COMP2207
Loose Coupling

Professor Timothy Norman

Department of Electronics and Computer Science, University of Southampton
T.J.Norman@soton.ac.uk

Academic Year 2017/18
Loose Coupling

- We aim to remove our dependency on a specific language
- ... but, can we also relax the coupling between client and server at the same time?

Communicate through an intermediary
- Introduce infrastructure components that
  - Allow senders and recipients of messages to be decoupled wrt. identity
  - Allow the sending and receiving of messages to be decoupled wrt. lifetime
  - ⇒ more flexibility (easier to replace or restart component)

This also makes one-to-many communication easier
We aim to remove our dependency on a specific language

... but, can we also relax the coupling between client and server at the same time?

Communicate through an intermediary

Introduce infrastructure components that

- Allow senders and recipients of messages to be decoupled wrt. identity
- Allow the sending and receiving of messages to be decoupled wrt. lifetime
- ⇒ more flexibility (easier to replace or restart component)

This also makes one-to-many communication easier
## Uncoupling in Space and Time

<table>
<thead>
<tr>
<th></th>
<th><strong>Time coupled</strong></th>
<th><strong>Time uncoupled</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space coupled</strong></td>
<td>Communication directed towards given receiver(s). Receiver(s) must exist at that moment in time. Examples: message passing, remote invocation.</td>
<td>Communication directed towards given receiver(s). Sender and receiver(s) can have independent lifetimes.</td>
</tr>
<tr>
<td><strong>Space uncoupled</strong></td>
<td>Sender does not need to know <strong>identity</strong> of receiver(s). Receiver(s) must exist at that moment in time. Example: IP multicast</td>
<td>Sender does not need to know <strong>identity</strong> of receiver(s). Sender and receiver(s) can have independent lifetimes.</td>
</tr>
</tbody>
</table>
Advantages of Space/Time Uncoupling

- **Space uncoupling**
  - Participants can be replaced, updated, replicated or migrated

- **Time uncoupling**
  - Senders/recipient may come and go
  - Useful for mobile environments

- **Space and time uncoupling**
  - Useful for dissemination in systems where receivers are unknown/liable to change; e.g. event feeds in financial systems
Group Communication

- Message sent to group
- Sender unaware of recipients
  - Processes may join/leave group
  - Single multicast operation for sending a message (array of bytes)
- Usually implemented over IP multicast, with added guarantees
- Applications include:
  - Reliable dissemination of information to a large number of clients
  - Collaborative applications — serves to preserve common user view
  - Consistent update of replicated data
- JGroups (www.jgroups.org) is an example of group communication middleware — see Coulouris et al. for an overview
Advantages of Group Communication

- A single multicast operation may be more **convenient** for the programmer and more **efficient**
- Infrastructure may offer better **reliability** and support **message ordering**
- May have different **types of group** (important for reliability/ordering):
  - Only group members can multicast to a **closed group**
  - Any process can multicast to an **open group**
- **Synchronous systems**
  - Real-time **diffusion** of messages to group
  - Requires bounds on **communication delays, process scheduling delays** and **faulty components**, and requires sufficient **redundancy** in communication paths between processes
- **Asynchronous systems**
  - Message exchange via **datagram** — messages may get lost, and hence systems are characterised by **communication uncertainty**
  - For many problems (such as consensus, see distributed algorithms), we require bounds on **timeout delays** an assume timely communication most of the time
Implementation Issues

- **Reliability**
  - **Integrity** — message received is *identical* to the one sent, and no message is delivered twice
  - **Validity** — every message is *eventually* delivered
  - **Agreement** — if a message is delivered to one group member, then it is delivered to all members

- **Ordering**
  - **FIFO (source) ordering** — sender order is preserved
  - **Causal ordering** — causality order between messages is preserved (e.g. message $m_2$ sent in response to $m_1$ is received by all group members after $m_1$)
  - **Total ordering** — all messages delivered to all processes in the same order, regardless of causality
Group Communication Middleware

Group address expansion

Group send

Multicast communication

Leave

Fail

Join

Group membership management

Process group
Group Membership Management

- We require an interface for group membership management
  - Create/delete a group
  - Add/remove a process from a group
- Failure detection
  - Monitor group members
  - Respond to process crashes, or unreachable members
- Notify group members of membership changes
- Perform group address expansion prior to message delivery
Publish-Subscribe Systems

- Most **widely used** form of indirect communication
- Also known as **distributed event-based systems**
- **Publishers** publish events to an **event service**
- **Subscribers** express interest in particular **events**
- Publish-subscribe system **delivers** event notifications

**Applications**
- Financial information systems
- Cooperative working systems
- Monitoring applications
- Asynchronous, cloud-based workflows
- Google Ads
Example Infrastructure: Google Cloud Pub/Sub

- Many-to-many *asynchronous* communication
- **Heterogeneity**: designed for applications with *components* written by different developers
- Cloud-based location enables *low latency*
Example Application: A Dealing Room System

- Market price for individual stocks represented by a single object
- Information provider receives updates from single external source
- Events delivered to all dealers interested in corresponding stock

Tim Norman (ECS)
Events are published to **topics** advertised by publishers

Topic **subscribers** are notified of **events** via messages

If a subscriber is not running, events will be **queued** until it rejoins
Pub/Sub Models

- **Subscription models**
  - **Channel-based** Events are published to named channels; subscribers receive all events sent to channel
  - **Topic-based** Events contain topic information; subscriptions identify topics of interest
  - **Content-based** Subscribers use queries (constraints over values of event attributes) to define content-based filters

- **Implementation models**
  - **Centralised** A single broker interacts with publishers and subscribers via point-to-point messages — lack of resilience and scalability
  - **Distributed** Network of cooperating brokers; also referred to as federated
  - **Peer-to-peer** Some fully peer-to-peer implementations possible
Message Queues

- A message queue provides indirection: space/time uncoupling
- (Traditionally) a point-to-point service
  - Sender places message in a queue
  - A single receiver removes the message from the queue
- Provides loose coupling suitable for integrating applications within an enterprise
RabbitMQ

- A more modern take on distributed systems infrastructure
- “Polyglot”
  - Java
  - Node.js
  - Erlang
  - PHP
  - Ruby
  - .NET
  - Haskell
  - . . . even COBOL
RabbitMQ

- A more modern take on distributed systems infrastructure
- "Multi-Protocol"
  - Advanced Message Queuing Protocol (AMQP) — an open standard application layer protocol
    - Message-oriented
    - Queuing (hence MQ support)
    - Routing (point-to-point or pub/sub)
    - Reliability and security
  - Message Queue Telemetry Transport (MQTT) — ISO standard
    - Pub/sub-based protocol
    - Designed to support small code footprint locations
    - Designed to support low bandwidth connections
  - Simple Text-Oriented Messaging Protocol (STOMP)
    - Similar to HTTP
    - Commands such as CONNECT, SEND, SUBSCRIBE, UNSUBSCRIBE
Example: Instagram

**INSTAGRAM STACK**

- **RabbitMQ** for scalable messaging
- **Celery** for distributed task queue
- **Django** web framework
A Simple RabbitMQ Producer (Java)

```java
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
public class Send {
    private final static String QUEUE_NAME = "hello";
    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        String message = "Hello World!";
        channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
        System.out.println("Sent: "+message);
    }
}
```
A Simple RabbitMQ Consumer (Python)

```python
#!/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("Received %r" % body)

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

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

- The RabbitMQ tutorials provide examples of different communication patterns
- Demonstrates flexibility of infrastructure
- Pub/sub, topic-based routing, even RPC
Summary

- **Indirect communication**
  - Group communication
  - Point-to-point
  - Publish and subscribe

- **RabbitMQ**
  - Offers **scale and flexibility**
  - Language agnostic ("polyglot")
  - Support for multiple underlying protocols
  - Can enable numerous communication patterns

- **We now have**
  - No longer tied to **Java**
  - Have not lost **support structures**
  - Can operate at **scale**
  - Have no direct **coupling**
    - Client → **producer**
    - Server → **consumer**