AARHUS UNIVERSITET

Software Architecture in Practice

Connectors

Henrik Bærbak Christensen

AARHUS UNIVERSITET

Motivation

- *A good craftsman have a large toolbox*
- Connectors are central for many QA
  - Perhaps me, but I postulate that architects are pretty conservative in choice of connector (including myself ☺)
    - 'We talk REST' or 'We talk gRPC'
      - Last decade it was 'We talk SOAP'
      - Before that it was .Net Remoting and Java RMI
      - Before that it was CORBA (?)

The same connector type: RPC

- Remember
  - Connectors are not just 'the wire', it is the protocol and the 'driver' code in each end of the wire…

Decoupling Middleware

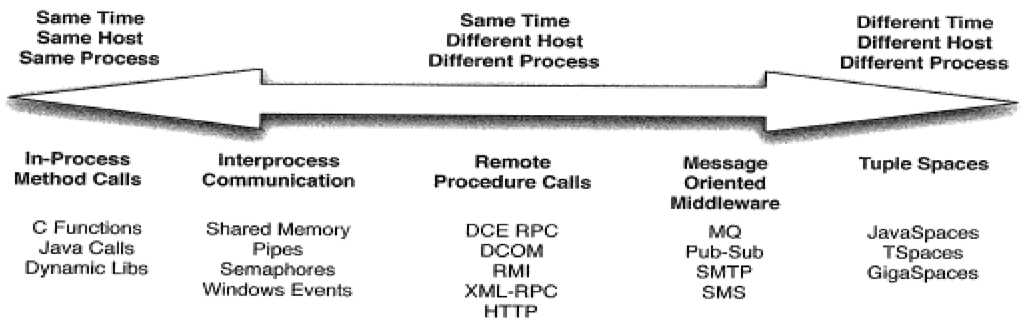


Figure 5.4: Coupling Spectrum of Middleware

- Middleware decisions effect the implementation cost of systems significantly
  - Learn many architectural styles to ensure you pick the right one

Source: Nygard 2017, “Release it”. (but little info in the chapter except the diagram.)

AARHUS UNIVERSITET

The Axes

The Axes

- Same time – different time
  - Same time: both sender and receiver is synchronized
    - Alias a **synchronous** call
  - Different time: sender and receiver operate at own pace
    - Alias an **asynchronous** call
- Same process – different process
  - In the same JVM or across different JVMs
    - Or ‘processes’/’programs’
- Same host – different host
  - On the same machine or across machines
    - Remote or local

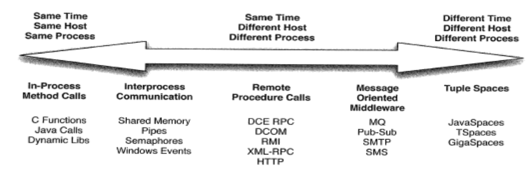


Figure 5.4: Coupling Spectrum of Middleware

AARHUS UNIVERSITET

In-Process Method Calls

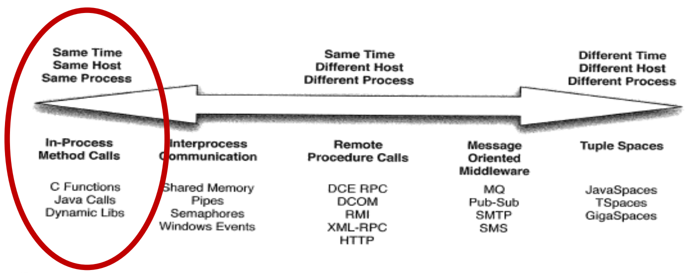


Figure 5.4: COUPLING SPECTRUM OF MIDDLEWARE

AARHUS UNIVERSITET

Method Call

- Synchronous
  - account.deposit(7);

- Java does not excel in async, but has library support

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> factorial(number));
while (!completableFuture.isDone()) {
    System.out.println("CompletableFuture is not finished yet...");
}
long result = completableFuture.get();
```

- JavaScript, Go and C#'s language support is way better…

Shared Memory

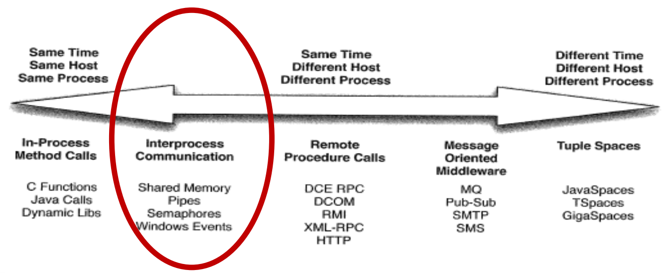


Figure 5.4: Coupling Spectrum of Middleware

AARHUS UNIVERSITET

Linux

4 - File structure

```
/dev/shm
```

shm / shmfs is also known as tmpfs.

tmpfs means temporary file storage facility. It is intended to appear as a mounted file system, but one which uses virtual memory instead of a persistent storage device.

- In this mounted file system, files are stored *in-virtual-memory*, not on the disk-based file system
  - Much faster than spinning disks, and faster than SSD
  - But of course does not survive reboots…

AARHUS UNIVERSITET

Simple Example

```java
// From https://stackoverflow.com/questions/25396664/shared-memory-between-two-jvms
// A Memory Mapped File - shared memory communication between Java processes

import java.io.*;
import java.nio.*;
import java.nio.file.*;
import java.nio.channels.*;
import java.nio.channels.*;

public class MMServer {

  public static void main( String[] args ) throws Throwable {
    File f = new File("/dev/shm/sharedmem");

    FileChannel channel = FileChannel.open( f.toPath(),
                                            StandardOpenOption.READ,
                                            StandardOpenOption.WRITE,
                                            StandardOpenOption.CREATE );

    MappedByteBuffer b =
      channel.map( FileChannel.MapMode.READ_WRITE, 0, 4096 );
    CharBuffer charBuf = b.asCharBuffer();

    char[] string = "Hello client\0".toCharArray();
    charBuf.put( string );

    System.out.println( "Waiting for client." );
    while( charBuf.get( 0 ) != '\0' );
    System.out.println( "Finished waiting." );
  }
}
```

```java
public class MMClient {

  public static void main( String[] args ) throws Throwable {
    File f = new File( "/dev/shm/sharedmem" );
    FileChannel channel =
      FileChannel.open( f.toPath(),
                        StandardOpenOption.READ,
                        StandardOpenOption.WRITE,
                        StandardOpenOption.CREATE );

    MappedByteBuffer b =
      channel.map( FileChannel.MapMode.READ_WRITE, 0, 4096 );
    CharBuffer charBuf = b.asCharBuffer();

    // Prints 'Hello server'
    char c;
    while( ( c = charBuf.get() ) != 0 ) {
      System.out.print( c );
    }
    System.out.println();

    charBuf.put( 0, '\0' );
  }
}
```

```
csdev@m1:~/proj/programmingkata/learn-memory-map-buffer$ java MMServer
Waiting for client.
Finished waiting.
csdev@m1:~/proj/programmingkata/learn-memory-map-buffer$
```

```
csdev@m1: ~/
csdev@m1:~/proj/programmingkata/learn-memory-map-buffer$ java MMClient
Hello client
csdev@m1:~/proj/programmingkata/learn-memory-map-buffer$
```

Remote Procedure/Method Call

RPC, RMI, REST, gRPC, …

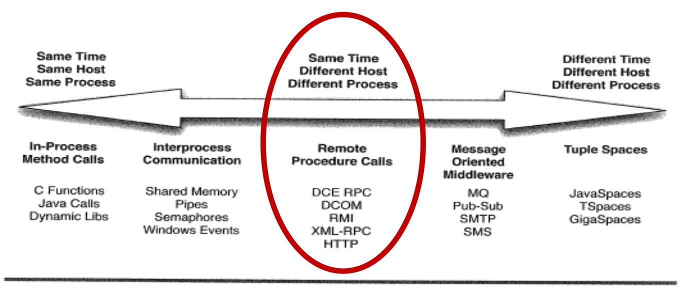


Figure 5.4: Coupling Spectrum of Middleware

Introduction

Explicit Invocation

- RPC suits the client-server style
  - Client sends **request** to **remote server** (and await reply)
  - Server compute answer and sends **reply** back to client
- **Broker** Pattern
  - Mimic modern object oriented style programming
    - Use programmatic interface via a Client Proxy
- Examples
  - CORBA, DCOM, .NET, Java RMI, gRPC, SOAP+WSDL
- But…
  - Could code it using raw Socket, raw HTTP, raw RS232, …

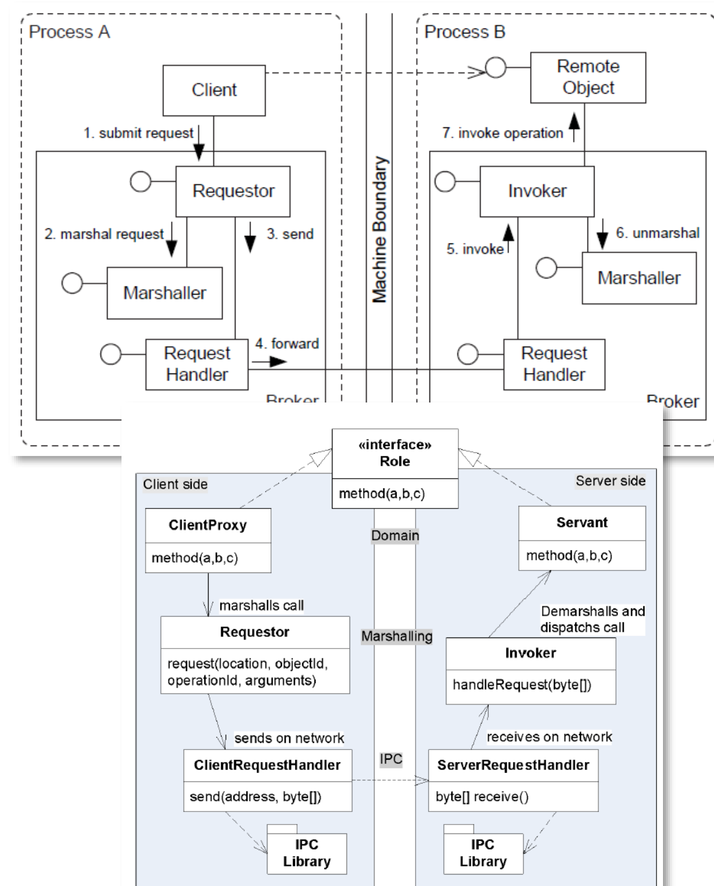
Key Characteristics

- It is a **same time** connector
  - The server must be present when the client makes it request
    - Availability QA is a problem with RPC
  - High coupling between client and server – direct call
    - Request-reply protocol; server ‘address’ must be known
- It is **control oriented**
  - You ask for something to be done, usually handing over **control**
    - Asynchronous is of course less ‘control’ oriented
- It is **event oriented**
  - A request is an **event** that it is important to handle
    - Ala, do not drop events, as the system is then not reliable

The Pattern

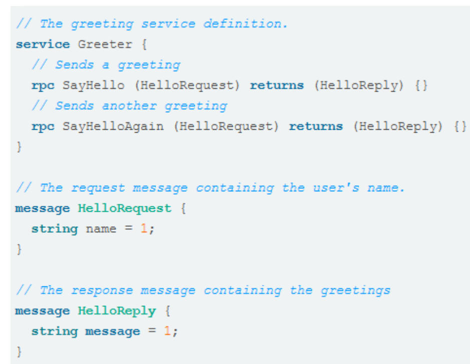
- Repeating what you know

- The Demise of Broker
  - My opinion
    - The tooling killed it
    - The P2P killed it
    - The lack of QA control killed it



Broker Post-Mortem

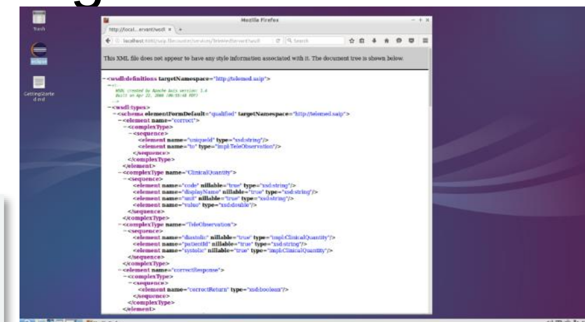
- The tooling! Became much to heavy weight
  - WSDL + SOAP
  - Corba IDL
  - gRPC .proto files
- I.e. you *generate* code from another source format
  - Keeping two disjoint code bases in sync is not *agile* and hinder refactoring…



```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```



Broker Post-Mortem

- RMI is not a client-server architecture, it is Peer2Peer
  - Clients can pass local (remote) objects *by reference* to the server
  - That is, **servers can call methods on client objects !!!**
- Why is this really, really, bad?
  - A) it is no more a client-server architecture, it is p2p
  - B) no control of performance on server
    - Server needs to call methods on 1.000 remote, slow, clients
  - C) hard security issues
    - Will you let a server call code on your machine? Naah
      - Java 7 totally broke the existing RMI framework!
  - D) designers does not get slapped when making ‘big ball of mud’ architectures

Broker Post-Mortem

- You are mostly stuck with auto-generated code, having little QA control
  - i.e. if there is an getX() and getY() method, then each will become a remote call
    - Return to ‘chatty interface’ issue in next course!

- Option is to redesign your interfaces to it is avoided
  - That is, good design is second to tooling ideas ☹

FRDS.Broker

- Sales pitch
  - FRDS.Broker is *real* client-server. Server cannot call methods on client objects. Period! (Similar to REST)
  - No autogenerated code
    - Con: you have to code client proxies and invokers your self
    - Pro: **Now you can control QA explicitly**
- Example
  - Client proxy ‘getX()’ will do nothing if local cache is less than 10 seconds old
  - Otherwise, it will bulkload (x,y,z, ….) and cache them
  - **Bulky interface**

Sales Pitch

- Everybody was flocking towards REST as the silver bullet
  - It is architecturally really attractive for scalable systems
  - But it is programmatically unattractive
    - Very low level coding paradigm
    - All abstractions levels (networking/marshalling/domain) are ‘one big ball of mud’
- So consider going for ‘hand coded’ brokers instead
  - Same architectural qualities
  - But much nicer programming model
    - And clear separation of abstraction levels…

AARHUS UNIVERSITET

REST

- ReST is an architectural pattern, with emphasis on QA like performance and scalability

- But at its heart it is still a RPC connector.
  - Same time, control oriented, event type

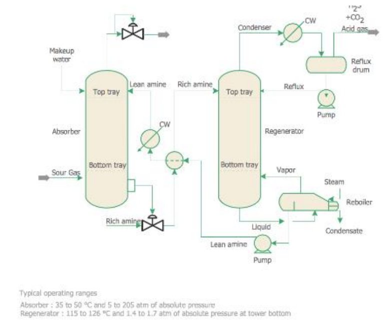
- More detail later…

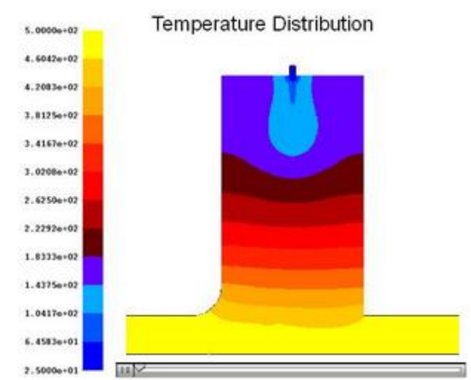
Our Case Study

A Temperature Monitoring System

Chemical Plant

- Case Study: *Chemical Plant*
  - *Sensor:*
    - *Read temperature of chemical process*
  - *Monitor:*
    - *Show temperature on graphical display*
    - *Alerting if temperature gets high*
    - *Emergency actions if T gets critical*
- Connector *between source and dest.*
  - Produce values (from sensor 217)
  - Consume values (from sensor 217)
- *Note: A bit unfair to RPC, not the normal use case ☺*





RPC

- Rest server: Measure T every 1.5 s; store internally
- Rest monitor: RPC to fetch T (as JSON) every 2.5 s

```
csdev@m1: ~/proj/evuproject/connectors
csdev@m1: ~/proj/evuproject/connectors 149x18
2020-11-12T11:02:40.752+01:00 [INFO] rest.RestServer :: Sampling temperature: 102.8134017662073
2020-11-12T11:02:42.253+01:00 [INFO] rest.RestServer :: Sampling temperature: 102.74159099008216
2020-11-12T11:02:42.463+01:00 [INFO] rest.RestServer :: RPC call from client...
2020-11-12T11:02:43.755+01:00 [INFO] rest.RestServer :: Sampling temperature: 103.26950606697552
2020-11-12T11:02:44.969+01:00 [INFO] rest.RestServer :: RPC call from client...
2020-11-12T11:02:45.256+01:00 [INFO] rest.RestServer :: Sampling temperature: 102.80007462742068
2020-11-12T11:02:46.759+01:00 [INFO] rest.RestServer :: Sampling temperature: 102.47493680865954
2020-11-12T11:02:47.482+01:00 [INFO] rest.RestServer :: RPC call from client...
2020-11-12T11:02:48.261+01:00 [INFO] rest.RestServer :: Sampling temperature: 102.82209273649657
2020-11-12T11:02:49.762+01:00 [INFO] rest.RestServer :: Sampling temperature: 103.51399903376408
2020-11-12T11:02:49.992+01:00 [INFO] rest.RestServer :: RPC call from client...
2020-11-12T11:02:51.265+01:00 [INFO] rest.RestServer :: Sampling temperature: 103.76336140774508
2020-11-12T11:02:52.508+01:00 [INFO] rest.RestServer :: RPC call from client...
2020-11-12T11:02:52.767+01:00 [INFO] rest.RestServer :: Sampling temperature: 104.40998278500773
2020-11-12T11:02:54.270+01:00 [INFO] rest.RestServer :: Sampling temperature: 105.14712379612932
<===========--> 87% EXECUTING [33s]
> :rest:restserver

csdev@m1: ~/proj/evuproject/connectors 149x18
csdev@m1:~/proj/evuproject/connectors$ gradle restmonitor
Starting a Gradle Daemon, 1 busy and 2 stopped Daemons could not be reused, use --status for details

> Task :rest:restmonitor
=== Starting Temperature Monitor - ReST Variant ===
2020-11-12T11:02:37.439+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "104.17888917415263" }
2020-11-12T11:02:39.959+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "102.77369209172089" }
2020-11-12T11:02:42.464+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "102.74159099008216" }
2020-11-12T11:02:44.975+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "103.26950606697552" }
2020-11-12T11:02:47.485+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "102.47493680865954" }
2020-11-12T11:02:50.003+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "103.51399903376408" }
2020-11-12T11:02:52.512+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor" : 217, "value" : "103.76336140774508" }
<===========--> 87% EXECUTING [19s]
> :rest:restmonitor
```

And of course…

- Availability is not high in RPC architectures…
  - … without some good tactics implemented ☺

```
2020-11-12T11:03:37.842+01:00 [INFO] rest.RestServer :: Sampling temperature: 101.86319927222858
<===========--> 87% EXECUTING [1m 18s]
> :rest:restserver
^Ccsdev@m1:~/proj/evuproject/connectors$ 

csdev@m1: ~/proj/evuproject/connectors 149x18
2020-11-12T11:03:22.655+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:25.171+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:27.680+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:30.184+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:32.695+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:35.208+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
2020-11-12T11:03:37.719+01:00 [INFO] rest.RestMonitor :: Reading RCP from Rest Server - T = {  "sensor"
Exception in thread "main" java.net.ConnectException: Connection refused
        at java.net.http/jdk.internal.net.http.HttpClientImpl.send(HttpClientImpl.java:561)
        at java.net.http/jdk.internal.net.http.HttpClientFacade.send(HttpClientFacade.java:119)
        at rest.RestMonitor.monitorTemperature(RestMonitor.java:40)
        at rest.RestMonitor.<init>(RestMonitor.java:34)
```

Messaging

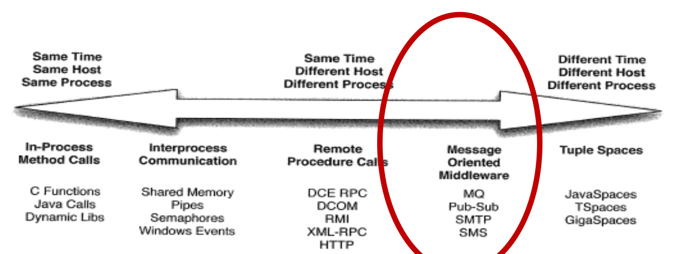


Figure 5.4: COUPLING SPECTRUM OF MIDDLEWARE

Introduction

Implicit Invocation

- Introduce an *intermediary*
- Wider application than just RPC
  - Client sends **Message** to a **MessageQueue** with properties
  - Consumers register interest in **Messages** with given properties
  - **MessageQueue** delivers (copies of) **Messages** to consumers

- Examples
  - RabbitMQ, ActiveMQ, Kafka, …

Key Characteristics

- It is a **different time** connector
  - Producer and consumer are *decoupled* in time
  - Highly decoupled, any producer and any consumer, in any number
- It is **data oriented**
  - You provide data, that 'someone' may get, no control hand-over
- It is **event oriented**
  - A message is an **event**
    - MQ systems usually do not drop events, and assume someone will read them

Demo

- Using *RabbitMQ* as Message Broker

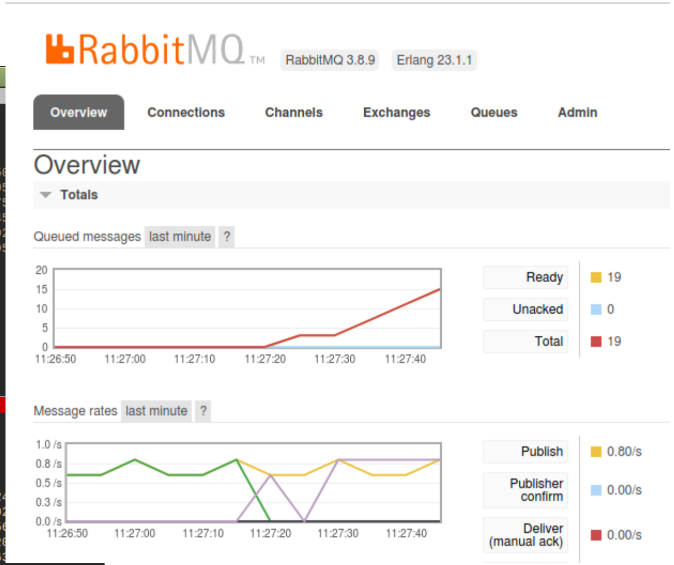
```
csdev@m1:~/proj/evuproject/connectors$ ./startmq.sh
f767505b3e8b8ba68dfe51da75aa18c4dc22eba781e392e327df06c834a3f549
docker rm -f mq, to stop the rabbit MQ again!
```

```
csdev@m1: ~/proj/evuproject/connectors
csdev@m1: ~/proj/evuproject/connectors 152x19
csdev@m1:~/proj/evuproject/connectors$ gradle mqserver

> Task :mq:mqserver
=== Starting Temperature Sensor - Messaging Variant ===
2020-11-05T11:26:31.167+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 105.51480856
2020-11-05T11:26:32.669+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 106.11083005
2020-11-05T11:26:34.170+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 105.56609075
2020-11-05T11:26:35.675+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 105.46579345
2020-11-05T11:26:37.177+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 106.41847892
2020-11-05T11:26:38.679+01:00 [INFO] mq.MqServer :: Publishing to exchange (routingkey: 'plant.temperature') - T = 107.10475195
<===========--> 87% EXECUTING [8s]
> :mq:mqserver
```

```
csdev@m1: ~/proj/evuproject/connectors 152x19
csdev@m1:~/proj/evuproject/connectors$ gradle mqmonitor

> Task :mq:mqmonitor
=== Starting Temperature Monitor - Messaging Variant ===
2020-11-05T11:26:36.898+01:00 [INFO] mq.MqMonitor :: Starting monitoring...
2020-11-05T11:26:36.909+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 105.5148085060524
2020-11-05T11:26:36.911+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 106.1108300526702
2020-11-05T11:26:36.911+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 105.5660907527356
2020-11-05T11:26:36.911+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 105.4657934570220
2020-11-05T11:26:37.182+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 106.4184789241563
2020-11-05T11:26:38.680+01:00 [INFO] mq.MqMonitor :: Consuming from queue (bindingkey: '*.temperature') - T = 107.10475195116584
<===========--> 87% EXECUTING [2s]
> :mq:mqmonitor
```



Tuple Space

A Distributed Shared Memory

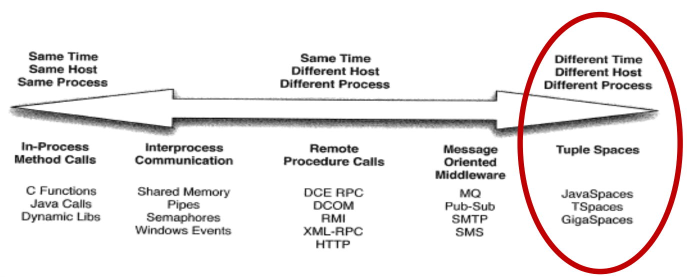


Figure 5.4: Coupling Spectrum of Middleware

Introduction

Implicit Invocation

- Introduce an *intermediary*
- A **State** oriented connector
  - Client **writes** state to the tuplespace
  - Consumers **read** data from the tuplespace

- Examples
  - Few… And obviously not main-stream, see other slide set…
  - But – any *fast* database is actually a tuplespace connector, right?

Demo

- Using *Redis* as intermediary

```
csdev@m1:~/proj/evuproject/connectors$ ./startredis.sh
a775dd021e6b44f71535375dfa2db36ee913eac3e5b9596cf7c488ae2cc08b0d
docker rm -f tuplespace, to remove the container again
```

```
csdev@m1: ~/proj/evu
csdev@m1:~/proj/evuproject/connectors$ doc
127.0.0.1:6379> get sensor_217
"100.09899811062759"
127.0.0.1:6379> get sensor_217
"99.47881100606956"
127.0.0.1:6379> get sensor_217
"99.7306947357706"
127.0.0.1:6379>
```

```
csdev@m1: ~/proj/evuproject/connectors
csdev@m1: ~/proj/evuproject/connectors 152x19
csdev@m1:~/proj/evuproject/connectors$ gradle tupleserver
Starting a Gradle Daemon, 1 busy and 1 stopped Daemons could not be reused, use --status for details

> Task :tuplespace:tupleserver
=== Starting Temperature Sensor - TupleSpace Variant ===
2020-11-05T11:35:19.896+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 104.79066701192264
2020-11-05T11:35:21.402+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 104.05408589307004
2020-11-05T11:35:22.906+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 103.66769996152195
2020-11-05T11:35:24.407+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 103.25770923872052
2020-11-05T11:35:25.908+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 102.90889015492154
2020-11-05T11:35:27.410+01:00 [INFO] tuplespace.TupleServer :: Writing to tuple space - T = 102.37008682847316
<===========--> 87% EXECUTING [9s]
> :tuplespace:tupleserver
```

```
csdev@m1: ~/proj/evuproject/connectors 152x19
csdev@m1:~/proj/evuproject/connectors$ gradle tuplemonitor

> Task :tuplespace:tuplemonitor
=== Starting Temperature Monitor - TupleSpace Variant ===
2020-11-05T11:35:10.353+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.94021309868778
2020-11-05T11:35:12.857+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.94021309868778
2020-11-05T11:35:15.361+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.94021309868778
2020-11-05T11:35:17.861+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.94021309868778
2020-11-05T11:35:20.362+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.79066701192264
2020-11-05T11:35:22.864+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 104.05408589307004
2020-11-05T11:35:25.368+01:00 [INFO] tuplespace.TupleServer :: Reading from tuple space - T = 103.25770923872052
<===========--> 87% EXECUTING [17s]
> :tuplespace:tuplemonitor
```

AARHUS UNIVERSITET

Key Characteristics

- It is a **different time** connector
  - Producer and consumer are decoupled in time
  - Highly decoupled, any producer and any consumer, in any number
- It is **data oriented**
  - You provide data, that ‘someone’ may get, no control hand-over
- It is **state oriented**
  - In the sense of *opposite that of an event*
    - There are no events to fear is lost
    - But history is lost, only *latest data* is available

Case

- Danfoss Drives collaboration many years ago
  - Requirement:
    - Hard real-time motor control system — Performance
    - Component ‘plug-in’ architecture — Modifiability
  - Challenge
    - Hard real time loop: read values, compute, actuate
      - Within N milliseconds — for N very small!
    - But, if I plug in a component method call in this loop
      - If method call lasts more than M micro-seconds, the hard real time constraints cannot be held ☹
  - Solution
    - **Connector was a tuple space** — read/write value is ‘constant time’
      - Easy to calculate the ‘time budget’

Summary ☺

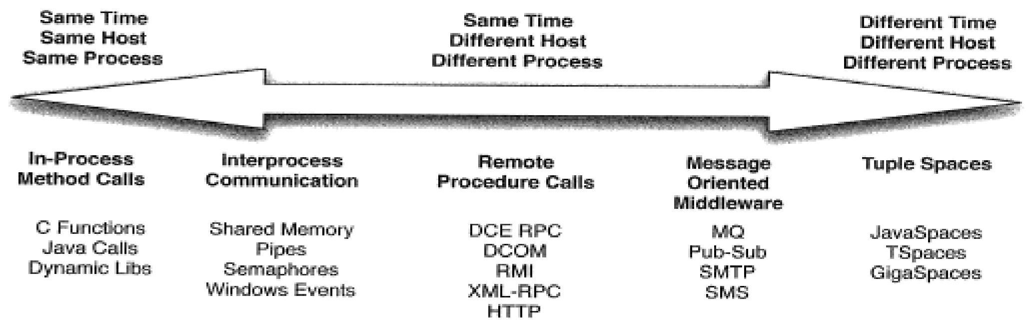


Figure 5.4: Coupling Spectrum of Middleware

- Middleware decisions effect the implementation cost of systems significantly
  - Learn many architectural styles to ensure you pick the right one

Source: Nygard 2017

AARHUS UNIVERSITET

Summary

- Metaphors for communication
  - RPC — Phone call
  - Messaging — Send/Receive a letter
  - Tspace — Put/Read a message up on a bulleting board