06: Message-oriented Communication

Distributed Software Architectures

Stefan Huber <shuber.lba@fh-salzburg.ac.at>

June 7, 2019

Section 1

Message-oriented Communication

Process coupling

Goal: For some applications it should be easy for processes to leave and join

- Direct communication requires that processes are always up and running and we know their name.

Idea:

- Stronger separation between *processing* and *coordination*
- Looser coupling
- Message-oriented communication rather than RPC

Publish-subscribe Architectures

Coupling dimensions between communicating processes:

- **Temporal**: Need to be executing at the same time
- **Referential**: Need to know each other by name

Examples for combinations:

	Temporally coupled	Temporally uncoupled
Referentially coupled	Direct	Mailbox
Referentially uncoupled	Event-based	Shared data space

- Direct: Like a phone call
- Mailbox: Like a postcard
- Event-based: Publish-subscribe, with subscribers required to be up and running
- Shared data space: A tuple space with put and get-and-remove operations

Publish-subscriber addresses the bottom row: referentially uncoupled.

- A process does not need to know the other's name.

Publish-subscribe Architectures

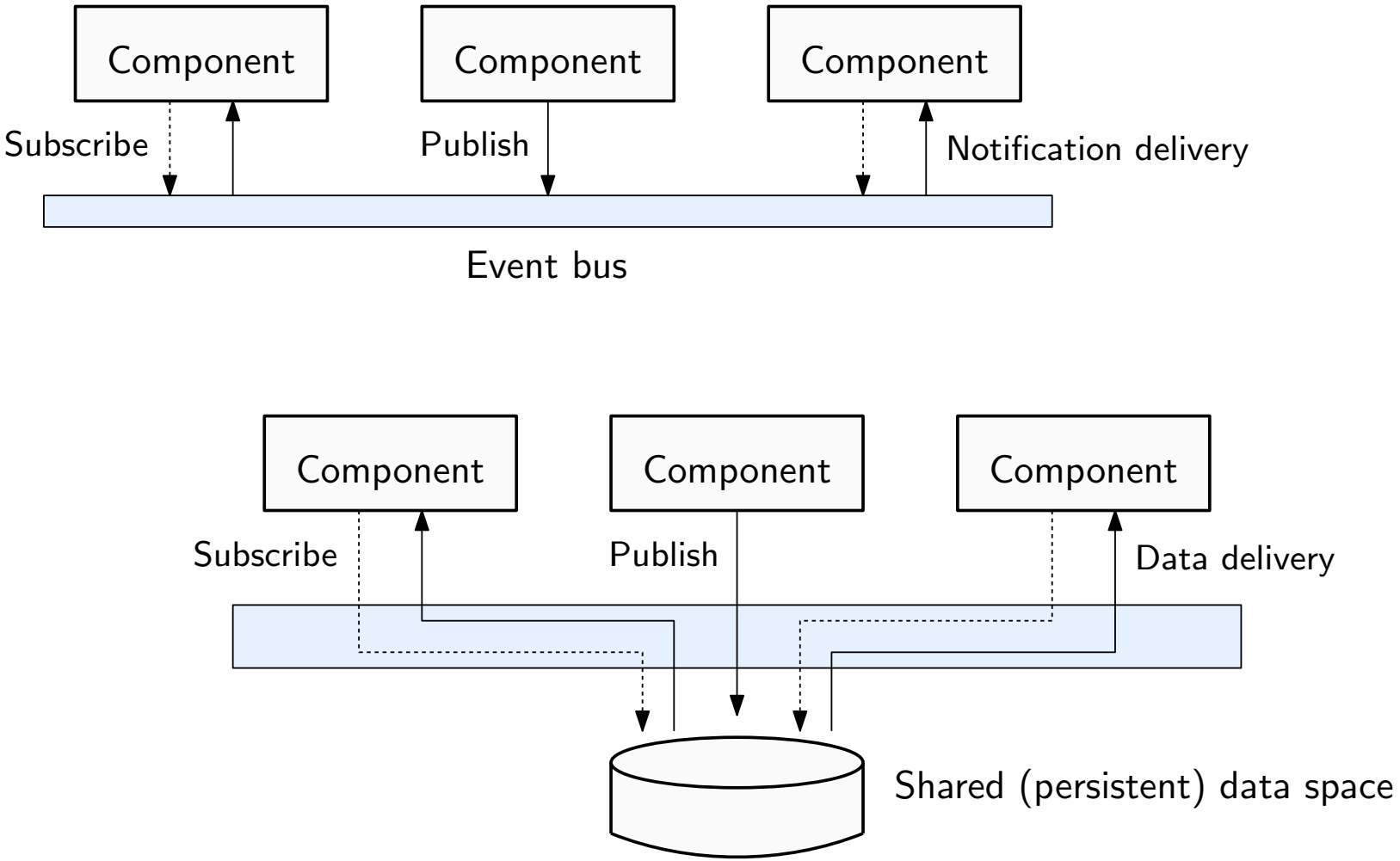


Figure: Event-based versus shared data space architectural styles. Both follow a publish-subscribe architecture with loose referential coupling.

Message-oriented communication

RPC increases the access transparency.

- But RPC requires the receiver to be up and running.
- Same is true for simple socket-based communication: Server needs to be up before the client.
- Both are a form of direct communication in the table of temporal and referential coupling.

Message-queuing systems allow for a looser temporal coupling.

ZeroMQ

ZeroMQ:

- ZeroMQ (ØMQ, 0MQ, ZMQ) is a asynchronous messaging system
- Lightweight, high-performance
- Socket-like programming, but with higher-level abstractions
- Typically uses TCP for transport, but also IPC, or in-process.
- Supports C++, C#, Erlang, Go, Haskell, Java, Lua, Node.js, PHP, Python, ...

Asynchronous and connection oriented:

- Receiver does need to be up and running (loose temporal coupling)
- Messages are queued and the ZeroMQ takes care for transmission

More info: `http://zeromq.org/intro:read-the-manual`

```
1 pip3 install pyzmq
```

ZeroMQ

A socket might be bound to multiple addresses:

- One-to-one communication
- Many-to-one communication
- One-to-many (multicast) communication

Different message patterns, including:

- Request-response (req-rep)
- Publish-subscribe (pub-sub)
- Pipeline

Sockets have types:

- For a req-rep pattern a REQ socket is connected to a REP socket.
- For a pubsub pattern a SUB socket is connected to a PUB socket.
- For a pipeline pattern a PUSH socket is connected to a PULL socket.

ZeroMQ: Communication patterns

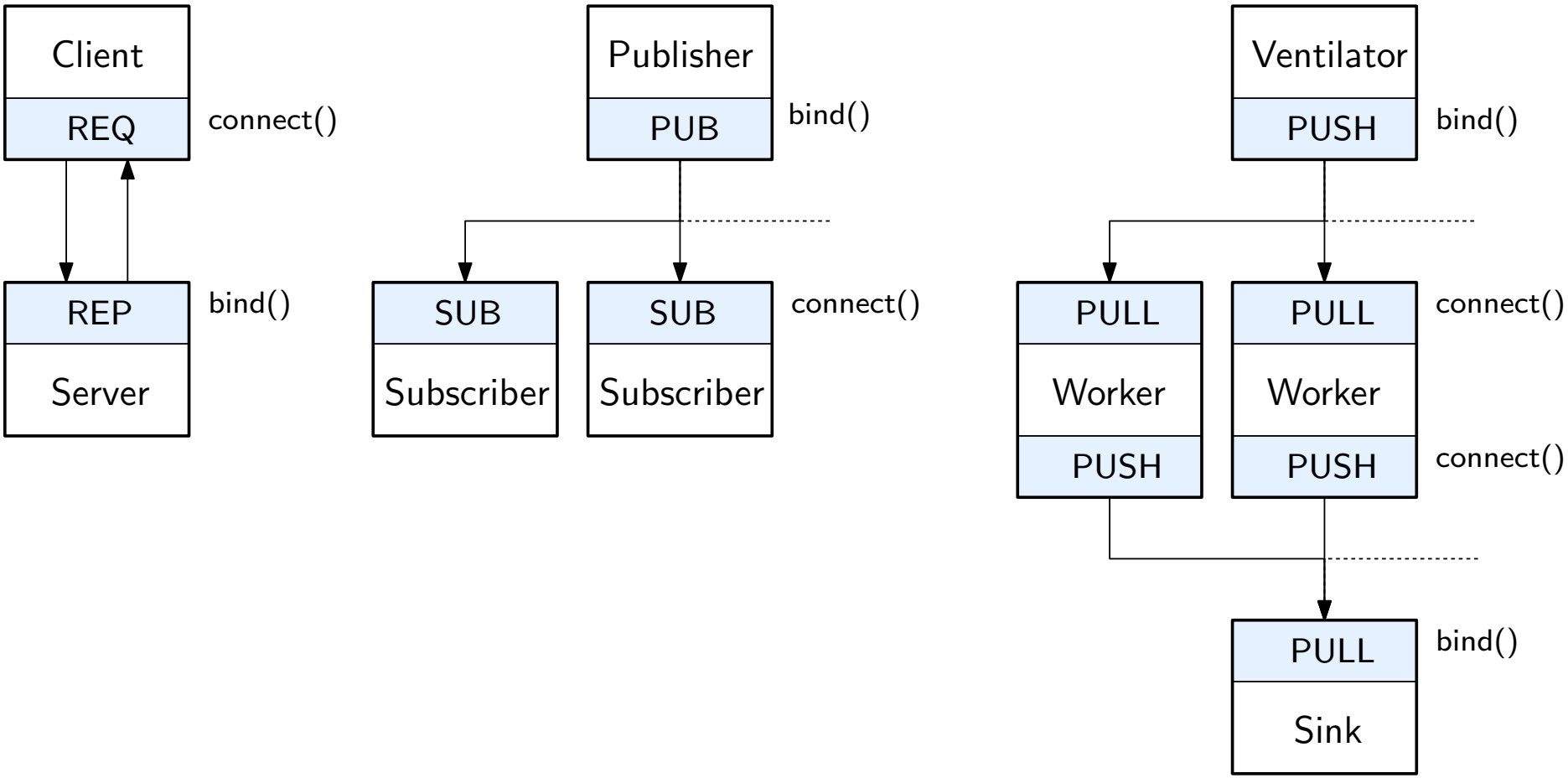


Figure: The three patterns request-response, publish-subscriber and pipeline.

ZeroMQ: Req-rep example

```python
import zmq

if __name__ == '__main__':
    context = zmq.Context()
    # We create a response socket (server) for a req-rep pattern
    s = context.socket(zmq.REP)
    # We can bind on more than one address
    s.bind("tcp://*:5555")
    s.bind("tcp://*:5556")

    while True:
        # Receive message
        message = s.recv()
        print("Received request: %s" % message)
        # Send reply back to client
        s.send(b"* " + message)
```

ZeroMQ: Req-rep example

```python
import sys, zmq

if __name__ == '__main__':
    context = zmq.Context()
    # We create a request socket (client) for a req-rep pattern
    s = context.socket(zmq.REQ)
    # REP socket does not need to be up yet
    s.connect("tcp://localhost:" + sys.argv[1])

    while True:
        req = input("Message> ")
        if len(req) == 0:
            break

        s.send(req.encode())
        msg = s.recv()
        print("> " + msg.decode())
```

ZeroMQ: Pubsub example

```python
import zmq

if __name__ == '__main__':
    context = zmq.Context()
    s = context.socket(zmq.PUB)
    s.bind("tcp://*:5557")

    while True:
        top = input("Topic> ")
        cont = input("Content> ")
        # Pubsub is inherently one-way: There is no reply!
        s.send_string("%s %s" % (top, cont))
```

```python
import zmq

if __name__ == '__main__':
    context = zmq.Context()
    s = context.socket(zmq.SUB)
    s.connect("tcp://localhost:5557")
    # We _must_ have a filter string
    s.setsockopt_string(zmq.SUBSCRIBE, "")

    while True:
        msg = s.recv_string()
        print("Got:", msg)
```

ZeroMQ: Pipeline example

Sink binds a PULL socket:

```
1  import zmq
2
3  if __name__ == '__main__':
4      context = zmq.Context()
5      s = context.socket(zmq.PULL)
6      s.bind("tcp://*:5000")
7
8      while True:
9          msg = s.recv()
10         print("sink > " + msg.decode())
```

ZeroMQ: Pipeline example

Ventilator binds to a PUSH socket:

```
import zmq

if __name__ == '__main__':
    context = zmq.Context()
    disp = context.socket(zmq.PUSH)
    disp.bind("tcp://*:5001")

    sink = context.socket(zmq.PUSH)
    sink.connect("tcp://localhost:5000")

    while True:
        input("Hit return to start batch processing> ")

        sink.send_string("Go")
        for i in range(10):
            print("Send job {}".format(i))
            disp.send_string("job {}".format(i))
```

Fair queuing distributes messages uniformly to all connected workers.

ZeroMQ: Pipeline example

Each of *n* workers connects to a PULL socket towards ventilator and a PUSH socket towards sink:

```
import time, zmq

if __name__ == '__main__':
    context = zmq.Context()
    disp = context.socket(zmq.PULL)
    disp.connect("tcp://localhost:5001")

    sink = context.socket(zmq.PUSH)
    sink.connect("tcp://localhost:5000")

    while True:
        msg = disp.recv_string()
        print("got > " + msg)
        time.sleep(1.0)
        sink.send_string("Done with" + msg)
```

Message-oriented Middleware

ZeroMQ is:

- Advanced, relative to Berkley sockets
- Lightweight, relative to message-oriented middleware or message passing systems.

A message-oriented middleware provides persistent asynchronous communication.

- Further reduces temporal coupling between processes.
- Queue managers provide an intermediate-term storage for messages.
- At the cost of higher administrative work.
- Support for message transfer times in the range of minutes.

Message queuing model

Communication model:

- Communication by inserting messages into queues.
- Messages passed over series of servers until it eventually reaches destination.
- Applications may share queues or each application may have its own queue.

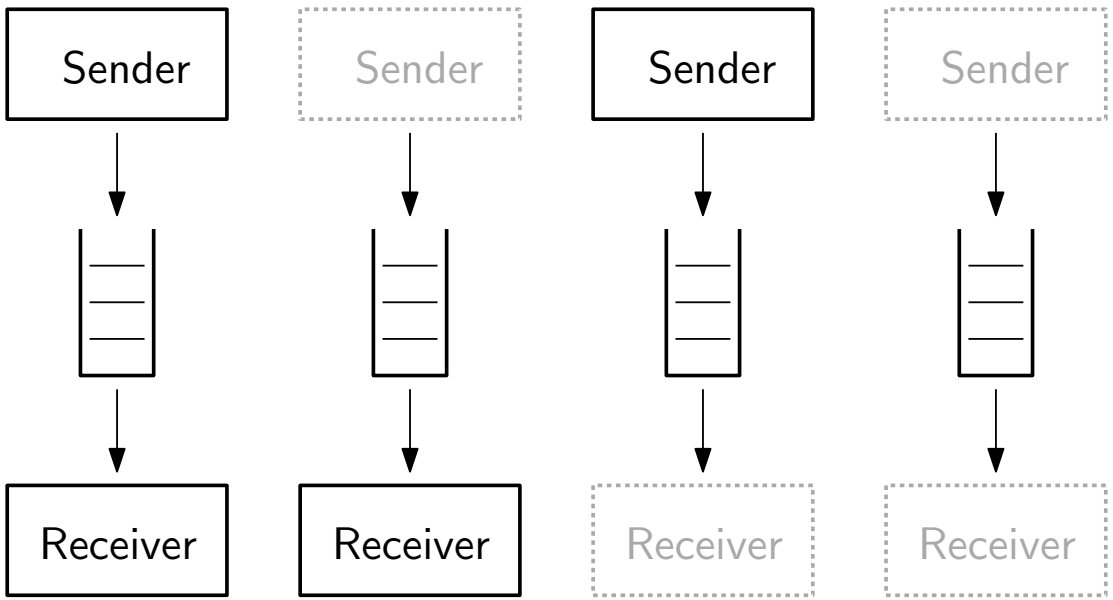
Guarantees:

- There is only one guarantee: Message reaches recipient *eventually*.
- There is no guarantee for item the time until it is received.

Persistent messaging

Messages reach recipient eventually:

- Once a message is in a queue, it is not removed even if sender or receiver are not running.
- Four combination regarding the state of receiver and sender.
- Only if also the fourth combination, both are not running, is supported, we speak of persistent messaging.



AMQP

AMQP:

- Advanced Message Queuing Protocol
- In 2006 AMQP was developed as an open alternative to proprietary solutions.

AMQP refers to:

- A messaging service
- A messaging protocol
- A messaging interface

Tutorials:

- `https://www.rabbitmq.com/tutorials/tutorial-one-python.html` and the following episodes.
- `https://www.rabbitmq.com/tutorials/amqp-concepts.html`

AMQP

AMQP communication:

- Application sets up connection to queue manager.
- Connection contains several one-way channels.
- Channels can be very dynamic, but connections are long-lived.
- A session is a pair of channels for bidirectional communication.

AMQP network:

- An AMQP network consists of nodes and links between them.
- Messages are logically transferred over links.
- Links keep track of message status.
- Credit-based flow control: Specifies the message rate over a link.

AMQP provides persistent messaging.

AMQP: Message passing between nodes

Message passing from an application to queue manager consists of three steps:

1. Message gets unique identifier and is recorded locally in sender's queue in state *unsettled*. Then message is sent to queue manager, which also denotes the unsettled state.
2. Queue manager handles the message and reports back to sender, which brings message into *settled* state.
3. Original sender tells queue manager that message is in settled state, and the sender may forget about the message.

The three steps provide end-to-end communication reliability.

Nodes of a AMQP network:

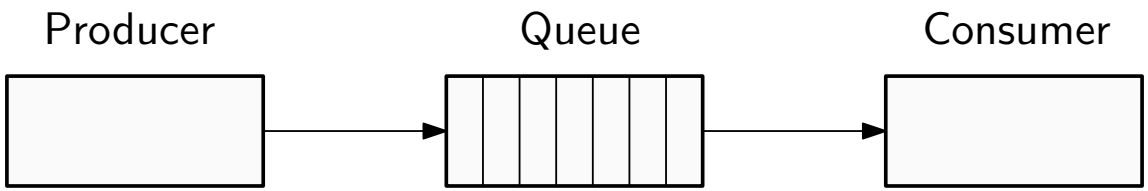
- Three types: Consumer, producer, queue
- A queue manager typically consists of many queue nodes
- Queue managers can be connected to other queue managers, forming an overlay network in which messages are routed.
- Messages can be marked to be durable such that intermediate nodes can recover in case of failures.

AMQP Demo

Software:

- We use RabbitMQ for an AMQP server implementation.
  - It uses tcp/5672 as the standard port.
- We use pika as a Python client implementation.

Our "hello world" demo will look like this:



- The queue is provided by RabbitMQ.
- The producer and the consumer are written in python using pika.

AMQP Demo: Producer

```python
import pika

if __name__ == '__main__':
    # Establish a connection
    conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    ch = conn.channel()

    # Create a queue on the server
    ch.queue_declare(queue='hello')

    cnt = 0
    while True:
        cnt += 1
        # Publish a message
        ch.basic_publish(exchange='',
                         routing_key='hello',
                         body='Hello World {}!'.format(cnt))
        input("Press return for next...")

    conn.close()
```

After the queue has been created, we can list it:

```
# rabbitmqctl list_queues
Timeout: 60.0 seconds ...
Listing queues for vhost / ...
name    messages
hello   1
```

AMQP Demo: Consumer

```python
import pika

def on_message(ch, method, properties, body):
    print("Received > {}".format(body))

if __name__ == '__main__':
    conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    ch = conn.channel()

    # Create a queue, if not existing
    ch.queue_declare(queue='hello')

    # Register a callback that is called when a message is received
    ch.basic_consume(queue='hello',
                     auto_ack=True,
                     on_message_callback=on_message)

    # Start the forever-loop
    ch.start_consuming()
```

We can also start multiple consumers. Or producers.

- Distribution is uniform among consumers; each is getting roughly the same number.
- By setting a *prefetch count* to 1 a consumer only gets a next message after having acknowledged the previous one.

AMQP: Message acknowledgment

```
def on_message(ch, method, properties, body):
    print("Received > {}".format(body))
    input("Press return to continue...")
    # Explicit acknowledgement
    ch.basic_ack(delivery_tag = method.delivery_tag)

if __name__ == '__main__':
    conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    ch = conn.channel()
    ch.queue_declare(queue='hello')
    # No auto_ack=True is passed
    ch.basic_consume(queue='hello',
                     on_message_callback=on_message)
    ch.start_consuming()
```

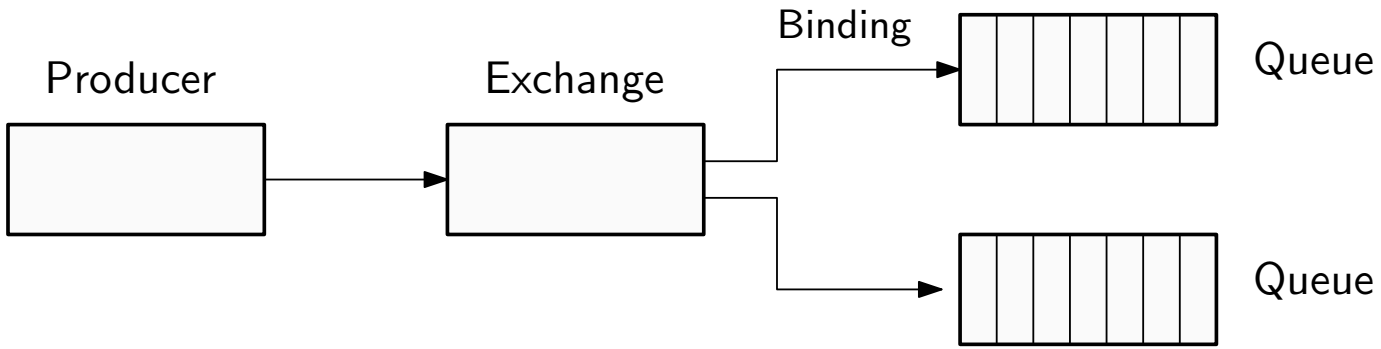
Messages are acknowledged by consumers:

- If channel or connection is dropped or client dies then the message will be sent to another consumer.
  - There is no timeout.
  - Acknowledgment must be sent over the same channel.
  - `rabbitmqctl list_queues name messages_ready messages_unacknowledged`
- By passing `auto_ack=True` to `basic_consume()` acknowledgment is done automatically. Otherwise, we can explicitly invoke `basic_ack()` to acknowledge the message.

AMQP: Exchanges

Exchange:

- A message is actually not directly sent to a queue, but to an exchange.
- An exchange may put a message to zero or more queues.
- The exchange type tells how an exchange acts.
  - The default exchange, which is unnamed, is of type direct.
  - The type fanout can be used for a publish-subscribe pattern.
- Exchanges have names and can be listed: `rabbitmqctl list_exchanges`



AMQP Demo: Log exchange producer

```
if __name__ == '__main__':
    conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    ch = conn.channel()
    # Declare the fanout exchange
    ch.exchange_declare(exchange='logs', exchange_type='fanout')

    cnt = 0
    while True:
        cnt += 1
        ch.basic_publish(exchange='logs',
                         routing_key='',
                         body='Hello everybody {}!'.format(cnt))
        input("Press return for next...")

    conn.close()
```

- We do not publish to a queue, but to an exchange.
- In the previous demo we published to the default exchange and specified a routing key.

AMQP Demo: Log exchange consumer

```python
def on_message(ch, method, properties, body):
    print("Received > {}".format(body))

if __name__ == '__main__':
    conn = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
    ch = conn.channel()
    # Declare the fanout exchange
    ch.exchange_declare(exchange='logs', exchange_type='fanout')
    # Create a temporary queue, but delete it when connection is closed
    res = ch.queue_declare('', exclusive=True)
    queue_name = res.method.queue
    ch.queue_bind(exchange='logs', queue=queue_name)

    # Register a callback that is called when a message is received
    ch.basic_consume(queue=queue_name,
                     auto_ack=True,
                     on_message_callback=on_message)
    ch.start_consuming()
```

- Each consumer creates a temporary, exclusive queue and binds it to the logs exchange.
- Hence, the exchange places a copy of the message in *each* queue, which gives a publish-subscribe pattern.

AMQP: Message routing

An exchange of type direct can perform message routing:

- Bindings have a `routing_key`, and so have messages.
- If the messages key matches the binding key then the exchange places the message into the queue.

Every queue is automatically bound to the default exchange:

- The binding's routing key equals the queue's name.