A modular and uniform programming environment for high-level distributed computing

Laurent Prosperi[1], Marc Shapiro[2], and Ahmed Bouajjani[3]

[1]ENS Paris-Saclay
[2]Sorbonne-Universités UPMC-LIP6 & Inria Paris
[3]IRIF, Université Paris Diderot

February 19, 2019

Abstract

The goal of the PhD is to identify the abstractions needed for writing distributed applications or distributed systems and to ease distributed programming for non experts. Distributed programming is by nature concurrent, error-prone and subjected to variable latency which make computations highly non-deterministic.

Furthermore, applications have conflicting requirements: correctness, i.e. controlling what the system does, versus application performance, which includes availability, responsiveness or throughput. There is no single right solution to this trade-off; it depends on application requirements, environment, workload, the available resources, etc.

One common approach to address high-level distributed programming is to restrict expressiveness, as for instance with TensorFlow or MapReduce. The closest to a general-purpose programming environment is Orleans: however, part of its runtime (e.g. deployment, consistency, security) is provided by a separate subsystem, and not programmable with the same first-class abstractions.

This is ripe for a general-purpose approach for both application developers who should not be bothered with low-level details, and system developers, who require fine control at runtime. It should be possible for both to create, protect, ensure correctness and use powerful abstractions, making first-class what is built-in in the above systems. An abstraction would be able to compose basic primitives that provide access to the full power of distribution.

We aim to provide a **uniform and modular programming environment** for this purpose. This will increase the reliability of such programs, by helping to ensure correctness at all levels of the environment. The environment should include: (1) a **programming model**, to be able to reason precisely about the behaviors of systems; (2) a **language** to be able to write applications and systems; and (3) a **runtime system** to assist in running and deploying code and data on a distributed infrastructure.

In the first year we will study state-of-the-art and derive requirements. The second year will be devoted to implementing the language and a runtime programmable as a first-class abstractions. Moreover, this will generate useful feedback. Finally, we plan to run a large experimental evaluation, to assess success and provide a reality check, for instance by implementing a geo-distributed database.

1 Societal, economic and/or industrial context

The Internet has revolutionised the ways we interact, conduct commerce, and use information. Users are ever more dependent on distributed applications. Conversely, providers must offer services that are constantly available, reliable, safe, and secure. Furthermore, programming highly-available distributed applications is no longer reserved to expert programmers. And, at the same time, users are ever more dependent on distributed applications especially with the Internet- of-Things (around 30 billion connected devices by 2020 [6]).

Distributed programming remains difficult and error-prone, exposing users, the economy, and critical infrastructure to bugs and security violations. Indeed, concurrency and failures, essential features of a distributed system, are difficult to abstract away. Interacting concurrent processes do not compose well. Moreover, *consistency* and *availability* are odds with each other: The CAP theorem [7] shows that, when network partitions[1] can occur (a fact of life on the internet) it is not possible to guarantee at the same time strong *consistency*[2] and *availability*[3]. In practice, either the service is unavailable at times (and users are unhappy), or data diverges at times (and developers are in trouble).

This raises a big problem. If the developer does not know precisely what is the current state, how can she develop a system that is correct? How to program such an application? How to ensure that it does not produce incorrect data (correctness)? How to guarantee that its data is safeguarded from third parties (security)?

Furthermore, applications have conflicting requirements. On the one hand, correctness (controlling what the system does), requires events to happen in a reliable, deterministic way. On the other, application performance (including availability, responsiveness and throughput), requires concurrent, asynchronous execution.

There is no single right solution to this trade-off; it depends on the application requirements, the expected environment and workload, the available resources, etc. Getting this right is difficult: current practice in building distributed systems rests on programmer expertise, i.e., trial and error, which is costly and dangerous. Thus, currently, large numbers of non-expert programmers to play it by ear among uncomfortable and momentous trade-offs, in the presence of non-composable, non-deterministic, and weak consistency.

When correctness of distributed application is not ensured this lead to errors or attacks with damaging effects for users. As or instance with the DAO bug that have been affected Ethereum in 2017 which leads to a robbery of millions of dollars.

[1] Due to failure for instance

[2] Strong Consistency is the property that all participants observe the same updates in the same order.

[3] Availability is the property that the application can always read and write its data.

2 Scientific context

We believe that the situation is ripe for a new, high-level approach. We propose to develop methods, tools and languages to aid the programmer of general distributed programs.

Highly successful and explicative abstractions already exist, such as (at opposite ends of the spectrum) consensus or data flow. Frameworks and languages are making distributed programming easily accessible in some restricted domains, for instance MapReduce (for batch processing), Flink (for stream processing) or TensorFlow (for deep learning and numerical computation). These approaches work well in their restricted environment, but only by severely restricting the developer's capabilities.

The closest to a general-purpose programming environment is Orleans [3] which allows the developer to define and compose abstract, location-independent actors (using promises to handle communication). The runtime environment is in charge of connecting them together and of deploying them, elastically as the service demands and the availability of resources change over time. Another important concept is dataflow or reactive programming, whose main abstraction is a graph whose edges carry flows of information, and whose vertices are computation entities [9]. Finally, tierless programming describes distributed computation subsuming how sub-computations are deployed and placed at the different tiers of a cloud computing environment [2].

These approaches provide orthogonal computation, composition, communication and deployment abstractions, and are designed to maximise parallelism and to enable flexibility and elasticity. However, in order to hide the complexity of distribution to application developers, they come with arbitrary restrictions; for instance, communication abstractions are often unidirectional. Deployment, consistency, security and fault-tolerance are assumed to be addressed by a separate system, not programmable with the same first-class abstractions.

We believe that it is possible to take a unified approach for both application and system developers themselves. It should be possible to create and protect abstractions, including what is built-in or second-class in the above systems, by composing basic primitives that provide access to the full power of distribution. Such an environment should be based upon: 1) a programming model which includes a data model, a consistency meta-model, a communication model and a computation model. 2) a language to be able to write applications and systems following the previous models. 3) a runtime system (handling deployment, fault-tolerance, elasticity, security, high-availability). It should be possible, for instance, to build the equivalent of Antidote [1] or Orleans with flexible consistency levels and a pluggable, non-compromisable security architecture.

2.1 Supervision

This PhD will be supervised by Marc Shapiro (at LIP6, Sorbonne Universités, in the Delys team) and Ahmed Bouajjani (at IRIF, Université Paris Diderot).

3 Approach

This section provides a detailed description of each part of the environment (programming model, language and runtime).

3.1 Programming model

Instead of having a monolithic model (hard to modify, to update), a hierarchy of models can be used. Moreover, this hierarchy hides part of the complexity for application developers (if they don't need it) and allows the building of tailored systems by doing as little modifications as possible. This model will be structured into four parts: a communication model (describing transmission of data between computing units), a data model (handling shared and persistent data objects), a consistency meta-model and a computation model (expressing program). Each parts relies upon the precessing one.

The communication model might contain the following primitives:

- Asynchronous and synchronous communication operation invocation

The data model might contains the followings primitives:

- Data objects with the capability to implement replication and versioning.
- Data flow carries any mixture of state, delta, or operation. Communication respects programmer-defined abstraction boundaries.

The consistency meta-model might contains the followings primitives:

- Transactional guarantees and the level of consistency needed (e.g. causal consistency)

The computation model might contains the followings primitives:

- Publish-subscribe/data flow, with forward and backward paths, and combiners.
- Asynchronous (concurrent) and synchronous (consensus-based) operation invocation on shared objects.

To ensure correctness, all of the above primitives should have formal specifications in order to describe their requirements, their actions and also their interactions.

3.2 Programming language

The programming language will be an instantiation of the programming model. It will provide interaction with pre-existing base of codes, for instance to reuse sequential libraries, and ensure abstraction isolation. It aims to ensure formal correctness by providing tools to express the specification of an abstraction and to guarantee abstraction combination. Specifications may be based on types (e.g. F^* [5]) or on invariants (e.g. the Eiffel programming language [4]) or on TLA+ [8]. An extra care should be put on error handling.

One direction, to keep focused on the distributed aspects of the language, will be to create a Domain Specific Language instead of a standalone one. A DSL is a meta-library (namely a set of modules) transforming a general purpose language (e.g. Java) into a specialised one (e.g. Flink).

3.3 Runtime

The runtime will assist in running and deploying codes and data on a distributed and heterogeneous infrastructure. An infrastructure can be represented, at a high-level, by a set of nodes (e.g. processors, processes, servers and devices). The runtime should be modular and programmable with first-class abstraction in order to allow easy runtime customisation. Such a customisation can be used for instance to change the deployment pattern or the fault-tolerance of a system. Hence, this runtime should be written in the previous language. The runtime might provide the following primitives:

The data primitives:

- Transparent piggy-backing of metadata, such as timestamps, provenance, security labels, or accounting information. This metadata should be automatically added by the runtime without programmer interactions (for instance to automatically debug thanks to provenance).
- Data replication for fault-tolerance.

The consistency primitives:

- High-availability through code replication when possible according to the consistency requirements.

The computation primitives:

- Programmable deployment and elastic configuration of computation and data entities, transparently to their functional program text.
- Dynamic check, for instance by analysing provenance information (automatically added by the runtime).
- Hot swapping in order to deploy new code during execution (e.g. to deploy an update without service interruption).
- High performance by scheduling computations on nodes and by using locality awareness-optimization.
- Check-pointing of computations for fault-tolerance.

4 Planning for the first year

This year will be dedicated to study state-of-the-art and to define the specifications.

4.1 Define a programming model

The first goal will be to study the source code of state-of-the-art distributed systems, to determine which kind of abstractions, and which interactions between them are needed. Potential targets are AntidoteDB for the database aspect, Flink for the stream processing part, Ceph for the distributed filesystem and TensorFlow for computation and deeplearning. From this abstractions, we will build the models and make them as modular as possible, with a good trade-off between isolation and expressiveness. Then, we should formally defined then in order to guarantee their correctness.

4.2 Define a language

From this model, a language can be derived. Making a DSL (instead of a standalone language) is a promising way in order to reuse existing codes and libraries and to keep focus on the distributed aspects of the computation.

4.3 Define a modular runtime

Then the last part will be to design the shape of the runtime and the specifications of its components. The runtime shall be composed from modular components, and programmable with first-class abstractions.

4.4 Publication plan

We plan to write an article about the design of a modular environment, how we ensure correctness across the three levels (models, language and runtime) and with toy examples to illustrate the capabilities of our environment.

5 Planning for the second year

The second year will be centered around the implementation of a prototype of this environment. This is a two-phase approach. First, the language can be implemented thanks to a compiler or as a meta-library (in case of a DSL if the syntax does not need to be extended). Second, an implementation of the runtime with basic components in order to provide fault tolerance, hot swapping, deployment, elasticity in an extensible and modular way. The runtime should be written with our defined language. This implementation will be used to retrofit the model and the language specifications since it runtime is, by itself, a distributed system. At the end of the second year, we might plan to write an article about the presentation of the whole structure of the environment and present its capabilities throughout the implementation of the default runtime.

The main track for the third year would be an experimental evaluation to assess success and provide a reality check. We shall re-implement (or retrofit) one of the systems within the environment we developed. Experiments will measure the differences, in terms of functionality, availability, latency, throughput, lines of code, density of defects, etc. This could lead to an article focusing on the evaluation of the environment.

References

[1] *AntidoteDB, a planet-scale, available, transactional database with strong semantics*. `http://antidoteDB.eu/`. [En ligne; accès le 16/01/2019].

[2] Gérard Boudol et al. "Reasoning about web applications: An operational semantics for hop". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34.2 (2012), p. 10.

[3] Sergey Bykov et al. "Orleans: cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 16.

[4] *Eiffel*. `https://www.eiffel.org/`. [En ligne; accès le 13/02/2019].

[5] *F* is a general-purpose functional programming language with effects aimed at program verification.* https://www.fstar-lang.org/. [En ligne; accès le 13/02/2019].

[6] *Forbes: 2017 Roundup Of Internet Of Things Forecasts.* URL: https://www.forbes.com/sites/louiscolumbus/2017/12/%2010/2017-roundup-of-internet-of-things-forecasts.

[7] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[8] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] *The Reactive Manifesto.* https://www.reactivemanifesto.org/fr. [En ligne; accès le 16/01/2019].