March 2025 | **Alexandru Bita**

Concurrency 101: the Actor Model, with a glimpse into Apache Pekko

Table of Content

01 Concurrency in a nutshell

Concurrency in a nutshell

Concurrency = **multiple computations** are happening **at the same time**.

- *Multiple computers in a network*
- *Multiple applications running on one computer*
- *Multiple processors in a computer / multiple processor cores on a single chip*

Advantages: background calculations, non-blocking IO, exploiting multi-core processors, etc.

- *High responsiveness*
- *Scalability*

Concurrency in a nutshell

Concurrency = **multiple computations** are happening **at the same time**.

- *Multiple computers in a network*
- *Multiple applications running on one computer*
- *Multiple processors in a computer / multiple processor cores on a single chip*

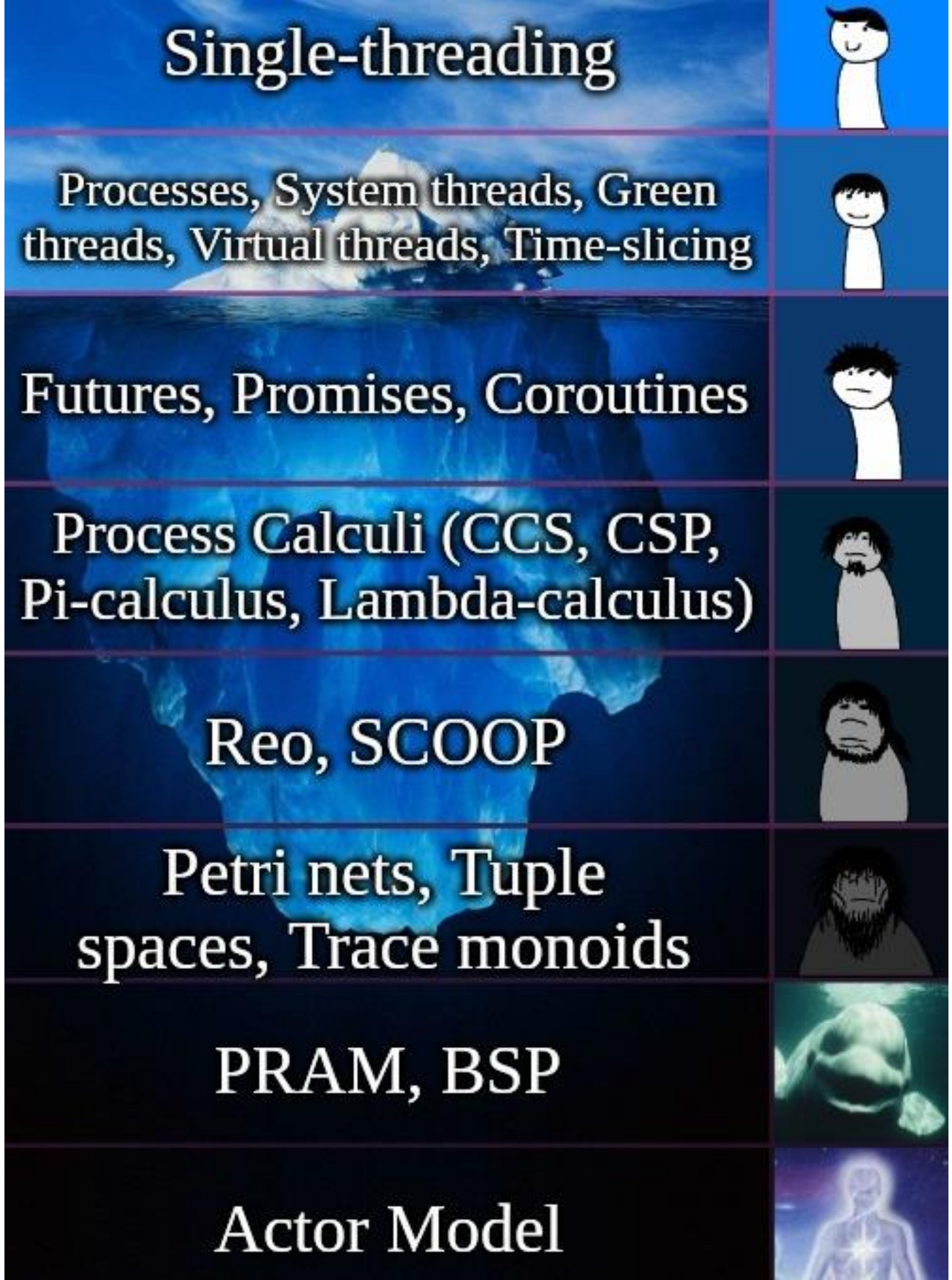
Advantages: background calculations, non-blocking IO, exploiting multi-core processors, etc.

- *High responsiveness*
- *Scalability*

Concurrency in a nutshell

Building concurrent solutions is complicated…

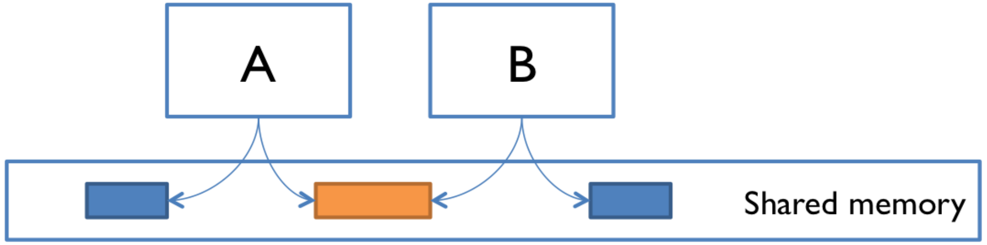
… but it gets easier when you pick the right tools



Concurrency in a nutshell

Shared State

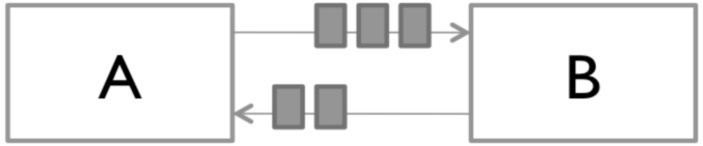
Interaction = read & write shared objects in memory.



- *processors (or processor cores) in the same computer,*
- *programs running on the same computer,*
- *threads in the same Java program, etc.*

Message Parsing (Private State)

Interaction = directly/indirectly send messages to each other. Incoming messages are queued up for handling.



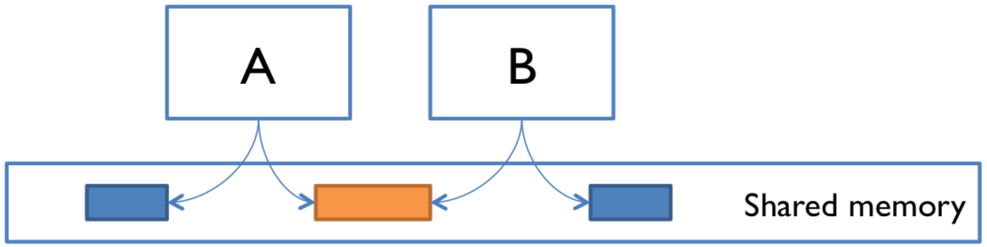
- computers in a network,
- client and server,
- programs running on the same computer with "pipe-able" input/output `ls | grep`, etc.

levi nine

Concurrency in a nutshell

Shared State

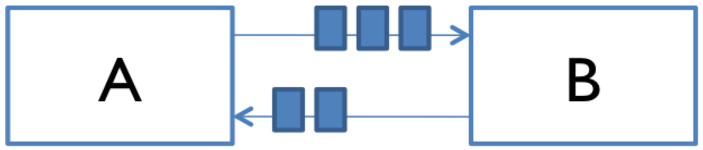
Interaction = read & write shared objects in memory.



- *processors (or processor cores) in the same computer,*
- *programs running on the same computer,*
- *threads in the same Java program, etc.*

Message Parsing (Private State)

Interaction = directly/indirectly send messages to each other. Incoming messages are queued up for handling.



- computers in a network,
- client and server,
- programs running on the same computer with "pipe-able" input/output \`ls | grep\`, etc.

02 | Difficult aspects of "traditional" concurrency

Difficult aspects of "traditional" concurrency

```java
static int balance = 0;

static void deposit() {
    balance = balance + 1;
}

static void withdraw() {
    balance = balance - 1;
}

static void makeTransactions() {
    for (int i = 0; i < NUM_OF_TRANSACTIONS; i++) {
        deposit();
        withdraw();
    }
}
```

```java
public static void main(String[] args) {
    for (int i = 0; i < NUM_OF_ATMS; i++) {
        new Thread(Main::makeTransactions).start();
    }

    Thread.sleep(SLEEP_TIME);
    System.out.println(balance);
}
```

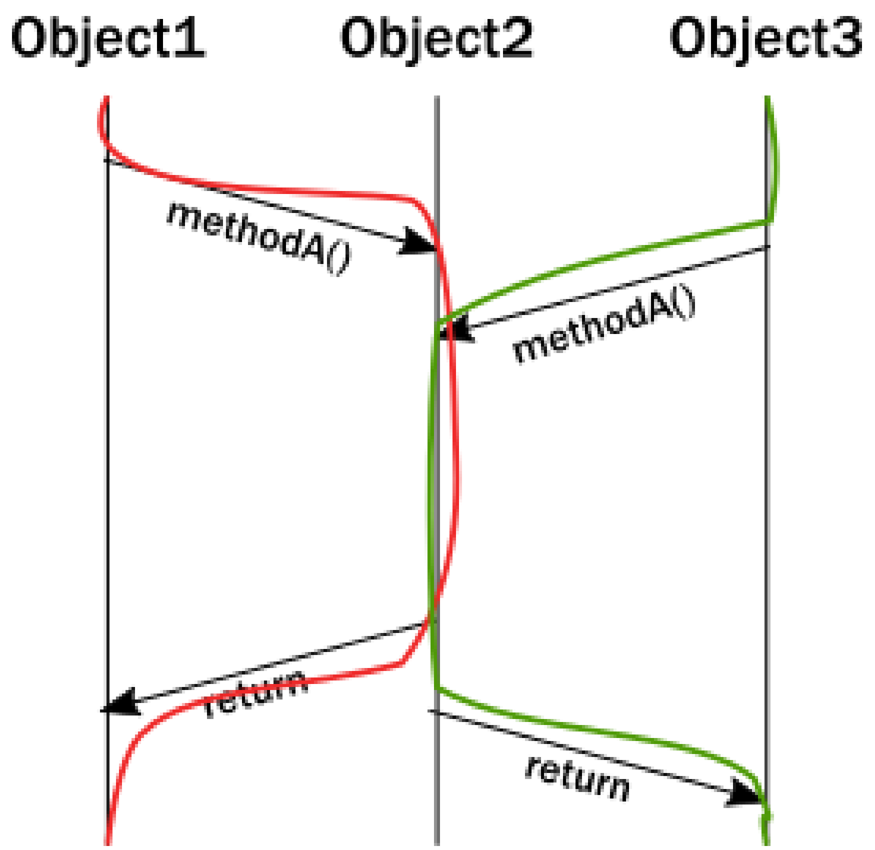
What's wrong with this code?

Difficult aspects of "traditional" concurrency

Interleaving

- A get balance (balance = 0)
    - B get balance (balance = 0)
- A add 1
    - B add 1
- A write result (balance = 1)
    - B write result (balance = 1)

-> Race condition



Difficult aspects of "traditional" concurrency

Shared memory is not so shared

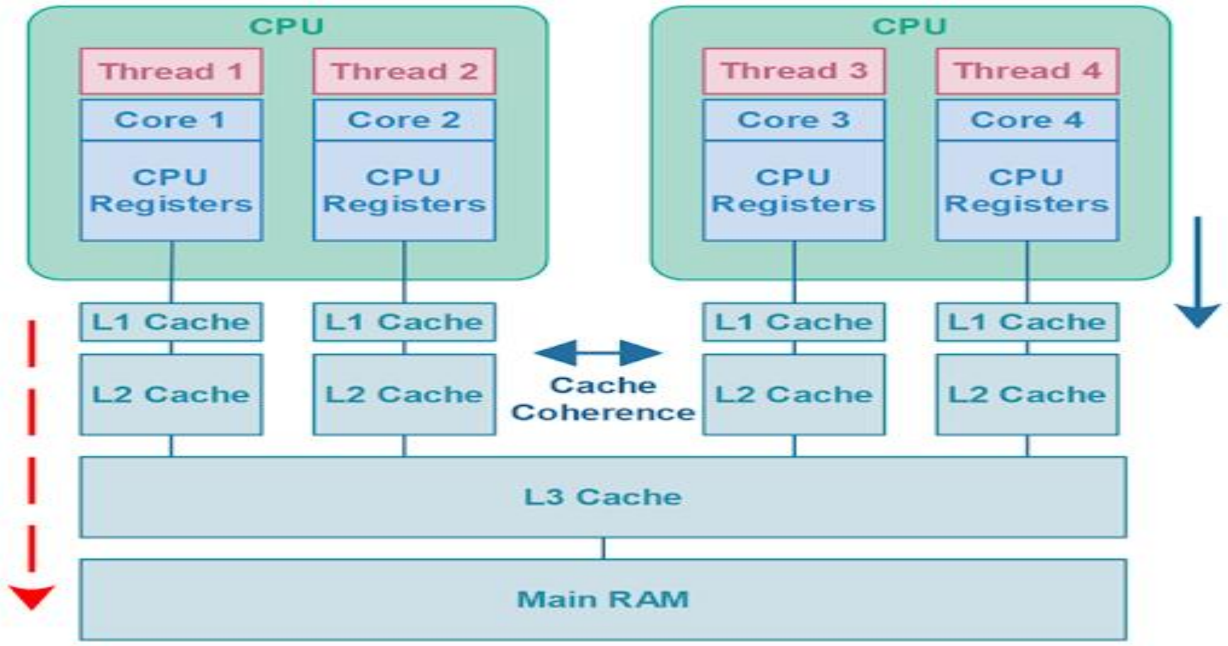
- ❖ CPU cache lines
- ❖ Global memory? Cache coherence, son.
  - ➢ Costly

Synchronization hell

- ❖ Mutual exclusion (mutex), Locks, Semaphores, Monitors, Latches, etc. Coordinating distributed systems?
  - ➢ Costly
- ❖ Deadlocks, Livelocks
  - ➢ Deadly
- ❖ Idle time

Exception handling

- ❖ Propagation? Call stack?
- ❖ Who will assume responsibility?



Difficult aspects of "traditional" concurrency

Shared memory is not so shared

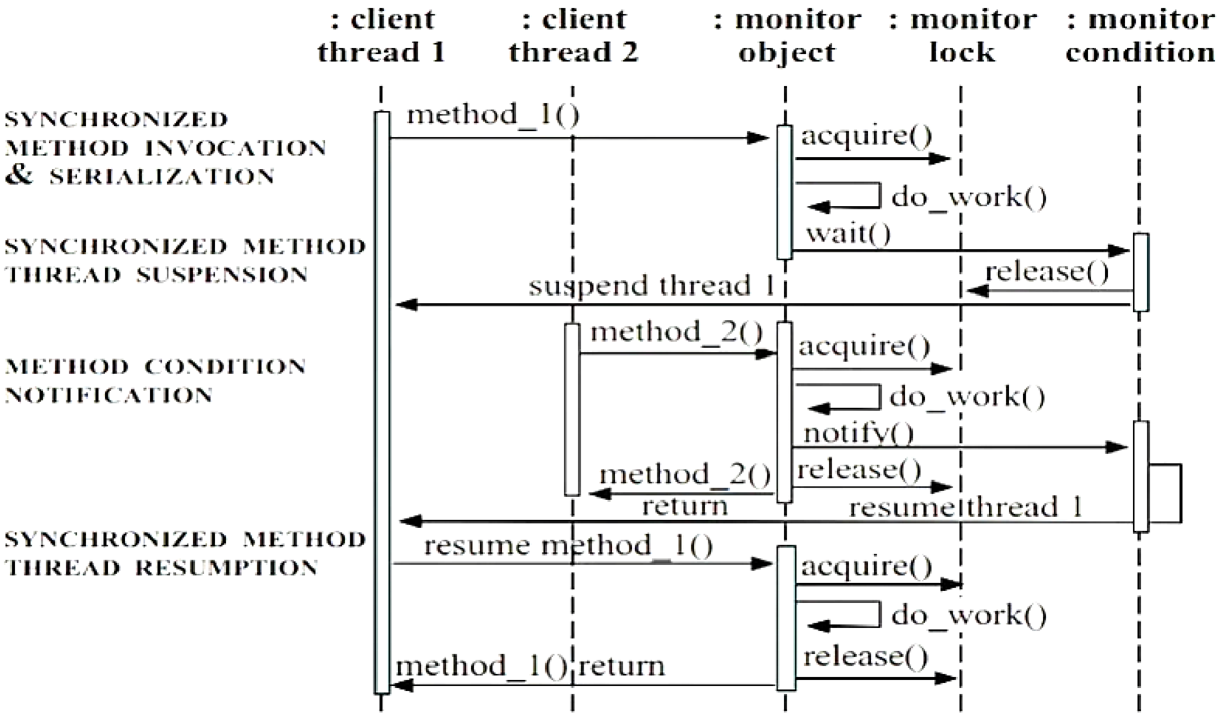
- CPU cache lines
- Global memory? Cache coherence, son.
    - Costly

Synchronization hell

- Mutual exclusion (mutex), Locks, Semaphores, Monitors, Latches, etc. Coordinating distributed systems?
    - Costly
- Deadlocks, Livelocks
    - Deadly
- Idle time

Exception handling

- Propagation? Call stack?
- Who will assume responsibility?



Difficult aspects of "traditional" concurrency

Shared memory is not so shared

❖ CPU cache lines

❖ Global memory? Cache coherence, son.
   ➢ Costly

Synchronization hell

❖ Mutual exclusion (mutex), Locks, Semaphores, Monitors, Latches, etc. Coordinating distributed systems?

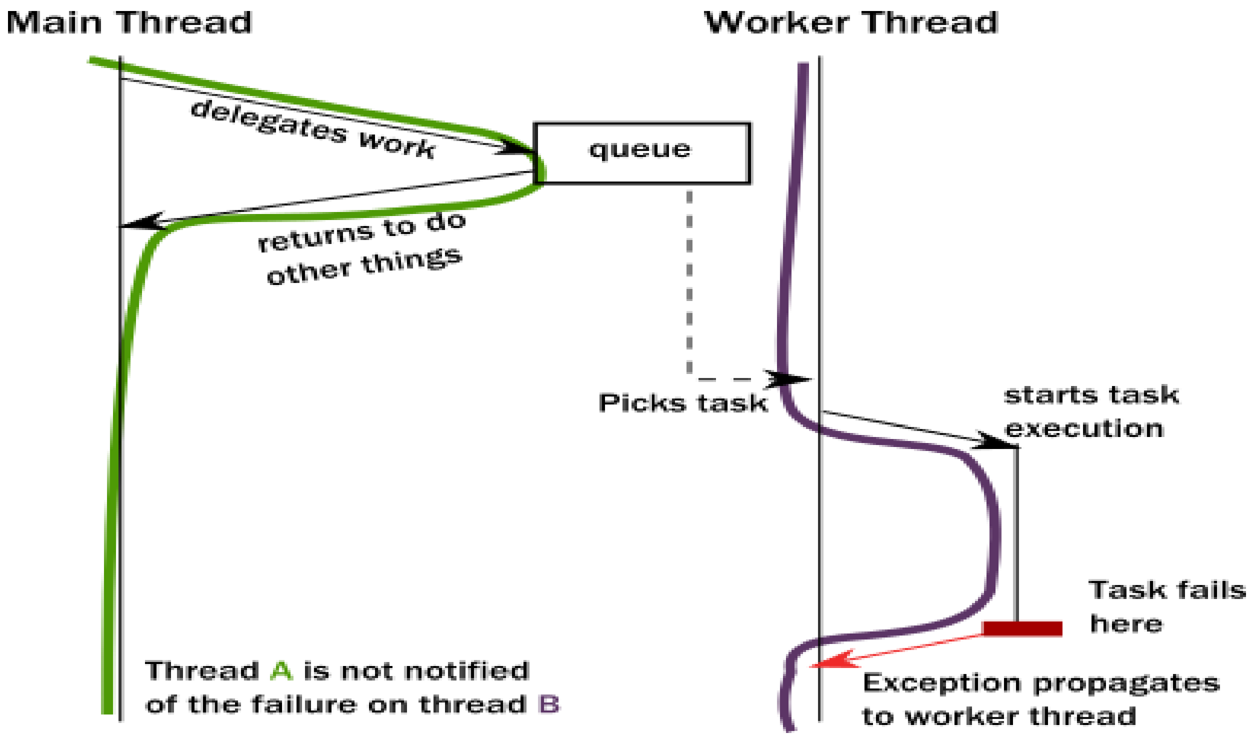
❖ Costly
   ➢ Deadlocks, Livelocks
   ➢ Deadly

❖ Idle time

Exception handling

❖ Propagation? Call stack?

❖ Who will assume responsibility?



Difficult aspects of "traditional" concurrency

Shared memory is not so shared

- CPU cache lines
- Global memory? Cache coherence, son.
  - Costly

Synchronization hell

- Mutual exclusion (mutex), Locks, Semaphores, Monitors, Latches, etc. Coordinating distributed systems?
- Costly
  - Deadlocks, Livelocks
  - Deadly
- Idle time

Exception handling

- Propagation? Call stack?
- Who will assume responsibility?

Race conditions.

Fault tolerance.

Performance.

Complexity.

03 | The actor model

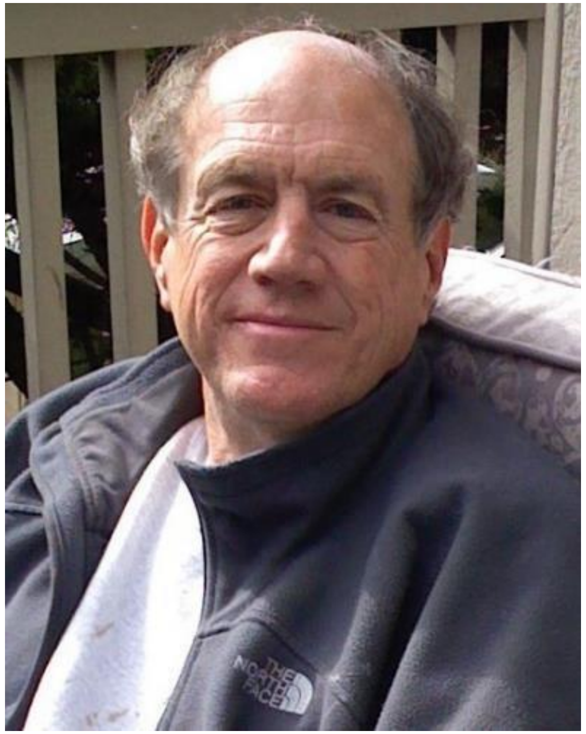
The actor model

Let's watch together: [Actor Model Explained]

Actor = the basic building block of concurrent computation.

- send a finite number of messages to other actors
- create a finite number of new actors
- designate the behavior to be used for the next message it receives

Everything is an actor.



Carl Hewitt – Aptos, CA
(1944–2022)

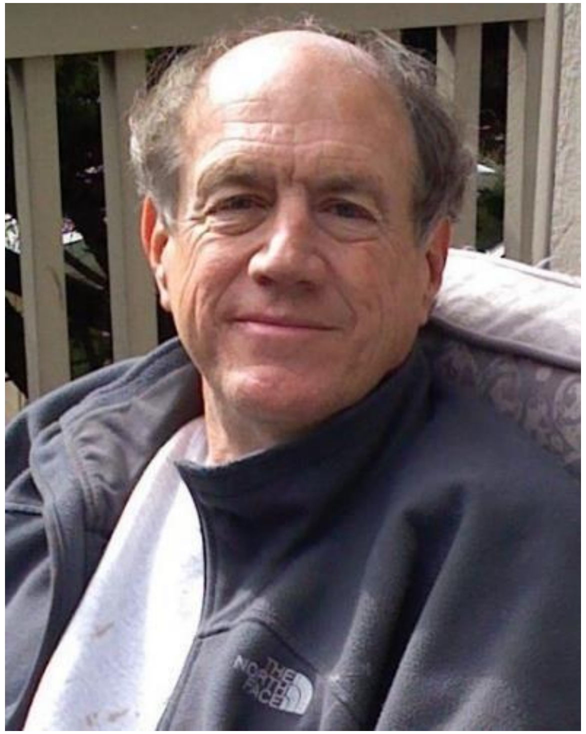
The actor model

Let's watch together: [Actor Model Explained]

Actor = the basic building block of concurrent computation.

- send a finite number of messages to other actors
- create a finite number of new actors
- designate the behavior to be used for the next message it receives

Everything is an actor.



Carl Hewitt – Aptos, CA (1944–2022)

The actor model

Do not communicate by sharing memory; instead, share memory by communicating.

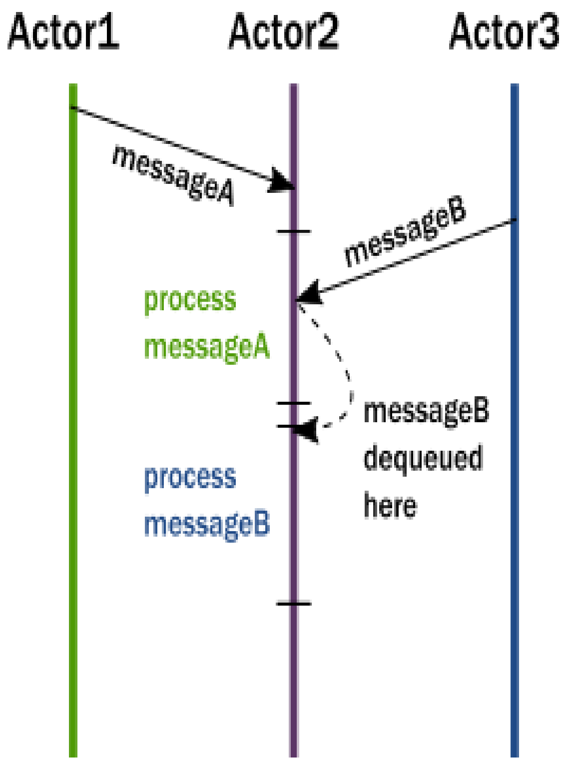
Communication properties:

- Asynchronous
- No channels, no intermediaries
- "Best efforts" delivery -> At-most-once delivery
- Messages can take arbitrary long to be delivered
- No ordering is guaranteed

The actor model

When an actor receives a message:

1. The actor adds the message to the end of a queue.
2. If the actor was not scheduled for execution, it is marked as ready to execute.
3. A (hidden) scheduler entity takes the actor and starts executing it.
4. Actor picks the message from the front of the queue.
5. Actor modifies internal state, sends messages to other actors.
6. The actor is unscheduled.



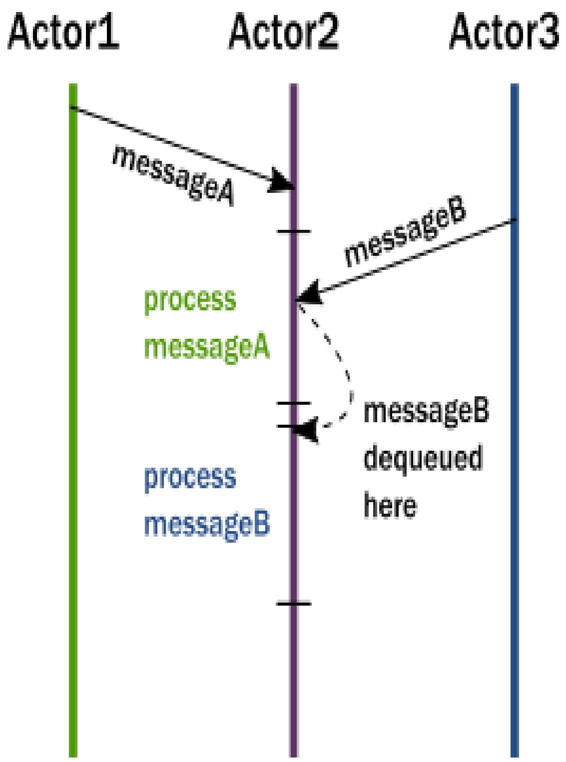
Actors have:

- A mailbox (the message queue)
- A behavior (the state of the actor)
- Messages (data representing a signal)
- An execution environment (the machinery that transparently drives an actor's actions)
- An address (or multiple)

The actor model

When an actor receives a message:

1. The actor adds the message to the end of a queue
2. If the actor was not scheduled for execution, it is marked as ready to execute
3. A (hidden) scheduler entity takes the actor and starts executing it
4. Actor picks the message from the front of the queue
5. Actor modifies internal state, sends messages to other actors
6. The actor is unscheduled



Actors have:

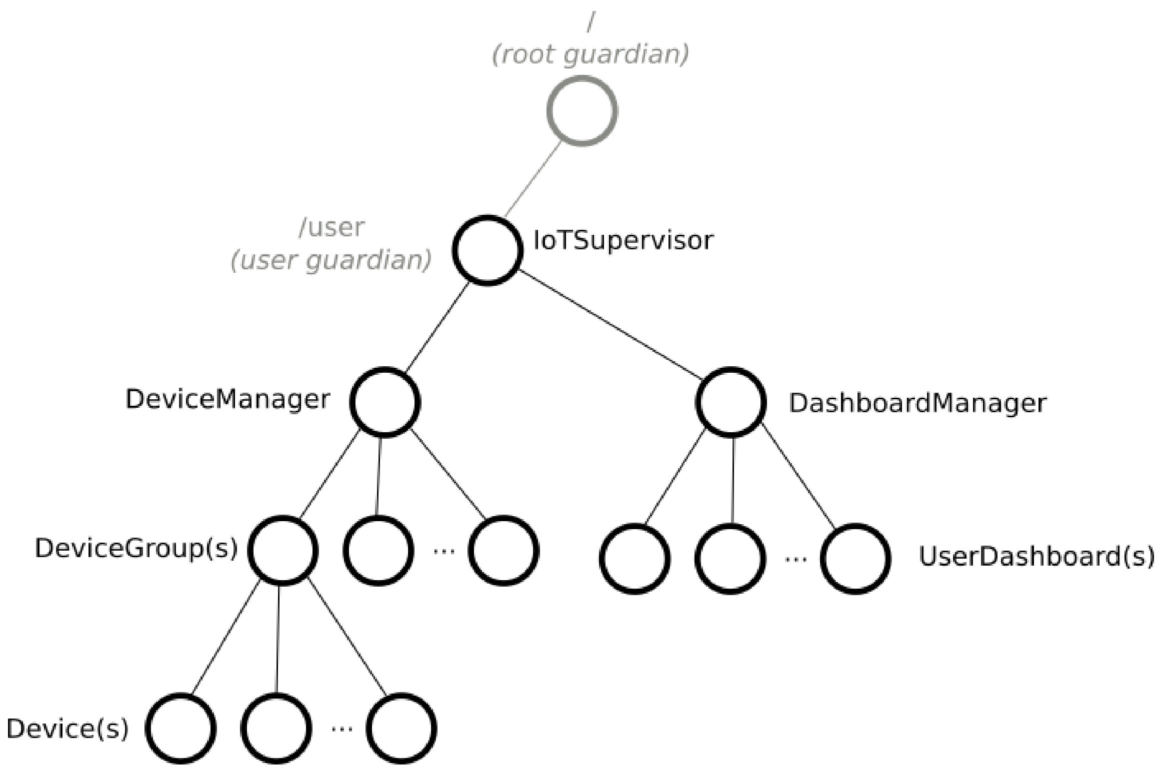
- A mailbox (the message queue)
- A behavior (the state of the actor)
- Messages (data representing a signal)
- An execution environment (the machinery that transparently drives an actor's actions)
- An address (or multiple)

Children must be supervised at all times

The state of an actor is monitored and managed by its parent actor (**Supervisor**).

A strategy is typically defined by the Supervisor when it is starting a child actor.

- React on a child's failure
- Restart/Stop the child as per the strategy
- **Child failures are never silent**
- Other actors can keep on sending messages



The actor model

- ❖ Graceful error handling, Fault tolerance, Self-healing
- ❖ Encapsulation is preserved
- ❖ Local state
- ❖ No race conditions, No need for locks and other mambo-jambos
- ❖ No idle time
- ❖ Easy to scale, Easy to distribute
- ❖ Better refactor-ability
- ❖ Less solution complexity, Improved testing

The actor model

Use-cases:

- Applications with shared state
- Event-driven applications
- Processing pipeline
- Distributed applications
- Highly-concurrent applications (online-games, finance)
- IoT
- Digital twins

Anti-patterns:

- Working on a non-concurrent system
- There is no mutable state
- You need extreme control over critical sections
- You need high composability

The actor model

Use-cases:

- Applications with shared state
- Event-driven applications
- Processing pipeline
- Distributed applications
- Highly-concurrent applications (online-games, finance)
- IoT
- Digital twins

Anti-patterns:

- Working on a non-concurrent system
- There is no mutable state
- You need extreme control over critical sections
- You need high composability

04 | Apache Pekko

Apache Pekko

Apache Pekko is an open-source framework designed to simplify the development of *concurrent*, *distributed*, *resilient*, and *elastic* applications.

Leveraging the Actor Model, Pekko offers high-level abstractions for concurrency, allowing developers to focus on business logic rather than low-level implementation details.

- Open-source successor of Akka 2.6.2
- Project enters incubation on October 10, 2022
- Apache Pekko released version 1.0 on July 13, 2023
- The Apache Pekko project graduated on 2024-03-20

Supports both Java and Scala.

Apache Pekko

Libraries

Actor library: Pekko's core library; actors are used all across other libraries

Remoting: enables actors that live on different computers to seamlessly exchange messages

Cluster: manage multiple actor systems in a disciplined way; provides an additional set of services on top of Remoting that most real world applications need

Persistence: provides patterns to enable actors to persist events that lead to their current state

Streams: to handle streams of events or large datasets with high performance

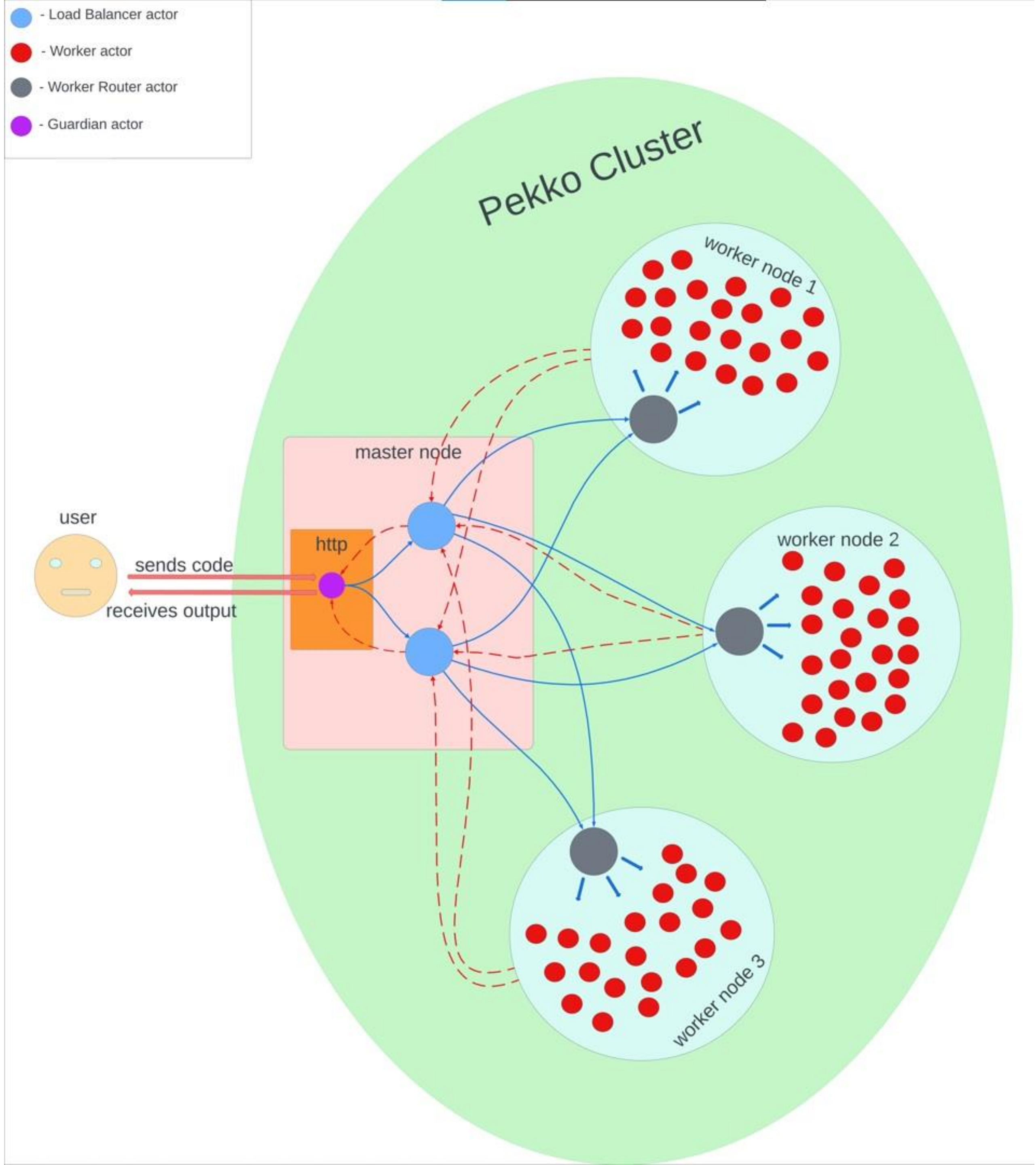
HTTP: the de facto standard for providing APIs remotely, internal or external

gRPC: provides an implementation of gRPC that integrates nicely with the HTTP and Streams modules

Apache Pekko

How does a fairly complex solution based on the actor model actually look like?

Take a look here: [A Distributed Code Execution Engine in Pekko with Scala]



levi nine

05 | Demo

Demo

Problem description

Multiple ATMs are accessing a single bank **B**.

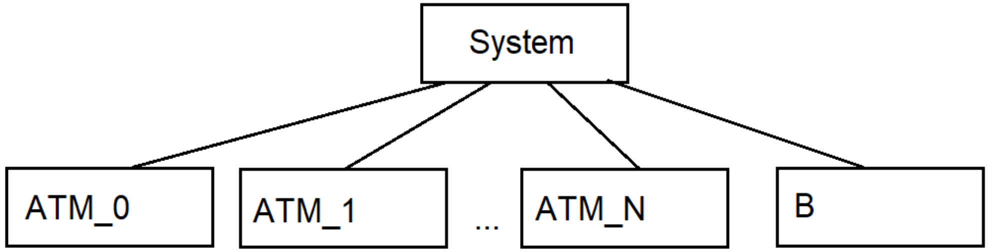
Each one executes a set number of transactions in a day.

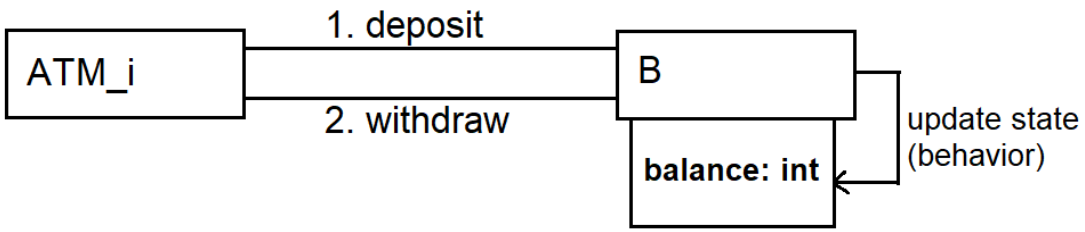
A transaction is made of two ordered tasks:

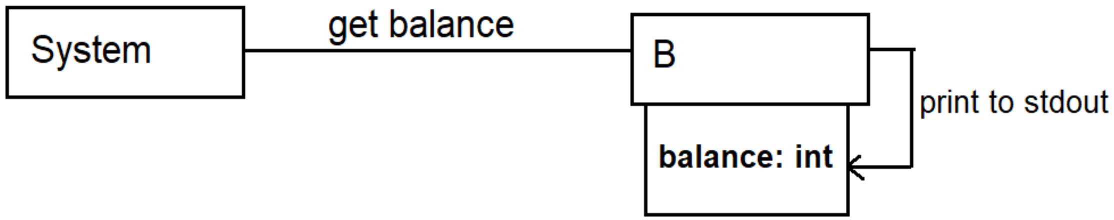
1. **Deposit**: adds 1 monetary unit (*mu*) to **B**
2. **Withdraw**: extracts 1 *mu* from **B**

At the beginning of the day, B holds 0 *mu*.

At the end of the day, after all ATMs executed their transactions, **B** should hold 0 *mu*.







06 Conclusions

Conclusions

The actor model is not perfect. No concurrency model is. Mindfully selecting and combining the most appropriate ones is what we can do best.

Know your tools. Both in **theory** and in **practice** (implementations).

Resources

Read

- Apache Pekko Documentation | Akka Documentation
- Concurrency (MIT)
- Concurrency (InfoUAIC)
- Concurrency (Wikipedia)
- Concurrency Models (Jenkov)
- Actors Model (WikiC2)
- Actor model (Wikipedia)
- Actor Model of Computation (arXiv / Carl Hewitt)
- Deadlock analysis with behavioral types for actors (CEUR-WS / Vincenzo Mastandrea)
- A Distributed Code Execution Engine in Pekko with Scala (RockTheJVM)

Watch

- The Actor Model
- Actor Model Explained
- A beginner's guide to programming with actors
- An introduction to the actor model for software developers
- A brief introduction to the actor model & distributed actors
- When and How to Use the Actor Model An Introduction to Akka NET Actors
- Introduction to the Actor Model for Concurrent Computation: Tech Talks
- The Actor Model (everything you wanted to know...)
- 10 Lessons From Implementing The Actor Model
- LISA17 - The Actor Model

Resources

Bonus

- Actors are not a good concurrency model (Paul Chiusano)
- What's Wrong with the Actor Model (Jaksa)
- Don't use Actors for concurrency (Chris Stucchio)
- Why I Don't Like Akka Actors (Noel Welsh)
- Why has the actor model not succeeded? (Paul Mackay)