Distributed Systems 1: Hands-on Labs

davide.vecchia@unitn.it
timofei.istomin@unitn.it

Hands-on Labs

What will we (you) do?

- Implement some distributed algorithms seen in class
  - We together during the labs
  - You in groups as the course project
- Learn a different programming model (Actor-based)
- Deepen the knowledge of Java

akka

http://akka.io

Build powerful reactive, concurrent, and distributed applications more easily

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

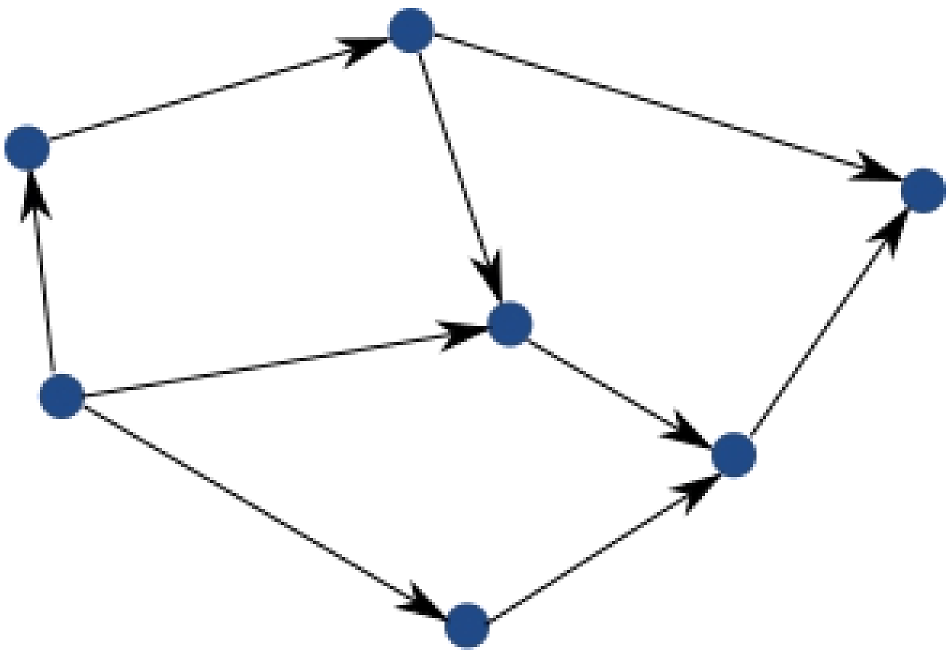
What actors provide

- Multi-threading without low-level concurrency constructs (threads, shared memory, locks)
- Network Transparency: explicit communication among remote objects as with local ones
  - Unlike hidden remoting approaches, e.g., RPC/RMI
- Elastic scalable architecture

We will not go there

Actor Model

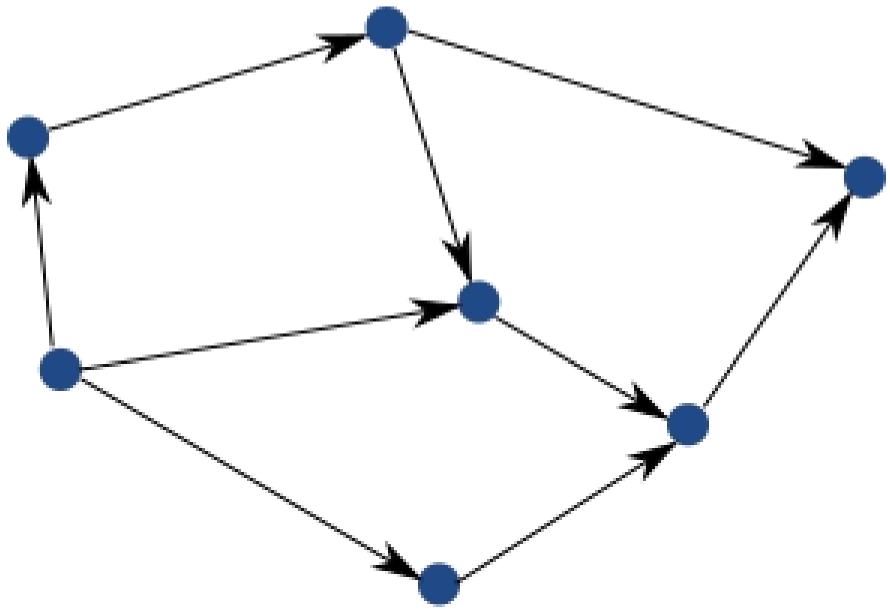
- A (distributed) program consists of Actors
- Actors encapsulate state and behaviour
- They interact by sending messages to each other



Actors interacting with each other by sending messages to each other

Does it remind you of anything?

Is it different from OOP?



Objects interacting with each other by calling methods on each other

- A program consists of Objects
- Objects encapsulate state and behaviour
- They interact by calling each other's methods

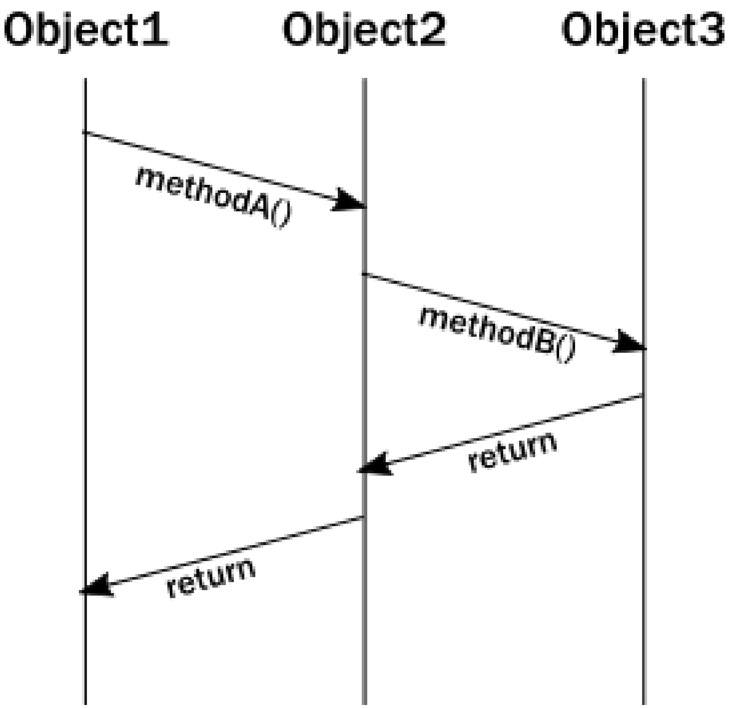
This is fine unless there are threads or network

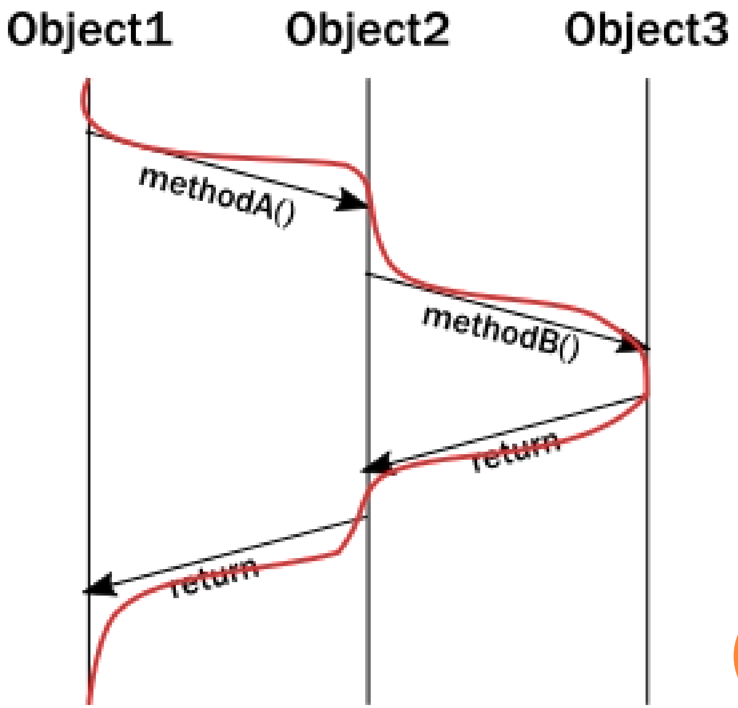
Control flow in OOP

A promise of a (correct) object:

Its state remains correct after any method call

- Preserves certain internal invariants

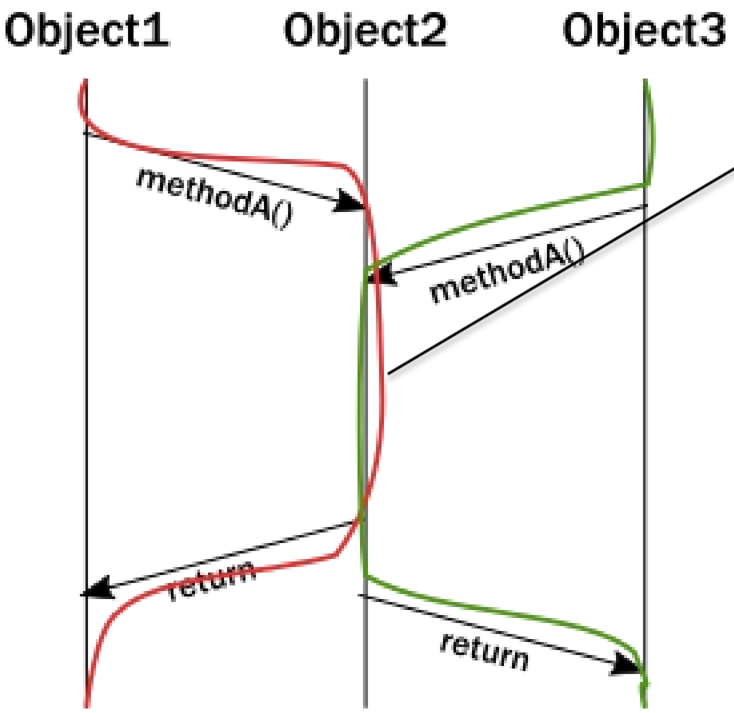




OOP WITH THREADS

What if we add threads?

- Why do we need threads?



Multiple threads are working on the same variables simultaneously

- **We'll get race conditions, cache problems**
  - Objects don't hold their invariants and therefore break their main promise

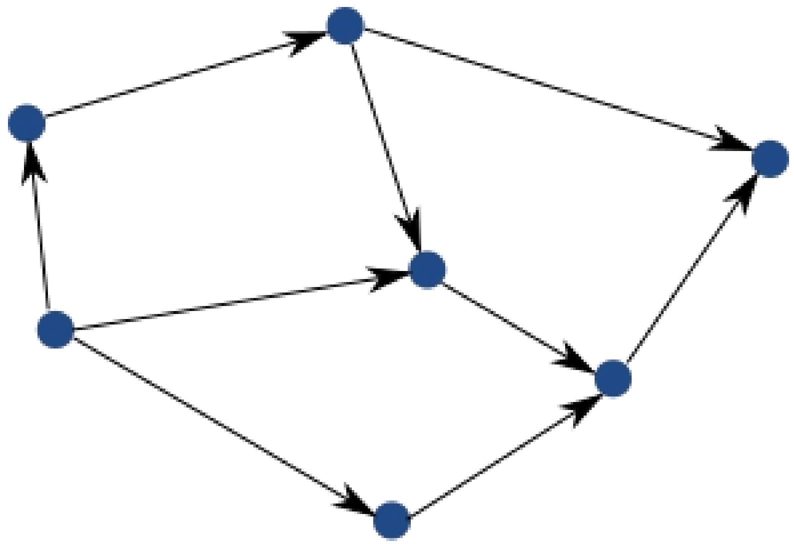
Working with threads

- Synchronisation mechanism are used to avoid race conditions and other problems:
  - **Memory barriers**
  - **Locks**
  - **Condition variables**, etc.
- … but they are tricky, don't eliminate thread blocking and <u>create other problems</u>:
  - **Deadlocks**: some threads might get stuck forever waiting for each other
  - **Thread contention**: locks are expensive and in some cases can become a bottleneck creating "traffic jams" in the system

It gets even more complicated in distributed programs

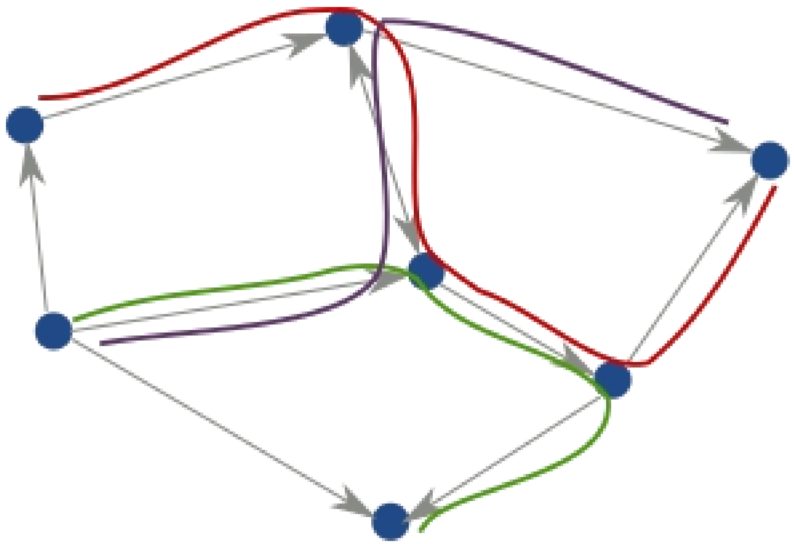
OOP WITH MULTITHREADING

The original idea



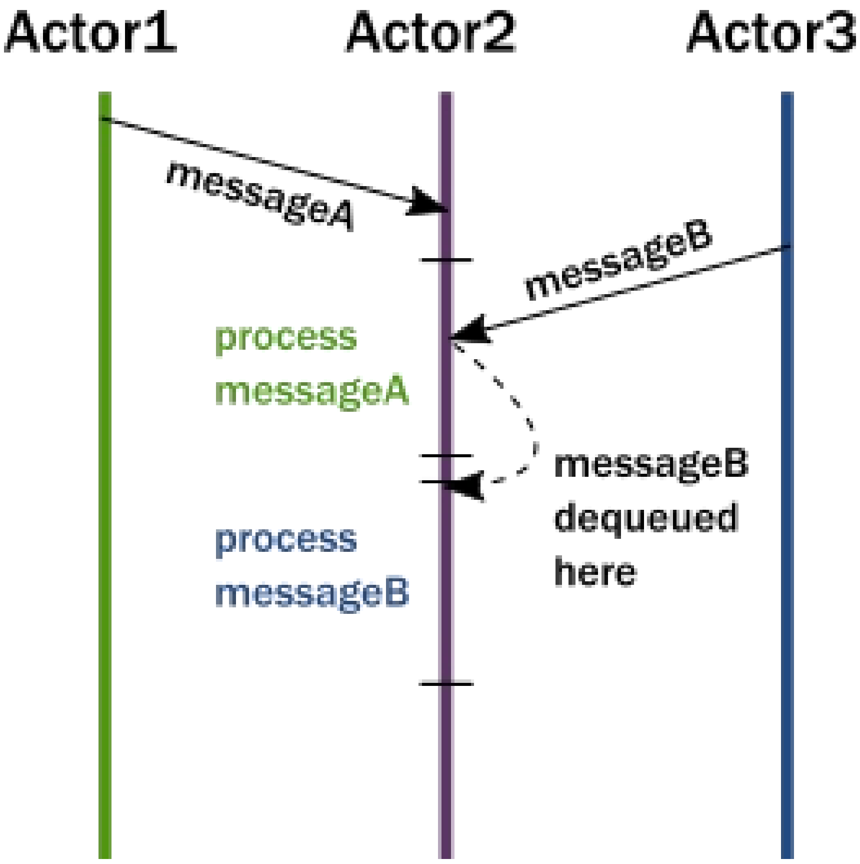
Objects interacting with each other
by calling methods on each other

The reality



Objects interacting with each other
Threads A, B, C, interacting with each other,
traversing method calls on objects

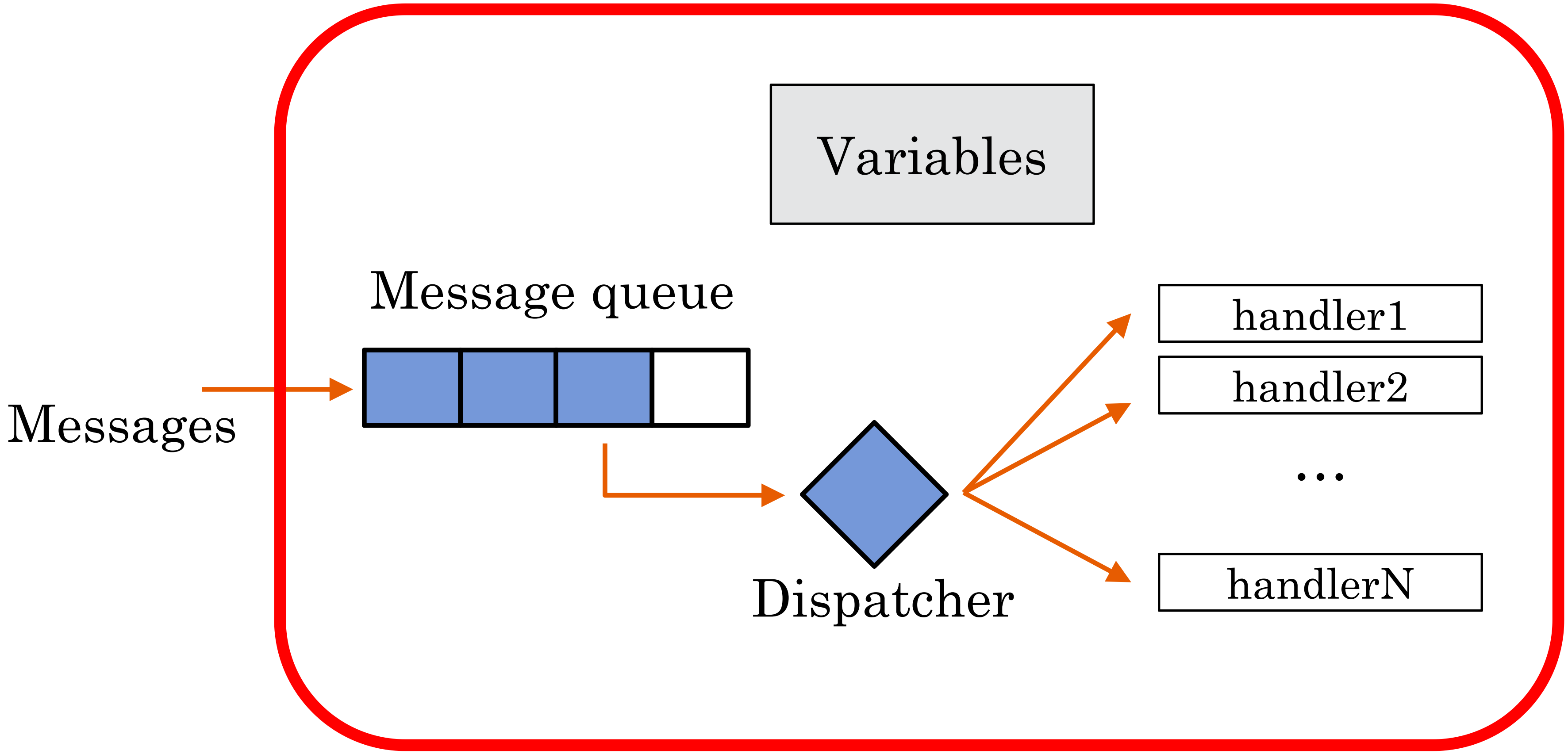
THE ACTOR MODEL



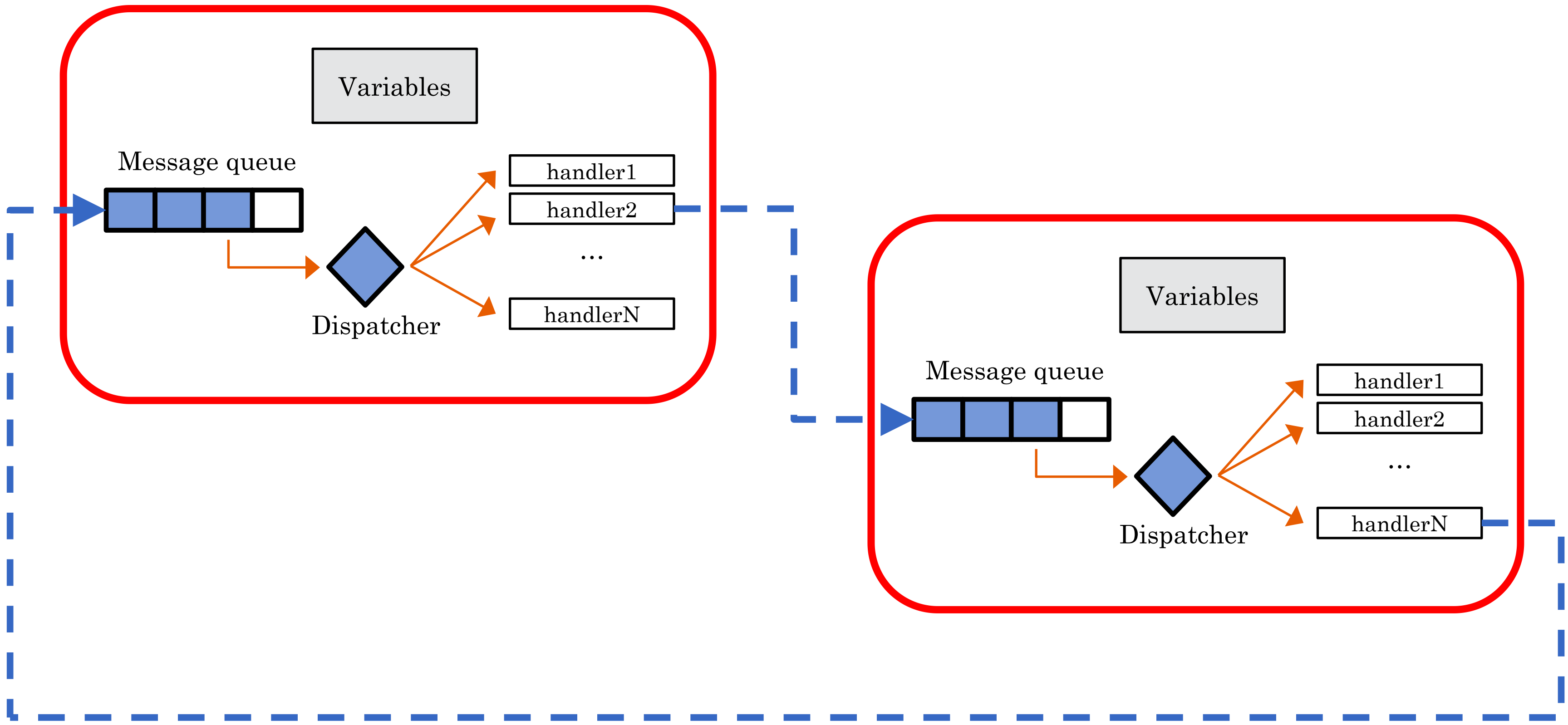
- Send messages to objects (actors) instead of calling their methods
- Don’t wait till the object completes the task
  - Sending a message is non-blocking
  - No return value
- If the actor is busy at the moment, the message is queued

Akka is based on the actor model

An actor


Variables
Message queue
Messages
Dispatcher
handler1
handler2
...
handlerN

Actor System



- There are many actors in the system
- They communicate by sending messages to each other

Guarantees

- Actors *may* be running in parallel in different threads or even on different computers

- It is guaranteed that **no** actor is running in more than one thread at a time
  - No race conditions, no locks needed

- Message queues are FIFO

- Messages *might* be lost

- Message **send** operation is non-blocking

Remote actors?

- The message-passing semantics is natural for remote communications

- Therefore, actors may reside on different computers and communicate using the same primitives as with local actors
  - all messages should be serializable

- Actor-based programs can be very scalable and easy to deploy on a cluster
  - even without changing the program code

AKKA

- AKKA is a Java and Scala framework implementing the Actor Model

- We will use the Java version (akka classic)

- Code, tools and documentation: https://doc.akka.io/docs/akka/current/index-classic.html

- Other implementations of the Actor Model exist, e.g., Erlang: http://erlang.org

Ensuring encapsulation

Java cannot automatically ensure proper encapsulation of actors

- Some discipline is required on the programmer side!

General rule: make sure no object (variable) is accessible from multiple actor instances, e.g.:

- **Don't** send references to ***mutable*** objects in messages. Send copies instead
- **Don't** use ***non-final static*** variables in the actor class nor other classes accessible from the actor
- **Don't** use threads that may access internals of an actor (There's no need for threads! Actors are all you need)

AKKA MESSAGES

Akka messages are user-defined *serializable* Java objects

```
public class Hello implements Serializable{
  private final String msg;
  public Hello(String msg) {
      this.msg = msg;
  }
}
```

Final! (the value of the reference to the object cannot be changed)

String is immutable (referenced object cannot be changed)

Common practice: define your messages as inner classes of the Actors that will receive these messages

E.g.: Receiver.java

AKKA MESSAGES

Send copies for mutable object (ArrayList).

```
public class JoinGroupMsg implements Serializable {
  private final List<ActorRef> group;
  public JoinGroupMsg(List<ActorRef> group) {
    this.group = Collections.unmodifiableList(
      new ArrayList<>(group));
  }
}
```

Would `new ArrayList<>(group)` **be enough?**

No! `group` is mutable!

What about `Collections.unmodifiableList(group)` **?**

No! *`unmodifiableList`* returns a "view", whoever holds the original could modify it.

Actor constructors

```
class MyActor extends AbstractActor {
  // internal variables can be defined here
  private int id;

  // constructor
  public MyActor(int id) {this.id = id;}

  // Actor "properties"
  // (used by the system to create actors)
  static public Props props(int id) {
    return Props.create(
                       MyActor.class,
                       () -> new MyActor(id));
  }
}
```

Calls to the constructor

INITIALISATION

```
public static void main(String[] args) {
  // Create an actor system named "helloakka"
  final ActorSystem system =
                 ActorSystem.create("helloakka");

  // Create an actor
  final ActorRef myactor = system.actorOf(
                     MyActor.props(352),
                     "actor352");
}
```

The actor class that we want to create

Constructor parameters go here

Actor reference

New actor name, unique in the actor system

ACTOR REFERENCE

- To protect actors from unauthorized access the system does not reveal direct Java references to the actor objects

- Instead, you can use an **ActorRef** object associated with an actor to send messages to it

- **ActorRef** objects can be passed inside messages to other actors

- **ActorRef** works both locally and remotely

SENDING MESSAGES

```
Hello m = new Hello("Hi there!");
myactor.tell(m, getSelf());
```

Destination actor reference

Sender reference: can be **null**

Handling incoming messages

```
@Override
public Receive createReceive() {
  return receiveBuilder()
    .match(Message1.class, this::onMessage1)
    .match(Message2.class, this::onMessage2)
    .match(Message3.class, this::onMessage3)
    .build();
}

private void onMessage1(Message1 msg) {...}
private void onMessage2(Message2 msg) {...}
private void onMessage3(Message3 msg) {...}
```

Define the mapping between incoming message classes and the methods of the actor

Define the "reaction" upon reception in private methods of the actor, for each message type

Useful methods of an actor

- Abstract (for you to define, if needed)
  - `void preStart()` – called after the actor has been initialized but before processing any messages

- Defined (for internal Actor use)
  - `getSelf()` – get **ActorRef** of myself
  - `getSelf().path().name()` – get my name
  - `getContext().system().scheduler().schedule()` – schedule an action in the future
  - `getSender()` – get the reference to the current message sender

SCHEDULING A FUTURE ACTION

- Preferred way: schedule a message in the future (maybe even a message to self)

```
Cancellable timer =
getContext().system()
.scheduler().scheduleWithFixedDelay(
      Duration.create(1, TimeUnit.SECONDS),
      Duration.create(1, TimeUnit.SECONDS),
      receiver,
      new Hello("Hi there!”),
      getContext().system().dispatcher(),
      getSelf());
```

After 1s…

Then every 1s…

Send to (can be getSelf())

Sender ref

- It is possible to schedule a *runnable* instead, but this is not recommended!
  - The runnable **may not** access Actor's variables otherwise race condition may happen

Example

Let's look at the example together

Open Java source files located in the `hello/src/main/it/unitn/ds1` directory, using a text editor or an IDE

To run it from the command line, go to `hello` directory and type `gradle run`

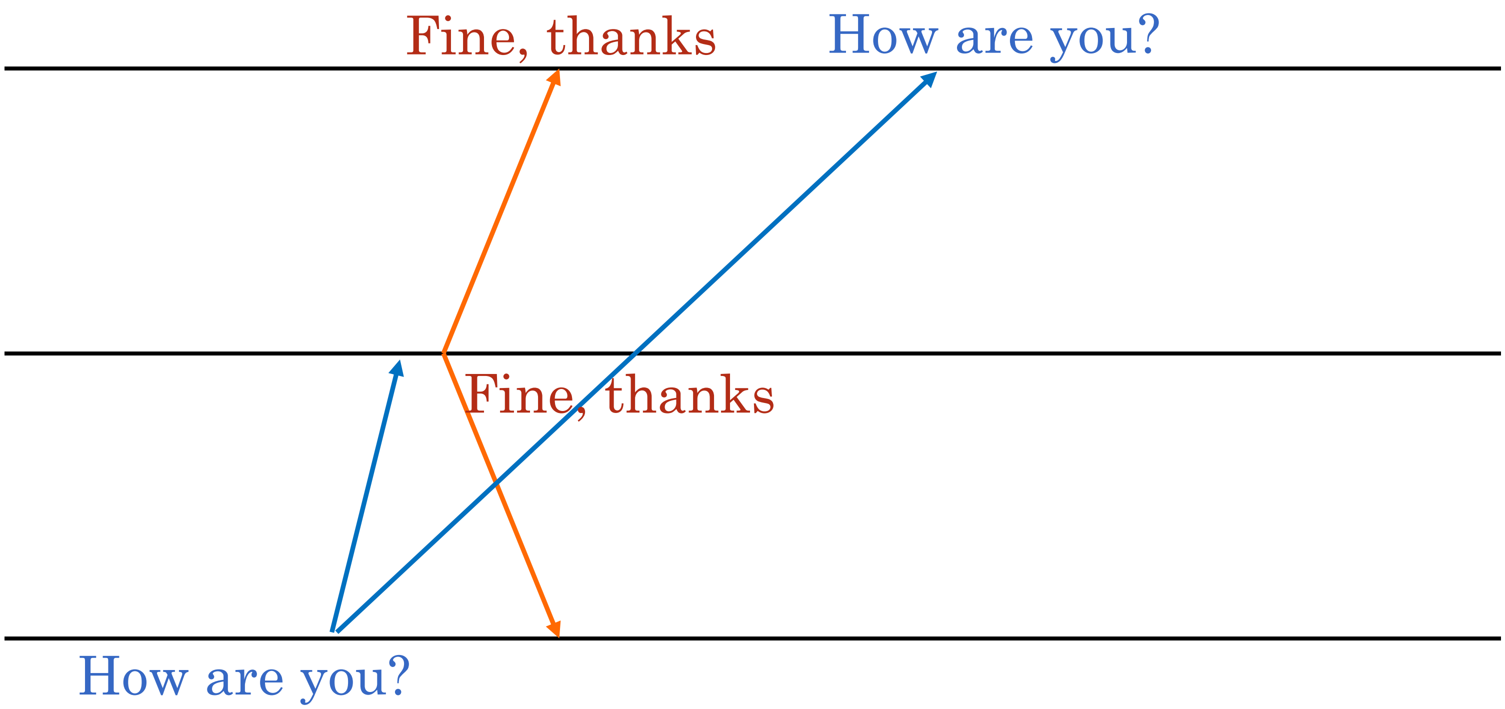
Exercise: Causal Delivery

- We'll create a toy group chat application
- There will be a group of actors that send chat messages to the whole group (multicast)
- For simplicity: all the actors will run locally

- The chat system should guarantee the property of **causal delivery**:

> Nobody can deliver a reply to a message M before delivering the message M itself

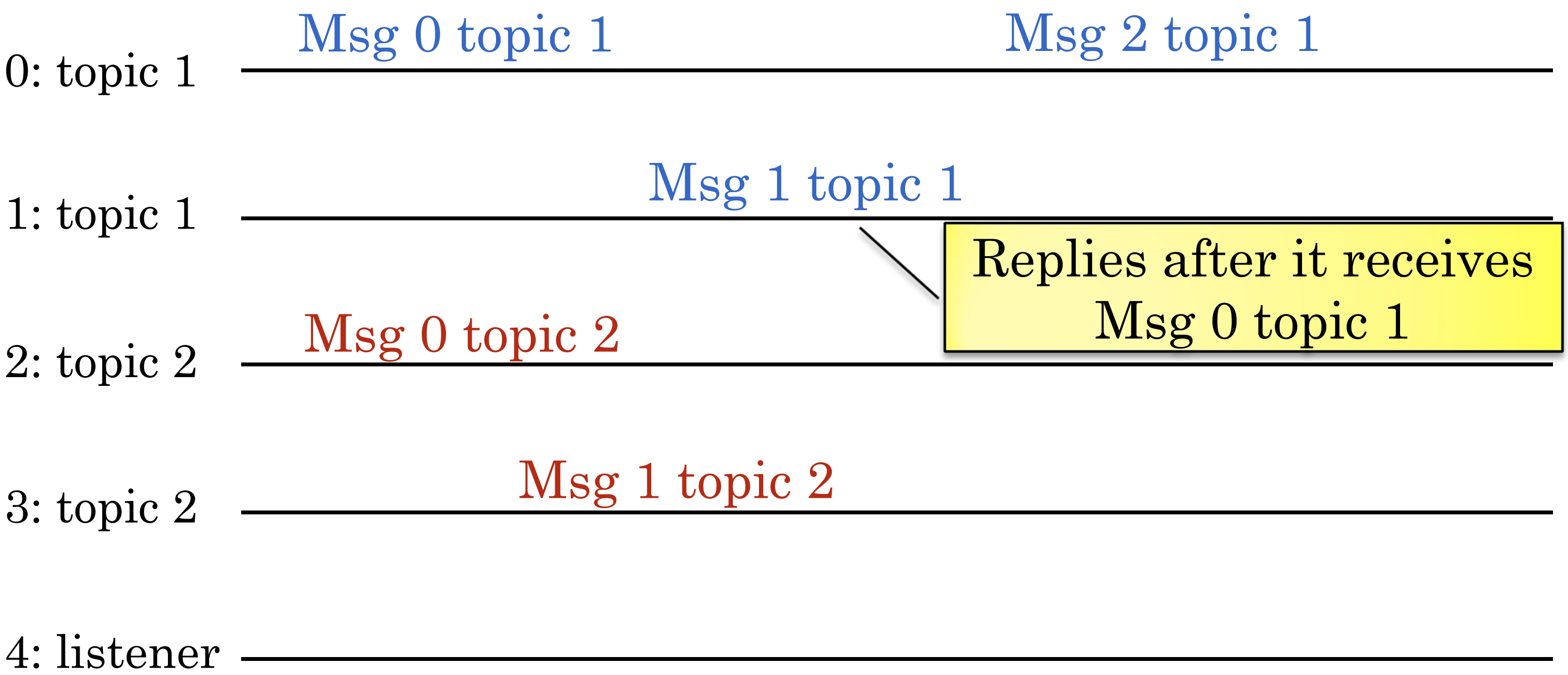
Why reordering is possible?



Simulating a group chat

- We will create several pairs of "chat users" that will be talking on different "topics"
  - We'll start from having only one pair
- We will create a number of "listeners" that see all the messages but don't participate in the talks
- To check that the system preserves the message order we need to know the right order ourselves!
  - To do so, we will put a sequence number into every message in a conversation
  - But we will **NOT** use these sequence numbers inside the "chat system" to preserve the ordering
  - That would be cheating.

Chat timeline



- All messages are sent to everybody (multicast)
- Chatters reply on their topic only, incrementing the sequence number

Chat history

- At the end we will print the chat histories of all participants in the order of message delivery
- The system is correct if all messages on the same topic are always in the order of sending
- Messages on different topics might be swapped

Listener1: T1:0 T1:1 T2:0 T1:2 T2:1 T2:2

Listener2: T1:0 T1:1 T1:2 T2:0 T2:1 T2:2

BEFORE WE START…

- Before we start with vector clocks, complete a simpler exercise
- Open the **multicast** example
  - Let's look at the code together
- Compile it
- Run and check the output

- Exercise: add more pairs of chatters to the system

Emulating network delays

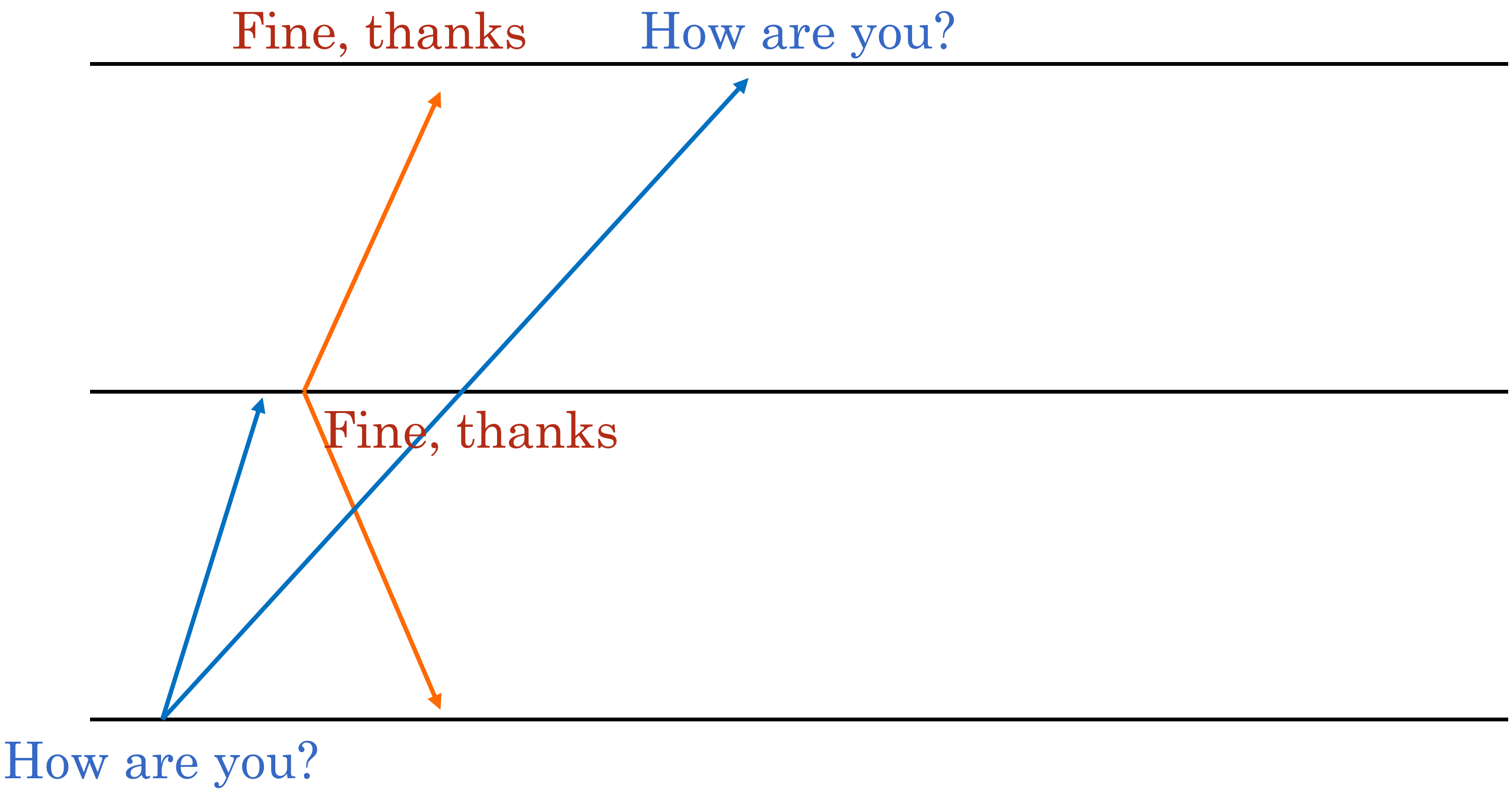
We *emulate* random network delays with this code:

```
private void multicast(Serializable m) {
  Collections.shuffle(group);
  for (ActorRef p: group) {
    if (!p.equals(getSelf())) {
      p.tell(m, getSelf());
      try {
        Thread.sleep(rnd.nextInt(10));
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

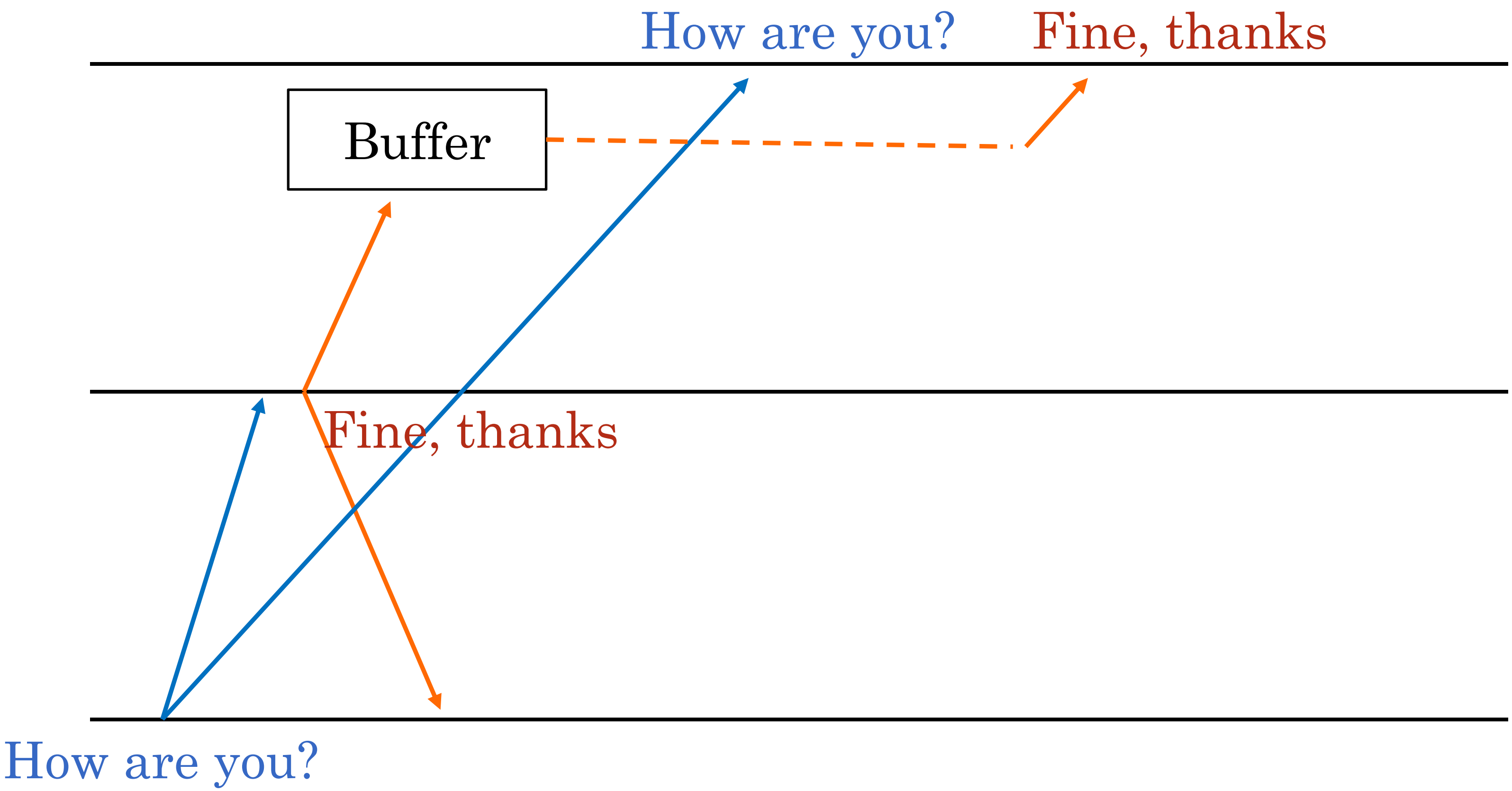
Within 10 ms

Never use **Thread.sleep** to schedule activities in the future!

Reordering…

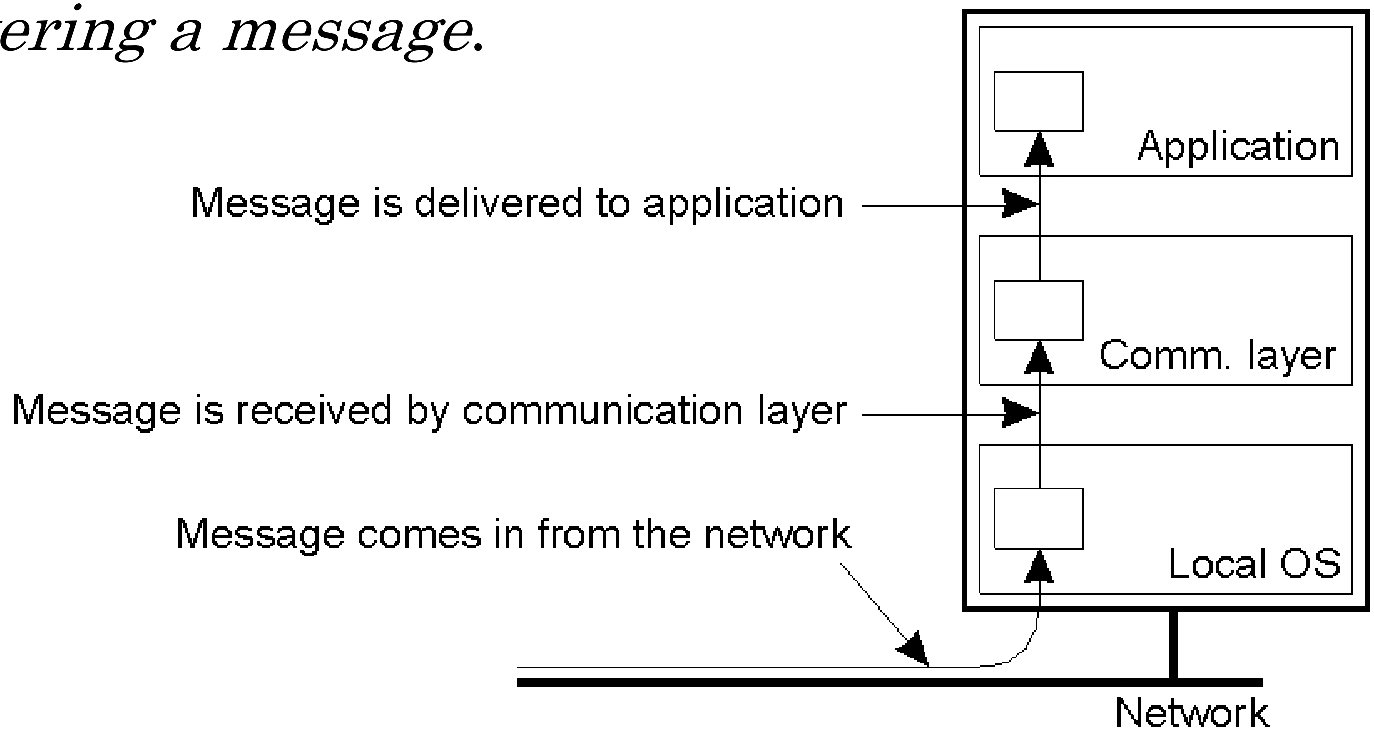


How it should work



Delivering a message

- We'll make a distinction between *receiving* and *delivering a message*.



- You might have already seen this with TCP:
  - messages might be *received* out-of-order;
  - but TCP buffers them and *delivers* them in order to the application.

READING

- https://doc.akka.io/docs/akka/current/guide/actors-motivation.html
- https://doc.akka.io/docs/akka/current/guide/actors-intro.html

Vector clock for "Bulletin Board"

Each process i has an array V_i where $V_i[j]$ denotes the number of events that process i knows have taken place at process j.

- In this application "events" refers to the sending of a message.
- Thus if $V_i[j] = 6$ then i knows that j has sent 6 messages.

Vector clock for "Bulletin Board"

- Just before S sends a message, it does the following:

  $$V_S[S] = V_S[S] + 1;$$

- This is basically saying that the number of messages process S has sent is incremented by one.

- It includes V_S into the outgoing message.

Vector clock for "Bulletin Board"

- Message m (from S) is delivered to R iff both the following conditions are met:
  - $V_S[S] = V_R[S]+1$
    - Meaning that m is the next message that R was expecting from process S
  - $V_S[i] \leq V_R[i]$ for all $i \neq S$
    - Meaning that at the moment of sending m the sender S did not see more messages than the receiver R did.
- Otherwise the message is buffered

Vector clock for "Bulletin Board"

Before **delivering** a message from process S, process R updates its own vector clock with the timestamp V_S from the message:

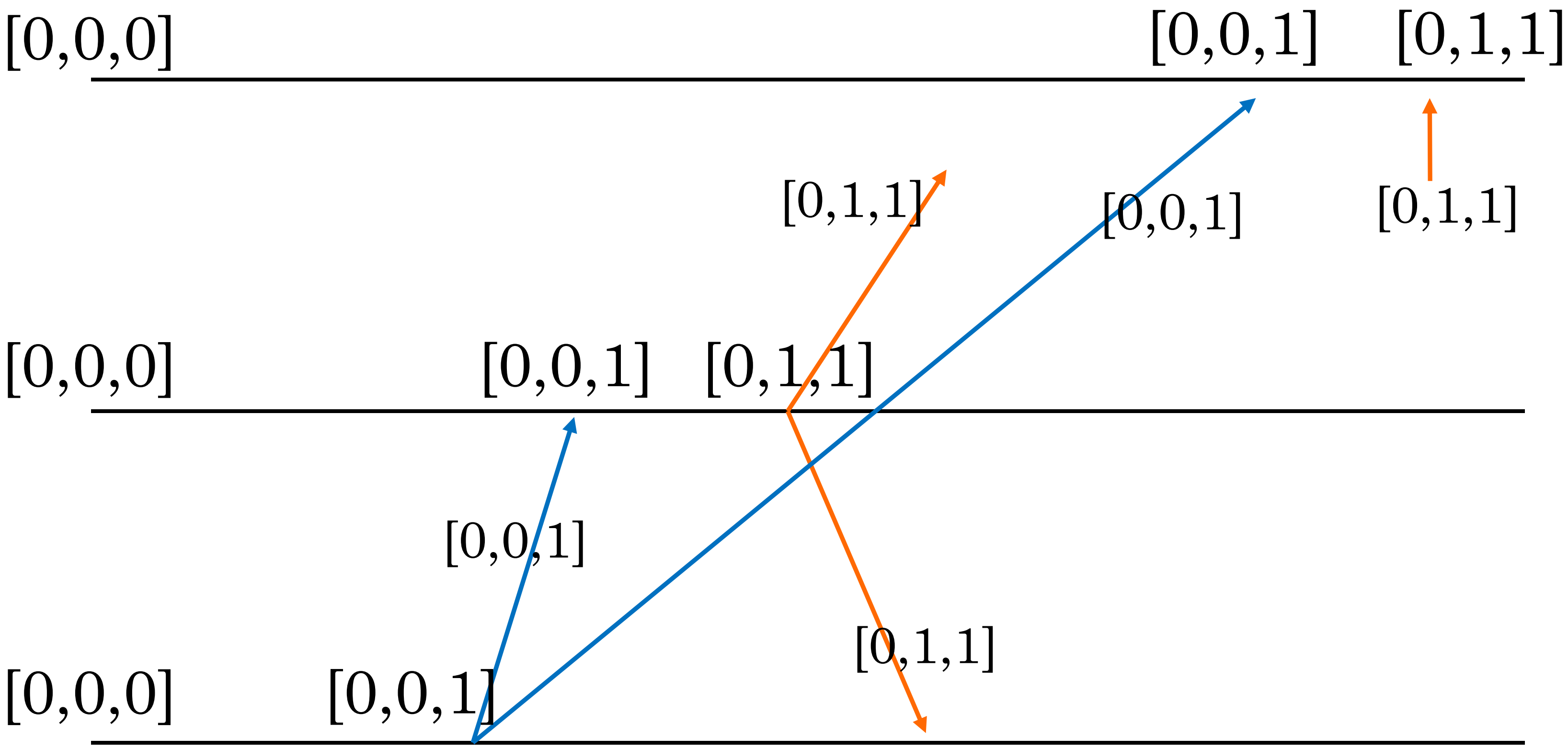
for k = 1 to n do

$V_R[k] = \mathbf{max}(V_R[k], V_S[k]);$

Delivering buffered messages

Each time a message is delivered (and therefore the local vector clock is updated), go through the messages on hold to check if now they can be delivered as well.

An example execution



Implementation hints

```
// message buffer
private List<ChatMessage> buffer =
                                new ArrayList<>();


private void updateLocalClock(ChatMsg m)
{...}


private boolean canDeliver(ChatMsg incoming)
{...}


// find a message that can be delivered
// now and remove it from the buffer
private ChatMsg findDeliverable() {...}
```

IMPLEMENTATION HINTS

- In `sendChatMsg()`, increment the vector clock element of the current actor
- In `onChatMsg(ChatMsg msg)`:
  - If cannot deliver `msg`, add it to the buffer
  - Otherwise, deliver it, as well as all other messages from the buffer that can be delivered now
  - Update local clock before delivering a message

Java hints

```
import java.util.Iterator;

// iterate over the message buffer
Iterator<ChatMsg> I = buffer.iterator();

while (I.hasNext()) {
  ChatMsg m = I.next();
}

// Remove the current element
I.remove();
```

Java hints

```java
import java.util.Arrays;

// print an array
Arrays.toString(this.vc)
```