An environment for composable distributed computing

Benoît Martin
Sorbonne Université, CNRS, LIP6
Paris, France
benoit.martin@lip6.fr

Laurent Prosperi
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
laurent.prosperi@lip6.fr

Marc Shapiro
Sorbonne Université, CNRS, INRIA, LIP6
Paris, France
marc.shapiro@acm.org

Abstract

Modern applications are highly distributed and data-intensive. Programming a distributed system is challenging because of asynchrony, failures and trade-offs. In addition, application requirements vary with the use-case and throughout the development cycle. Moreover, existing tools come with restricted expressiveness or limited runtime customizability. Our work aims to address this by improving reuse while maintaining fine-grain control and enhancing dependability. We argue that an environment for composable distributed computing will facilitate the process of developing distributed systems. We use high-level composable specification, verification tools and a distributed runtime.

ACM Reference Format:
Benoît Martin, Laurent Prosperi, and Marc Shapiro. 2020. An environment for composable distributed computing. In *Proceedings of Eurosys Doctoral Workshop (EurosysDW20).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Author contribution

B.M. and L.P. are PhD student of M.S. They did the design together equally. B.M design and implemented the runtime. L.P design and specified the conceptual and language framework.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EurosysDW20, April 27, 2020, Heraklion, Crete, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Distributed computing has become crucial across many areas of computing. Programming highly-available distributed applications is no longer reserved to expert programmers. Consider mobile computing, Internet of Things (IoT), High Performance Computing, Network Function Virtualisation (NFV), neural networks, or internet gaming.

However, programming distributed system remains difficult and error-prone, exposing users and critical infrastructure to bugs. Furthermore, applications have to scale from a single node to geo-distributed infrastructures. Moreover, distributed systems need to deal with failures, non-determinism, asynchrony and/or concurrency. This leads to trade-offs (e.g. CAP [6], FLP [12]) without a one-size-fits-all solution. Hence, a flexible, dynamically evolving, customizable and elastic programming environment is required.

Approaches [2, 5, 18] provide orthogonal computation, composition, communication and deployment abstractions. These paradigms are designed to maximize parallelism. However, in order to hide the complexity of distribution to developers, they come with arbitrary restrictions. Moreover, distributed system features (e.g. consistency, fault-tolerance) are rarely first-class abstractions in programming language.

We address the complexity and the difficulty of building distributed systems by improving code-reuse and composition in order to ease the work of the developer and to help balance the different trade-offs. Moreover, since distributed programming is error-prone, we provide formal specifications and tools to help mitigate bugs. As there is no one-size-fits-all solution, we give flexibility to the programmer to specialize the environment for its specific use-case. This is achieved by enabling the creation of new distributed abstractions, customizing their implementation and exposing distributed system features as first-class abstractions.

Software engineering methodologies identify a separation between an application's specification and implementation [1]. Our approach applies this separation to the development of a distributed application. To increase code-reuse and dependability, we define a formal composition model **specification language** based on **distributed abstractions** (Section 2.1). Moreover, to increase flexibility, we design and implement a customizable **distributed runtime** based on an **reactive actor programming model** (Section 2.2).

2 Proposed solution

Environment. To build a distributed system, our proposed environment (Figure 1) starts with (i) a specification of the underlying runtime written by the programmer or provided by the runtime maintainer ; then (ii) programmers write the specification of the system in our specification language ; (iii) our compiler processes these specifications by verifying the soundness of the composition and generating skeleton code with embedded dynamic checking to detect specification violations; then (iv) programmers write the implementations (or use some external one) for each abstractions ; finally (v) the code is run on the target customized distributed runtime.

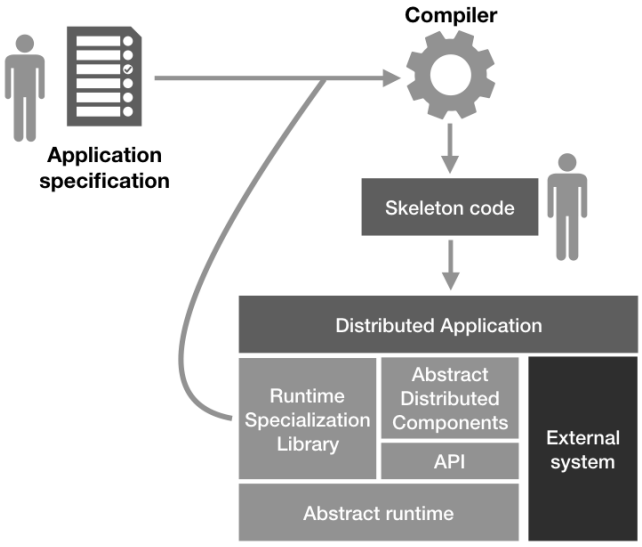

Figure 1. Overview of the environment used to build a distributed system.

2.1 Specification

In order to facilitate the programmer's job in term of reusability, productivity and bugs mitigation, a formal compositional model is needed. This model should go beyond the traditional API specification limitations [10]. It has to cope with network failures and describe the consistency of operations. Given that building a distributed system involves multiple components written in different programming languages, a polyglot specification language[1] is needed to write composition specification. Expressing formal specifications does not guarantee that implementations match their specification nor that the composition is sound. Therefore dependability must be ensured from the soundness (safety and liveness properties) of a composition to the detection of specification violations at runtime.

[1]We use the term *polyglot* [16] since we express the specification independently to the languages used for each implementation.

Specification model. A **distributed abstraction** provides functionalities with a defined interface and a distributed behavior. It is represented by a specification consisting of an *internal description*, a list of *ports* and a *description of the concurrent interactions*. The internal description is an abstract state annotated with invariants. A port represents an interaction point (e.g. a socket) that has a type, guards (pre-conditions) and effects (post-condition). Finally, concurrent interactions at the granularity of ports are defined by using synchronization annotations.

A composition of abstractions is a high-order abstraction that encapsulates the composition of its components. More formally, the composition is a graph such that vertices are abstractions and dynamically evolving edges connect ports together. Furthermore, component properties cannot always be composed in a sound way. Let us take two datastores A and B such that A is weakly consistent and B is strongly consistent. If a program reads read from A and write its read to B, the consistency guarantee assumed by the programmer for B can be broken. Therefore, parent abstraction will specify the desired properties of the composition and check their soundness.

The specifications are orthogonal with the implementations. However, this prohibits certain optimizations that take into account the nature of the distribution, like the deployment patterns. One use-case is to collocate two unrelated implementations for performance purposes. To leverage these optimizations, we increase the expressiveness of our specification model by allowing implementation dependent properties.

Enhancing the dependability of a specified system. For this, we need i) to ensure the soundness of the composition, ii) to detect and react to violations of a specification by a component implementation and iii) check the dynamic distribution properties (e.g. placement or provenance based properties).

We wish to provide automated static checking of the soundness of the composition. This is done by ensuring that we soundly compose APIs by using type checking. We use static session types [9] to deal with port composition. However, the limits of static verification[2] in term of expressiveness, scalability and simplicity will be reached quickly. Hence, the default fallback is to use dynamic checking.

From declarative properties (pre-conditions and post-conditions), our tools will automatically derive dynamic checks that are injected into the implementation. Injection can not be done directly inside black-box implementations. We mitigate this limitation by analysing the observables of the implementation inside the implementation of the parent abstraction (if any). However, in that case, part of the internal aspects are outside the reach of our tools and we cannot guarantee that fault-tolerance placement properties are ensured. In a

[2]A natural extension is to integrate a model-checker [14]

nutshell, this dynamic approach does not ensure soundness but detects its violation.

Furthermore, some distribution properties, like placement, are inherently dynamic and deployment dependent (i.e. relying on the interactions between implementation and infrastructure). In order to support these properties we need a view of the running system. This is achieved by oracles which provide dynamic deployment information, like the placement of the abstractions on a set of nodes. Oracles are implemented as external services, increasing the modularity of our approach by allowing programmers to define their own oracles. For instance a placement oracle could be implemented as a service of our distributed runtime .

Code generation. Since distributed abstractions export their dependencies and composition properties, they can be treated as black-boxes. Therefore, we design a specification language to declare and compose the distributed abstractions in an uniform way independently of the targeted languages used for writing the components. From this declarations, the compiler generates the related skeleton code for implementations (e.g. header files). Moreover, it performs static verification, generates and injects dynamic checks.

2.2 Implementation

In order to best accommodate for high-level abstractions and code-reuse, a dedicated **distribute runtime** is required. This will reduce arbitrary limitations by taking into account the target architecture (e.g. x86-32/64, ARM 32/64), the interconnect in a clustered/cloud environment and the distributed context. This increases expressiveness while maintaining fine-grained control. Furthermore, a new runtime makes it possible to use a specific programming model that will facilitate the programming of distributed abstractions. This model uses known paradigms from the programming language community.

Programming paradigms. In the interest of greater code-reuse and concurrency handling, the use of an object-oriented model that features message-passing will ease development [21]. Message-passing, as opposed to shared-state concurrency, is less prone to dead-locks. This feature is of great help as concurrent programming is widely considered one of the hardest concepts to program with. An object-oriented programming model presents the concepts of data and type abstraction that enables code-reuse and composability.

Programming model. We choose to use an Actor Model[13] as a base to define our programming model. This model, defines the smallest component as an *Actor* that sends and receives messages in a reactive (asynchronous and non-blocking) manner. The advantages of such a model are numerous: (i) the concurrency model is abstracted which simplifies the developers job (ii) multicore scalability is naturally obtainable (iii) placement depends on where actors are positioned relative to a physical CPU core (iv) actor communication is abstracted using the actor's unique identifier.

However, the main downside to reactive programming is the learning curve that is due to the requirement of using asynchronous and non-blocking calls. Be that as it may, properly defined abstractions and well structured pre-defined components can overcome this learning curve.

Programming language. Choosing a programming language in which to implement a runtime that will run a programming model, is a non-trivial task. In order to best apply our programming model, the implementation language must be object-oriented and exposes zero-overhead generic programming. In the interest of performance, a strongly typed and compiled language will best suite.

Customizable Distributed Runtime. Our Abstract Distributed Runtime makes customization possible by exposing low-level abstractions such as the inter-runtime communication (transport protocol, serialization and logic), scalar, container and memory allocator types (Figure 2). This fine level of customization is crucial to improve portability and code-reuse.

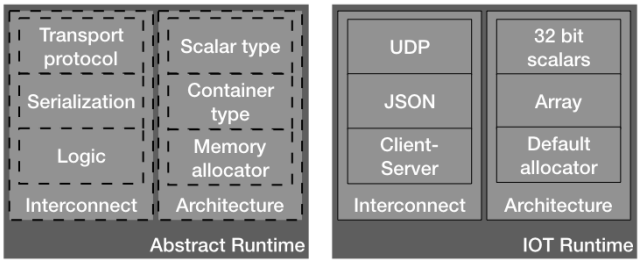

Figure 2. Overview of the Abstract Distributed Runtime and an example of an embedded IoT runtime

The numerous runtime customization possibilities may be a non-helping feature in easing the development of distributed systems. For this reason, a pre-defined set of customizations are available in our Runtime Specialization Library. These customizations enable a flexible construction of multiple runtimes types such as a specialized embedded runtime for IoT (Figure 2) or a multi-socket, multi-core node.

Latency predictability is taken into account in order to include future porting of the runtime to realtime and embedded systems. Fine-grained runtime optimizations are possible in the following form:

- pin a thread to a core (when possible) to minimize the scheduler's context switching which increases predictability
- change the scheduler thread priority (using a real-time kernel) which increases predictability
- change the allocator type that is used within the runtime. We provide a custom allocator that is optimized for a multicore context

- scalar types can be specified for better portability

3 Further work and challenges

The idea of distributed abstractions is central in easing the development of distributed applications. A correct and canonical definition of these distributed abstractions is the main challenge and will contribute to define high-level abstractions and low-level components.

High-level programming environment. We face two main issues, on the one hand finding the sweet spot in the design space to maximize the expressiveness of the specification semantics without breaking efficiency nor reaching verification limits; on the other, finding a way to map our abstract properties variables to implementation states in a generic way in order to easily generate dynamic checks for the different targeted languages.

The first underway step is writing the multi-pass compiler doing parsing, composition verification, dynamic checking generation and skeleton generation. A first usable version (with a minimal syntax, a type checker for abstractions API and some basic generation toward our own runtime) is expected for the end of June. The next step is specifying our runtime to expose some runtime behaviors as first class abstraction. This will five us feedbacks on our sweet spot selection and we will iterate on its design. A further step is to increase the coverage of static checks by extending the type system.

Runtime. The main difficulty is in both defining a runtime and a programming model while keeping in view our needs for portability, predictability and performance. The programming model is based on the previous work from the reactive programming community and the runtime implementation must take into account requirements that are imposed by the programming model (synchronous and non-blocking calls).

Choosing the right programming language will be crucial structuring element to our framework. This language must allow zero-overhead generic programming in order to maximize code-reuse without sacrificing performance (See 4).

A first version containing the basic runtime fundamentals (functional actor system, multicore scaling and limited networking) is expected for May/June. The next step consists in defining a simple abstract distributed component that will be used as a running example that will help with defining the high-level abstractions. A number of iterations will then be made in order to better define the following useful and needed abstract distributed components.

Iterative process and evaluation. An iterative process will take place throughout the work on both high-level abstraction and low-level component and will be used as a feed-back loop to ease convergence to canonical distributed abstractions.

An evaluation phase for both the runtime and the specification model will be crucial to asses the performance evolution throughout the project. First, a number of self-contained and representative benchmarks will have to be identified and implemented using the framework. The initial results will be used as a baseline for comparison with existing solutions and as a reference point to future work.

4 Related work

Distributed programming paradigms. Several paradigms are currently explored to ease distributed programming : *actor model*, *service oriented computing*, *dataflow or reactive programming* and *tierless programming*. In the *actor model*, states are encapsulated by actors which communicate through message passing allowing transparent scaling, robustness to failures [2] and efficient implementation [18]. A system, in *service oriented computing*, is composed of a set of services; each of them provides some functionalities and they are composed by orchestrators. The main abstraction of a *dataflow or reactive programming*[3] is a graph whose edges carry flows of information, and whose vertices are computation entities. Finally, *tierless programming* describes distributed computation subsuming how sub-computations are deployed and placed at the different tiers of a cloud computing environment [5]. In order to hide the complexity of distribution, these paradigms come with arbitrary restrictions; for instance, communication abstractions are often unidirectional. Furthermore, distribution specific issues (e.g. deployment, consistency, security and fault-tolerance) are assumed addressed by a separate system, not programmable with the same first-class abstractions. Moreover, except for service oriented programming, none of them can easily use existing systems as building blocks and ensure the correctness of the composition.

Specification and verification. Several specification languages try to overcome the traditional limitations of high-level languages (i.e. ignoring low level implementations de tails critical for robustness and performances). Mace [14] is a C++ language extension, for both writing specifications and programs, focusing the analysis of systems behaviours thanks to causal path debugging and embedded model checking. Madeus [7] targets deployment specification of components in order to improve commissioning parallelism, correctness is guarantee by a model checker Mada [8] expressing both qualitative and quantitative properties related to both safety and efficiency. Maude [19] tries to arrive, by using some high-level specifications, at a good design before implementing and verifying in depth properties. An other direction is explored by [16] in the case of synchronous-reactive model by composing components (called reactors)

that can be treated as a black box. Therefore, reactors are defined and composed by a polyglot language.

Existing approaches uses type systems to ensure correctness of composition of components. [17] proposes to use subtyping and parametric polymorphism to check API composition and to allow code evolution. ScalaLoci [22] targets placement correctness, placement types are used to define code location and restrict the topology of communication. MixT [20] and ConSysT[11] see consistency as a property of information expressed as types and use compile-time information flow control to forbid illegal consistency mixing.

Programming language and model. Peter Van Roy explored and classified all the main programming paradigms into several categories that express the features of these programming languages [21]. This is helpful to give a broad view to help us choose the right concepts to best fit our needs (composability, concurrency and scalabilty). The difficulty of choosing a programming language and model lies in the fact that there is no one-size-fits-all language or model. However, several valuable key concepts can be extracted from Peter Van Roy's work. He argues that (i) message-passing is the correct default for general-purpose concurrent programming instead of shared-state concurrency ; (ii) object-oriented programming brings forwards the principle of data abstraction, polymorphism (i.e. an object that can take several forms) and inheritance (i.e. common relationship between abstractions) that will enable code-reuse and thus composability.
The reactive programming community is using the term "reactive system" to describe a system that is flexible, loosely-coupled and scalable. This is made possible by decomposing systems into non-blocking, asynchronous tasks that communicate via messages or events [15]. The ideas behind the Reactive Manifesto [4] can be seen as a revival behind the Actor Model by Hewitt and al [13]. The manifesto argues that a system built in a loosely-coupled manner, that can be executed in an asynchronous and non-blocking fashion is able to be scalable, resilient, elastic and responsive.

These concepts are very helpful in helping choose the programming language and shape the programming model that we are going to use for our runtime environment.

5 Conclusion

In this paper, we address the complexity of building distributed systems while maintaining fine-grain control and enhancing dependability in order to ease the work of the developers and to help balance the different trade-offs (e.g. CAP). For this we argue for a new programming environment composed of a specification model based on distributed abstractions, a polyglot language to express this, a programming model to write the implementation of abstractions and its related runtime. In a nutshell, this novel environment gives the ability to compose preexisting systems, to have an hybrid checking of the soundness of the composition and to improve control, performances and predictability thanks to our runtime.

Our contribution using distributed abstractions will ease the development of performant distributed applications by enabling sound compositions using a high-level specification language. The ambition is to release the framework under an open-source license and to build a community composed of software engineers, project architects, academics and industrials. Ultimately delivering complex distributed systems concepts to mainstream application programmers.

References

[1] Suad Alagić. 2017. *Software Engineering: Specification, Implementation, Verification.* Springer.

[2] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75.

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 1–34.

[4] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. *The reactive manifesto.* https://www.reactivemanifesto.org/fr.

[5] Gérard Boudol, Zhengqin Luo, Tamara Rezk, and Manuel Serrano. 2012. Reasoning about Web applications: An operational semantics for HOP. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 2 (2012), 1–40.

[6] Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, 7.

[7] Maverick Chardet, Hélène Coullon, Dimitri Pertin, and Christian Pérez. 2018. Madeus: a formal deployment model. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 724–731.

[8] Hélène Coullon, Claude Jard, and Didier Lime. 2019. Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning. In *International Conference on Integrated Formal Methods*. Springer, 120–137.

[9] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. 2009. Sessions and session types: An overview. In *International Workshop on Web Services and Formal Methods*. Springer, 1–28.

[10] Chas Emerick. 2014. *Distributed Systems and the End of the API.*

[11] Nafise Eskandani, Mirko Köhler, Alessandro Margara, and Guido Salvaneschi. 2019. Distributed object-oriented programming with multiple consistency levels in ConSysT. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 13–14.

[12] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., 235–245. event-place: Stanford, USA.

[14] Charles Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: Language Support for Building Distributed Systems. (2007), 10.

[15] Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017-02. *Reactive Design Patterns.* Manning Publications.

[16] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A Lee, and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A Deterministic Model for Composable

Reactive Systems. *Model-Based Design of Cyber Physical Systems (CyPhy'19)* (2019).

[17] Christopher S Meiklejohn, Zeeshan Lakhani, Peter Alvaro, and Heather Miller. 2018. Verifying Interfaces Between Container-Based Components. (2018).

[18] Christopher S. Meiklejohn, Heather Miller, and Peter Alvaro. 2019. PARTISAN: Scaling the Distributed Actor Runtime. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. 63–76.

[19] José Meseguer. 2018. Formal Design of Cloud Computing Systems in Maude. In *Formal Methods: Foundations and Applications*, Tiago Massoni and Mohammad Reza Mousavi (Eds.). Vol. 11254. Springer International Publishing, 5–19.

[20] Matthew Milano and Andrew C Myers. 2018. MixT: a language for mixing consistency in geodistributed transactions. *ACM SIGPLAN Notices* 53, 4 (2018), 226–241.

[21] Peter Van Roy et al. 2009. *Programming paradigms for dummies: What every programmer should know*. Vol. 104. IRCAM/Delatour. 616–621 pages.

[22] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 129.