Department of Information Engineering, CUHK
MScIE – 2nd Semester, 2015/16

IEMS 5722
Mobile Network Programming and Distributed Server Architecture

Lecture 9
Asynchronous Tasks & Message Queues

Lecturer: Albert C. M. Au Yeung

10th March, 2016

The HTTP Request-Response Cycle

- The application server only performs work whenever there is a *request* from the client
- The client *waits* for the response before the application server has completed its work and returns a *response*

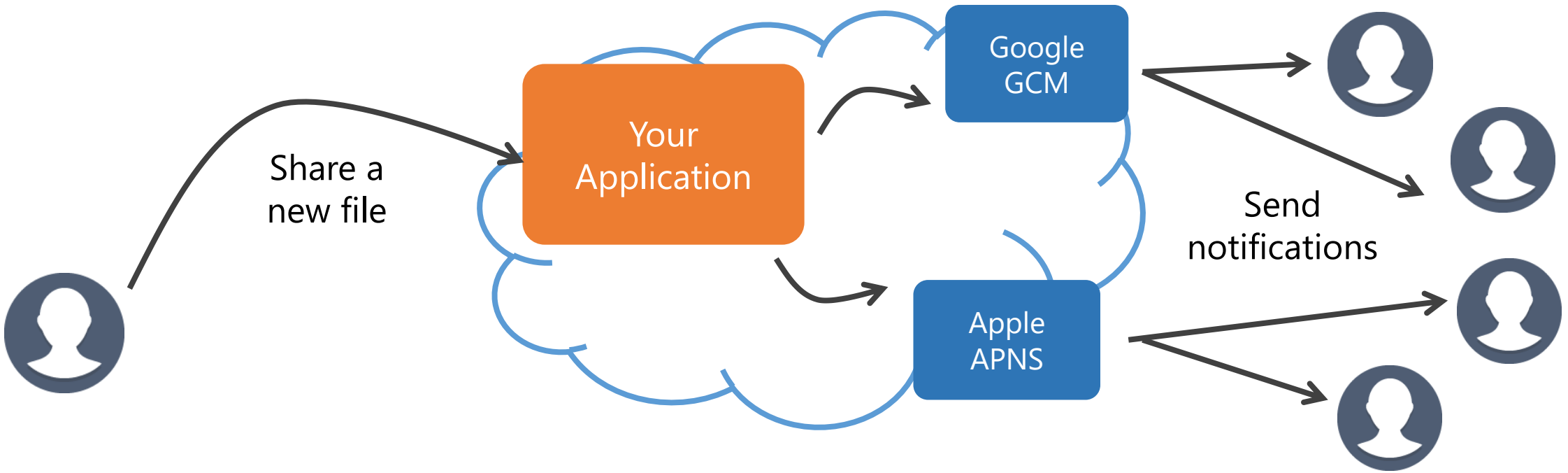
The HTTP Request-Response Cycle

- The HTTP request-response cycle is expected to complete in a *short* time (no one likes waiting!)
- However, not all tasks can be completed in a short time
- Therefore, it is necessary to carry out some tasks in the background (i.e. *outside* the HTTP request-response cycle)

Example

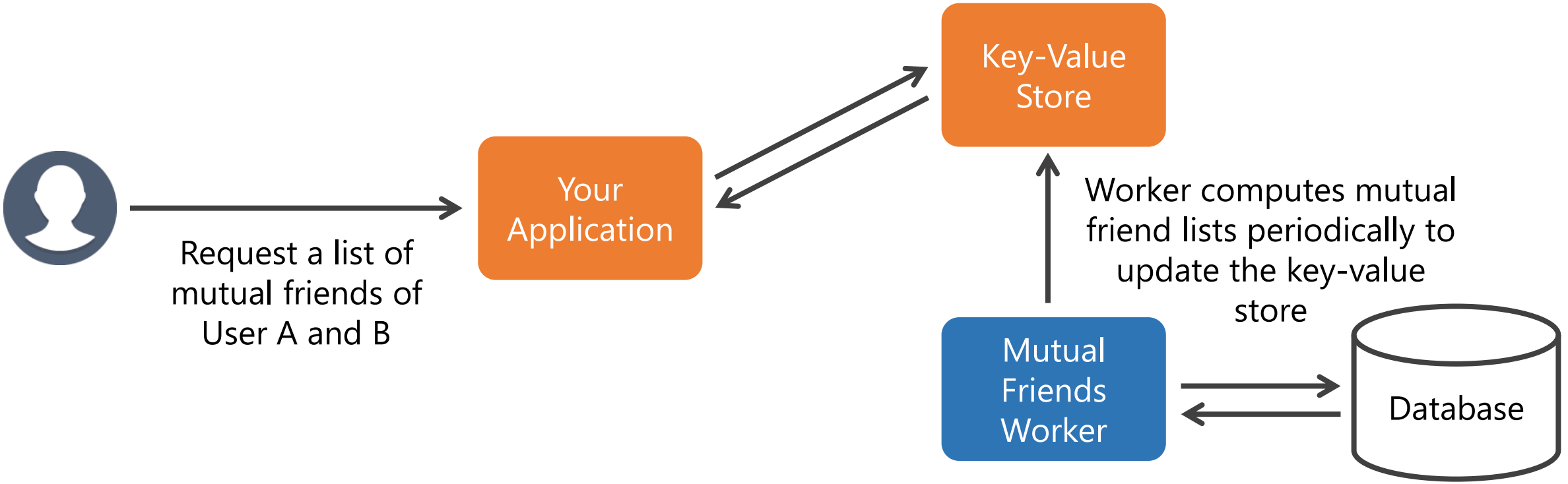
There are a lot of scenarios in which background tasks are necessary

- Consider a mobile app in which ***notifications*** will be sent to your friends when you have shared a new file



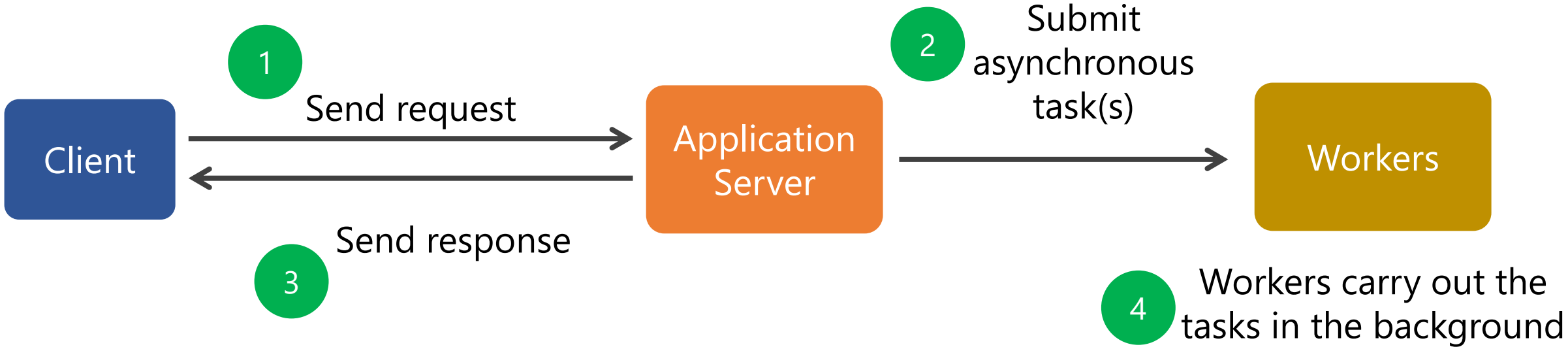
Example

- Consider a social network in which you need to quickly retrieve the list of mutual friends of two users



Asynchronous Tasks

- Task carried out outside the HTTP response-request cycle are called *asynchronous (non-blocking) tasks*
- Achieve de-coupling by separating the application server and other services



Regular Jobs

A simple way to implement asynchronous tasks is to schedule "jobs" in the system to be run at specific time (and periodically)

For example:

- Update various counters in the system (e.g. number of users, posts, messages)
- Generate a list of recommended friends for a user
- Compute the similarity of two products based on user feedbacks in a e-commerce site
- Collect information from an external source (e.g. news or weather info)

Cron

Cron is a scheduler in Unix/Linux for setting up regular jobs

- Type "crontab -e" to edit the configuration file (cron table)
- Each line in the configuration file defines a single job

```
  * * * * *  command to execute
# | | | | |
# | | | | |
# | | | | +---- day of week (0 - 6)
# | | | +------ month (1 - 12)
# | | +-------- day of month (1 - 31)
# | +---------- hour (0 - 23)
# +------------ min (0 - 59)
```

Cron Examples

- Run at 10:30am everyday

```
30 10 * * * /home/user/program.py
```

- Run at the 0th minute of every hour

```
0 * * * * /home/user/program.py
```

- Run at 6pm on every Wednesday

```
0 6 * * 3 /home/user/program.py
```

- Run every 10 minutes on Monday to Friday

```
*/10 * * * 1-5 /home/user/program.py
```

Limitation of Cron Jobs

Cron is used to schedule periodic tasks. However:

- Requests send to your server may not be evenly distributed across time
- Not all tasks can be finished within a certain amount of time
- It is not trivial to incorporate logic or load balancing in carrying out a set of related jobs

And there are other requirements too…

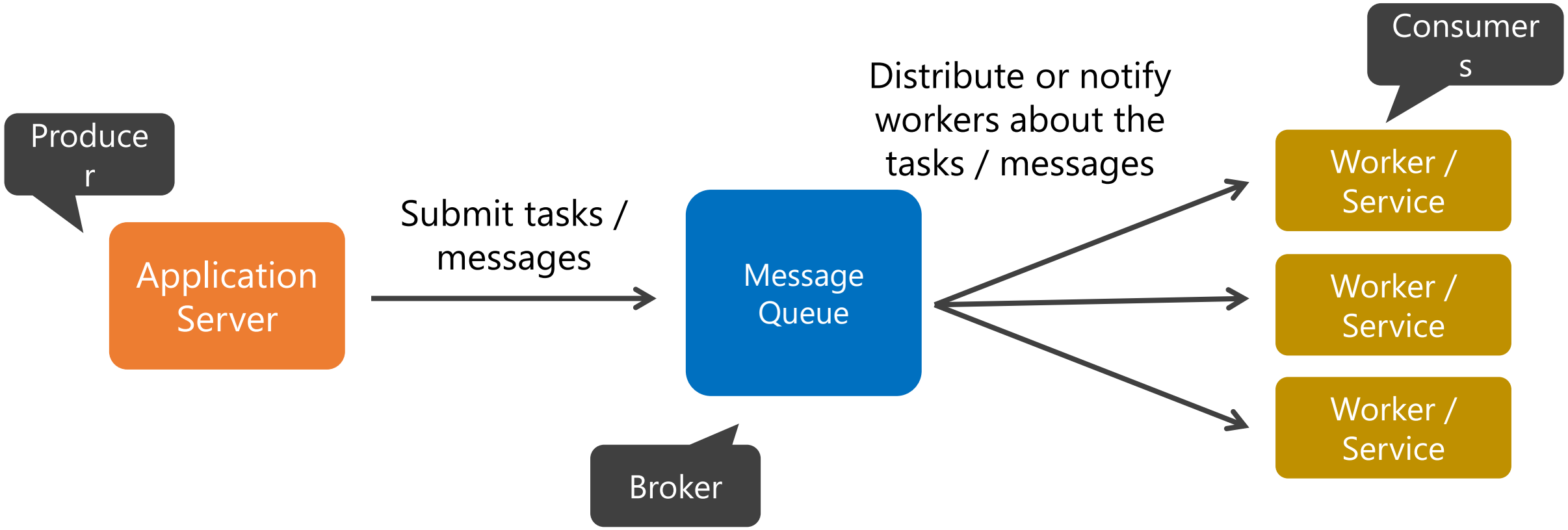
Asynchronous Tasks

It can be non-trivial to implement asynchronous tasks:

- Tasks may have to be executed sequentially
- There can be a lot of concurrent tasks
- Tasks may have different priorities
- Different tasks may require different amount of resources

We need some "*middle man*" to help us manage the tasks/messages between the server application and the workers

Message Queues



Message Queues

By having a “broker” between the application server and the services, we actually have a more robust system

- Free the HTTP request-response cycle from heavy tasks
- Clients are shielded from failures of background tasks
- If there is a failure, the broker can make sure that the task is submitted again for retry

Asynchronous Message Queues

A popular way of implementing asynchronous message queue in Python applications involves using the combination of the following:

- Celery – as the task queue
- RabbitMQ – as the message broker
- Redis – as the backend data store

RabbitMQ

RabbitMQ

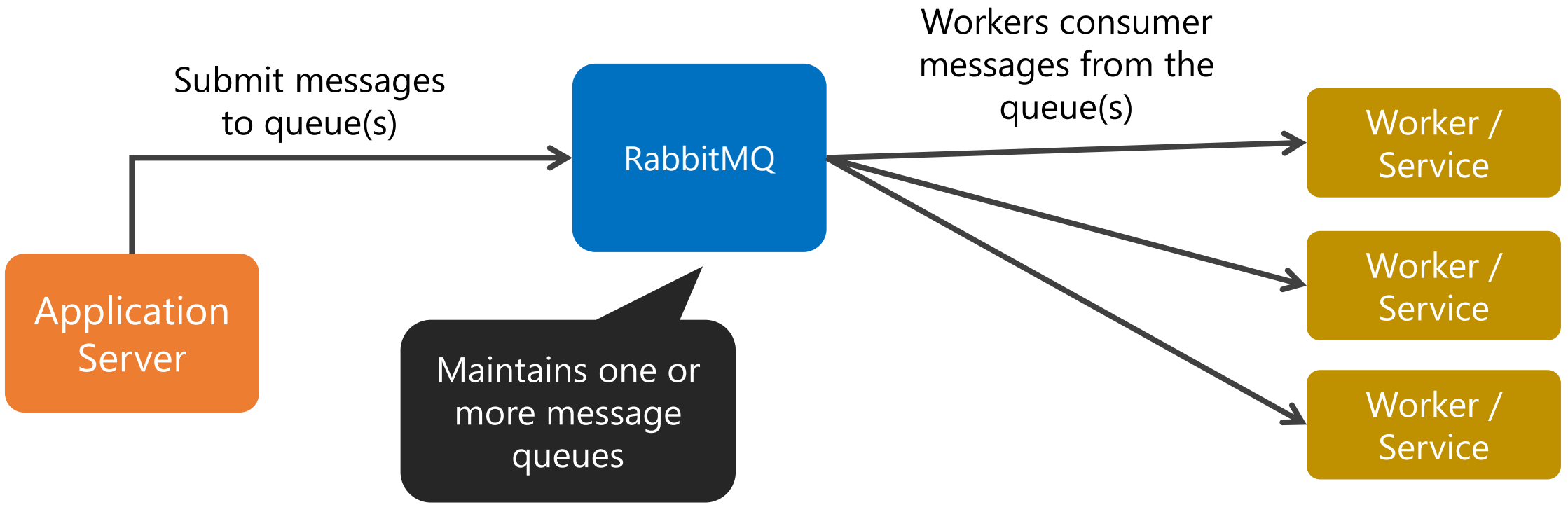
One of the commonly used open source message broker software. Some features include:

- Delivery acknowledgement
- Clustering and queues mirroring
- Client libraries available in many programming languages
- Management UI

References:
https://www.rabbitmq.com/
http://blogs.vmware.com/vfabric/2013/03/scaling-real-time-comments-huffpost-live-with-rabbitmq.html
http://blogs.vmware.com/vfabric/2013/03/how-indeed-com-handles-35-million-job-postings-per-day-using-rabbitmq.html

RabbitMQ

Architecture when using RabbitMQ as the message broker (Note: the components below do not necessarily reside on the *same machine*)



RabbitMQ

For installation, refers to https://www.rabbitmq.com/download.html

RabbitMQ uses the AMQP protocol for message passing

- AMQP = *Advanced Message Queuing Protocol*
- An application layer protocol for sending and receiving messages
- Defines how messages are routed and stored in the broker
- Defines how communications are done between clients and server (broker)

RabbitMQ – Sending Messages

In Python, we can install a module that help us use AMQP to talk to RabbitMQ (e.g. pika)

The block on the right shows an example of submitting a message to a queue in RabbitMQ

```
import pika

connection = pika.BlockingConnection(
    pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.queue_declare(queue='queue001')

channel.basic_publish(
    exchange='',
    routing_key='queue001',
    body='This is a message!')

connection.close()
```

Put this in your server application

Ref: https://www.rabbitmq.com/tutorials/tutorial-one-python.html

RabbitMQ – Consuming Messages

Consuming a message requires more complex codes, as it involves creating a *callback function* to handle incoming message

```
import pika

connection =
    pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='queue001')

def callback(ch, method, properties, body):
    print "Received %r" % (body,)

channel.basic_consume(callback, queue='queue001', no_ack=True)
channel.start_consuming()
```

Execute this script on the worker machine

Ref: https://www.rabbitmq.com/tutorials/tutorial-one-python.html

RabbitMQ

The above example refers to the most basic usage of RabbitMQ

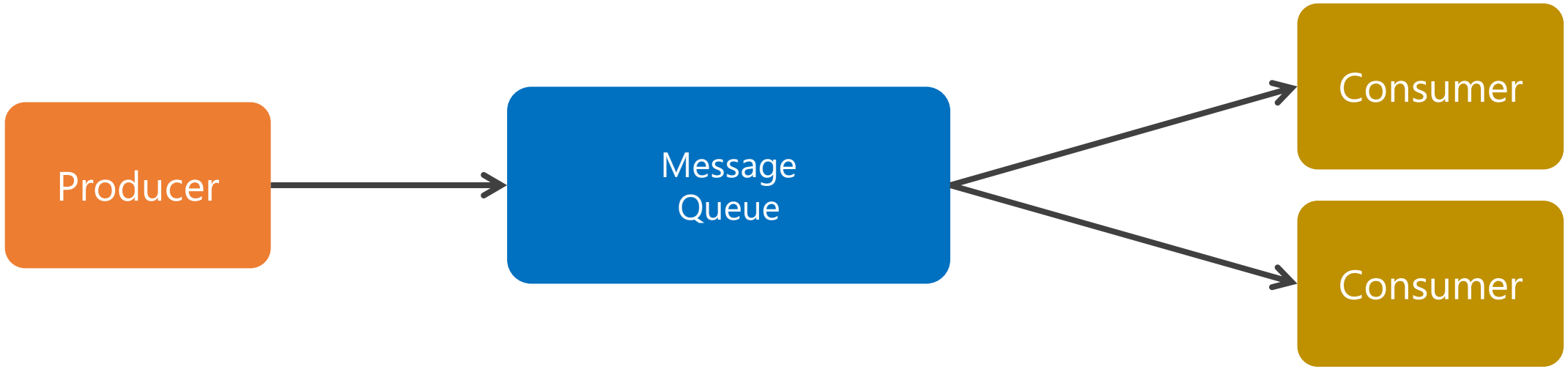
- A single queue with a single consumer

What if we want to have more consumers to share the load, introduce different types of messages, …?

RabbitMQ

Distributing messages among multiple workers

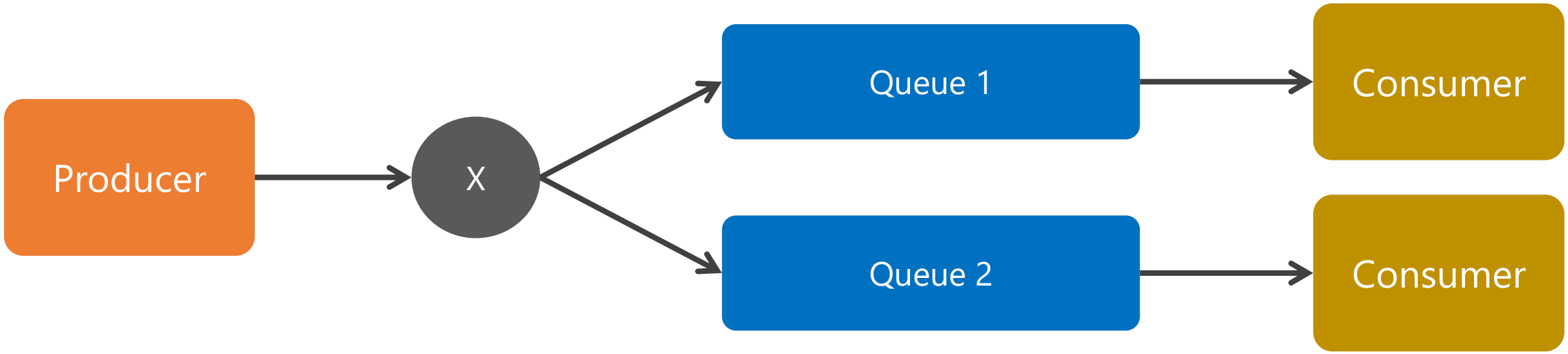
- Simply create more workers and make them consume *from the same queue*
- RabbitMQ will distribute messages to them in a *round-robin* fashion



RabbitMQ

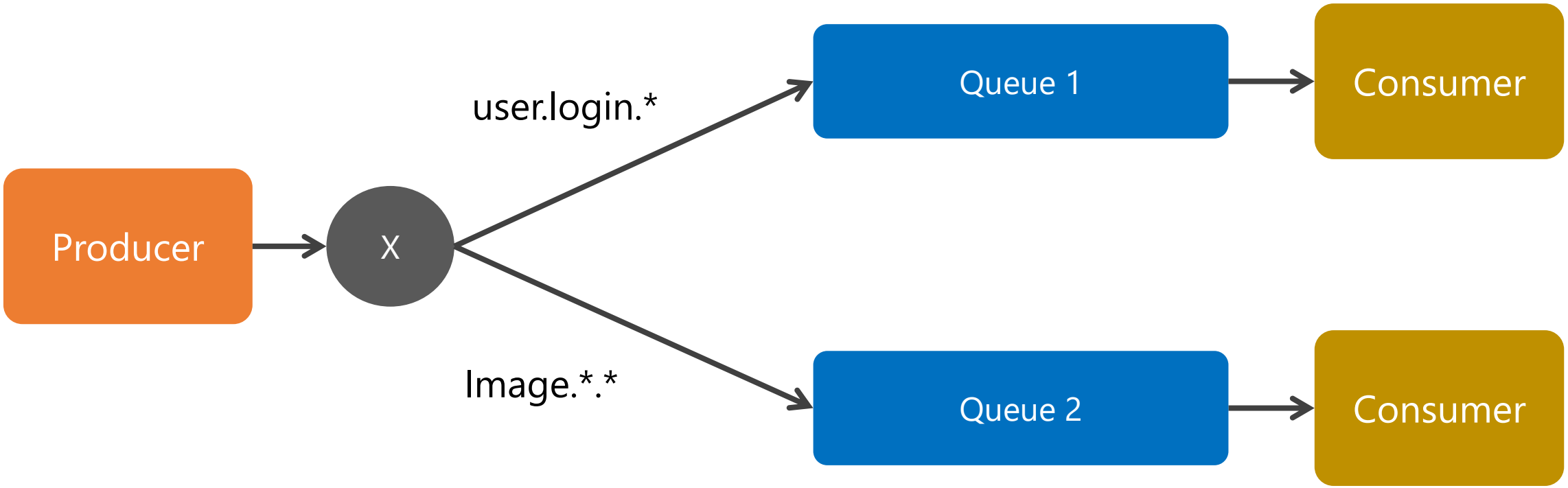
Broadcast message to all consumers at once (publish/subscribe model)

- Setup an *exchange* and let the exchange submit messages to multiple queues for multiple consumers



RabbitMQ

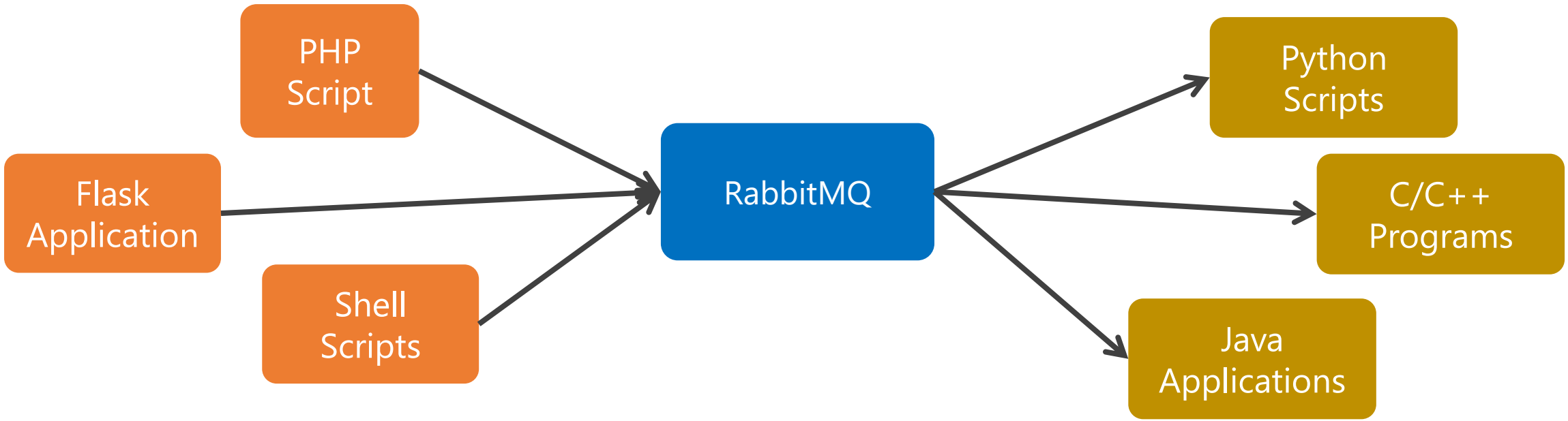
Messages can also be routed by the exchange based on their "topics", e.g.:



Ref: https://www.rabbitmq.com/tutorials/tutorial-five-python.html

RabbitMQ

RabbitMQ is a general purpose message broker, you can implement the producers and consumers using different technology and languages (as long as you use the AMQP protocol)



Events vs. Commands

The content of the message can be customised to your needs

- You can issue "commands". E.g. increment user's number of likes
- You can also issue "events". E.g. user ID=3 likes photo ID=5
- In general, it is better to send messages of "events"
  (make your application event-driven!)
  - Events can be consumed by workers responsible for doing different tasks
  - Allow better isolation between systems
  - Do not need to hard-code actions in your application

Celery

Celery

- A **distributed task queue written in Python** for Python apps
- It has to be supported by a message broker (e.g. **RabbitMQ**)
- Allow implementation of asynchronous tasks to be more integrated into your Python application

Confused?

Ref: http://www.celeryproject.org/

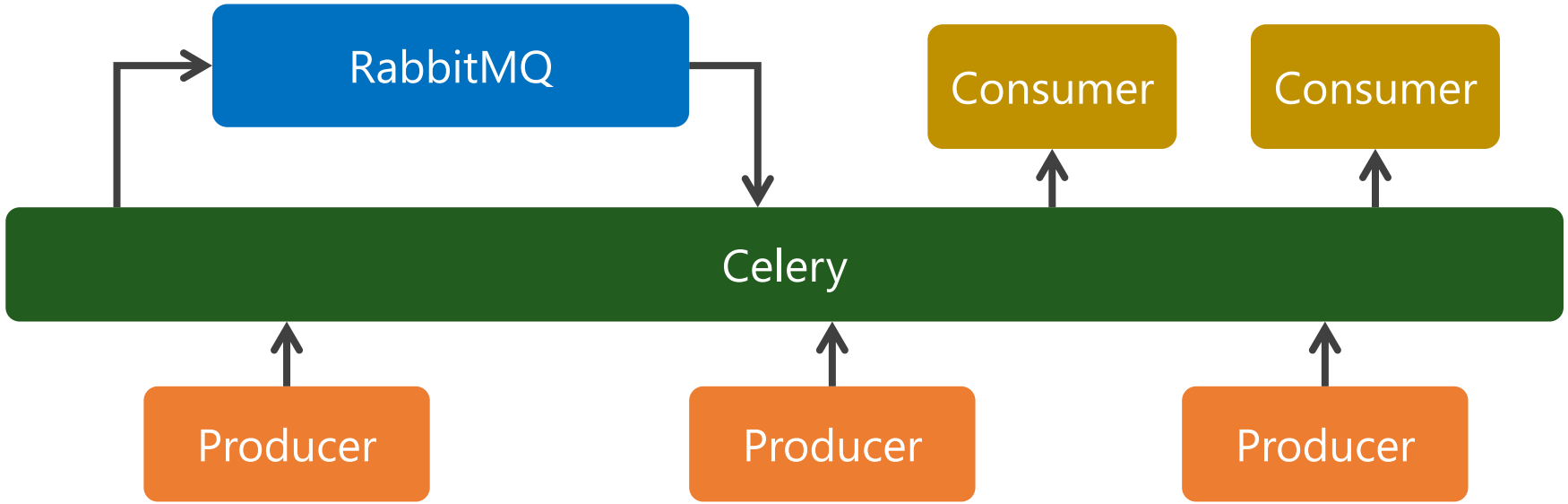
Celery

Before you use Celery:

Celery

After you use Celery

- Allows you to send messages / invoke asynchronous tasks by simply making a function call in Python
- Helps you manage your workers (e.g. restart them in case of a failure or if an exception is raised)



Celery – Example

Let's start from defining the asynchronous task you want to perform

Give the module a name (e.g. 'tasks')

Specify the URI of the RabbitMQ server

```
from celery import Celery

app = Celery('tasks', broker='amqp://guest@localhost//')

@app.task
def countWordsInWebPage(url):
    content = downloadURLContnt(url)
    words = countWords(content)
    hash = updateDatabase(words)
    return hash
```

Define a task

Celery – Example

After defining the tasks, run the Celery worker server with:

```
$ celery –A tasks worker
```

Start Celery, which will host the worker

And then you can perform the defined tasks asynchronously in your application:

```
from tasks import countWordsInWebPage
url = “https://en.wikipedia.org/wiki/RMS_Titanic”
countWordsInWebPage.delay(url)
...
```

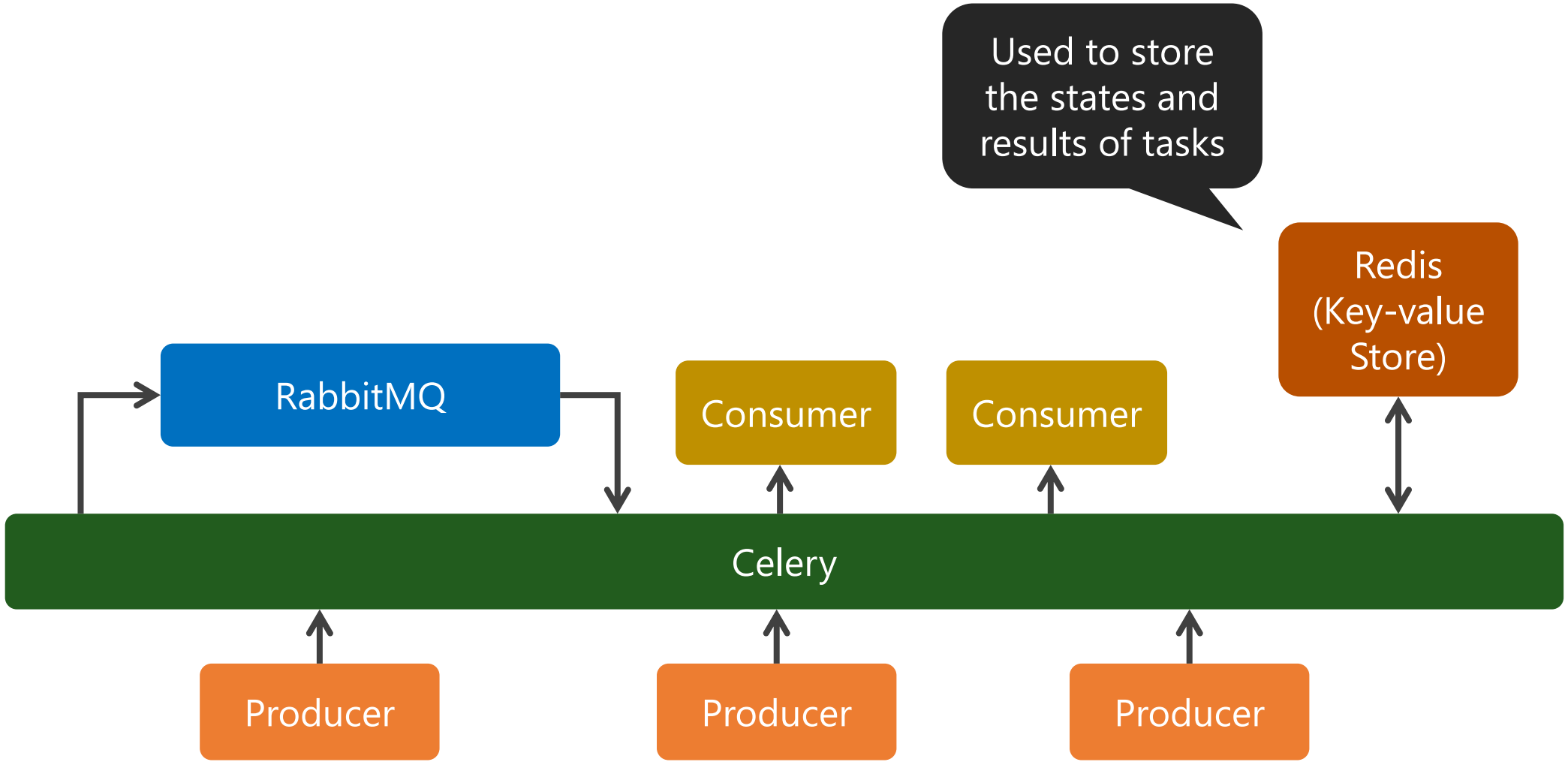
Import the function from your task module

Invoke the ‘delay’ method to perform the task asynchronously

Celery

- In many cases, you simply want to submit a task and are not concerned about the result (e.g. user likes and article, user comments on a photo, etc.)
- In other cases, you may want to *keep track of the status of the task* (e.g. let the user know about the progress of uploading a file)
- For the latter case, Celery needs a "backend" storage to temporarily stores the states of the asynchronous tasks
- Usually Redis is used as the backend

Celery


Used to store the states and results of tasks
Redis (Key-value Store)
RabbitMQ
Consumer
Consumer
Celery
Producer
Producer
Producer

Celery

Setting the backend of Celery:

```
from celery import Celery

app = Celery('tasks',
    backend='redis://localhost', broker='amqp://guest@localhost//')

@app.task
def countWordsInWebPage(url):
    content = downloadURLContnt(url)
    words = countWords(content)
    hash = updateDatabase(words)
    return hash
```

Celery

Once you have a backend, you can check the status of a task submitted:

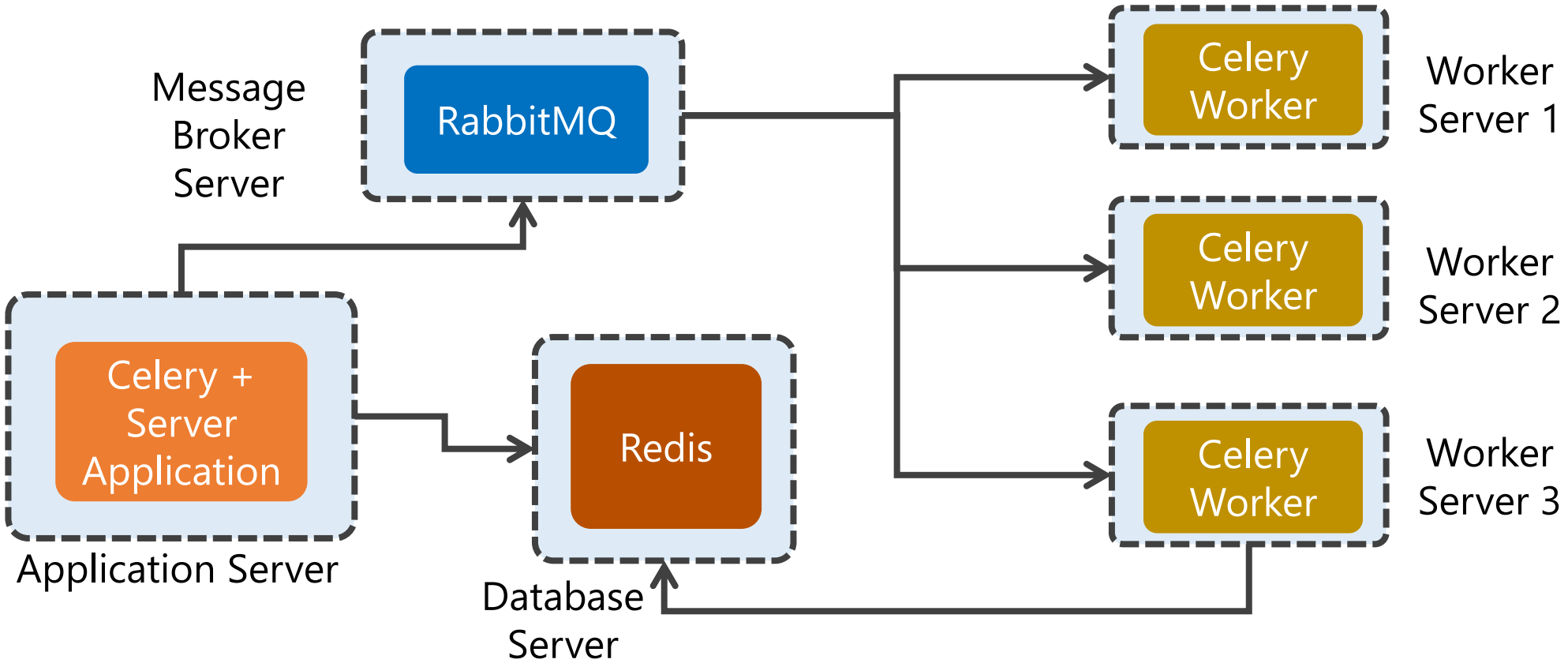
```
...
result = countWordsInWebPage.delay(url)
task_id = result.task_id
...
```

```
from celery.result import AsyncResult
result = AsyncResult("my-task-id")
result.ready() # True if the task is finished
result.result  # The return value of the task
result.state   # The current state of the task
```

Celery

The components can all be set on different machines for scalability

(Workers should be pointed to the URI of the same message broker)



Celery + Flask

You can easily integrate Celery with your Flask application

Firstly, define a function that creates a new Celery object:

```
from celery import Celery

def make_celery(app):
    celery = Celery(app.import_name,
broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task

    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery
```

Celery + Flask

Then, create a Flask app and use it to initialise the Celery object and define some tasks using the **@celery.task()** decorator

```
from flask import Flask

app = Flask(__name__)
app.config.update(
    CELERY_BROKER_URL='amqp://guest@localhost',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)
celery = make_celery(app)

@celery.task()
def add(a, b):
    return a + b
```

Celery + Flask

Then, you can create asynchronous tasks by invoking the functions in your application. For example:

```
@app.route('/')
def do_add():
    add.delay(2,3)
    return “Done”
```

Ref:
http://flask.pocoo.org/docs/0.10/patterns/celery/
https://github.com/miguelgrinberg/flask-celery-example

Celery + Flask

Before deploying your Flask app, you will also have to start the worker using Celery, for example:

```
$ celery –A app.celery worker
```

If you need more debug information, use the –loglevel parameter:

```
$ celery –A app.celery worker --loglevel=DEBUG
```

Ref:
http://flask.pocoo.org/docs/0.10/patterns/celery/
https://github.com/miguelgrinberg/flask-celery-example

Asynchronous Message Queues
+
NoSQL Database

Case Study

Consider a *social application*, in which users can be *friends* of each other.

What are some of the common operations?

- Get a list of friends of a given user
- Get a list of common friends of two given users
- Recommend a user add other users as friends if they are friends of his/her friends
- …

Case Study

A straight-forward way of storing the friendship network is to use a table with the following schema in a relational database:

TABLE friendship

Column	TYPE
user_id	INTEGER
friend_id	INTEGER
is_friend_since	DATE

Data

user_id	friend_id	is_friend_since
1	2	2015-05-07
2	1	2015-05-07
3	2	2015-09-20
2	3	2015-09-20
…	…	…

Case Study

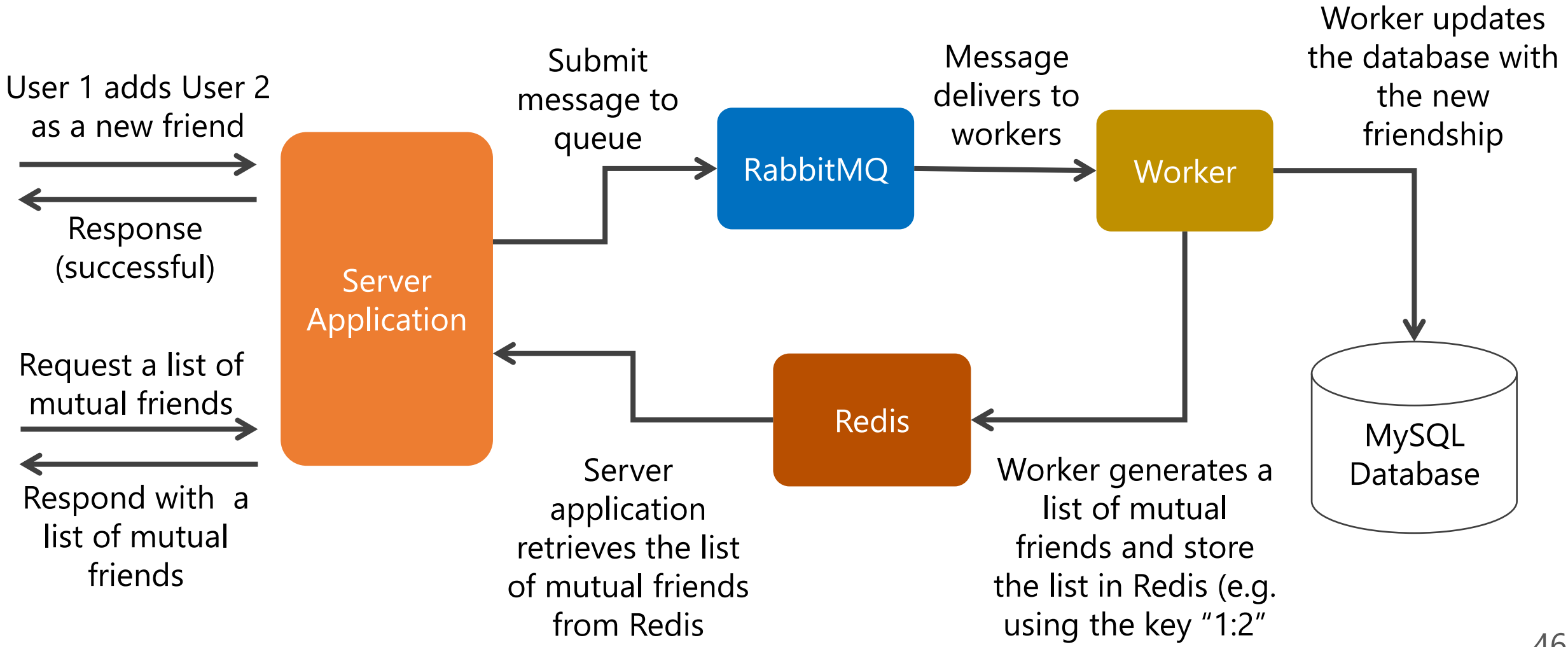
How do you get a list of mutual friends of two given users (say user 2 and user 3)?

- Retrieve the lists of friends of these two users, and then find the overlap of these two lists in your application code
- Perform a complex SQL query by joining the friendship table with itself and let the database return the list of mutual friends

It would not be efficient to compute this list on-the-fly. It is also likely that this list is not going to be changed very frequently.

Case Study

How can we improve the efficient of this function using asynchronous tasks?



End of Lecture 9