Why Erlang?

This course should be devoted to the concepts that are widely implemented in the programming languages available nowadays. So why Erlang only?

The reason is that you already must follow classes in Java and C++, so the **object-oriented programming** is something you are learning anyway.

You also learn C, as a part of C++ or by itself (for system programming), so you are also familiar with **structured programming**.

You also have the possibility to choose my class of *Artificial Intelligence*, which is mainly an introduction to the Prolog programming language, allowing you to gain some understanding of **logic programming**.

Why Erlang? (cont)

What is missing is

- concurrent programming,
- distributed programming,
- functional programming.

Concurrent programming consists in making a software run on separate *threads* or *processes*, i.e. each part of the software is logically running in parallel on the same run-time environment (e.g. a virtual machine or the operating system in case of native-code compilation), therefore on the same operating system (no networking involved).

Why Erlang? (cont)

Of course, if the hardware includes only one processor (CPU), the concurrent execution will not be truly in parallel, but interleaved, i.e. one thread is run for a while, then another is scheduled to use the CPU etc. If several processors are present, true concurrency can be achieved, depending on the compilers.

In the class of *System Programming*, you learn how to program concurrently in C, but the C language, despite it was designed for this task, has no built-in features to handle the concurrency: it relies instead on the operating system, which provides the support for threading and processing.

Because of this dependence by design on low-level layers (i.e. the operating system), which obscures the concepts at stakes, concurrency in C is not an easy task.

Why Erlang? (cont)

Distributed programming is an extension of concurrent programming.

In the former case, the threads are running on different run-time environments on different physical machines, which are interconnected by means of a network (each machine may run several threads of the whole application).

Again, using C for this task, for example, is possible but can be made easier if the programming language includes features for distributing the computations across a network.

Why Erlang? (cont)

Functional programming is a form of *declarative programming* (which includes also logic programming), in which disciple the programmer describes what he wants instead of how to make it.

For example, the language XSLT, which allows the specification of transforms of XML files, is declarative. The query language SQL is also declarative.

Declarative languages are recognisable by the purposeful lack of variables whose values can be modified, in other words: the programmer has no access to the memory whatsoever. In particular, there are no looping constructs, no pointer arithmetics, no explicit dynamic memory allocation or deallocation etc.

The declarative paradigm makes debugging of concurrent and distributed application easier since there are no **side-effects**.

Why Erlang? (cont)

The sequential subset of Erlang is functional because the lack of side-effects eases the debugging.

Concurrency and distribution make the application much more difficult to debug in presence of side-effects and shared memory.

Even in the case of sequential programming (i.e. one thread), if there is no global state shared by several functions and if every function call returns the same result with the same arguments — which is not the case always if side-effects are allowed —, the debugging is much easier.