Aalto University
School of Science

# CS-E4110 Concurrent Programming

Week 4 – Exercise session

2021-11-26

14:15 – 16:00

Jaakko Harjuhahto

# Today

- Distributed computing
- Partial order
- Actor model
- The Akka toolkit
- General Q&A

# Distributed Computing

# Beyond Threads

- Thus far we have taken a thread centric view
  - Threads as independent streams of execution
  - Communicate by reading and writing to shared memory
- There is a fundamental limit to what can be done with a single computer
  - Resource driven: need more processing power, memory
  - Problem driven: multiple computers must interact to solve their own local problems
  - Reliability driven: more than one computers are needed to guarantee that the system survives a single faulty computer
- Concurrent systems beyond a single computer (= node) must adopt a different strategy for concurrent programming
  - Without shared physical memory, threads cannot interact just by reading and writing

# Multiple Levels of Concurrency

- Distributed system with multiple nodes
- Threads within a single node
- Data and instruction level parallelism

# Message Passing

- Nodes in a distributed systems interact with by sending and receiving messages
  – No shared memory available, as with threads
- Often integrated into a framework or toolkit to abstract away at least some of the challenges with communication
  – Communication patterns
    - Channels, queues, broadcast, publish/subscribe, etc
  – Message ordering
    - Total order, topic order, etc
  – Delivery guarantees
    - At-most-once, at-least-once, exactly-once
  – And more...

# Message Interleaving

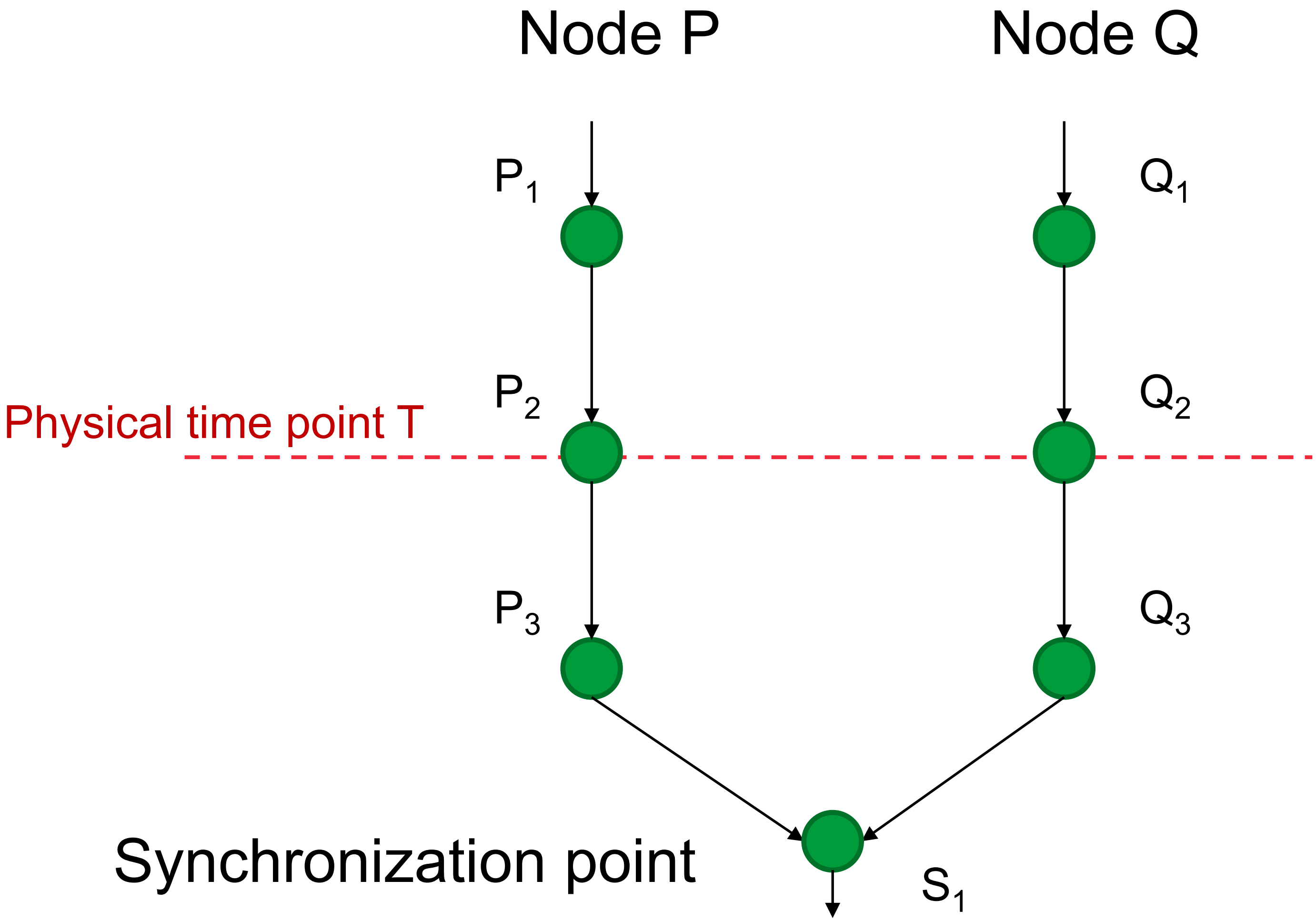
- We don’t know the order in which messages are received and must reason about all possible message schedules
  - Sounds familiar?
- Our computational model of states and state transitions still works!
  - Transitions are not memory reads and writes, but instead receiving and sending messages
  - We can reason about distributed systems

# Some Select Challenges in Distributed Computing

- Hard or impossible to know system state
  - Sharing node state always includes a delay, the original state has changed before other nodes receive the state message
- More failure modes
  - A single computer usually either works fully or doesn't work at all
  - A distributed system can have partial and/or temporal failures, either for nodes or communication channels
- Many distributed algorithms are complex
  - E.g. Paxos and it's variants for consensus
- Many more...

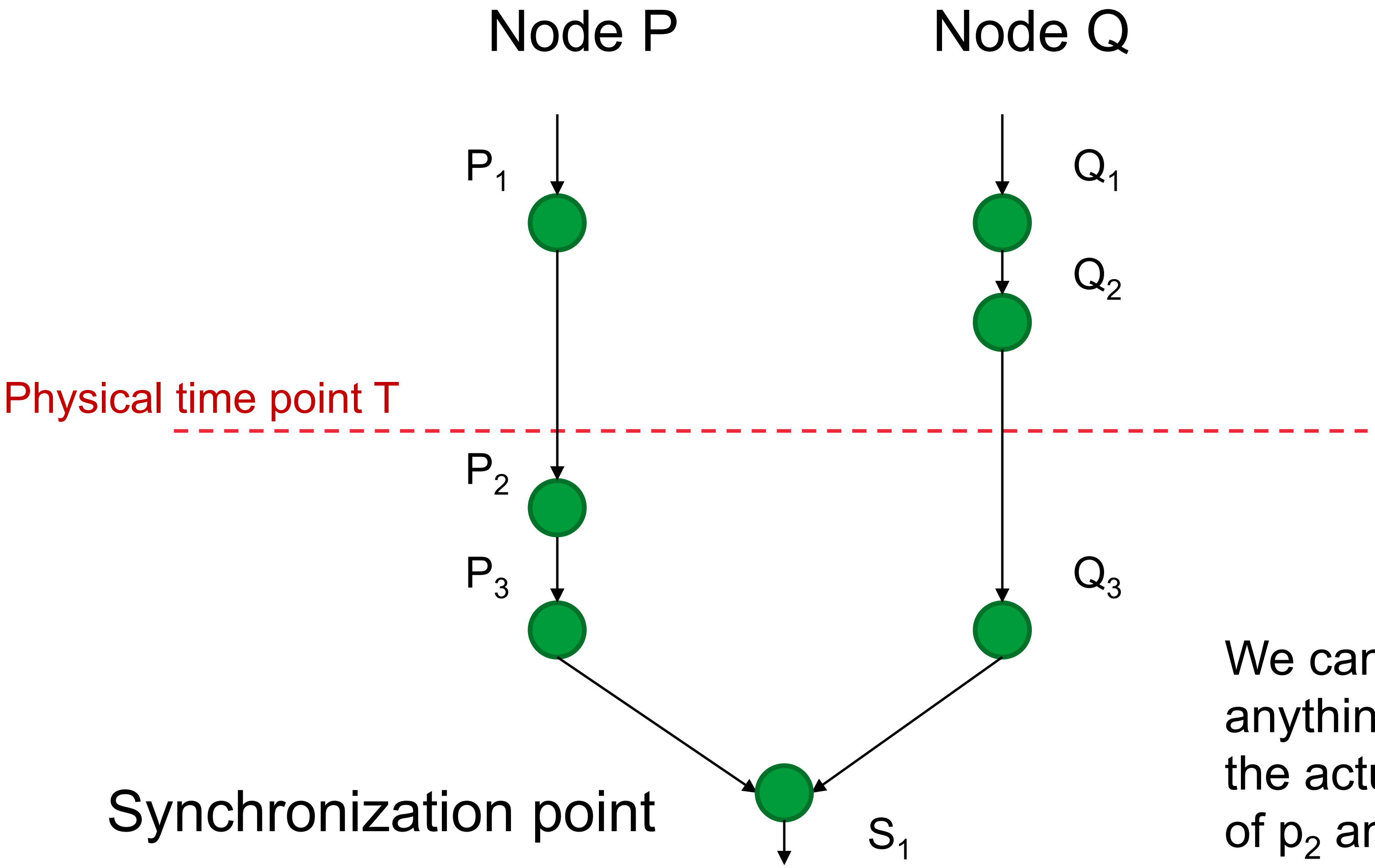
# Partial Order

# Partial Order



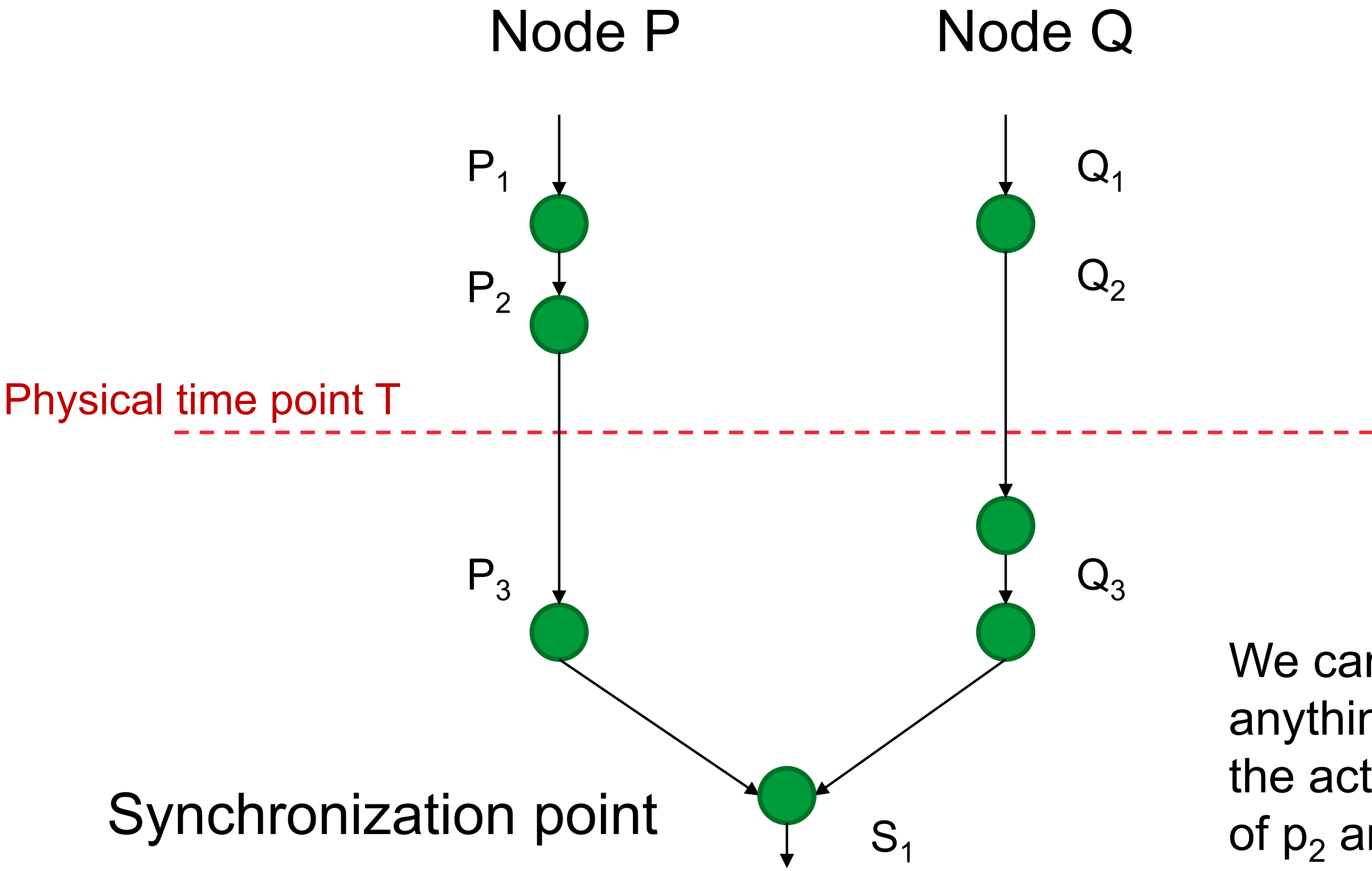

We know:
$hb(p_3, s_1)$
$hb(q_3, s_1)$
$hb(p_2, p_3)$
$hb(q_2, q_3)$
$hb(p_2, s_1)$
$hb(q_2, s_1)$

# Partial Order




We can't say anything about the actual order of $p_2$ and $q_2$

# Partial Order




We can't say anything about the actual order of $p_2$ and $q_2$

# Actor Model

# Motivation

- Thus far we have implemented concurrent data structures and programs using monitors, locks, atomics and threads working with shared memory
  - Relatively low level concepts
  - Explicitly thinking about concurrency
- Explicit low level concurrency is hard
  - Composability is a challenge
  - Adding new elements requires understanding how the entire system is designed and works

# Restructuring

- Split the program into independent concurrent and/or parallel parts, let the runtime, framework or toolkit manage execution
  - Paraphrasing: ”explain **what can be** executed concurrently, let someone else worry about how and when it is executed”
  - Map units of execution onto available threads
  - The execution environment offers a programming model: abstractions and guarantees on behaviour

# The Actor Model

- The actor model eliminates shared state
  - Actors have their private state (= variables)
  - Actors interact by sending immutable messages
- Actors are location transparent
  - Actors can run on the same computer or on different computers, or on different computers at different times
- As a happy side effect this model works both for concurrency on a single computer and an entire distributed system!

# The Akka Toolkit

# Akka

- De facto reference implementation of actors for Scala
  - Available also for Java and .NET
- Two variants of Akka Actors
  - Classic and Typed
    - The most significant change from classic to typed is actor type safety in regards to message types
  - We will use Akka Classic
  - https://doc.akka.io/docs/akka/current/index-classic.html
- The Akka ecosystem is much larger
  - Streams, HTTP, Clusters, Sharding, Persistence, etc
  - The Play framework for web apps

# Akka Actors

- Akka actors are implemented as classes extending the base class `Actor`
  - Must implement the receive function '`def receive: Receive`'
  - Note the actors are implemented as classes and objects, but should not have public methods or public variables
- Program logic is implemented in the `receive` function
  - Receive messages, perform correct actions based on message type and message contents
  - Actor state transitions are reactions to messages

# More Akka Actors

- Actors are lightweight and creating new actors is cheap, unlike processes or threads
  - Concurrency through ”just create more actors...”
  - Typically even single requests or functions can be actors
  - The Akka runtime will effectively use system resources to run the entire actor systems
- Actors are hierarchical
  - An actor supervises the actors it has created, i.e. it’s children
  - If the children fail, the parent must handle how this error is handled

# Messages

- Messages must not contain mutable shared data
  - Consider the case of a distributed actor system, messages must be serialized for transfer over the network, as distributed nodes do not share memory
  - Actors are location transparent, you do not know which machine will process an actor's `receive` invocation
  - Avoid object references, if the objects have mutable state
- Scala case classes are immutable and well suited for messages
  - If necessary, implement serialization for the message classes
    - I.e. How to turn your class instances into JSON, ProtocolBuffers, binary blobs

# Q&A