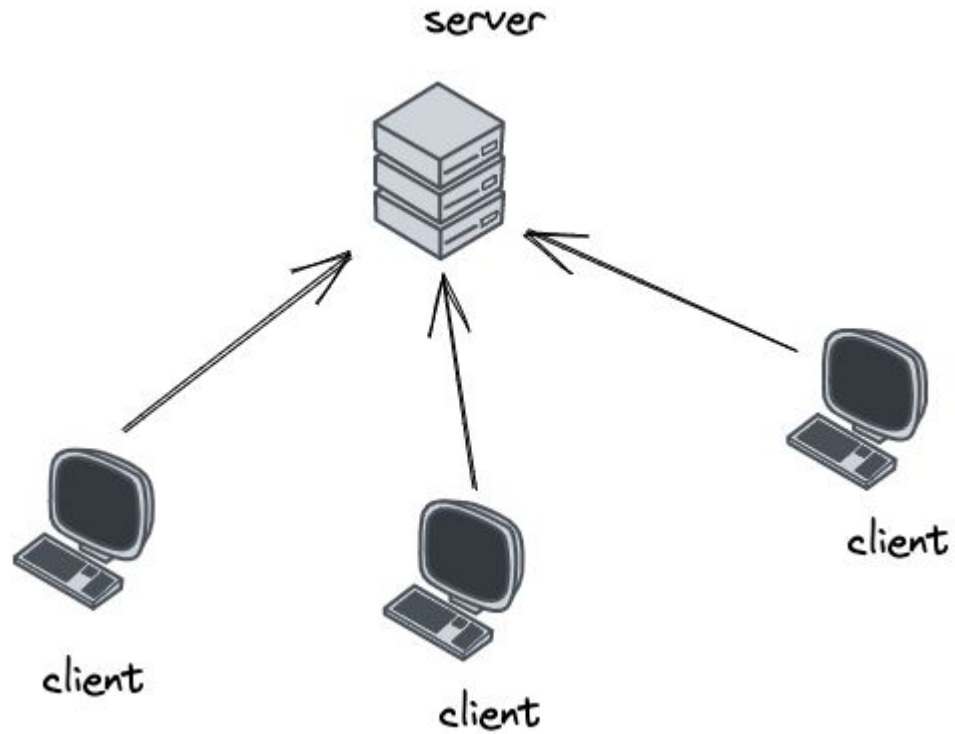
- Messaging / Communication system
 - Message queues
 - Brokers / Backends
- Execution system
 - Threads / coroutines / greenlets ...
 - concurrent models
 - Can be in another language, runtime, ...

Example: Celery for Python

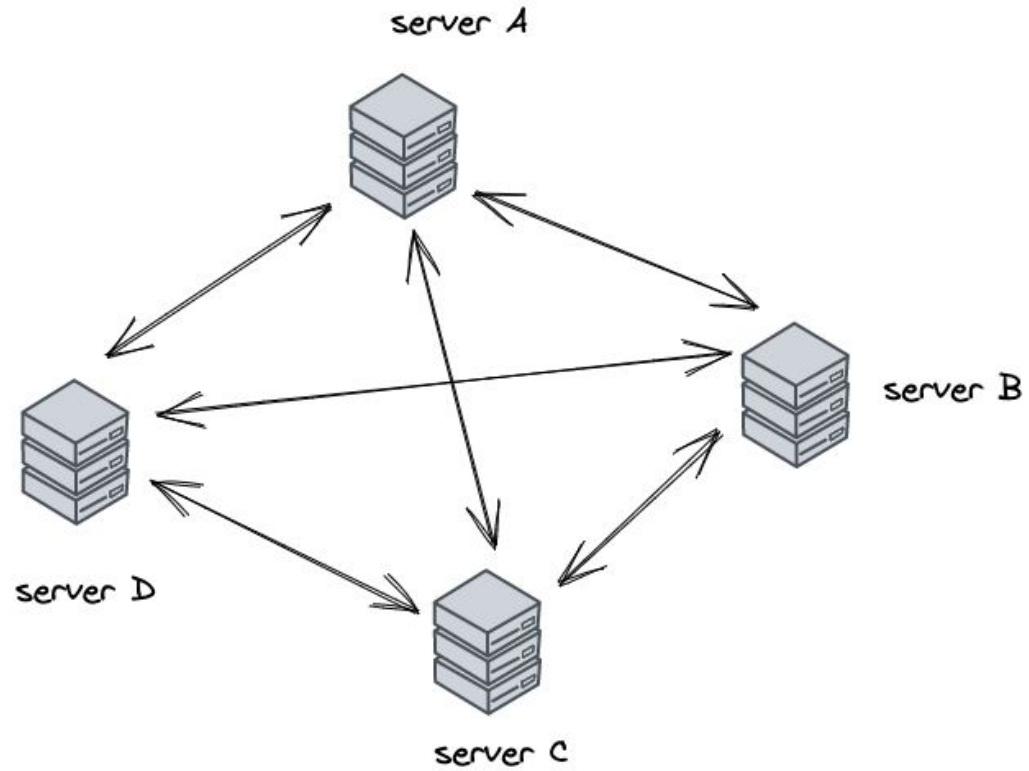
Message Queues

- Messaging
- Channels, Queues, Exchanges...
- Publish/Subscribe

Client-Server



Server-Server



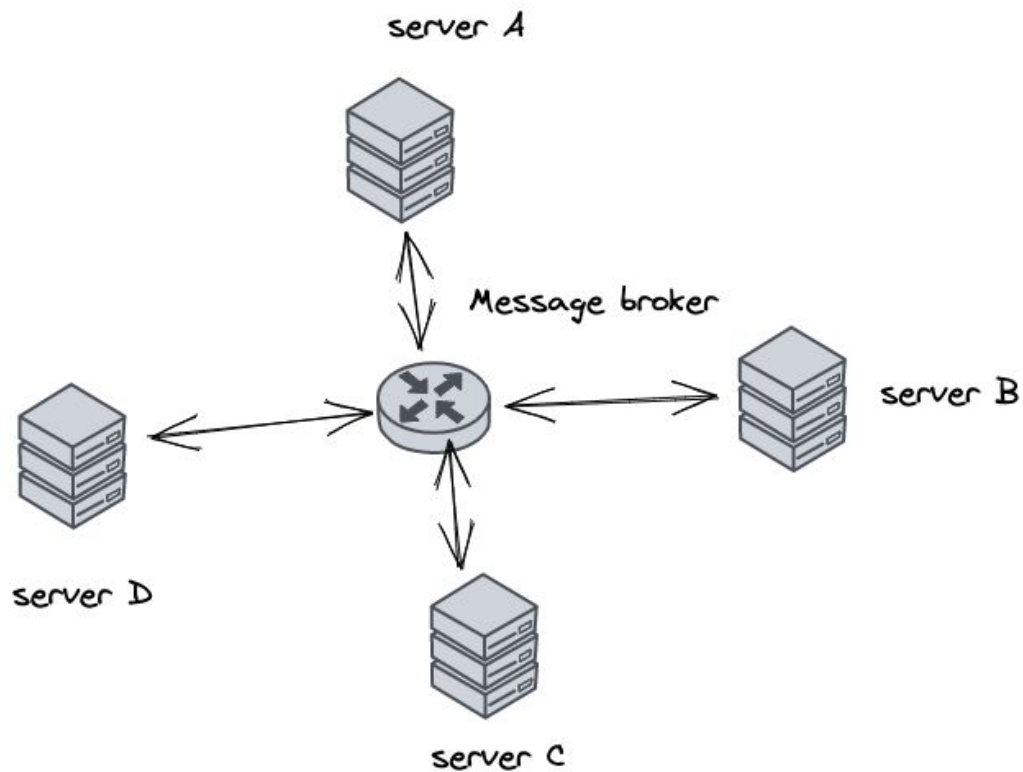
Communication between servers

- Many-to-many
 - Point to point: too many connections
- Scaling
 - Add new servers
- Asymmetric
 - Not all servers need to talk to all others
 - Some produce messages, others consume
- Failure tolerance
 - Offline servers - failover
 - Busy servers - retry

Messaging

- Communicate through message queues or brokers
- **Decouple** message from execution
 - Server for execution sends a message - another server picks it up
- **Asynchronous** communication
 - No need to wait for response - or response may be delayed
- **Dataflow** processing
 - React to presence of messages
 - Automatically adjust to rate of processing determined by activity
- **Ordered** transactions
 - First-in-first-out

Message Broker



Potential benefits

- Scalability
 - Can easily add servers to consume messages as needed
- Traffic spikes
 - Messages retained in queue until processed - may be delayed, but not lost
- Monitoring
 - Easy point of reference for monitoring performance: number of messages unprocessed
- Batch processing
 - Collect messages into a queue and process them at one shot

Variants

- **Message Queues**
 - Mostly point to point: producer -> queue -> consumer
- **Pub/Sub: Fanout**
 - Producers publish without knowing who will read, multiple subscribers consume
- **Message Bus**
 - Analogy to hardware bus: multiple entities communicate over shared medium, addressable
- **APIs / Web services**
 - Direct point to point - communicate between services directly: less resilient, no storage
- **Databases**
 - A message is a piece of information: store in database - not normally well suited

Advanced Message Queueing Protocol - AMQP

- Standard similar to HTTP, SMTP
 - Details of how to connect, initiate transfers, establish logical connections etc
- Many open-source implementations
 - RabbitMQ, Apache ActiveMQ etc
- Broker
 - Manage transfer of messages between entities
 - “Message exchange” intermediary - clients always talk to exchange
- RabbitMQ
 - Well suited for complex message routing

Redis

- In-memory database
 - Key-value store
 - Not originally designed for messaging at all
- Pub/Sub pattern
- Very high performance due to in-memory
 - But lacks persistence - data lost on shutdown
- Excellent for small messages
 - Performance degrades for large messages

Summary

- Distributed systems need messaging
- Complex messaging patterns possible
 - Point-to-point
 - Publish / Subscribe
- Many messaging systems exist
 - One more service to install and maintain
 - Useful at scale or for long running tasks
- Most useful in context of task queues

Asynchronous Tasks with Celery

Task Queues

How can we manage large numbers of long running tasks without interfering with ability to respond to user requests

- User request handler “pushes” task onto a queue
 - First-in-first-out: give priority in order that tasks issued, or can have separate priorities
- Separate queue managers to handle execution of tasks and returning results

Asynchronous: in general no guarantees of timely response,

Asynchronous Task Execution

- Language supported:
 - Python asyncio
 - JS async/await

Asynchronous Task Execution

- Language supported:
 - Python asyncio
 - JS async/await
- Guarantees of completion
- Reliable against server failure
- Ability to auto-retry

General Principles

- Pushing a task onto a queue should be faster than executing the task
 - Else not worth using a queue - just finish the task
- There should be enough worker resources to empty the queue eventually
 - Else build up backlog and eventually overflow

Potential problems

- Problems like any other distributed system
- Deadlock and related issues
 - Message system does not accept messages: block or lose data?
- Buffer sizing, overflows

Scenario: Push Queue

- Client pushes task onto server queue
- Server *should* start operation “immediately”
 - May be delayed based on availability of resources etc
- Closer to “real-time” operation
- Example:
 - Update friend list in social media application: push updates to DB for all friends
 - Send emails: push emails onto queue to be sent out individually

Scenario: Pull Queue

- Clients can push tasks at any time
- Server “polls” queue at regular intervals
- Better suited to “batch-mode” operation
 - Generally not real-time
- Example:
 - Batch update of high scores in gaming server: periodic updates
 - Dashboard updates - process many log entries in batch and update periodically

Pull mechanisms

- Polling
 - Periodically check on state of queue
 - CPU / network intensive - repetitive function calls
- Long poll
 - Server keeps connection open until data present
 - Client blocks until data received

Examples - High End

- Google AppEngine
 - TaskQueue - APIs
- Tencent cloud
- AWS
 - SQS - simple queue service - messaging
 - worker tasks implemented separately

Examples - General Use

- Celery - Python library
- RQ - Redis Queue
- Huey, Django-carrot, ...

We are mostly interested in Python APIs, but exist for most languages

- Messaging systems are language independent
- Task queue builds on top of message system + language

Celery

- Python package for handling asynchronous tasks
- Requires a separate broker for messaging
- Also a backend for collecting and storing results
- Multiple celery instances can “auto-discover” through the messaging system
- Abstracts away the messaging system to focus on tasks

Using Celery

- Problem: multiple moving parts
 - Message broker
 - Result collector
 - Celery instance to run workers
 - Actual code
- Installing and managing needs care
- Use when needed
- Can use on platforms like replit
 - Requires a little extra work