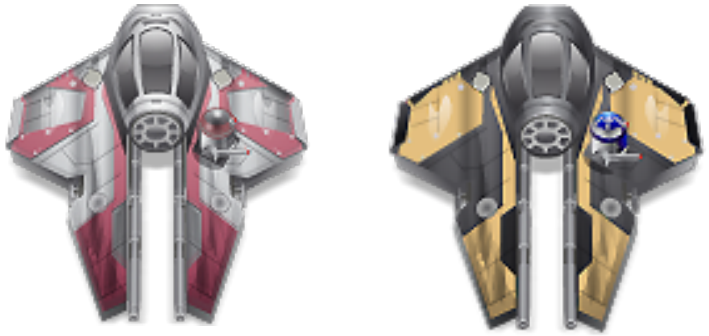
ESIR SR

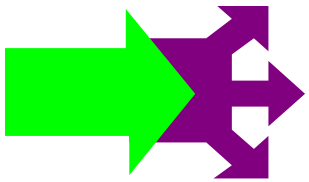
# Unit 10b: JGroups

François Taïani



# Overview of the Session

- Group Communication
- What is JGroups?
- JGroups' Architecture
  - ➔The Channel Class
  - ➔The Protocol Stack Infrastructure
  - ➔The Building Blocks
- Comparison with JMS



# Introducing Group Communication

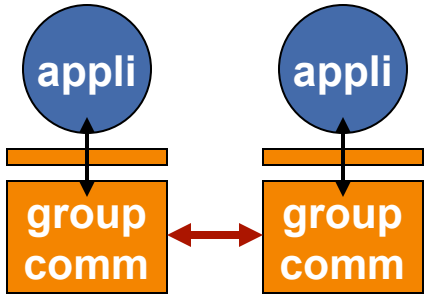
- What is group communication?
  - ➔ Enables the multicasting of a message to a group of processes as a single action
  - ➔ Sender is unaware of the destinations for the message
- Why group communication?
  - ➔ Support for replication: fault tolerance & scalability
  - ➔ Support the efficient dissemination of data Service discovery, publish/subscribe

Indirect Communication
Fault Tolerance

JMS

Associated reading: section 7.4 of Tanenbaum and Van Steen (2002 ed); Section 4.4 and 11.5 of Coulouris & al. (2nd ed)

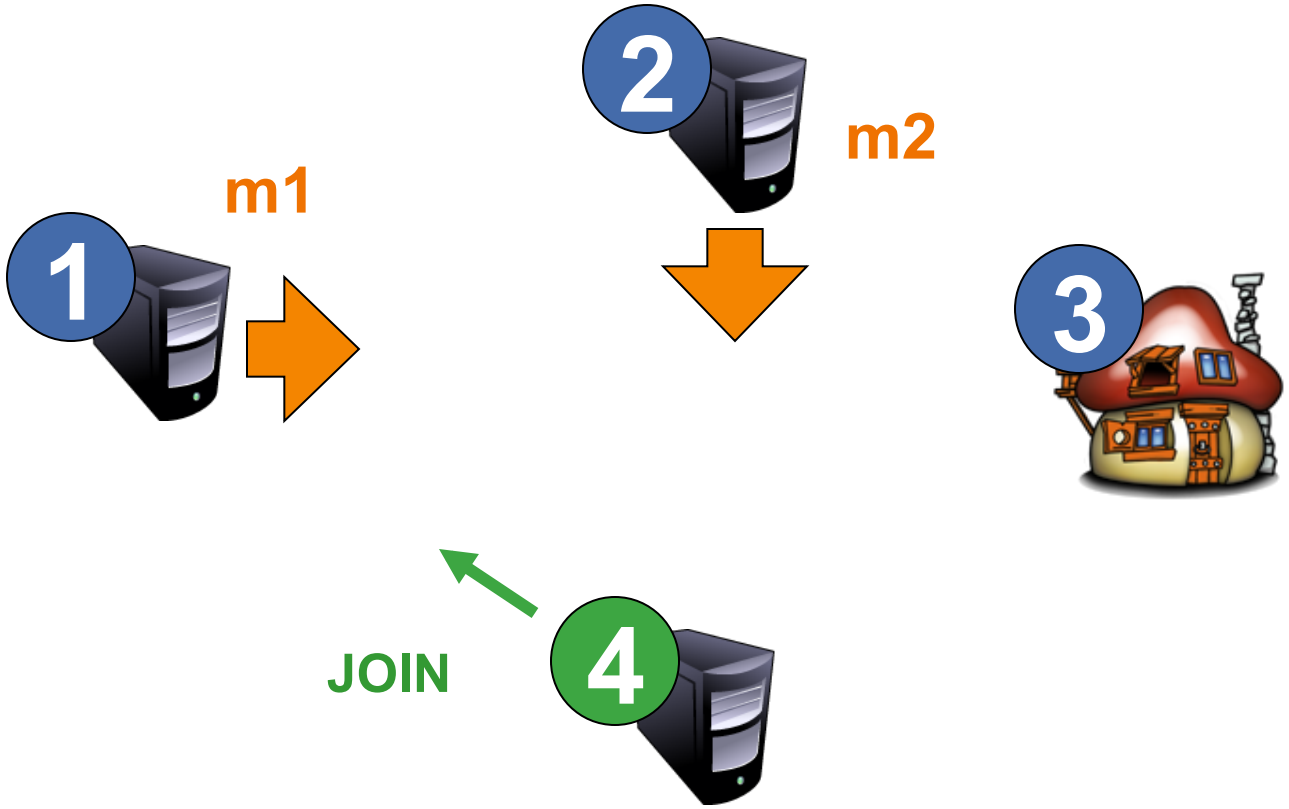
# A Typical Group Service



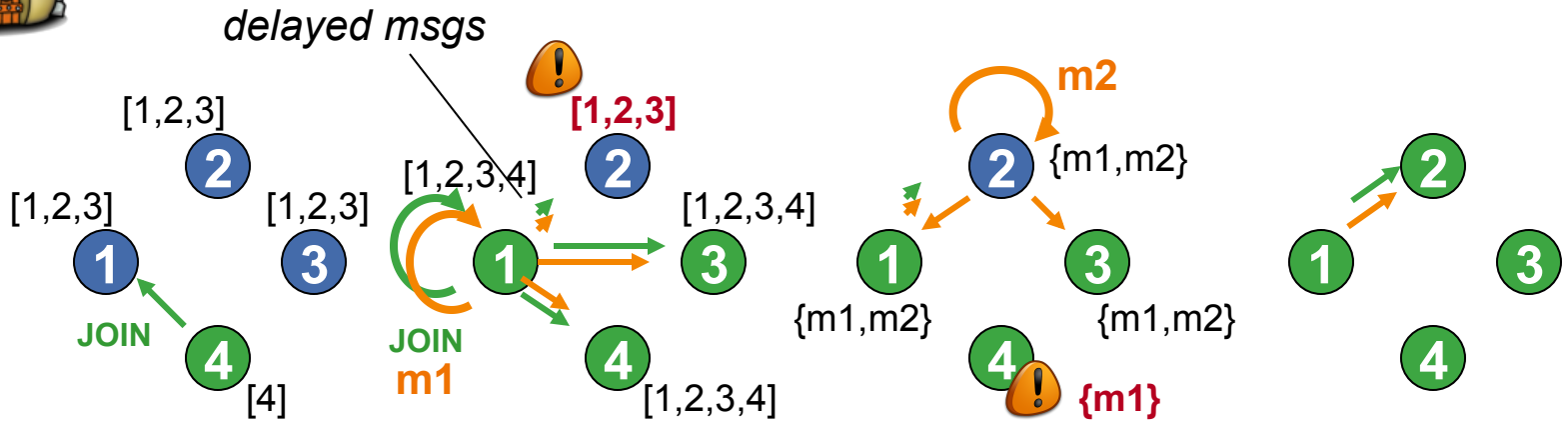

```
interface GroupCommunicationService {

    // creates a new group and returns the groups ID

    public GroupID      groupCreate();

    // Adding & Removing a member to/from a group

    public void         groupJoin   (GroupID group, Participant member);
    public void         groupLeave  (GroupID group, Participant member);

    // multicasts a message to the named group with
    // the specified delivery semantics, and
    // optionally collects a number of replies

    public Messages[]   multicast   (GroupID group, OrderType order,
                                     Messages message, int nbReplies)

}
```

# More Ordering Woes

- What happens if a process joins/ leaves during multicast op?
  - ➔ Which messages will 4 get?



# The Join/Leave Problem




- Once '4' has started receiving messages, it should get all the subsequent ones (I.e. again, all-or-nothing)
  - ➔ Participants need to agree on **who's in the group**!
- But agreeing requires a new multicast operation with **ordering** guarantees → **virtual synchrony**
  - ➔ The 'local view' of the sender piggybacked on sent message
  - ➔ Protocol must make sure message received in same local view as it was sent

# Example: JGroups

- A toolkit for reliable multicast communication
  - In Java
  - Open source (LGPL license from Free Software Foundation)
  - Used in commercial products like **JBoss** to implement JMS


JMS

- Key Feature: Flexible Protocol Stack
  - Can be **tailored** to application needs and network characteristics
  - Can be **extended**
  - Provides a wide spectrum of properties. Focuses on **reliability**.
- JGroups created by Belan Ban at Cornell Uni (US)
  - Incidentally the home of Isis
  - Where Werner Vogel (Amazon's CTO) worked for a long time

amazon.com

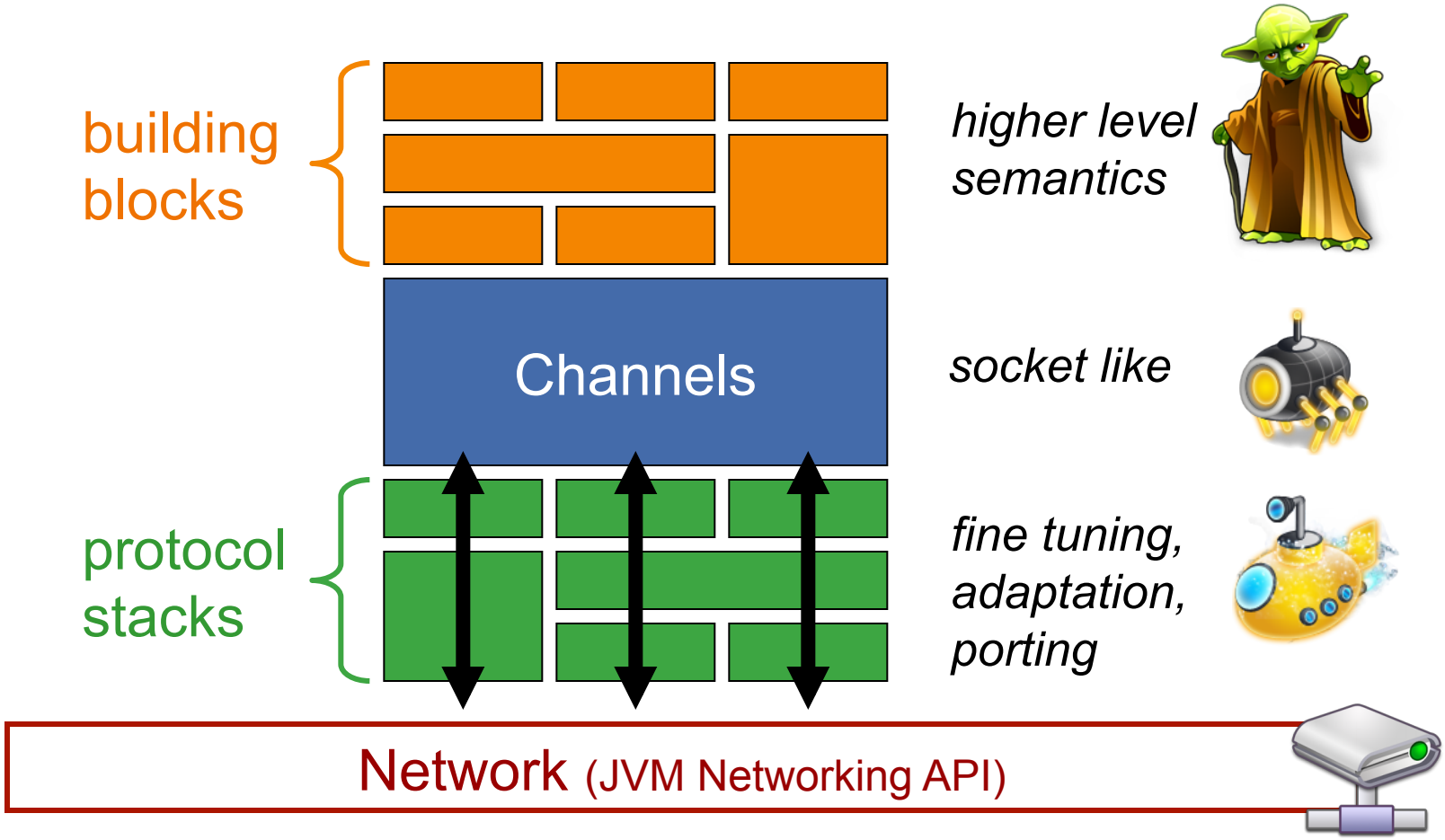
# Where do I get it?

- From the JGroups Web site
  - ➔http://www.jgroups.org
- Refs:
  - ➔Doc on the site (javadoc, manuals, tutorials)
  - ➔Virtual Synchrony see http://www.theserverside.com/news/1363871/New-Features-in-JGroups-25

# A developer's view

- JGroups still being actively developed by Bela Ban (2012)
  - ➔ http://belaban.blogspot.com/
  - ➔ Bela works for RedHat, who own JBoss (J2EE FOSS server)
- 1st hand experience of a distributed system developer
  - ➔ Insight into design decision, everyday problems, etc.

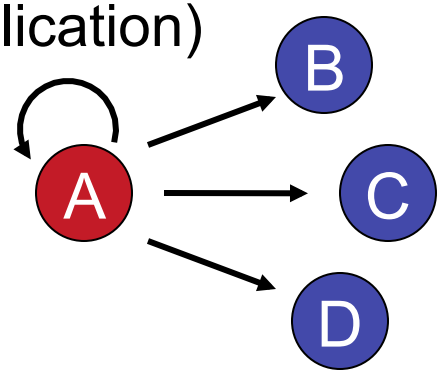
# JGroups Architecture



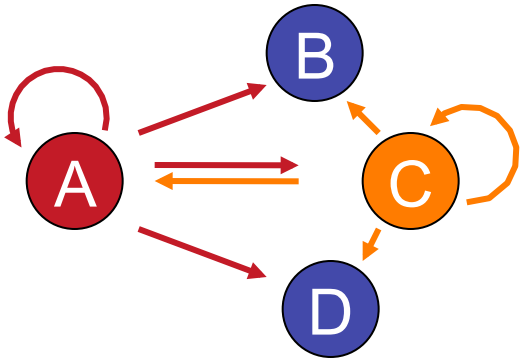
# Note on reception

**In JGroup any sending nodes receives its own message**

- encourages symmetric code (e.g. active replication)




- sender can observe reception context (order, stabilisation)
  - ➔ E.g. total order
  - ➔ A and C needs to receive their own msg to know which one arrived first



# JGroups API: Channels

- Design by **Simplicity**
  - ➔ One core class: **org.jgroups.Channel**

properties of the channel (see later)

```
String props="UDP:PING:FD:STABLE:NAKACK:UNICAST:" +
             "FRAG:FLUSH:GMS:VIEW_ENFORCER:" +
             "STATE_TRANSFER:QUEUE";

Message send_msg;
Object  recv_msg;
Channel channel=new JChannel(props);

channel.connect("MyGroup");

  send_msg=new Message(null, null, "Hello world");
  channel.send(send_msg);

  recv_msg=channel.receive(0);
  System.out.println("Received " + recv_msg);

channel.disconnect();
channel.close();
```

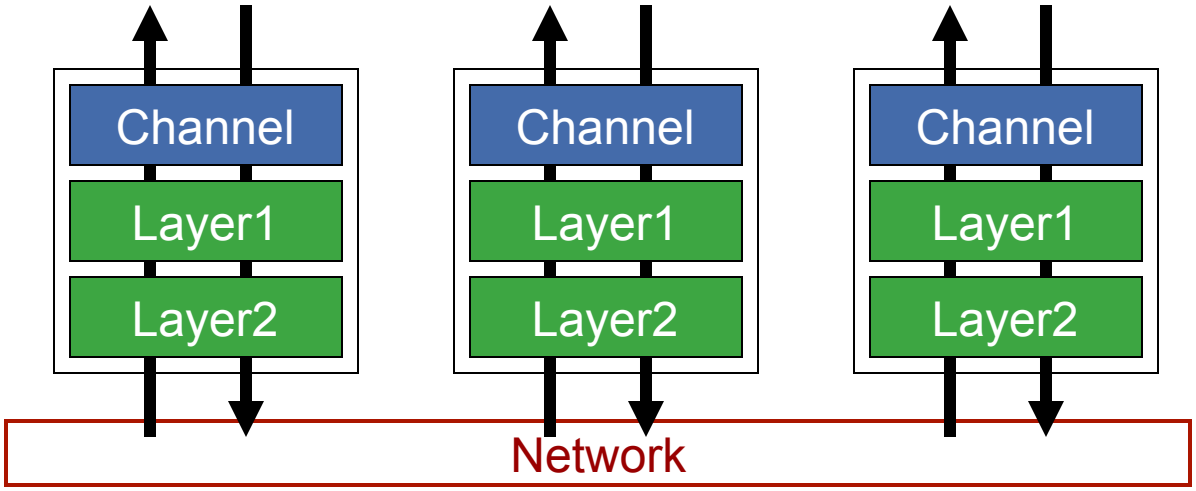
sender's address (filled up by protocol stack)

all group members

I receive my own message

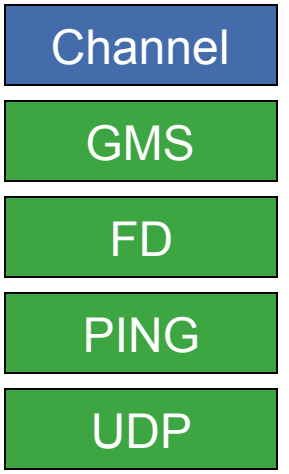
# Protocol Stacks

- Each Channel instance sits on top of a protocol stack
  - ➔ Stack content defined by the property string of the Channel:
  - ➔ `Channel myChannel = new JChannel("LAYER1:LAYER2")`
- Protocol stack composed out of "layers"
  - ➔ Messages go up and down the layer stack
  - ➔ Each layer can modify, reorder, pass, drop or add a header to messages



# Protocol Layers (1)

- `Channel myChan = new JChannel("UDP:PING:FD:GMS");`
  - Stack contains layers UDP, PING, FD, and GMS (bottom-up)
  - Corresponds to classes:
    ```
    org.jgroups.protocols.UDP
    org.jgroups.protocols.PING
    org.jgroups.protocols.FD
    org.jgroups.protocols.GMS
    ```
  - **UDP**: IP multicast transport based on UDP
  - **PING**: initial membership (used by GMS)
  - **FD**: Failure detection (heartbeat protocol)
  - **GMS**: Group membership protocol.




- Some expertise needed
  - Syntactically any combination possible
  - Does not necessarily make sense (e.g. GMS requires PING)

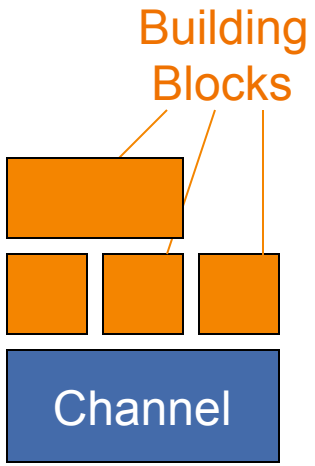
# Protocol Layers (2)

Changes from 2.x to 3.x

- Example of other protocol layers
  - ~~**CAUSAL**: causal ordering layer using vector clocks~~
  - **SEQUENCER**: total ordering layer using a message sequencer
  - **NAKACK**: negative ACKs (NAKs), paired with positive ACKs
  - **STABLE**: computes the broadcast messages that are stable
  - ~~**PERF**: measures time taken by layers to process a message~~
  - **COMPRESS**: compresses the payload of a message
  - **pbcast.FLUSH**: virtual synchrony
- Constrains must be obeyed
  - Usually same layers needed in all group members
  - Dependencies: e.g. GMS needs PING, STABLE needs NAKACK

# Building Blocks

- Channel class very simple
  - Similar to sockets, but group behaviour
  - Message based. No reply/request concept.
  - Active (aka blocking, pull-style) message reception
    - Explicit threading needed on top of channel for passive reception
- Building Blocks (org.jgroups.blocks package)
  - **Higher level** classes on top of Channel
  - Provides higher level programming abstractions
- Examples
  - **ReceiverAdapter**: passive reception
  - **RpcDispatcher**: remote invocation



# ReceiverTest

```
public class ReceiverTest extends ReceiverAdapter {


   public static void main(String args[]) throws Exception {









   }
}
```

# ReceiverTest

```
public class ReceiverTest extends ReceiverAdapter {
  public void viewAccepted(View new_view)
  { System.out.println("view: " + new_view); }
  public void receive(Message msg)
  { System.out.println("Received msg: " + msg.getObject()); }
  public static void main(String args[]) throws Exception {
    Channel chan=new JChannel();
    chan.setReceiver(new ReceiverTest());
    chan.connect("ReceiverTest");
    for(int i=0; i < 10; i++) {
      System.out.println("Sending msg #" + i);
      chan.send(new Message(null,null,"Hello "+ i));
      Thread.currentThread().sleep(1000);
    }
    chan.close();
  }
}
```

# Small Quizz

- Imagine 5 processes execute the previous code
  - ➔ What's the maximum number of "`Hello i`" being printed?
  - ➔ What would be the minimum number? (no crash)
  - ➔ How many would this be with ***n*** processes?

# RpcDispatcher

```
public class RpcDispatcherTest {

   public static void main(String[] args) throws Exception {

   }
}
```

# RpcDispatcher

```
public class RpcDispatcherTest {
  public static int print(int number) throws Exception
  { return number * 2; }
  public static void main(String[] args) throws Exception {
    JChannel        channel =new JChannel();
    RpcDispatcher   disp    =
       new RpcDispatcher(channel, new RpcDispatcherTest());
    RequestOptions opts     =
       new RequestOptions(ResponseMode.GET_ALL, 5000);
    channel.connect("RpcDispatcherTestGroup");
    for(int i=0; i < 10; i++) {
      Thread.sleep(100);
      RspList rsp_list=disp.callRemoteMethods
      (null,"print",new Object[]{i},new Class[]{int.class},opts);
      System.out.println("Responses: "+rsp_list); }
    channel.close();
    disp.stop();
  }
}
```

# RpcDispatcher: Quizz

- Assuming 3 processes execute the previous program
  - ➔ What's the max number of time `print` will be invoked?

# RpcDispatcher (2)

- Provides "**Group**" Remote Procedure Call behaviour
  - Looks up the invoked method (here "print")
  - In example collects answers from all group members
  - Each group member is potentially both a client and a server!
- Not completely **transparent**
  - No stub or skeleton to hide remote invocation
  - Arguments passed as an array of Object
- **Return behaviour** can be adapted
  - GET_ABS_MAJORITY: return majority (of all members, may block)
  - GET_ALL: return all responses
  - GET_FIRST: return only first response
  - GET_MAJORITY: return majority (of all non-faulty members)
  - GET_N: return n responses (may block)

# Comparison with JMS

- **JMS** is a standardised API
  - ➔ Various implementations
- **JGroups** is a library
  - ➔ Has its own API (not JMS compliant)
  - ➔ Only one implementation
  - ➔ Can be (and is) used to implement the JMS API (in JBoss)
- **JMS** is a Message Oriented Middleware
  - ➔ Higher level that plain group communication as in JGroups
  - ➔ E.g. persistence, durability, transactions
- **JMS** assumes a server based architecture
  - ➔ JGroups can be used in fully decentralised manner (or not)

# Expected Learning Outcomes

**At the end of this 5th Technical Session:**

- You should know what JGroups is about
- You should appreciate the basic working of the Channel class
- You should understand JGroups' protocol stack mechanism
- You should be able to compare JGroups to JMS
- You should be able to solve some of the small quizzes shown in the slides (and explain your solution)