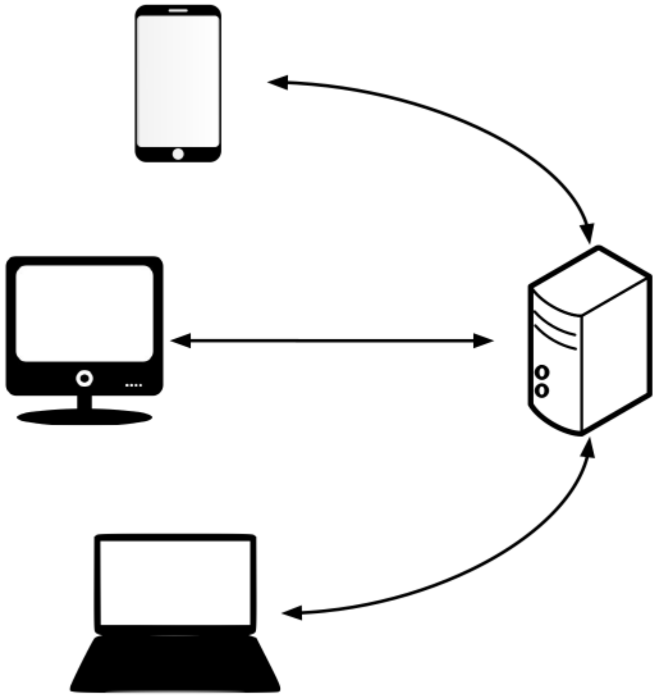
Concurrent Programming

Effective Programming in Scala

Distributed Computations Are Ubiquitous

Nowadays, distributed systems power a lot of applications (e.g., web and mobile applications).

These systems are made of several physical machines (a.k.a. nodes) communicating together:

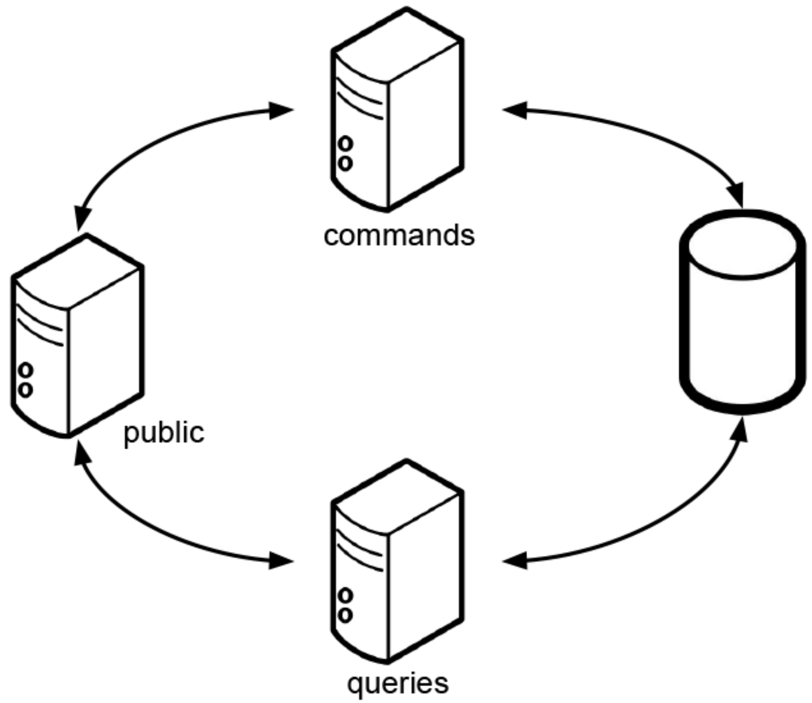


Challenges Raised by Distributed Systems

- What happens if multiple clients modify the state of the system at the same time?
- Can the server handle requests coming from other clients while it is already busy responding to one client?

Distributed Computations Are Ubiquitous (2)

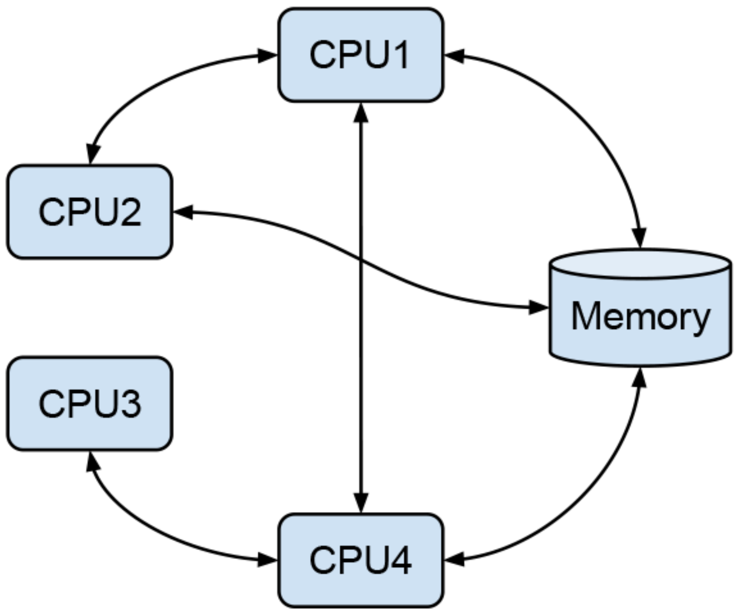
If we zoom on the server node, the same pattern might be used internally:



Similar concerns arise: can the system handle the application of two commands at the same time? Is it possible to read the system state while commands are still being processed?

Distributed Computations Are Ubiquitous (3)

Last, if we zoom again on one node it can use the same pattern again to leverage the multiple CPUs of the machine:



Again: what happens if multiple computations modify the memory at the same time? How can a computation running in a CPU depend on the result of another computation running in a different CPU?

Multi-Threading

- Leveraging multiple CPUs in a single program requires **multi-threading**,
- sharing data between threads requires using **thread-safe** data structures (similarly, sharing data between nodes requires some form of transactions in the underlying database system).

Sharing Data Between Concurrent Programs

- Within a single JVM, use thread-safe data structures such as the ones in the package `java.util.concurrent.atomic`, or `scala.collection.concurrent`,
- Between processes or nodes, you can rely on third-party database systems.

Dependencies Between Computations

Whether you perform a remote call, or you start a computation in a different thread, you eventually need a way to "do something" with the result.

In Scala, in both situations we use the type `Future` to model the result of such computations.

A `Future` represents a value that may not be available yet, but might become available in the future—when whatever is computing it has finished.

Future

As a first example, consider the following signature of a program that inserts a new user in a remote database and eventually returns a result of type `User`:

```
def insertUser(login: String, passwordHash: Seq[Byte]): Future[User]
```

As a second example, consider the following signature of a program that computes the cryptographic hash of a password using the BCrypt algorithm. This program starts the computation on a different thread and eventually returns a result of type `Seq[Byte]`:

```
def bcrypt(saltRound: Int, password: String): Future[Seq[Byte]]
```

Asynchronous Computations

We will see later how to implement an operation that executes code on a different execution thread (locally, or remotely). Let us first get a general idea of what it is like to work with operations returning `Future`.

The operations `insertUser` and `bcrypt` return a result to the caller thread possibly *before* the actual result has been computed.

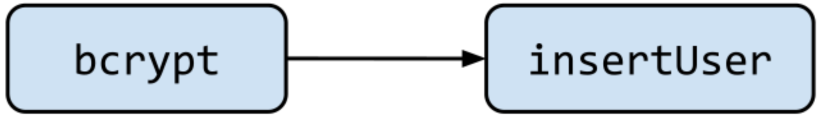
We say that `bcrypt` and `insertUser` are **asynchronous computations**.

The returned `Future` value provides methods to eventually use the actual result *when* it is available.

Manipulating `Future` Values

The most common way to work with `Future` values is to **transform** them into other `Future` values by using operations very similar to the ones we already use with collections or `Try`: `map`, `flatMap` and `zip`.

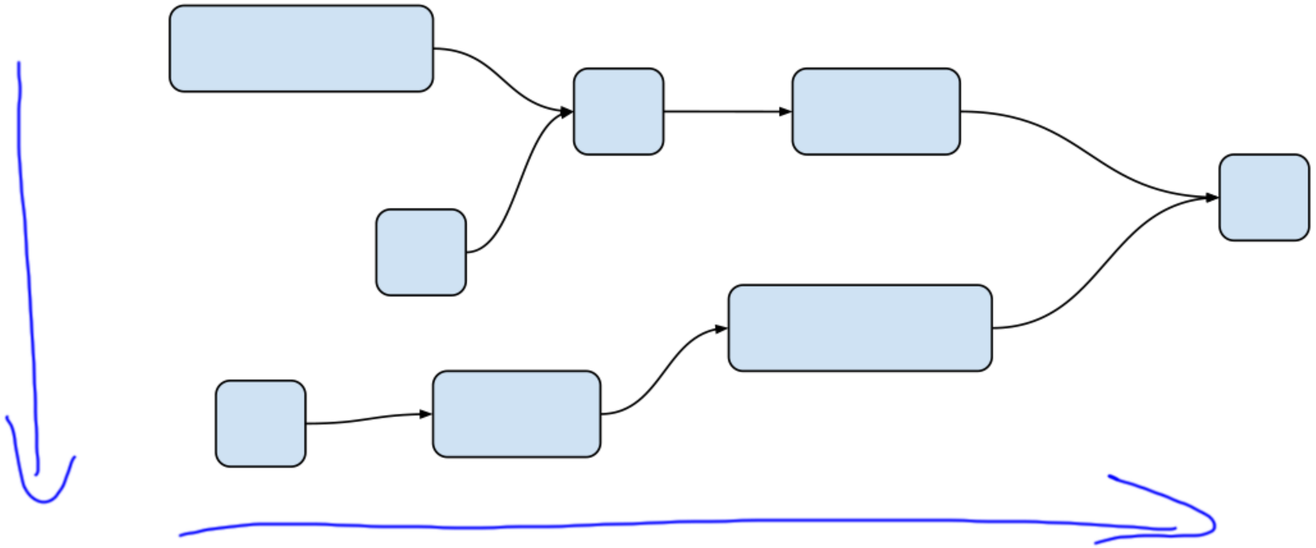
For instance, consider the complete scenario of adding a new user to the system. We first want to compute the user password hash with the operation `bcrypt`, and, after we get the password hash, insert the user row in the database with the `insertUser` operation:



The arrow indicates the direction in which the time flows.

`Future` Values All the Way Down

By combining `Future` values, we end up building trees of computations with sequential and parallel branches:



The result is necessarily a `Future` value again, which is finally consumed by your application framework or testing framework.

Failure Handling

When we run an asynchronous computation, there are two possible outcomes:

- the computation eventually returns a result,
- or, it fails (e.g., an exception is thrown, or the remote service is unreachable).

We will see in the next lesson how to handle such failures.

We can model the state of a `Future[A]` value at any point in time with the type `Option[Try[A]]`:

- `None` means that the outcome of the computation is not known yet,
- `Some(Success(a))` means that the computation succeeded,
- `Some(Failure(e))` means that the computation failed.

Summary

Most systems run on several execution threads.

Sharing data between several threads of execution requires using thread-safe data structures.

For a program running in a given thread, we model the result of computations possibly performed in a different thread with the type `Future`.

We usually combine values of type `Future` to build a graph of computations containing sequential and parallel branches.