Titre du sujet de recherche :
Programming language abstractions for the Internet of Things

Laboratory / laboratoire :
CITI-INRIA, Université de Lyon, INSA de Lyon
http://www.citi-lab.fr/

Research team / Equipe de recherche :
This is a joint work between DynaMid and INRIA UrbaNet.
http://dynamid.citi-lab.fr/
http://www.inria.fr/en/teams/urbanet

Supervisors / Directeurs de thèse :
Pr Fabrice Valois (HDR), Université de Lyon, INSA-Lyon, CITI-INRIA
fabrice.valois@insa-lyon.fr
Dr Julien Ponge, Université de Lyon, INSA-Lyon, CITI-INRIA
julien.ponge@insa-lyon.fr

Lab Language / Langue de travail : English

Abstract / Présentation du sujet :

The so-called Internet of Things marks the convergence of small connected devices (e.g., personal devices, body devices, wireless sensors) and the larger set of more traditional distributed applications as accessed over standard Internet protocols. The “software is eating the world” motto\(^1\) is no lie as more and more of devices communicate with cloud-based services. Still, developing and integrating software remains largely a crafting exercise with mainstream programming languages, while research languages tend to be too impractical.

The architecture of modern applications is converging towards distributed services that expose standard-based interfaces. A service tends to fulfill a single functional purpose (e.g., storing some data / logs, providing authentication, and so on). In

\(^1\)See http://www.wsj.com/articles/SB1000142405311903480904576512250915629468.
this setting an application shifts from a paradigm where it is made by assembling component libraries to a paradigm where many (distributed) processes form the application. Communications between such services are typically made using the general-purpose HTTP protocol, but more specific ones can be used when needed (MQTT for IoT devices, ZigBee in some wireless sensor networks, etc). Given that distributed services rely on the integration with other services through highly interoperable protocols, it is very wise to take advantage of many programming languages rather than follow a “one size fits it all” approach.

Interestingly, the characteristics of distributed services deployed on cloud infrastructures are quite similar to those of (sensor) network gateways. Among many problems, these applications need to cope with concurrency due to network requests, and they have to bind data from/to network protocols. While middleware can be used to, say, automatically expose a HTTP service interface and perform data binding, or to provide concurrent programming abstractions, this remains orthogonal to programming language operational semantics and type system.

The history of programming languages is paved with abstractions being moved from library support to first-class citizen language constructs: memory management (e.g., Java, Self), threads (e.g., Java), actor models (e.g., Erlang, Scala), communicating sequential processes over co-routines (e.g., Go), etc. Still, even with a modern programming language the development of distributed services involves lots of boilerplate code (e.g., types for network messages data-binding) and there is little to no static checks beyond types, especially with respect to the correctness of concurrent code. As an example, the Go programming language only provides runtime race condition detection.

In practice, one can observe that the code of a typical application based on distributed services involves a significant share of message processing and network operations. The literature lacks successful languages that were both practical and suitable for these kinds of networked applications. The Scala programming language is a prime example of a language effort that initially tried to address the need for the development of “XML services” with the support of XML semi-structured data elements in the language. Still, Scala does not enforce a concurrency model, it does not provided network programming helpers, and it merely focused its efforts on a sophisticated type system. Funnel (Functional Nets) was a predecessor of Scala with first-class support for concurrency primitives based on join-calculus. Still, it proved impractical to use in real-world applications, just like other attempts of join-calculus in the ML / OCaml families.

An alternative to composing distributed applications using programming languages is to rely on some orchestration language such as BPEL and workflow execution engines. Behavioral protocols can be extracted from BPEL processes, which is useful for checking correctness of distributed systems compositions. Still, the limited expressiveness of workflow languages combined with the complex tooling to develop, test and execute them limit their wider adoption in favor of more traditional programming languages.
The main scientific goal of this PhD thesis is to investigate which abstractions shall be part of the next-generation programming languages in the age of the Internet of Things. We are especially interested, but not limited to, the useful abstractions to cope with: concurrency, asynchronous programming, data processing, software dynamics, message passing, network membership discovery and distributed algorithms (e.g., consensus and transactions). Given the distributed / concurrent nature of the applications that we target, we are also interested in providing compilation-time verifications beyond classical type checking (e.g., deadlock detection, time-bound guarantees). We also want the research outcomes to be practical.

The anticipated challenges are as follows.

1. Establish an exhaustive state of the art on programming language and middleware abstractions. Consider which ones shall be part of a programming language, and which ones shall be relegated to library support, based on an extensive study of distributed services requirements.

2. Propose a programming language. Formalize and prove the soundness and correctness of its type system and operational semantics. Classify the ranges of static checks that can be performed at compilation time. Devise which remaining checks shall be done at runtime. Discuss their algorithms.

3. Propose an implementation on top of the Java Virtual Machine or the LLVM code generation infrastructure with state of the art performance. Develop a rigorous micro-benchmarks tests suite, and revisit some suitable larger benchmarks from The Computer Language Benchmarks Game\(^2\).

4. Validate the language usefulness for developing distributed applications, both in cloud and wireless sensor gateway settings. Provide metrics to evaluate programs against other languages. Perform a field study on practitioners to assess the language practicability, suitability and learning curve.

References:


\(^2\)See http://benchmarksgame.alioth.debian.org/.


