Distributed Systems

Indirect communication

Contents

- Indirect vs. direct communication
- Types
- Event-based systems
  - Subscription models
  - IceStorm + MQTT
- Message queues systems
  - RabbitMQ

Tip

All listings in this topic are available for download at:

`https://github.com/UCLM-ESI/ssdd.examples`

Indirect vs. direct communication

Direct communication is point-to-point:

- Participants must exist **at same time**.
  - Connection or session establishment
- Each participant require a way to know the address of other.
- Very inefficient when there are many participants.

Request-Reply, RPC/RMI protocols are direct communication.

Indirect vs. direct communication

Indirect communication:

- Communication implies an **intermediary**
- Sender/receiver(s) are decoupled in **time** and **space**:
  - In **space**: sender don't require to know the receiver identities or address, and vice-versa
    - sender/receiver(s) may be transparently replaced, migrated, replicated, etc.
  - In **time**: sender and receiver(s) may have different (even not overlapped) livetimes.
    - sender/receiver(s) don't require to coexist.

Indirect vs. direct communication

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5

Suitable for…

- scenarios where users connect and disconnect very often.
- scenarios with a large number of users
- event dissemination where recipients are unknown and change often
  - Mobile environments, messaging services, notification
  - Example: RSS, IoT, etc.

Disadvantages

- Overloading due to the introduction of a level of indirection
  - Reliable delivery of messages, order, etc.
- Increased management complexity due to decoupling between transmitters/receivers.
- Difficulty in achieving end-to-end properties
  - Real time, security, etc.

Indirect communication

Communication between the sender and the receiver through an intermediary:

Group communication

- Senders send messages to a "group". They don’t know the recipient identities.

Distributed event-based systems

- Disseminate events to multiple recipients through an intermediary (the event broker).

Message queues

- Senders put messages to a distributed queue, and recipients extract them from the queue.

Distributed shared memory

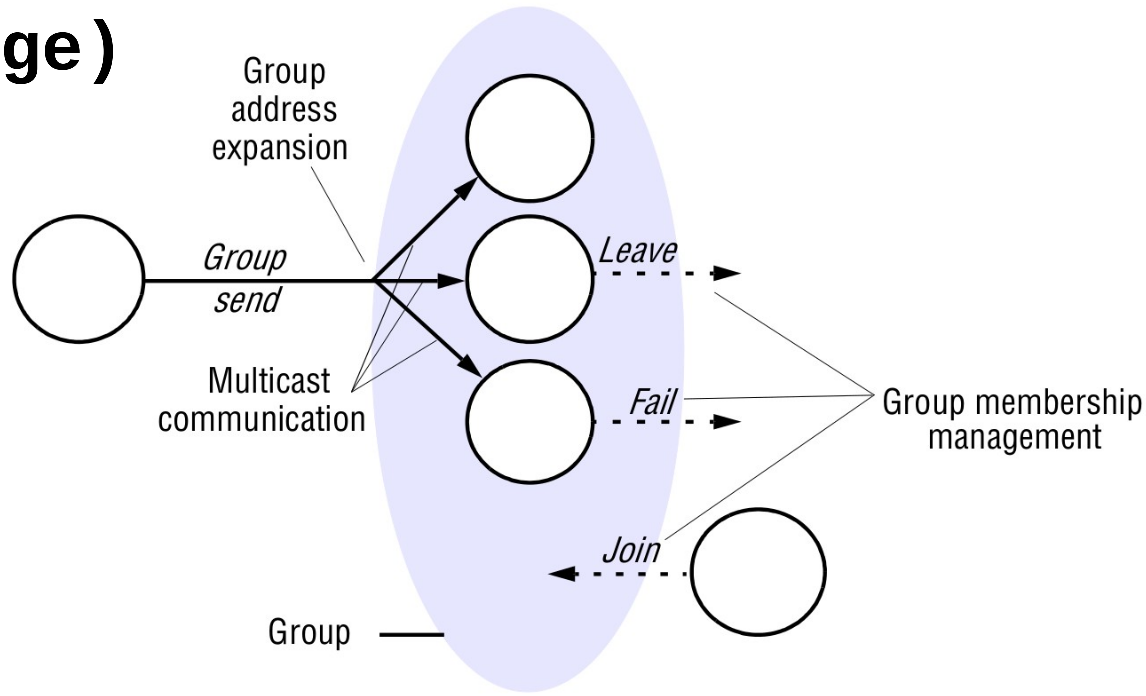
- Simulate centralized shared memory.

Group communication

- Central concept: GROUP
- Processes may become group members with **`join()`**, then may **`leave()`**.
- To send a message to a group:

  `group.send(message)`

- Very complex when reliability is required



Distributed event-based systems

Distributed Event-based systems implement one-to-many communications

- Heterogeneity & Asynchronism
- More decoupled and reactive style than RMI and/or RPC
- Public-subscribe systems

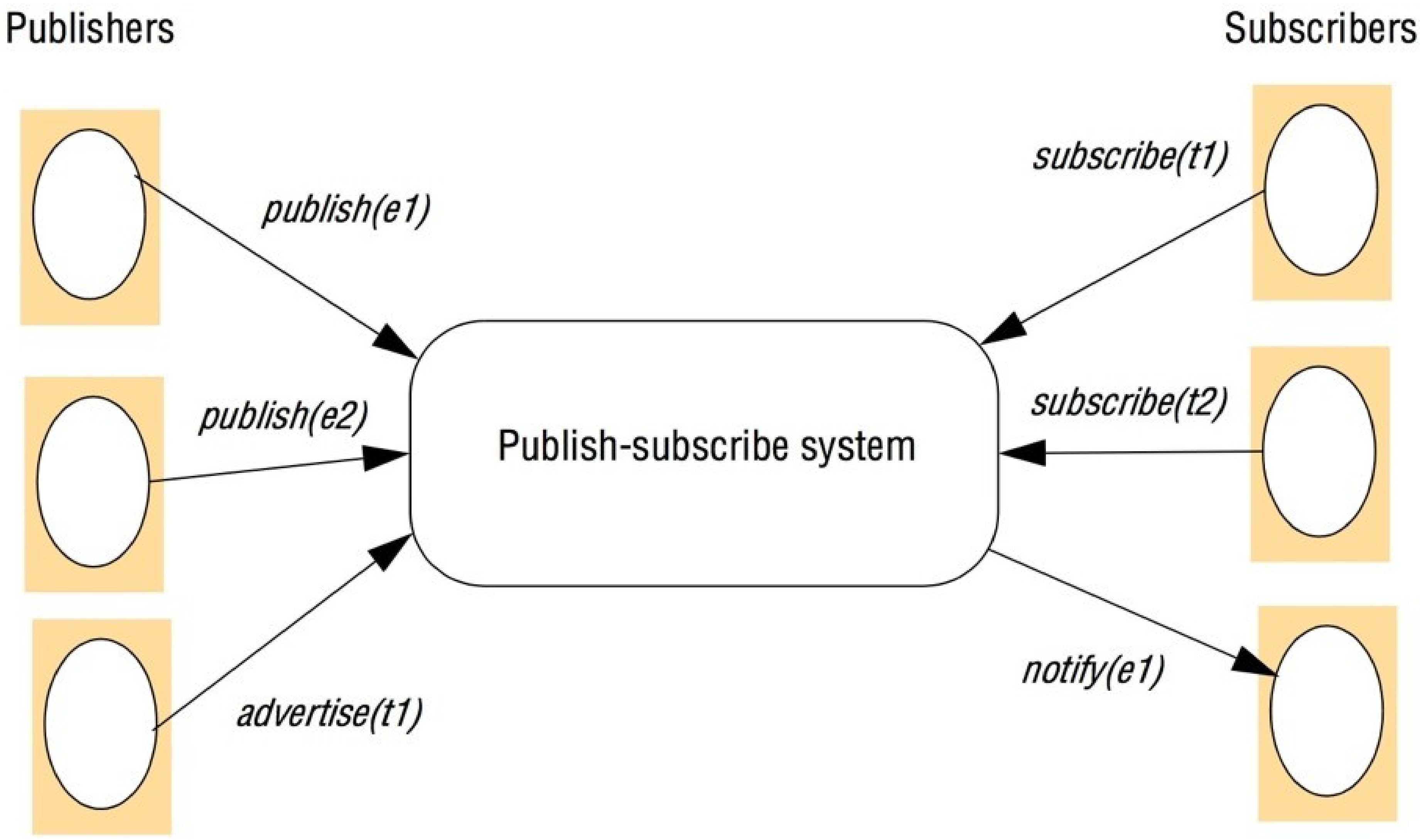
Roles:

- Publisher publishes structured events to an event service
- Subscriber expresses interest in particular events through subscriptions to an event service
- Event service or broker receives events from publishers and delivers events to subscribers according to its interests.

Programming model

- $e \rightarrow$ Event
  - any structured piece of information
- f → Filter
  - A pattern to filter the type of events
  - Expressed in specific format (e.g. regular expressions) or language (e.g. SQL-like languages)
- Basic operations :
  - `advertise(f) / unadvertise(f)`
  - `subscribe(f) / unsubscribe(f)`
  - `publish(e) / notify(e)`

Publish-subscribe paradigm



Subscription models

Channel-based: *publishers* publish events to **named channels** and *subscribers* then subscribe to one of these named channels to receive all events sent to the channel.

- Ex: ZeroC IceStorm

publisher

```
foo = broker.getChannel("foo")
foo.publish(e)
```

PSEUDOCODE

subscriber

```
foo = broker.getChannel("foo")
foo.subscribe(callback)
```

Topic-based (or subject based): Similar to channel-based but a topic (may be hierarchical) is explicitly defined as a **field**.

- Ex: MQTT

publisher

```
event = {
    topic = "temperature",
    value = 30.7}
broker.publish(event)
```

PSEUDOCODE

subscriber

```
broker.subscribe(
    "temperature", callback)
```

Subscription models

Content-based: generalization of topic-based where the filter is a query defined in terms of compositions of constraints over the values of event attributes.

- Ex: DDS

publisher

```
event = {
    color="red", size=12,
    speed=5}
broker.publish(event)
```

PSEUDOCODE

subscriber

```
broker.subscribe(
    color="blue", "size > 8",
    callback)
```

Type-based: subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter.

publisher

```
event = NoiseEvent(120)
broker.publish(event)
```

PSEUDOCODE

subscriber

```
broker.subscribe(
    NoiseEvent, callback)
```

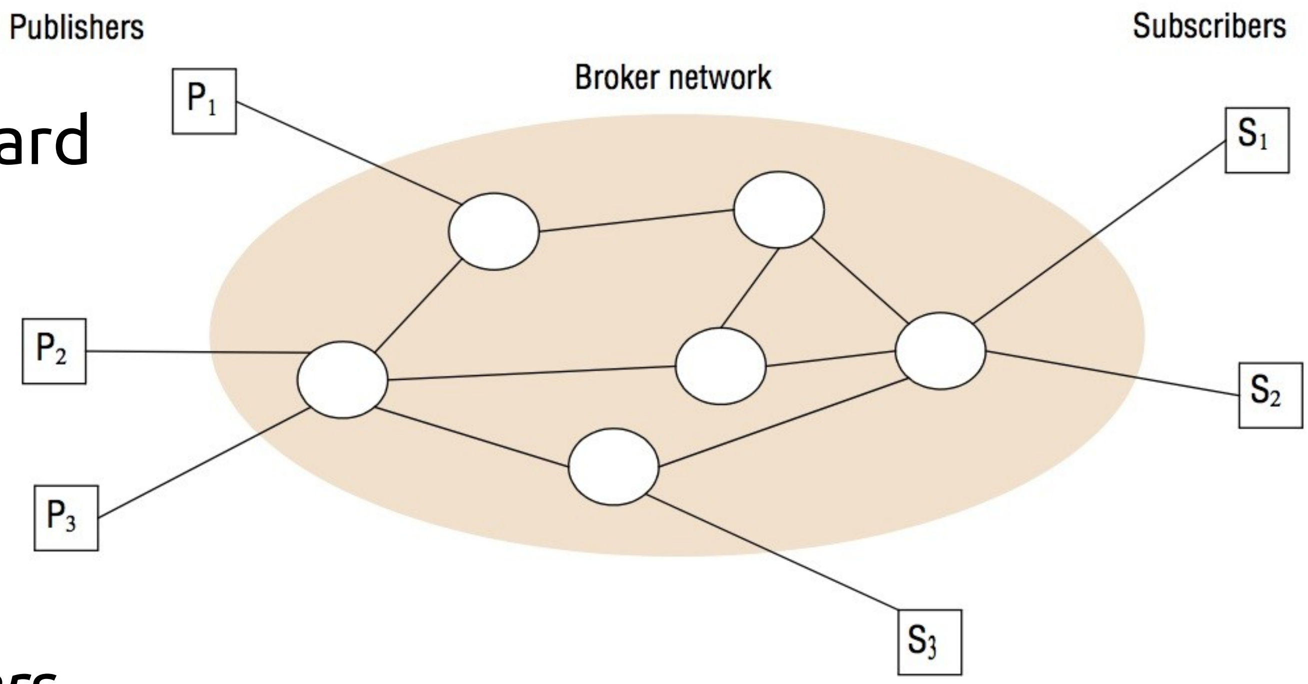
Centralized vs. Distributed

Centralized

- The *broker* is implemented in a single node
- **Pros**: straightforward to implement
- **Cons**: lack of resilience and scalability

Distributed

- A *network of brokers* cooperate for event distribution



MQTT

DIY

Description: A minimal MQTT working example

- `examples/mqtt`

See and play with:

- `subscriber.py`
- `publisher-humidity.py`
- `publisher-temperature.py`

Assure broker is running:

```
$ sudo service mosquitto restart
```

Run publisher(s)

```
$ python3 publisher-temperature.py &
  python3 publisher-humidity.py
```

Run subscriber(s) (try several)

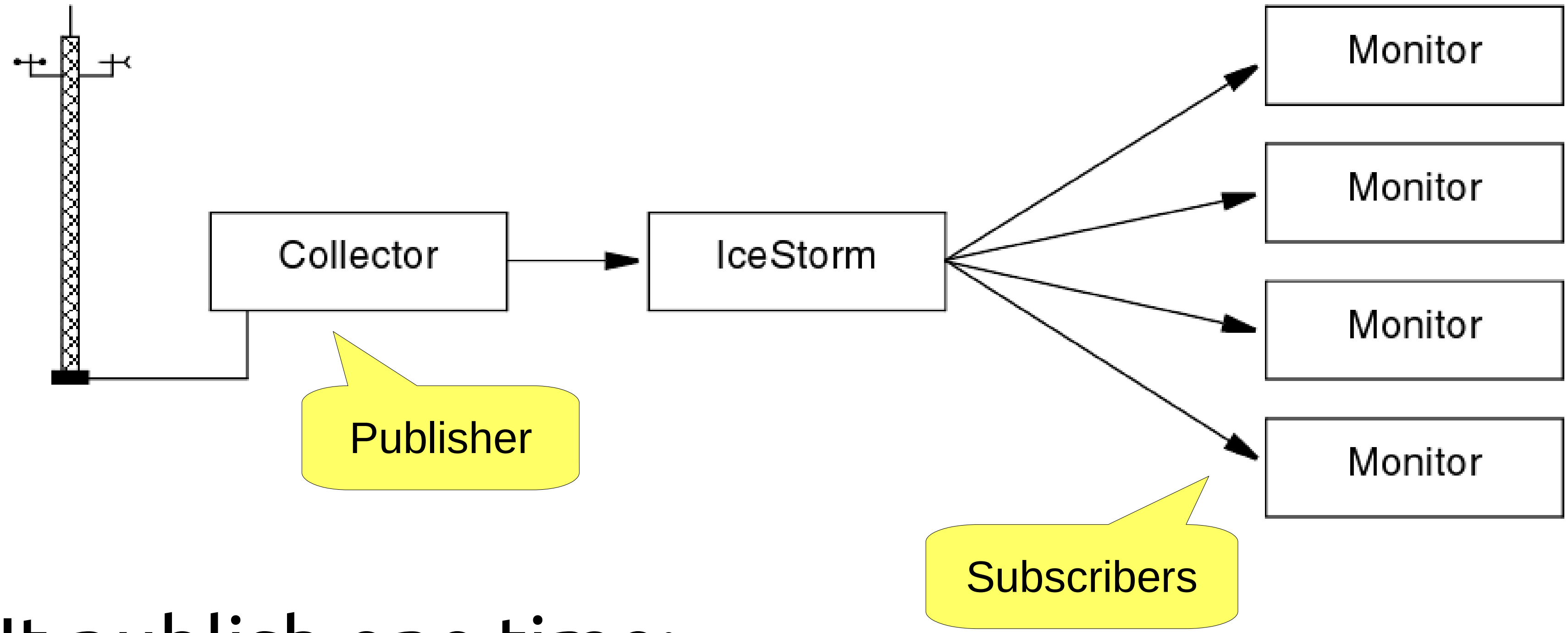
```
$ python3 subscriber.py
topic: temperature/X002, msg:
{
    'identifier': 'X002',
    'value': 36, 'unit':
    'Celsius',
    'timestamp': 1668347911.566947
} 36
```

IceStorm
Publish-subscribe example

It is the ZeroC Ice event service

- Invocation oriented (vs. data oriented)
- Channel based (named “topics”)
- Centralized broker (but replicated: HA-IceStorm)
- Basic filter features

IceStorm

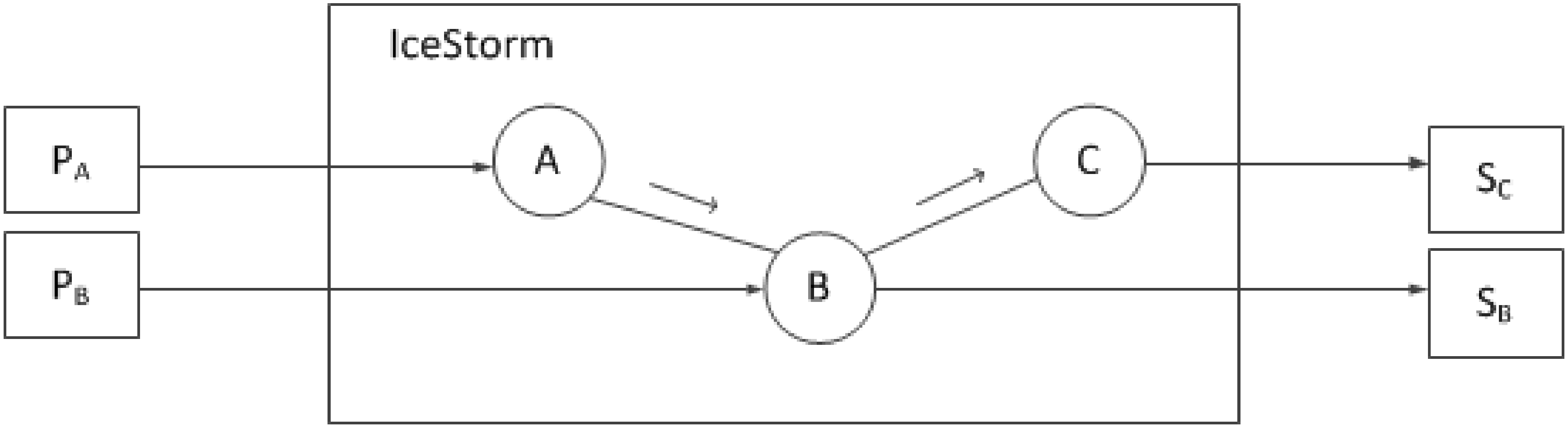


- It publish one time:
  - Broker creates copies
- Publish = invoke oneway method
  - Only provides push

IceStorm

Basic features

- Decouples production and consumption
- Allows to use a different transport protocol for any publisher or subscriber.
- Federation



IceStorm

DIY

Description: A minimal IceStorm working example

- **hello.ice/icestorm**

Choose branch:

```
$ git checkout branches/ice37
```

Start broker:

```
$ make start
icebox --Ice.Config=icebox.config &
```

See and play with:
- **subscriber.py**
- **publisher.py**

Run subscriber(s) (try several)

```
$ make run-subscriber
./subscriber.py --Ice.Config=subscriber.config
Using IceStorm in: 'IceStorm.TopicManager.Proxy'
Waiting events... 'CB1EF114-2937-4F8A-BADD-882EAAE203CC ...
Event received: Hello World 0!
```

Run publisher(s)

```
$ make run-publisher
./publisher.py --Ice.Config=publisher.config
Using IceStorm in: 'IceStorm.TopicManager.Proxy'
publishing 10 'Hello World' events
```

Message queues systems

- Distributed message queues decoupled one-to-one communication.
  - A sender put a message into an **explicit** queue.
  - A (single) recipient remove the message from queue.
- Commercial middlewares:
  - WebSphere MQ, Oracle AQ
  - JMS (Java Message Service)/Open MQ

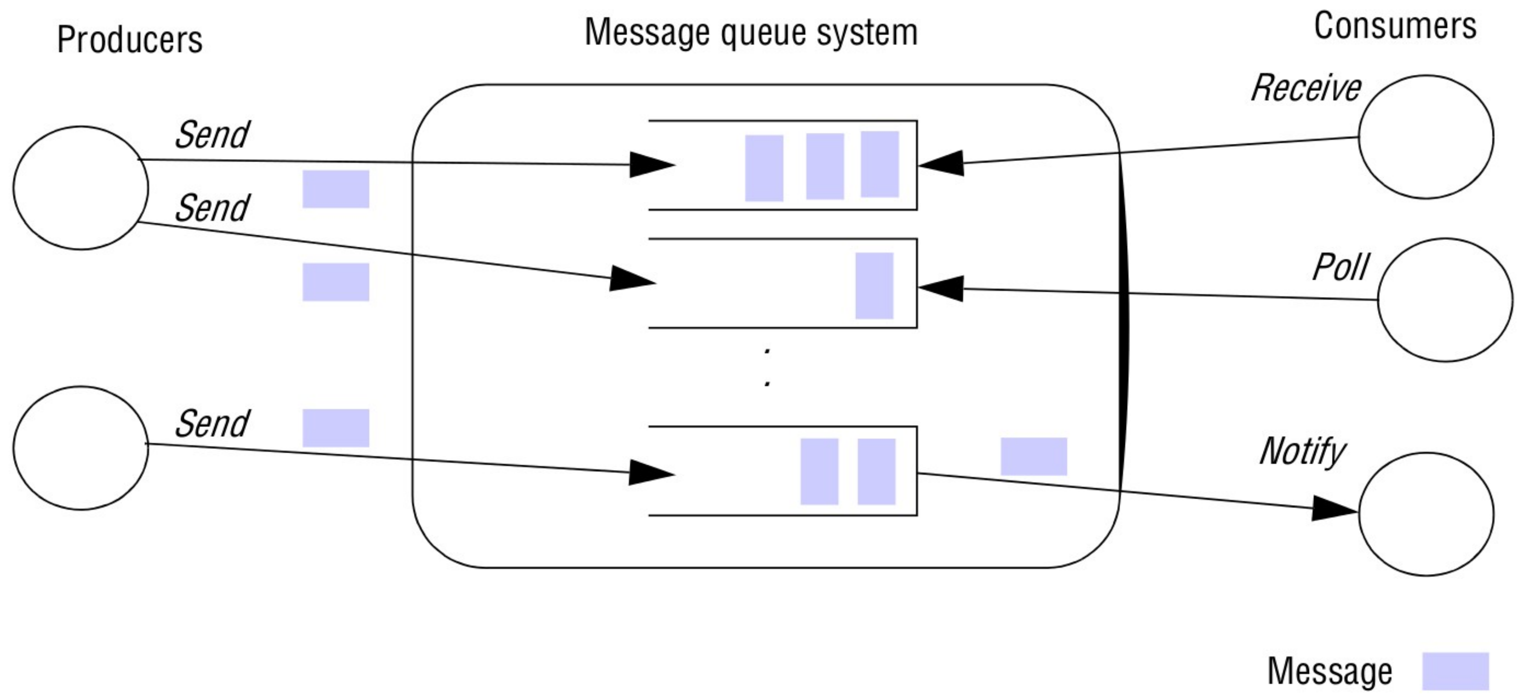
Message queues

- **Queues** are usually FIFO (**First In First Out**), but:
  - Some implementations provide priority or selection criterion.
- Messages consist of:
  - Destination queue identifier
  - Meta data
  - Payload
- Features:
  - Validity (message will be received)
  - Integrity (just one time)
  - Persistent

Programming model

Three styles for receiving messages:

- blocking receive (pull)
- non-blocking receive (polling)
- notify (push)



RabbitMQ
Message Queue example

RabbitMQ implements the standard AMQP (**Advanced Message Queuing Protocol**)

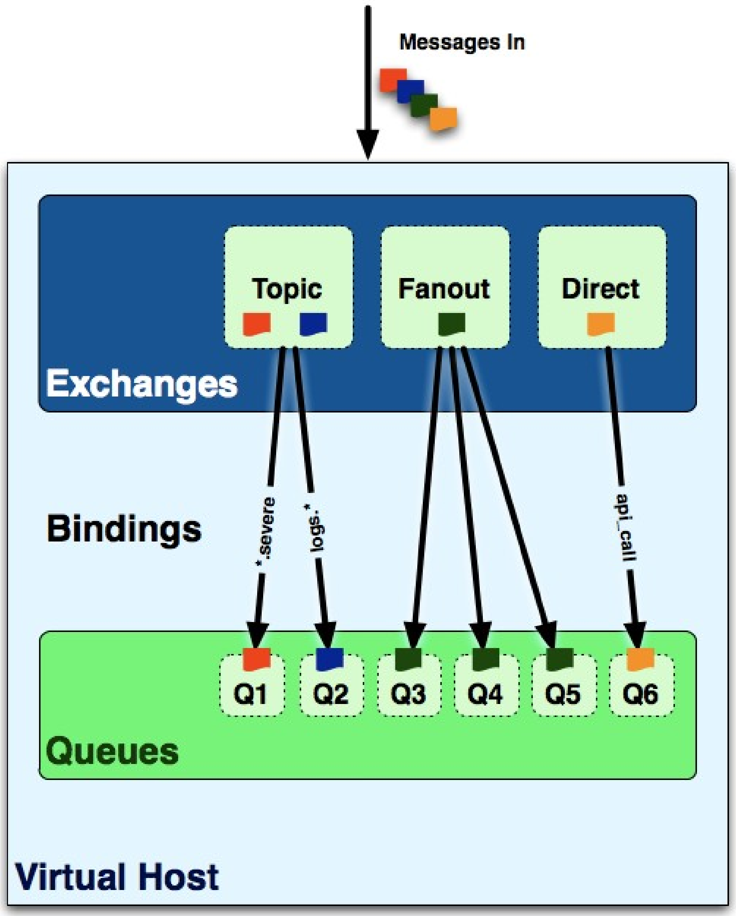
Concepts:

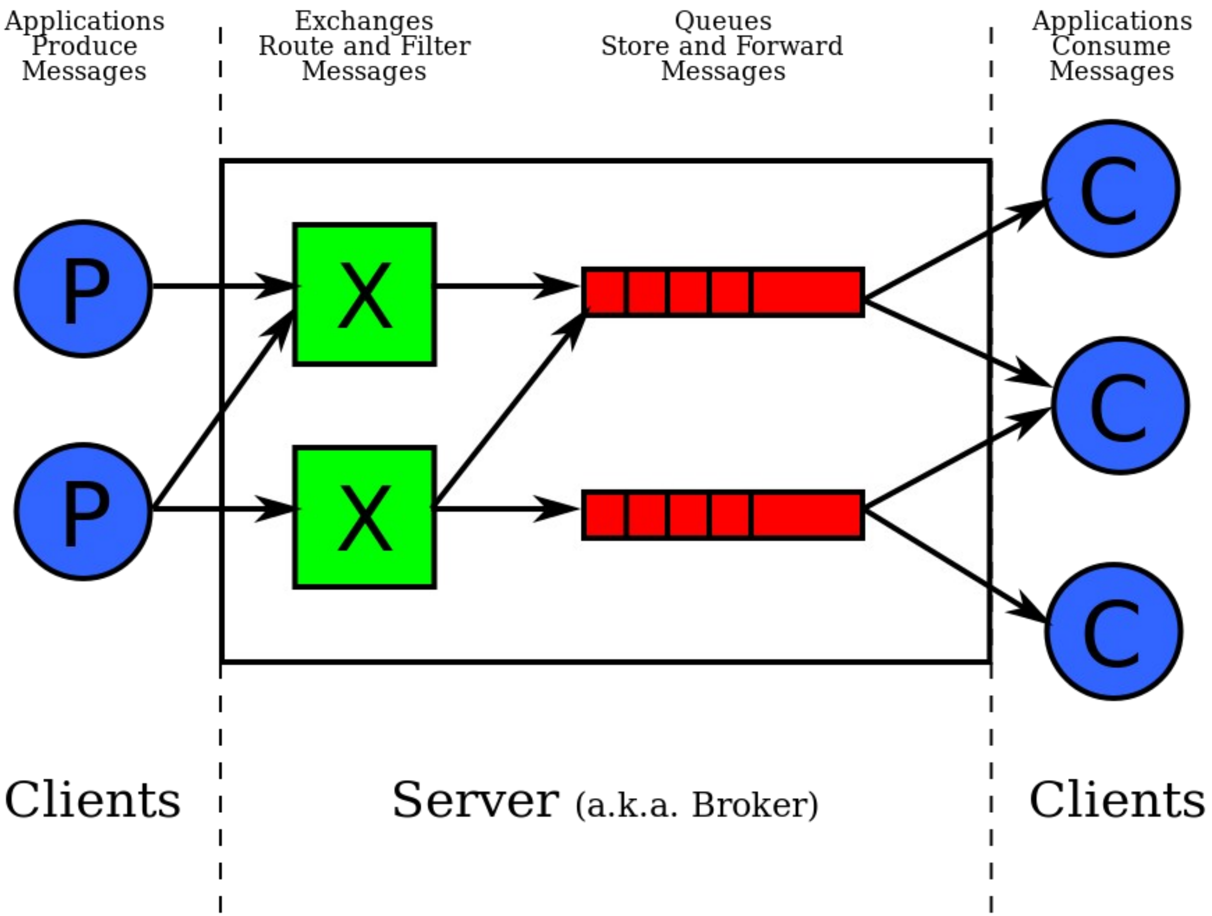
- **Publishers**: send messages to an exchange
- **Exchanges**: implement routing to send messages to consumers.
- **Consumers**: declare queue and bind it to a exchange to receive messages.
- **Queues**: Temporarily store messages from publishers
- **Routing**: Apply routing keys to perform message matching: *direct*, *fan-out* and *topic* exchanges.

It supports several messaging protocols:

- AMQP, MQTT, HTTP...

RabbitMQ
Message Queue example





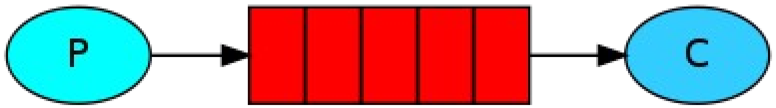
http://es.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
http://blogs.digitar.com/jjww/2009/01/rabbits-and-warrens/

RabbitMQ
producer-consumer

DIY

Description: minimal message queue working example

- **`[examples:rabbit-hello]`**



consumer.py

```
import pika

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

localhost = pika.ConnectionParameters(host='localhost')
connection = pika.BlockingConnection()
channel = connection.channel()
channel.queue_declare(queue='hello')
channel.basic_consume(callback, queue='hello', no_ack=True)

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

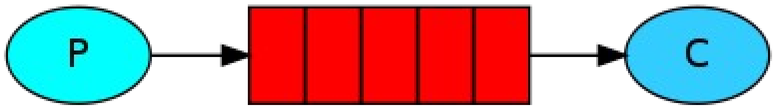
http://es.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

RabbitMQ
producer-consumer

DIY

Description: minimal message queue working example

- **[examples:rabbit-hello]**

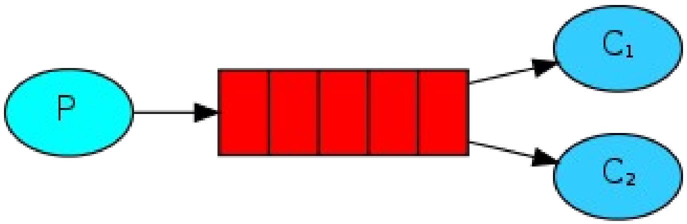


producer.py

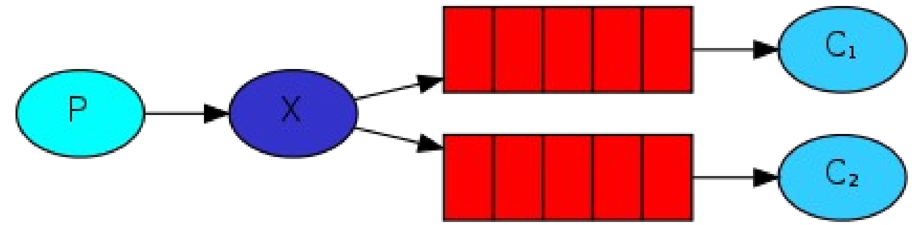
```
#!/usr/bin/env python
import pika

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

channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')

print " [x] Sent 'Hello World!'"
connection.close()
```

http://es.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

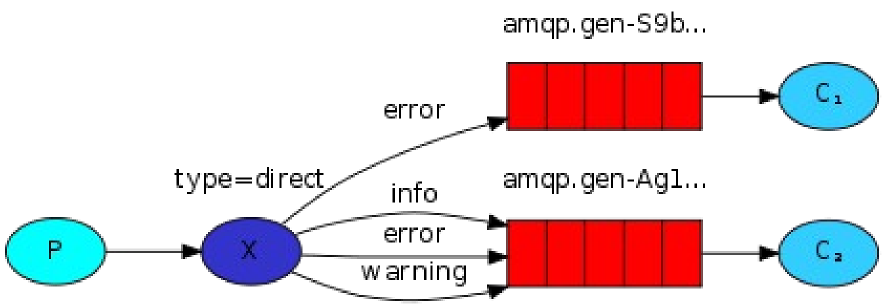
RabbitMQ
flexible binding



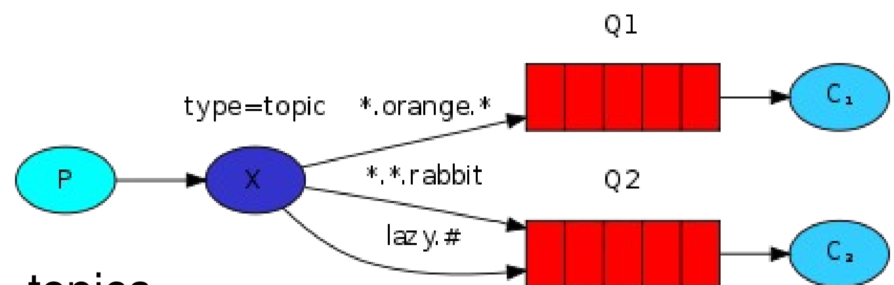
work queues



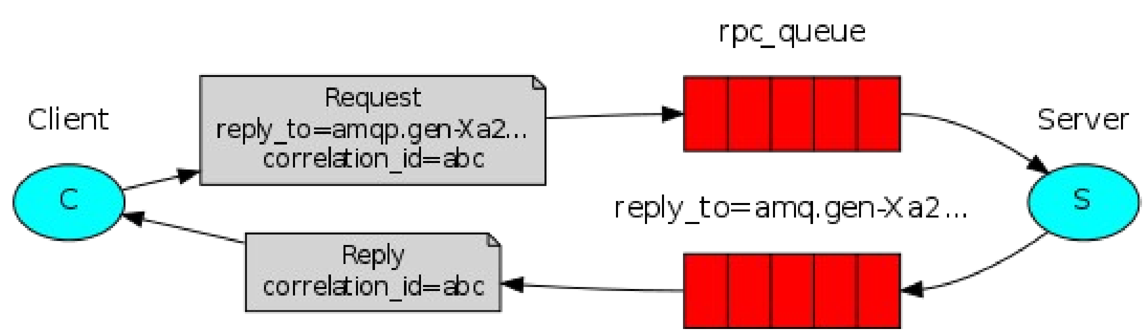
publish-subscribe



routing



topics



RPC

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

References

G. Coulouris, *Distributed Systems: Concepts and Design*, Addison Wesley 2011, Fifth edition

- Section 6.1 – Introduction
- Section 6.3 – Publish-subscribe systems
- Section 6.4 – Message queues