40.317 Lecture 3

Dr. Jon Freeman

26 May 2020

Slido Event #2248

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Agenda

- Distributed Computing
- Middleware
- ZeroMQ
- Using ZeroMQ in Python

Distributed Computing: Motivation

Why are we studying this topic?

- Every finance company with an in-house development team builds distributed systems.
- Distributed applications are the most typical form of “programming in the large.”

The Vision

We design a system as *a collection of <u>services</u> passing <u>messages</u> around*.

- Typically hosted on multiple machines.
- Each type of service has one clear, important business purpose.
- Each service is relatively easy to think about, troubleshoot, and extend.
- The system meets stated performance goals, and as usage increases we can keep meeting these goals by adding more instances / machines / resources.

The Vision, continued

This approach respects Gall's Law (1975):

> "A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system."

(We must apply Gall's Law together with this law from Fred Brooks, also from 1975:

> "Plan to throw one away, you will anyhow.")

The Vision, continued

We can apply Gall's Law to manage the complexity of the following over time:

- Each individual service
  - its capabilities (public interface)
  - its internal behaviour (private implementation)
- The number of distinct services
- How they interact with each other

Challenges

To achieve this we must tackle several challenges besides correctness and speed:

- Heterogeneity: We must support every relevant operating system, network, and programming language.
- Openness: We must maintain clear, accessible documentation for every version of every service’s interface.
- Concurrency: We must manage conflicting requests for shared resources quickly and fairly.

Challenges, continued

- Security
  - Strong *authentication* to obtain access
  - Appropriate *authorisation*, preferably service by service
- Scalability
  - Adding resources to handle heavier loads must be quick and easy.
  - In fact we should be able to add such resources *dynamically*.
    - Cloud Computing (⇨ Kubernetes ⇨ Software-Defined Everything)

Challenges, continued

- <u>Resilience to Failure</u>
  - Of our <u>network</u>: Messages must be sent reliably when so required.
  - Of our <u>machines</u> and our <u>services</u>: There should be redundant components and other failover strategies in place to ensure acceptable performance *when*, not *if*, one or more components fail.

A Thought Exercise

To better appreciate the difficulty of just one challenge, Resilience to Failure, consider a situation from the Warring States period:

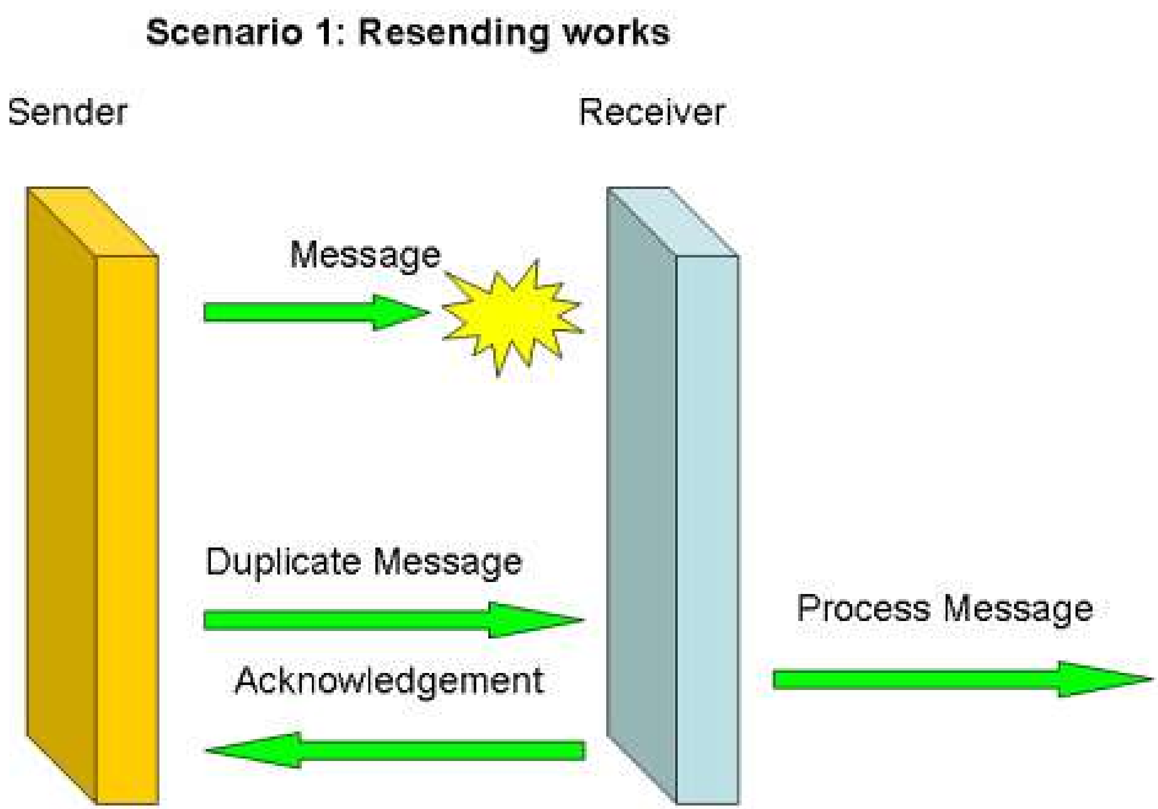
- 王陵 (Wang Ling) needs to contact 王齕 (Wang He) and receive a reply.
- The travel time between is up to ½ day.
- Messengers can be captured or killed.
- 王陵 sends one messenger, who does not return after one day.
- *Now what?*

How We Expect Messaging to Work

When we send a message from one service to another, we are seeking either or both of the following guarantees:

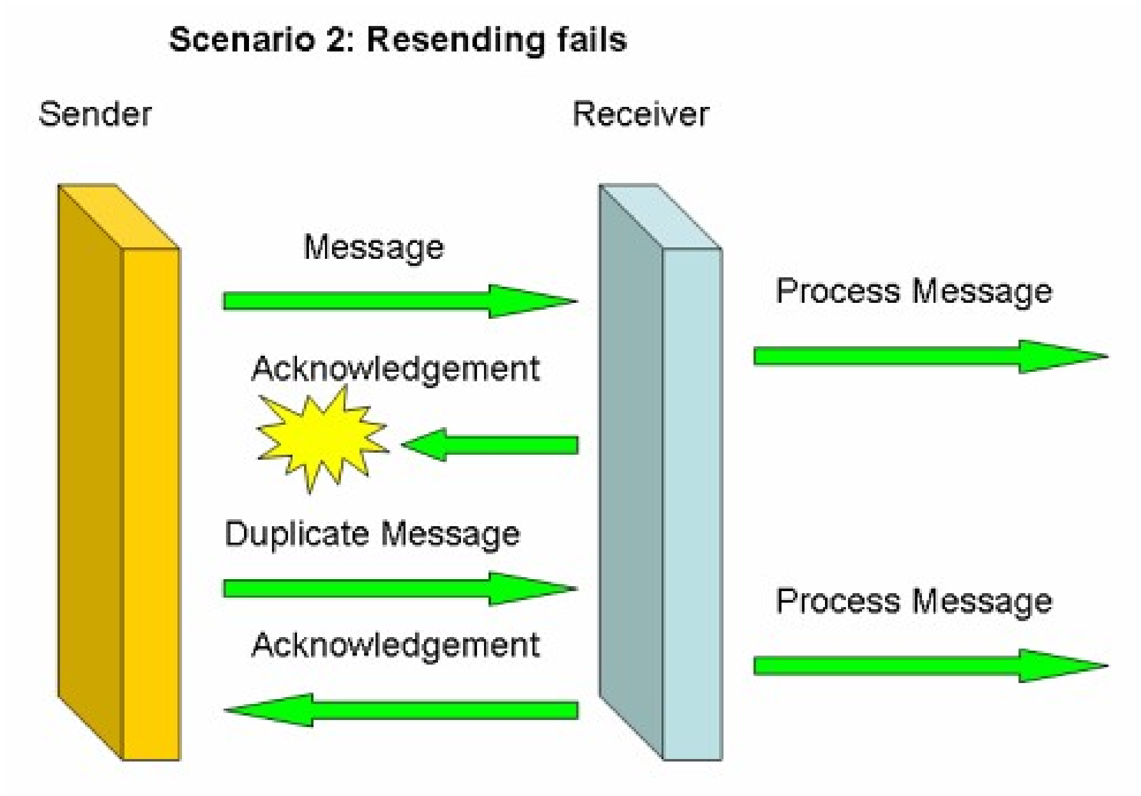
- Each *individual* message arrives [pick one]:
  - at least once
  - at most once
  - exactly once
- Each *sequence* of messages arrives in order

Reliable Messaging: First Attempt



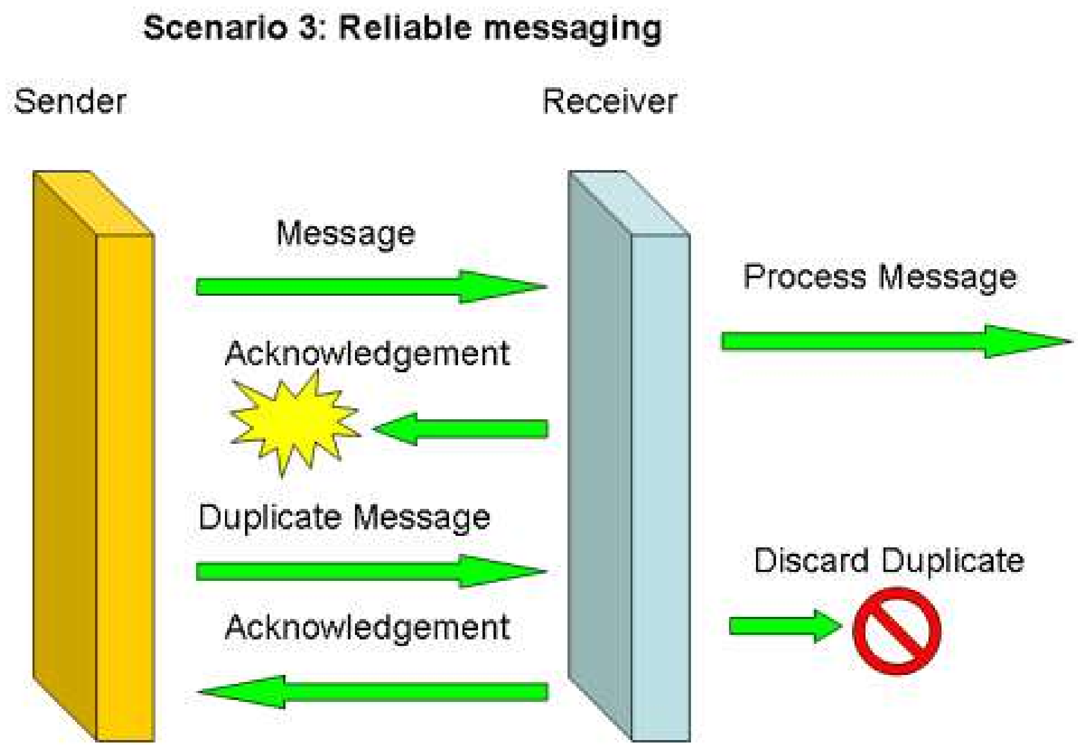
Source: https://www.infoq.com/articles/no-reliable-messaging

Reliable Messaging: Second Attempt



Source: https://www.infoq.com/articles/no-reliable-messaging

Reliable Messaging: Third Attempt



Source: https://www.infoq.com/articles/no-reliable-messaging

Typical Messaging Objectives

The messages sent inside a distributed system typically accomplish one of the following three objectives:

- Point-to-point communication
- Data distribution
- Workload distribution

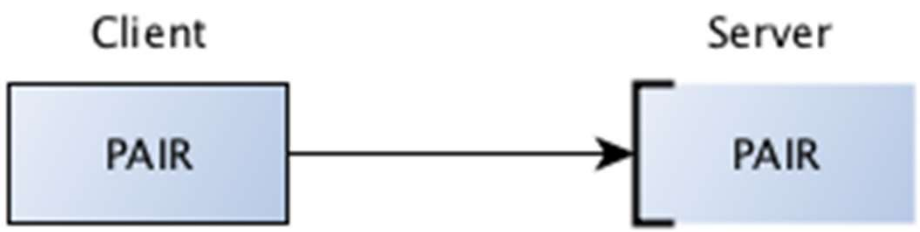
Typical Messaging Topologies

There are certain topologies, i.e. connection patterns, which occur again and again in distributed systems. We will examine four:

- Exclusive pair
- Request-reply
- Publish-subscribe
- Pipeline

Exclusive Pair Topology

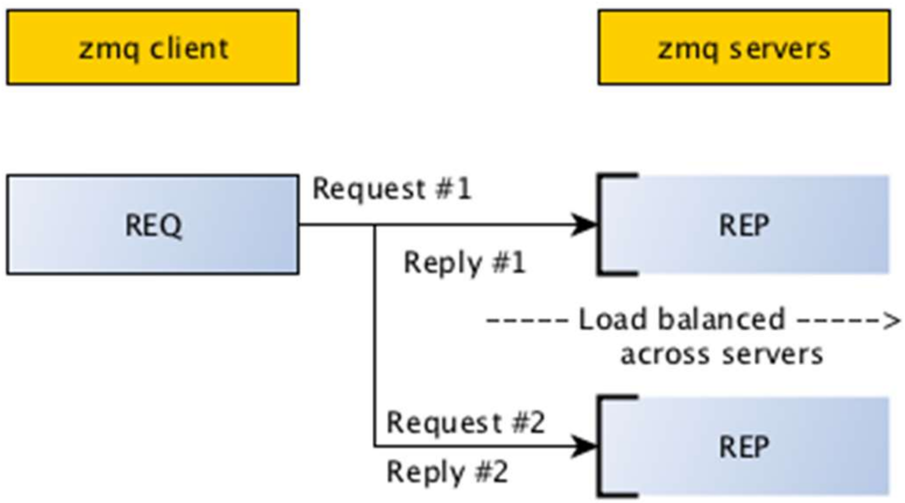
An *exclusive*, *bidirectional* connection between two peers.



Source: http://learning-0mq-with-pyzmq.readthedocs.io/en/latest/

Request-Reply Topology

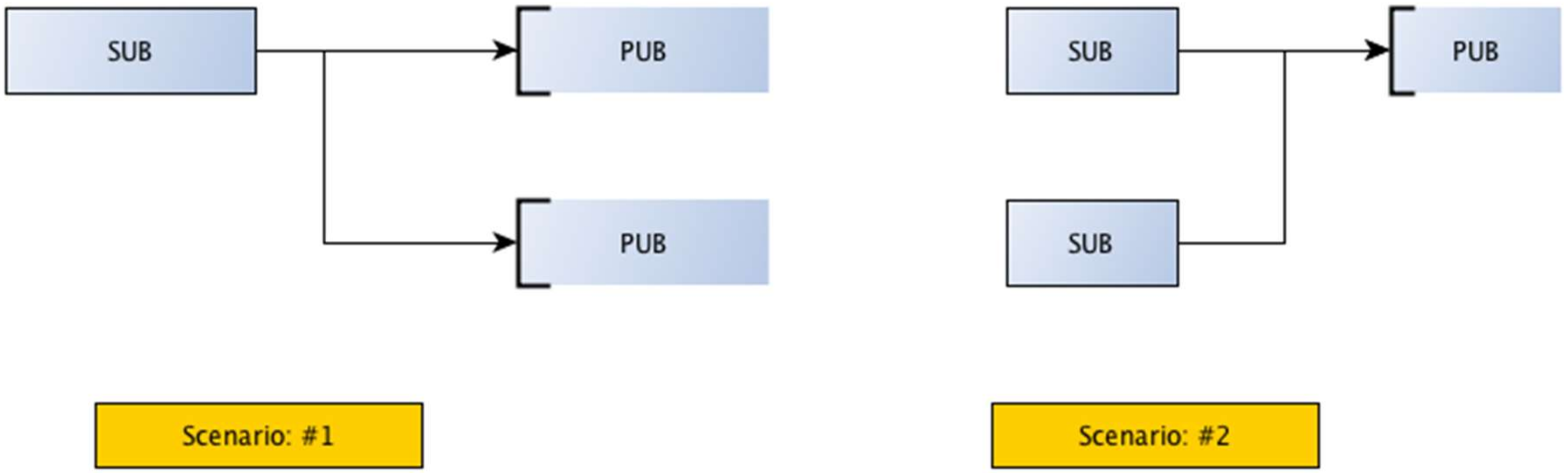
Connects a client to one or more servers.
Requests and replies must strictly alternate.



Source: http://learning-0mq-with-pyzmq.readthedocs.io/en/latest/

Publish-Subscribe Topology

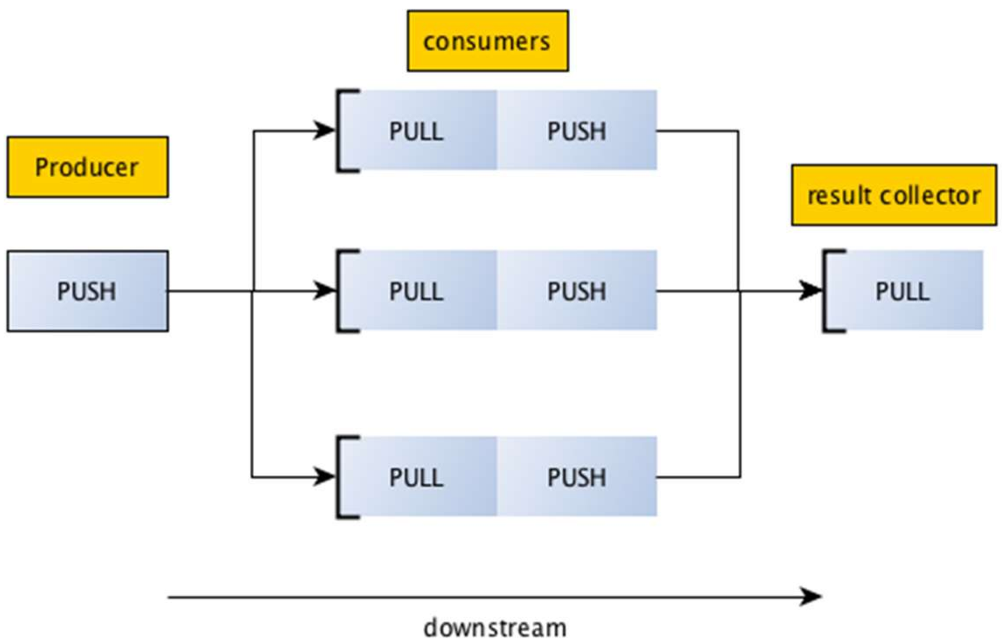
One-to-many or many-to-one connections between a set of publishers and a set of subscribers. A *data distribution* pattern.



Source: http://learning-0mq-with-pyzmq.readthedocs.io/en/latest/

Pipeline Topology

A fan-out/fan-in pattern which can have multiple stages, and even loops. A *parallel* task *distribution* and *collection* pattern.



Source: http://learning-0mq-with-pyzmq.readthedocs.io/en/latest/

Middleware

A standalone product for routing messages between the components of a distributed system.

We expect middleware to address three of the five challenges we listed earlier, namely:

- Heterogeneity
- Scalability
- Resilience to Failure

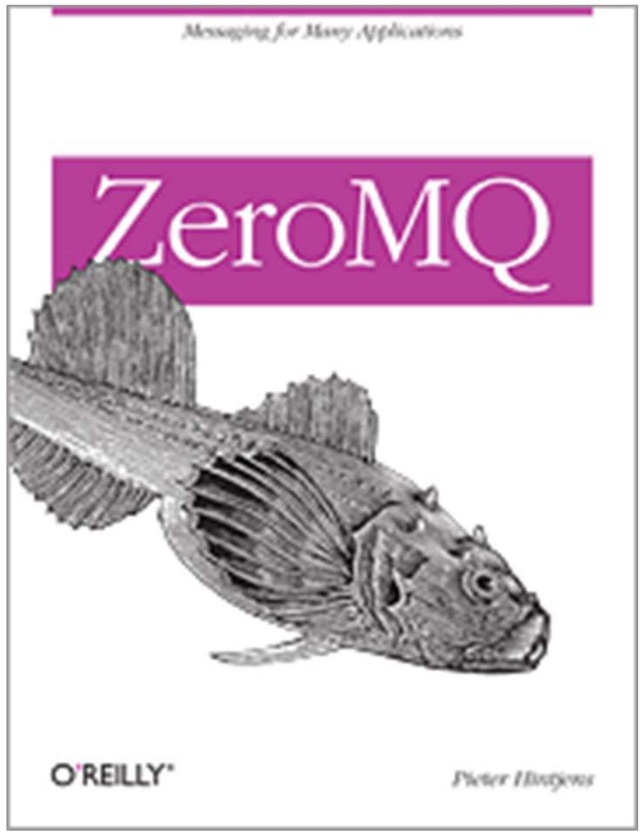
Popular Products

Popular middleware products currently include:

- RabbitMQ
- Amazon Simple Queue Service (SQS)
- Iron MQ
- Redis
- Beanstalkd
- …

Popular Products, continued

- ... and ZeroMQ (or “ØMQ”), *our recommendation and our focus.*



Is There a Standard Protocol?

There are several open specifications:

- AMQP, Advanced Message Queuing Protocol
- STOMP, Streaming Text Oriented Messaging Protocol
- XMPP, Extensible Messaging and Presence Protocol
- MQTT, a lightweight pub-sub protocol
- OpenWire, used by ActiveMQ

Many of ZeroMQ's design choices were a reaction to choices made for AMQP.

The Alternatives to ZeroMQ

Most alternatives to ZeroMQ require the existence of a centralised message broker which *routes* and *queues* messages.

Such a broker:

- is expensive to purchase and maintain
- causes network activity to double
- is a performance bottleneck
- *is a single point of failure*

ZeroMQ’s Big Difference

ZeroMQ is peer-to-peer, i.e. it does *not* require a central server.

Instead, it “pushes routing to the publisher edge, and queueing to the consumer edge.”

ZeroMQ's Philosophy

- Writing massively connected applications ought to be easy.
- Removing complexity is better than exposing new functionality.
- Performance is not optional.
- State should not be shared, i.e. messages should be self-contained.

What ZeroMQ Can Do

- Delivers data blobs (messages) to <u>nodes</u> with high throughput and low latency
  - < 5 secs to receive and filter 10M msgs!
- Provides a single API to work with, regardless of transport type
  - For all popular programming languages
- Automatically reconnects to peers as they come and go
- Queues messages at both sender and receiver, as needed

What ZeroMQ Can Do, continued

- Manages queues carefully, overflowing to disk when required
- Handles socket errors
- Does all I/O in background threads
- Uses lock-free techniques for talking between nodes

And last but not least,

- Has built-in support for the four common topologies / patterns mentioned earlier

What ZeroMQ Cannot Do

Its main limitations are lack of support for:

- *Guaranteed* message delivery, i.e. message delivery as a <u>transaction</u>

- Persistent queues

And one secondary limitation:

- ZMQ sockets are not thread-safe.
  - (Only ZMQ “contexts” are thread-safe.)

Installing ZeroMQ for Python 3

Anaconda 3 includes it, so if you are using Anaconda you just:

```
import zmq
```

which actually loads two things:

1. The core ZeroMQ library
2. Python-specific bindings

ØMQ Demos: Exclusive Pair

One client and one server.

- `zmq_pair_client.py`
- `zmq_pair_server.py`

ØMQ Demos: Request-Reply

One client sending requests to multiple servers.

- `zmq_reqrep_client.py`
- `zmq_reqrep_server.py`

ØMQ Demos: Publish-Subscribe

There are two possibilities, and ZeroMQ supports them both:

1. [More common] One publisher, many subscribers
2. [Less common, and interesting] Many publishers, one subscriber
   - `zmq_pubsub_publisher.py`
   - `zmq_pubsub_subscriber.py`

ØMQ Demos: Pipeline

One source; a single stage, with multiple nodes; and one sink.

- `zmq_pipeline_source.py`
- `zmq_pipeline_stage.py`
- `zmq_pipeline_sink.py`

More About Reliable Messaging

- It is usually not practical to *guarantee* reliable messaging *at the transport level*.
- Instead, we seek to achieve *high* reliability in the context of a particular reliability strategy.
- No reliability strategy is "the best." Instead we adopt the strategy which most closely matches the semantics of the work we need to do.

The Two Main "Semantics"

- <u>Pessimistic synchronous dialogue</u>: The receiver acknowledges every message with a success or failure response.

- <u>Optimistic asynchronous monologue</u>: The sender pushes data to the receiver as quickly as possible, not expecting an acknowledgement.

Three Main Reliability Strategies

- Request-response: The client has a retry mechanism; the service can detect and deal with duplicate requests.
- *Transient* publish-subscribe: If data is lost, subscribers simply wait for fresh data to arrive.
  - E.g. Video or voice streaming
- *Reliable* publish-subscribe: Subscribers acknowledge data using a low-volume reply back to the sender; the publisher resends data if it needs to.

Guaranteeing Message Delivery

It turns out we *can* guarantee reliable delivery, by working at the application level rather than the transport level.

- Consider a system which sends prescriptions from doctors to chemists.
- Every prescription *must* be processed exactly once!!
- We can ensure this if every message contains a unique “Prescription ID.”
- *These IDs can only come from the app logic, not from the middleware alone.*

Discussion Questions (1/2)

In this example, how can we ensure the system generates a *guaranteed unique* Prescription ID for each prescription?

- Multiple doctors might be trying to send a prescription at the same time.

Discussion Questions (2/2)

Is the interaction between a coffee shop and its customer synchronous or asynchronous?

Recommended Reading

- Lessons learned from designing AMQP: https://github.com/imatix/imatix.github.io/tree/master/articles:whats-wrong-with-amqp
- The ZeroMQ Guide for Python: http://zguide.zeromq.org/py:all
- "Nobody Needs Reliable Messaging:" https://www.infoq.com/articles/no-reliable-messaging/

Thank you

A BETTER WORLD BY DESIGN.

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN