Mobile Erlang Computations to Enhance Performance, Resource Usage and Reliability

Adrian Francalanza and Tyron Zerafa

University of Malta

1 Introduction

A software solution consists of multiple autonomous computations (i.e., execution threads) that execute concurrently (or apparently concurrently) over one or more locations to achieve a specific goal. Centralized solutions execute all computations on the same location while decentralized solutions disperse computations across different locations to increase scalability, enhance performance and reliability.

Every location affects its executing computations both directly (e.g., the lack of a resource may prohibit a computation from progressing) and indirectly (e.g., an overloaded location may slow down a computation). In a distributed environment, application developers have the luxury of executing each computation over its best-fitting location; the location (a) upon which the computation can achieve the best performance and (b) which guarantees the computation's livelihood. Ideally, the decision to execute a computation over a location instead of another also load-balances the use of available resources such that it has the least impact over other computations (e.g., a computation should not execute over an already overloaded location further slowing down its computations).

Application developers can only execute computations over their best-fitting location if their distributed programming language provides abstractions that allow them to control the locality of computations both before they are started and during their execution. In the rest of this document, section 2 briefly justifies why these two forms of locality control are required and section 3 outlines the issues that arise, and will be tackled in the talk to be held at CSAW 2014, by them.

2 Control over Locality

The ability to determine the requirements of a computation (and hence its best-fitting location) before it starts executing depends on the computation's type (i.e., whether it is of a functional or reactive nature).

Any computation of a functional nature is deterministic (i.e., its execution can be completely established from the computation's executed code and its initial inputs). For instance, the execution of the factorial algorithm is of a functional nature since it cannot be affected by any other computation; this is just a mathematical function. Thus, it is possible to application developers to determine its requirements (e.g., processing power) and **initialize** it over the best location (e.g., the most lightly-loaded location).

Any computation of a reactive nature is non-deterministic (i.e., its execution depends on the input it receives from other computations). For instance, the execution of a publish-subscribe server is of a reactive nature since its behaviour can only be determined at runtime as it starts receiving requests. This inability to foresee the execution of a reactive computation before it starts executing **limits** application developers from foreseeing all its requirements and initializing it over its best-fitting location (e.g., although frequently communicating computations are best co-located on the same location to reduce communication overheads, it is impossible to start the execution of publich-subscribe server closer to its clients' since these can only be determined at runtime).

Reactive computations can never be initialized over their best-fitting runtime systems since their requirements can never be determined before their execution starts. Furthermore, the runtime system best-fitting a (functional or reactive) computation may change throughout the computation's lifetime (e.g., the factorial algorithm may impose a huge processing load on its location which affects the execution of the publish-subscribe server) or fail (e.g., the failure such location would terminate both the server and factorial computations). In such cases, the performance and reliability measures of a computation can be greatly enhanced through **mobility**: the ability to dynamically relocate an executing computation from one location to another. This additional level of flexibility over immobile computations allows application developers to dynamically adapt their solutions to the ever-changing nature of distributed systems (e.g., by relocating the executing server computation closer to its new clients that are determined at run-time so as to reduce communication overhead), enhance load-balancing (e.g., by relocating the executing server computation to a lighter-loaded location) and increase fault tolerance (i.e., by relocating both executing computations away from a location that is about to fail to a more stable location).

3 Talk Overview

The ability to control the locality of a computation both before it is started and during its execution gives rise to a number of important questions. Erlang, a language designed specifically for distributed systems, only offers rudimentary support for code and computation mobility which greatly limit the flexibility of this language. The talk that will be presented at CSAW 2014 will discuss some of the issue surruounding mobility in the context of Erlang and outline how such functionality can enhance performance, resource usage and reliability of Erlang-developed solutions. Specifically, this talk will try to answer the following questions: (1) What is the desired semantics of a computation that is initialized over (or relocated to) a remote location? (2) How can mobile computations be referenced (and located) at run time? (3) Can mobile computations preserve the same message-passing semantics supported by Erlang for immobile computations? (4) Do conventional synchronous error-handling abstractions suffice for mobile computations? (5) Is it feasible to relocate computations?

This work is carried out following the award of a STEPS scholarship which is part-financed by the European Union - European Social Fund (ESF) under Operation Programme ll - Cohesion Policy 2007-2013.