Message Queuing

Embedded Interface Design

with **Bruce Montgomery**

Learning Objectives

- Students will be able to…
  - Recall typical message patterns
  - Recognize popular message tools/platforms
  - Apply ZeroMQ or RabbitMQ in message-based code
  - Use AWS SQS

IoT Messaging Challenges

- Interoperability
- Scalability
- Deployment
- Security

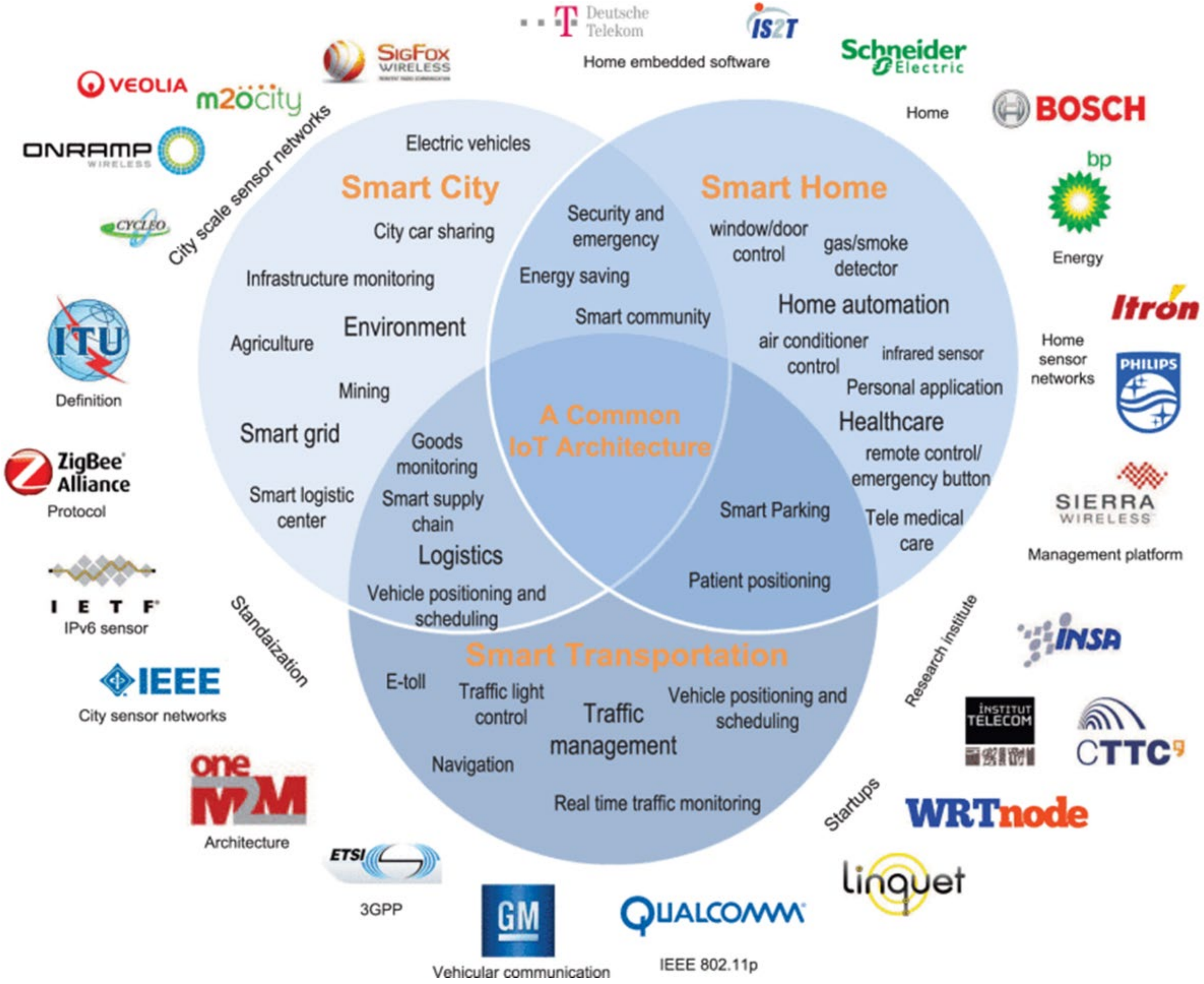
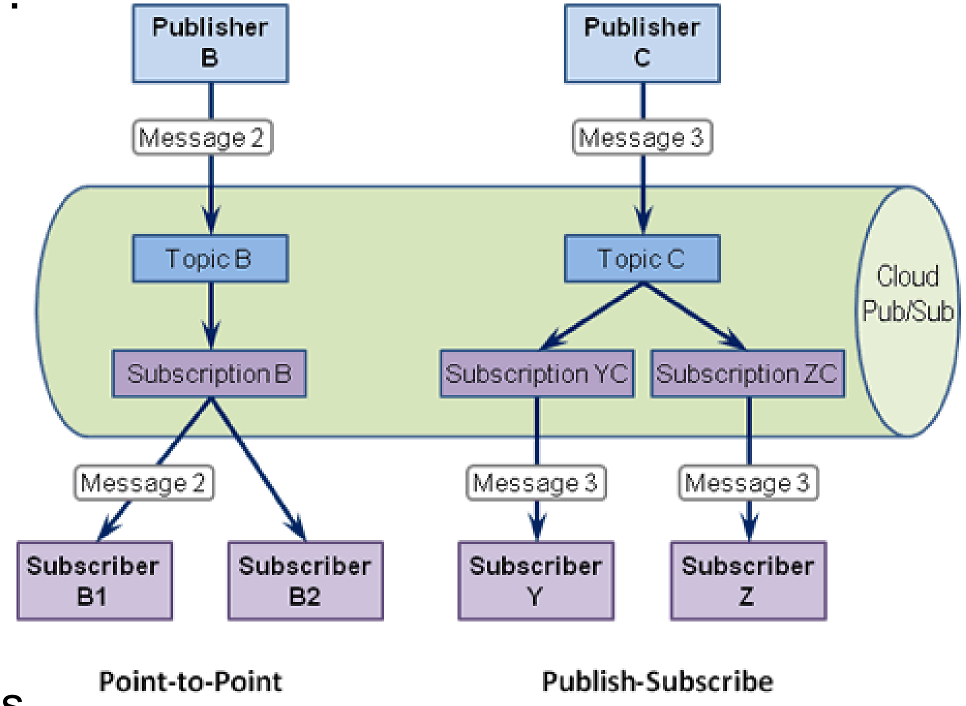


Image from [1]

Messaging Patterns (Revisited)

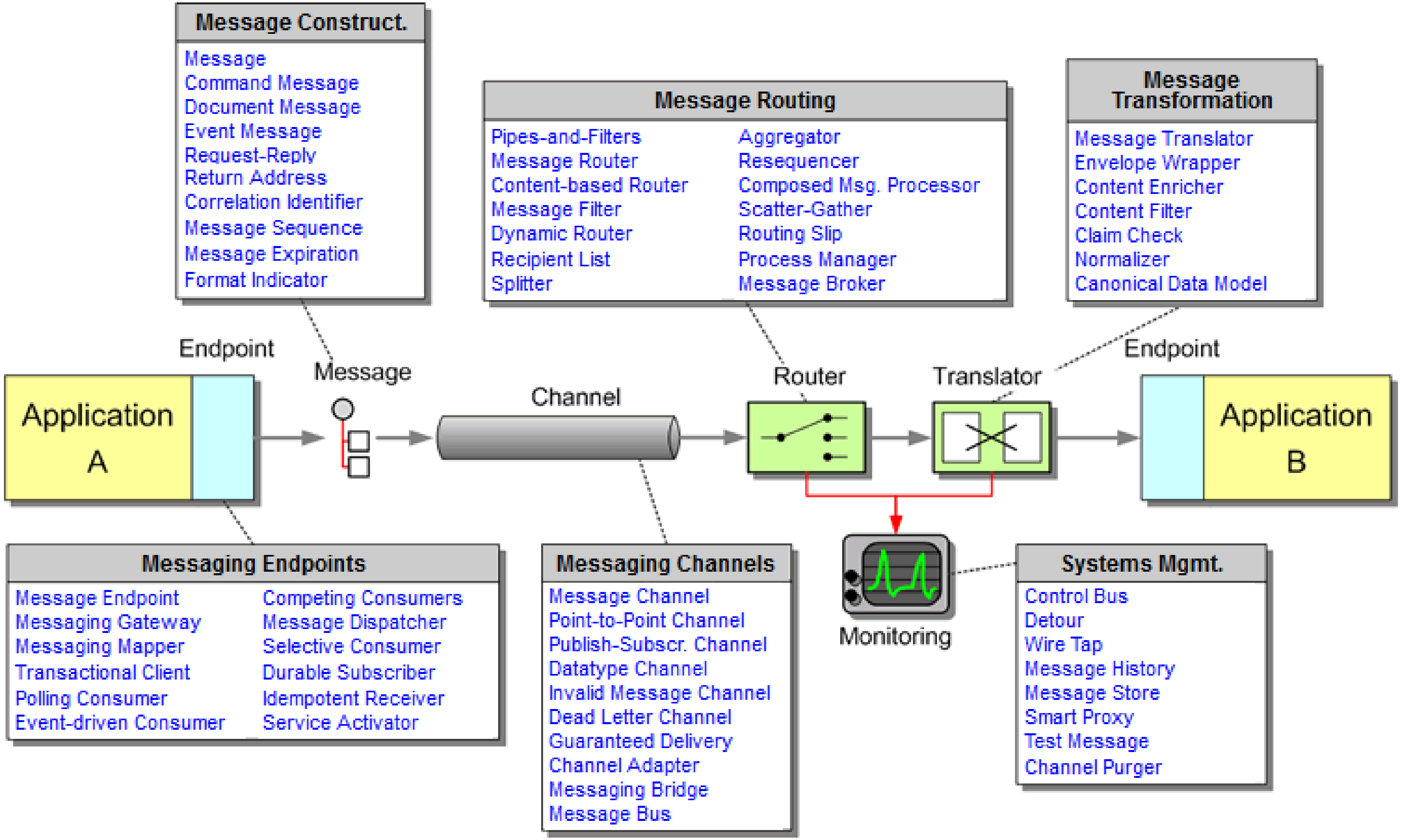
- Messaging Patterns describe the model messages follow to flow data between message producers and consumers…
- Typical Patterns
  - Publish/Subscribe (MQTT, AMQP, SMQ, STOMP, XMPP PubSub)
  - Request/Response (CoAP, HTTP, WebSocket)
  - Point to Point (aka Peer to Peer) (XMPP)
  - ACTive (Availability for Concurrent Transactions) – used by XMPP
- Other
  - Pipeline/Databus (aggregation, load-balancing)
  - Survey (1 request, multiple responses)
- Brokers – Tools to allow multi-protocol messaging
  - RabbitMQ – Primarily AMQP, supports MQTT, WebSockets
  - Others: ZeroMQ, ActiveMQ, Nanomsg, etc.



References [2], [3]

Messaging Patterns

- The Enterprise Integration Patterns book has defined 65 messaging patterns
- http://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html



Message Construct.

Message
Command Message
Document Message
Event Message
Request-Reply
Return Address
Correlation Identifier
Message Sequence
Message Expiration
Format Indicator

Message Routing

Pipes-and-Filters
Message Router
Content-based Router
Message Filter
Dynamic Router
Recipient List
Splitter
Aggregator
Resequencer
Composed Msg. Processor
Scatter-Gather
Routing Slip
Process Manager
Message Broker

Message Transformation

Message Translator
Envelope Wrapper
Content Enricher
Content Filter
Claim Check
Normalizer
Canonical Data Model

Messaging Endpoints

Message Endpoint
Messaging Gateway
Messaging Mapper
Transactional Client
Polling Consumer
Event-driven Consumer
Competing Consumers
Message Dispatcher
Selective Consumer
Durable Subscriber
Idempotent Receiver
Service Activator

Messaging Channels

Message Channel
Point-to-Point Channel
Publish-Subscr. Channel
Datatype Channel
Invalid Message Channel
Dead Letter Channel
Guaranteed Delivery
Channel Adapter
Messaging Bridge
Message Bus

Systems Mgmt.

Control Bus
Detour
Wire Tap
Message History
Message Store
Smart Proxy
Test Message
Channel Purger

Endpoint -> Message -> Channel -> Router -> Translator -> Endpoint

Messaging Tools

- **Message Brokers** like ActiveMQ, Apache Kafka, or RabbitMQ
- **Messaging Frameworks** like ZeroMQ
- **Web service- or REST-based integration**, including Amazon Simple Queue Service (SQS) or Google Cloud Pub/Sub
- **Other messaging approaches:**
  - **EAI (Enterprise Application Integration) and SOA (Service Oriented Architecture) platforms**, such as IBM WebSphere MQ, TIBCO, Vitria, Oracle Service Bus, WebMethods (now Software AG), Microsoft BizTalk, or Fiorano.
  - **Open source ESB's (Enterprise Service Bus)** like Mule ESB, JBoss Fuse, Open ESB, WSo2, Spring Integration, or Talend ESB
  - **JMS-based messaging systems (Java Message Service)**
  - **Microsoft technologies** like MSMQ or Windows Communication Foundation (WCF)
- Reference [4]

ZeroMQ (aka 0MQ)

- 0MQ was initially developed in 2007, and is now an open-source project [5]
- **Provides sockets that carry atomic messages across various transports** (in-process, inter-process, TCP, and multicast)
- Designed to be an ultra-simple API (Application Programming Interface) based on BSD sockets.
- Implements real messaging patterns like topic **pub-sub, fan-out, and request-response.**
- Available for most programming languages, operating systems, and hardware. It provides a single consistent model for all language APIs; It is simple to learn and use, with a learning curve of roughly one hour
- Designed as a library linked with applications - **no brokers to start and manage**, fewer moving pieces - less to break and go wrong - low CPU footprint
- Licensed as LGPL code

ZeroMQ “Hello World” Example (using zmq)

```
#
#   Hello World server in Python
#   Binds REP socket to tcp://*:5555
#   Expects b"Hello" from client, replies with b"World"
#

import time
import zmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv()
    print("Received request: %s" % message)

    #  Do some 'work'
    time.sleep(1)

    #  Send reply back to client
    socket.send(b"World")
```

From [6]

```
#
#   Hello World client in Python
#   Connects REQ socket to tcp://localhost:5555
#   Sends "Hello" to server, expects "World" back
#

import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server…")
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

#  Do 10 requests, waiting each time for a response
for request in range(10):
    print("Sending request %s …" % request)
    socket.send(b"Hello")

    #  Get the reply.
    message = socket.recv()
    print("Received reply %s [ %s ]" % (request, message))
```

Request-Reply Pattern

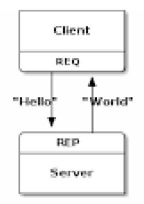
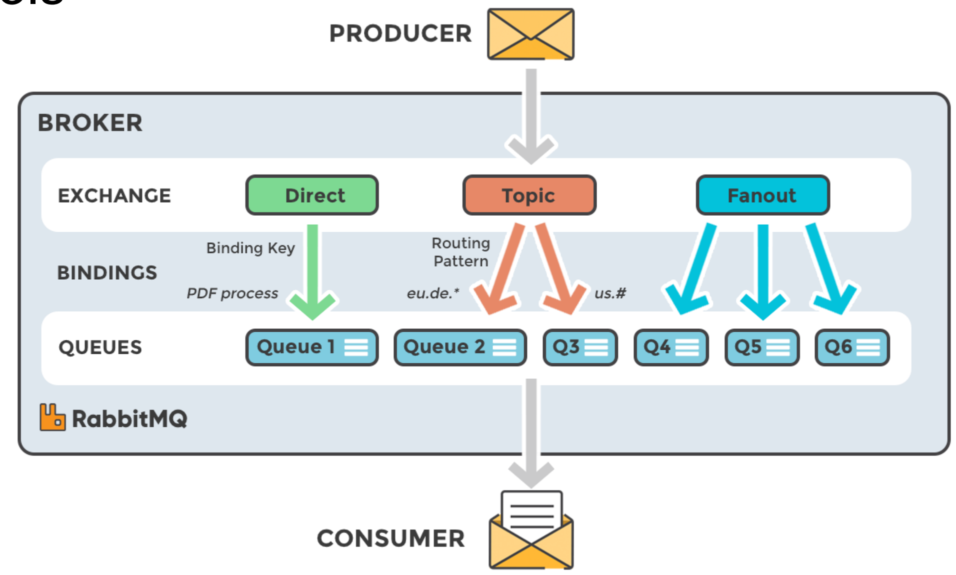


Figure 1 — Request-Reply

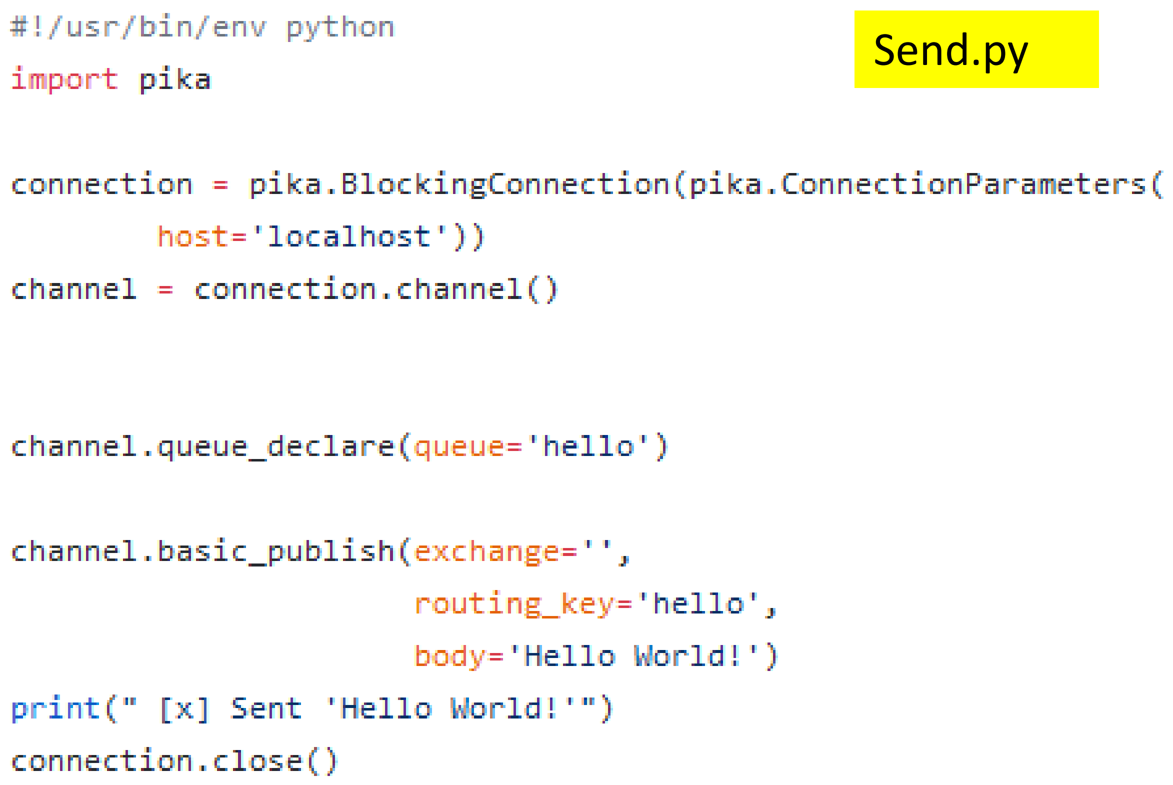
Info on zmq for Python is at [7]

RabbitMQ

- “Most Widely Deployed Open Source Message Broker”
- RabbitMQ is a message broker, containing and managing queues for messages, producers, and consumers
- Base RabbitMQ uses asynchronous messaging with AMQP 0.9.1, but it supports multiple alternative protocols including STOMP, MQTT, AMQP 1.0, and HTTP
- Support for most programming languages
- Supports work queues, pub/sub, request/reply, and other messaging architectures
- Supports TLS & LDAP for authentication and authorization
- Has a management and monitoring infrastructure – HTTP-API
- Reference [8], [9]



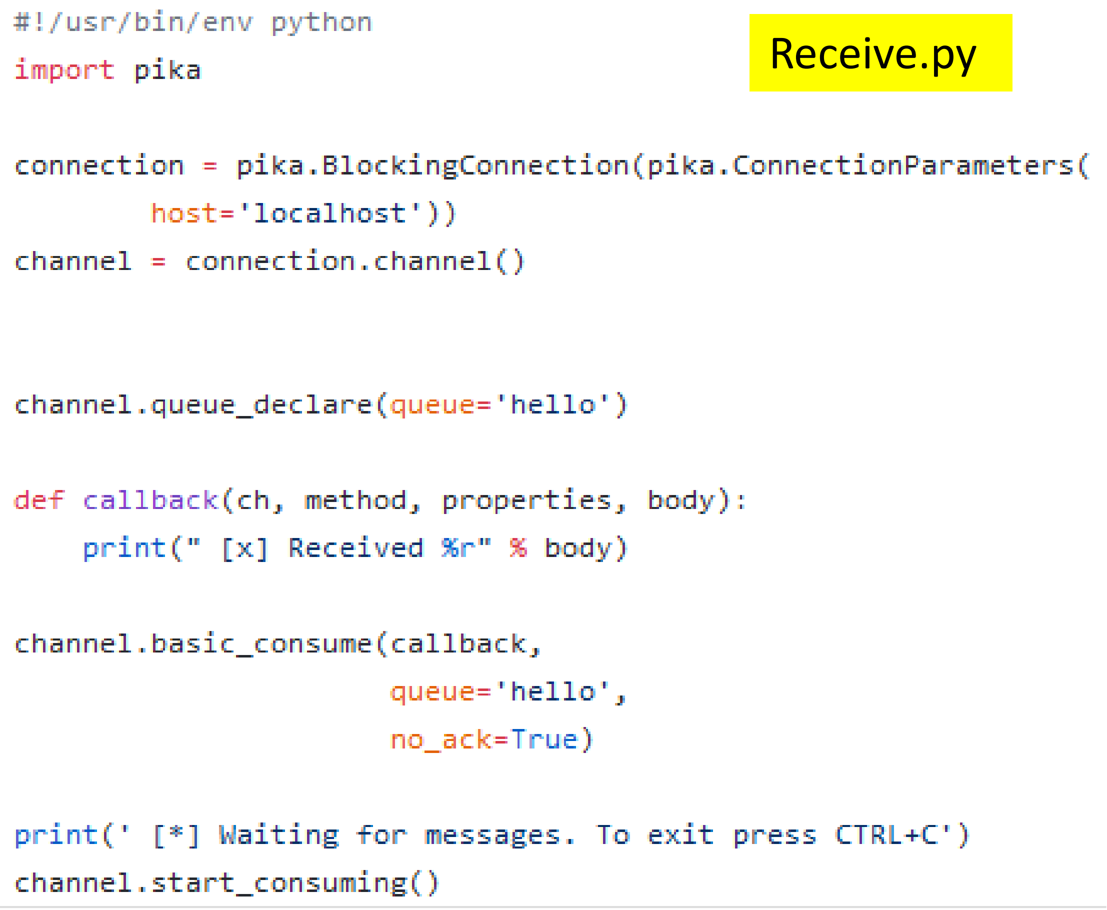
RabbitMQ Send/Receive Example (using Pika)



Send.py

```
#!/usr/bin/env python
import pika

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



channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

Example [10]



Receive.py

```
#!/usr/bin/env python
import pika

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


channel.queue_declare(queue='hello')

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

channel.basic_consume(callback,
                      queue='hello',
                      no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Info on pika at [11]

AWS SQS (Simple Queue Service)

- Fully managed cloud-based message queuing service
- Two types of message queues
  - Standard queues offer maximum throughput, best-effort ordering, and at-least-once delivery
  - SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they are sent
- Easily administrated, reliable
- Easy to integrate with IoT tools and Lambda
- Can integrate with server-side encryption
- Scale easily
- Reference [12]

References

[1] https://www.researchgate.net/figure/Industrial-IoT-ecosystem-including-major-applications-and-players-3_fig8_277562344
[2] http://www.eejournal.com/archives/articles/20150420-protocols/
[3] http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
[4] http://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html
[5] http://www.zeromq.org/local--files/whitepapers:multithreading-magic/imatix-multithreaded-magic.pdf
[6] http://zguide.zeromq.org/py:all
[7] http://zeromq.org/bindings:python
[8] https://www.rabbitmq.com/
[9] https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html
[10] https://www.rabbitmq.com/tutorials/tutorial-one-python.html
[11] https://pika.readthedocs.io/en/0.11.0/
[12] https://aws.amazon.com/sqs/