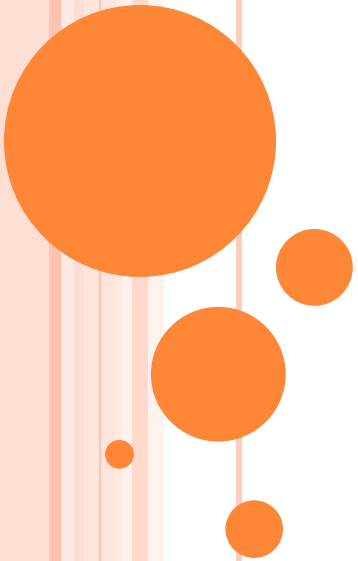


Distributed Systems 1: Lab 3, Virtual Synchrony

davide.vecchia@unitn.it

Reliable multicast

- Definition: A process is **correct** if it does not fail at any point.

- Reliable multicast of message *m* satisfies:
  - **Validity** - if the sender is correct, then it will eventually deliver *m*.
  - **Integrity** - a process delivers *m* at most once, and only if it was previously sent.
  - **Agreement** - if a correct process delivers a message *m*, then all correct processes deliver *m*.

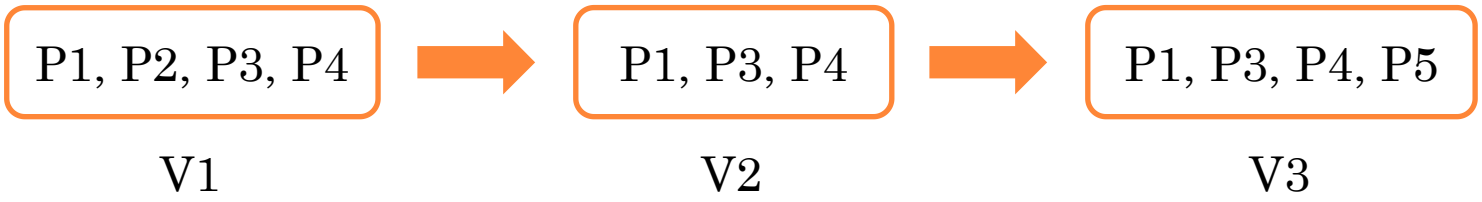
"All-or-none" multicast

Group view and view change

- **Agreement**: if a correct process delivers a message *m*, then all correct processes deliver *m*.

Nodes may join or leave the system, or crash…
The group of correct processes will change over time.

- Goal: consistent **group view** across all processes.
  - Add/remove nodes from the group with a **view change**.



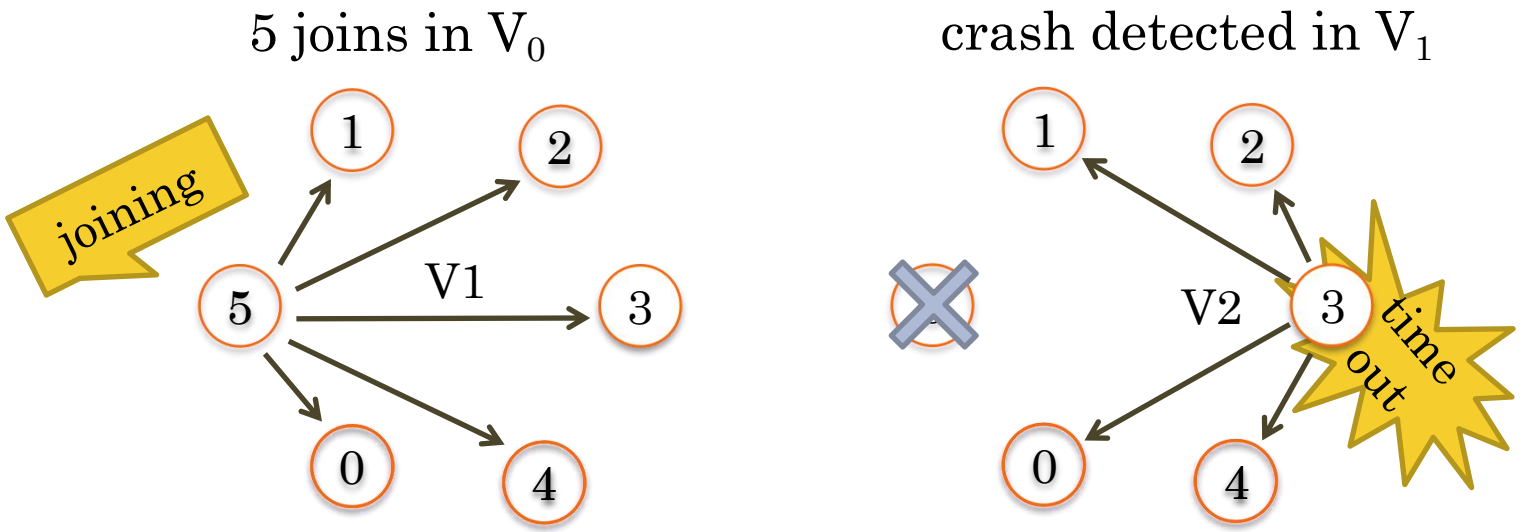
Virtual Synchrony

- Enables reliable multicast within a group of processes that can change over time.
- All processes maintain a view of the group, and correct processes will see the same sequence of views.
  - Therefore, each view defines a **global epoch**.
- Multicast messages cannot cross epoch boundaries.
  - New views cannot be installed until all multicasts in the previous views complete.
  - or, equivalently: if a correct process delivers a message *m* sent in epoch E, then all correct processes deliver *m* within epoch E.

"All-or-none"

VS IMPLEMENTATION (IN A NUTSHELL)

- A view change message is sent in multicast when a node(s) joins or leaves, or when a crash is detected.



- Before installing the view, all-to-all **FLUSH** messages are used to ensure all messages belonging to the old view have been delivered.

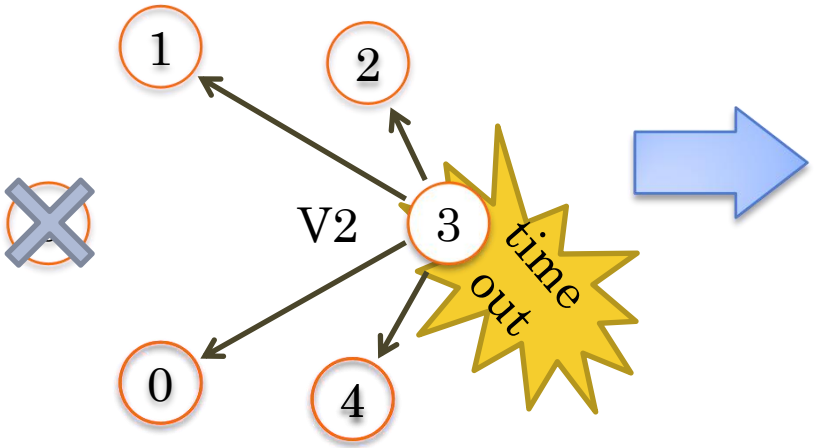
Exercise template

- `VirtualSynch.java`, `VirtualSynchActor.java`, `VirtualSynchManager.java` – incomplete VS implementation
  - Single-phase multicast: the initiator announces the message is stable right after sending it to the last recipient in the group.
  - No multicast ordering.
  - Nodes can join and "crash".
  - Upon receiving a view change message, nodes will install the view immediately!

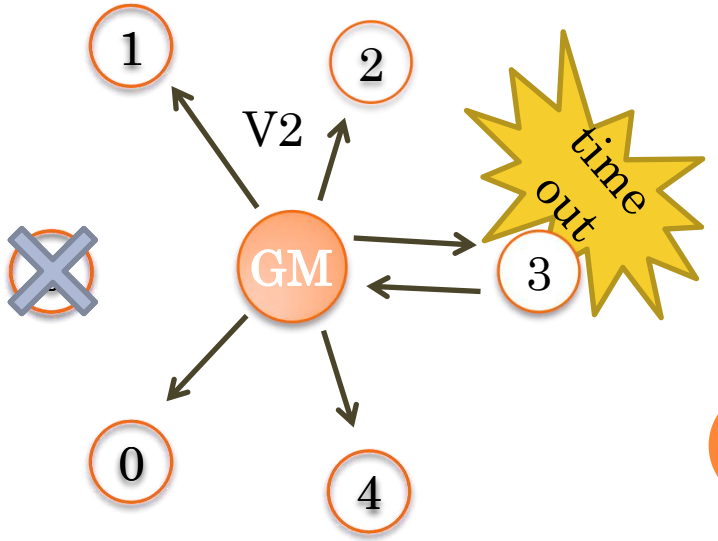
Simplified algorithm

- Implementing distributed group membership is non-trivial: multiple view changes may be issued concurrently, leading to an inconsistent state.
- Simplifying assumption: a special actor, the **group manager**, is the only actor that can propose a view change.

crash detected in V_1, general algorithm



simplified algorithm: node 3 reports the crash to the GM



Checking correctness

- Start the program and let it run the for some time
  - `gradle run`
- Save the output to `test.log`
  - `java -cp $AKKA_CLASSPATH:. VirtualSynch > test.log`
- Compile and run the provided `Check.java`
  - `javac Check.java`
  - `java Check test.log`
- The output should look like this:

```
View: 0 Nodes: 2 Sent: 85 Recv: 85
```

Checking correctness (join and crash)

- The VS system works correctly only if, in each view, the number of messages sent and received is the same.
- Otherwise, it means either reliable multicast properties are not respected or messages crossed epoch boundaries!
- Check again the output of the algorithm, but this time make **new nodes join** the system, then **simulate the crash** of one of them.

The incomplete VS implementation is not correct. Why?

Fixing the exercise template

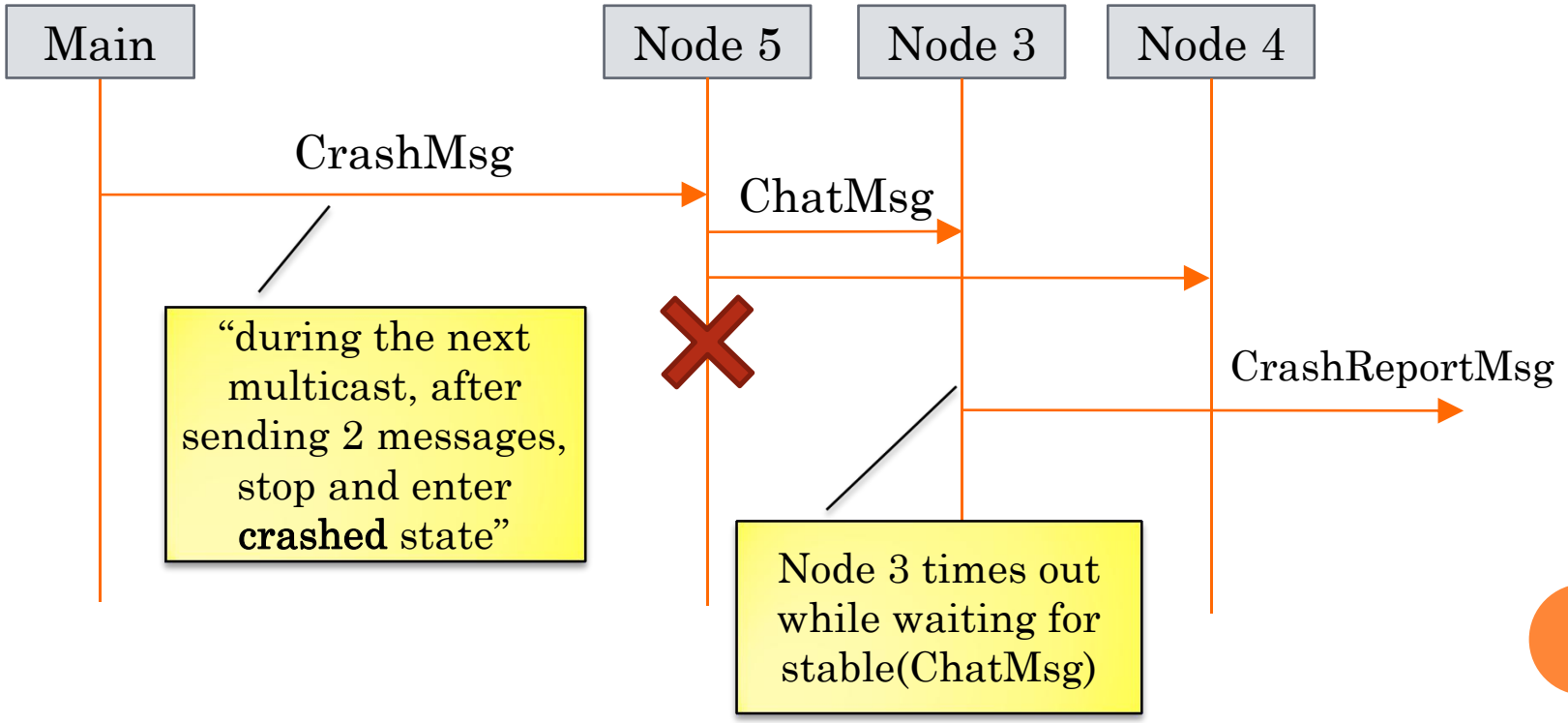
- Assuming reliable FIFO channels, all correct nodes should deliver all messages sent in the same view:

```
View: 0 Nodes: 2 Sent: 44 Recv: 44
View: 1 Nodes: 3 Sent: 8 Recv: 8
View: 2 Nodes: 4 Sent: 61 Recv: 61
View: 3 Nodes: 3 Sent: 5 Recv: 5
View: 4 Nodes: 2 Sent: 72 Recv: 72
View: 5 Nodes: 3 Sent: 3 Recv: 3
View: 6 Nodes: 4 Sent: 51 Recv: 51
...
```

- Fix the incomplete implementation:
  - Upon a view change, multicast **unstable** messages
  - Collect all **FLUSH** messages *before* installing a view
  - What if a node crashes during the **FLUSH** protocol?

Simulating a crash

- If a node times out while waiting for a message, it will report to the group manager that the sender crashed.
- How do we make a node "crash"?



Actor behavior: become

- We need to define the behavior of an actor in the **crashed** state. It should stop sending messages and it should discard incoming ones.
- One option is to keep a `state` variable and check the actor state in each message handling method before any other action. This makes the code less readable and more difficult to maintain.
- Instead, Akka allows the programmer to define multiple receive behaviors and to switch between them using:

```
getContext().become(newBehavior)
```

Switching actor behavior

- In the exercise template, 2 behaviors are defined.

```
@Override
public Receive createReceive() {
  return receiveBuilder()
    .match(ChatMsg.class,  this::onChatMsg)
    .match(StableChatMsg.class,  this::onStableChatMsg)
    .match(StableTimeoutMsg.class,  this::onStableTimeoutMsg)
    .match(ViewChangeMsg.class,  this::onViewChangeMsg)
    .match(CrashMsg.class, this::onCrashMsg)
        ...
    .build();
}
```

Default Receive behavior

```
final AbstractActor.Receive crashed() {
  return receiveBuilder()
    .match(RecoveryMsg.class, this::onRecoveryMsg)
        ...
    .matchAny(msg -> {})
    .build();
}
```

ignore messages not matched

to switch: `getContext().become(crashed())`