Varda: a language for programming distributed systems by composition

Laurent Prosperi

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Varda: a language for programming distributed systems by composition

---

Laurent Prosperi

Soutenue publiquement le : *5 septembre 2023*

Devant un jury composé de :

François Pottier, Directeur de Recherche, Inria	*Rapporteur*
François Taiani, Professeur des universités, IRISA, Université de Rennes	*Président, Rapporteur*
Cezara Drăgoi, Applied Scientist, Amazon Web Service	*Examinatrice*
Adrien Guatto, Maître de conférences, IRIF, Université Paris Cité	*Examinateur*
Mira Mezini, Professeure des universités, Technical University of Darmstadt	*Examinatrice*
Ahmed Bouajjani, Professeur des universités, IRIF, Université Paris Cité	*Co-Directeur*
Mesaac Makpangou, Chargé de recherche, Sorbonne Université, LIP6, Inria	*Directeur*
Marc Shapiro, Directeur de recherche (émérite), Sorbonne Université, LIP6, Inria	*Encadrant*

Remerciements

Je tiens tout d'abord à remercier mes rapporteurs, *François Pottier* et *François Taiani*, pour le temps qu'ils ont consacré à la lecture attentive et à l'évaluation de ce manuscrit. Merci également aux autres membres du jury, *Cezara Drăgoi*, *Adrien Guatto*, *Mira Mezini*, *Ahmed Bouajjani*, *Marc Shapiro* et *Mesaac Makpangou*. Un grand merci pareillement à *Nikolaos Georgantas* pour avoir suivi mes avancées tout au long de cette thèse.

Ce doctorat n'aurait pas été possible sans la supervision de *Marc Shapiro*. Merci à toi, Marc, pour ta confiance et pour m'avoir accompagné toutes ces années. Je tiens à exprimer ma profonde gratitude pour ton soutien inébranlable dans mes recherches et mes projets parallèles qu'ils soient académiques, dans d'autres disciplines, ou sportifs. Enfin sans toi et tes encouragements, je n'aurais pas franchi le cap de la (haute) montagne en hiver.

J'aimerais également adresser un remerciement très particulier à *Ahmed Bouajjani* pour nos enrichissants échanges hebdomadaires et pour tes conseils qui furent les bienvenus à maintes reprises.

Plus largement, je souhaite remercier tous les chercheurs et chercheuses en informatique avec qui j'ai collaboré ces dernières années et qui m'ont formé·e·s à la recherche. Je pense notamment à *Gustavo Petri*, *Basma El Gaabouri* et *Anthony Fox* que j'ai côtoyé lors de mon passage chez ARM. De nos discussions et de nos travaux ont germé nombres d'idées ayant façonnées Varda. Sans vous, ces travaux de doctorat auraient été bien plus fades. Je pense aussi à toutes les personnes avec qui j'ai travaillé et longuement échangé lors de mon passage à l'INRIA à Rennes: *Alexandru Costan*, *Pedro Silva*, *Gabriel Antoniu*, *Luc Bougé* et *Nathanaël Cheriere*. C'est avec vous que j'ai intégré le monde des systèmes distribués après mon séjour initiatique en terre formelle. Enfin, je souhaite remercier *Ulrich Schmid* et *Roman Kuznets* pour m'avoir donné goût à la recherche lors de mon séjour autrichien. Votre confiance m'a inspiré et motivé à donner le meilleur de moi-même, et je suis honoré d'avoir pu compter sur vous. Nos collaborations resteront pour moi une source d'inspiration et d'enrichissement.

Je souhaite ensuite remercier les membres du LIP6, en particulier les permanents de l'équipe Delys. Merci à *Pierre Sens* pour tes abondants conseils, et pour nos

discussions qui sont constamment une source d'encouragement et d'optimisme pour la recherche comme pour la course à pied. Merci à *Julien Sopena* pour nos longues discussions sous caféine qu'elles soient scientifiques ou non.

Ces dernières années, j'ai côtoyé plusieurs générations de doctorant·e·s et de post-doctorant·e·s au LIP6 que je tiens à remercier. Un énorme merci à mes camarades doctorants *Benoit Martin*, *Saalik Hatia* et *Ayush Pandey* avec lesquels j'ai parcouru la dernière ligne droite du doctorat. Merci à *Sreeja Nair*, *Gabriel Le Bouder*, *Dimitri Vasilas*, *Sara Hamouda*, *Lucas Serrano*, *Cédric Courtaud*, *Francis Laniel*, et *Jonathan Sid-Otmane* avec qui j'ai eu de nombreuses discussions et partagé de bons moments. Je tiens par ailleurs à remercier tout particulièrement *Guillaume Fraysse* qui m'a soutenu lors de mon pas de côté vers les sciences politiques et *Ilyas Toumlilt* sans qui le saut dans l'ultra-endurance n'aurait pas été possible. Enfin, j'encourage vigoureusement la génération suivante: *Célia Mahamdi*, *Aymeric Agon-Rambosson* et *Étienne Le Louët*.

Ces années de doctorat ont été une savante alchimie entre informatique et sciences politiques qu'il a fallu équilibrer avec plus ou moins de sueur, de café et de souplesse. Tout à commencer avec *Olivier Wieviorka*, que je remercie pour sa confiance et pour l'encadrement de mes travaux de recherche pendant mon premier "pas de côté" en histoire. Ce parcours n'aurait pas été possible sans *Louis Gautier* qui m'accepta malgré mon parcours atypique et encadra mes travaux d'abord dans le cadre de son master puis de la chaire *Chaire Grands Enjeux Stratégiques Contemporains*. Je le remercie pour sa confiance et son soutien. Enfin, je dois beaucoup à *Marc Shapiro* qui m'aida à concilier ces deux mondes.

Je tiens à remercier particulièrement *Dorian*, *Jean-Dominique*, *Gaëtan* et *Julien* pour nos échanges incessants qui ont enrichi ma réflexion par leurs débats (dominicaux) et leurs remarques pertinentes, avec une mention pour le premier seul non-informaticien qui connait, à son cerveau défendant, l'odeur de l'huile de vidange du compilateur Varda. En outre, je tiens à remercier *Rébecca*, qui par ses encouragements incessants m'a incité à me lancer dans cette double aventure.

Merci au groupe de grimpeur·euse·s et traileur·euse·s pour avoir aiguisé mon mental et ma résistance et grâce auquel j'ai gardé confiance en moi tout au long de ces années. En particulier, je souhaite remercier toutes les personnes m'ayant permis de prendre le départ et de finir la grande aventure qu'est le Marathon des Sables. Merci à la Tente 82, à mes six frères et sœurs des sables. Je n'oublierai jamais cette aventure. Avec un peu de méthode et un soupçon de volonté, nos limites sont beaucoup plus loin qu'il n'y parait.

Ces années de thèse auraient été moins agréables et moins enrichissantes sans mes ami·e·s de Paris à la Provence en passant par Marseille. Et bien sûr, *Emilie* et *Philippe* pour notre amitié sans faille depuis le lycée. Et toute la *Kolloc*, sans oublier son extension Covid, pour m'avoir supporté ces trois dernières années, souvent sous forme de courant d'air entre mes divers projets menés trop souvent de front. Pour celles et ceux encore engagé·e·s dans un doctorat: *Haut les cœurs !*

Pour finir, je tiens à remercier ma mère pour son soutien indéfectible depuis toujours. Yvonne, ma grand-mère, qui m'a forgé, sans toi je n'aurai jamais réussi ni académiquement, ni dans l'ultra-endurance, ni pour le reste d'ailleurs. Les mots ne suffisent pas pour exprimer ma reconnaissance pour tout ce que vous avez fait pour moi. Enfin, je pense à celles et ceux qui nous ont quitté-e-s trop tôt. Je ne vous oublierai jamais: *On lutte jusqu'au bout avec le sourire et on ne lâche rien, quelle qu'en soit l'issue!*

Abstract

Large distributed systems are often built by assembling off-the-shelf (OTS) components developed independently. The current approach is to interconnect their APIs manually. This is *ad-hoc*, complex, tedious, and error-prone.

To address this issue, Varda offers a higher-level language, motivated by a vision of safe-by design. To express a system, a programmer describes its architecture, involving OTS components, using well-defined entities and constraints. To provide safety, the compiler performs static verification, generates a correct-by-construction implementation and inject dynamic checks. To enhance programmers' productivity, Varda offloads the boilerplate plumbing to the compiler. To improve performance, the compiler applies property-preserving optimisations (e.g., component inlining).

Our experiments show that Varda applications are compact, exhibit modular and reusable design, and have a modest run-time overhead.

Keywords: Distributed System, Programming Language, Composition, Optimisations, Safety

Résumé

Les systèmes distribués sont souvent construits en assemblant des composants prêts à l'emploi (OTS) développés indépendamment. La pratique actuelle consiste à interconnecter manuellement leurs API. Cette méthode est complexe, fastidieuse et sujette aux erreurs.

Pour résoudre ce problème, Varda propose un langage de haut niveau prenant en compte la correction des systèmes dès la conception. Un programme Varda décrit l'architecture du système à l'aide d'entités et de contraintes formelles. Le compilateur vérifie statiquement l'architecture, génère une implémentation correcte et injecte des tests dynamiques. Pour gagner en productivité, Varda automatise la génération du code d'interconnexion. Pour améliorer les performances, le compilateur applique des optimisations préservant la sémantique du système.

Nos expériences montrent que les applications Varda sont compactes, modulaires, et ont un surcoût modeste à l'exécution.

Mots-clés : Systèmes distribués, Langage de programmation, Composition, Optimisation, Correction

Contents

Part I

Introduction

Distributed systems are everywhere, from smartphones to data centers. A distributed system is an assembly of logical units (compute or storage) located on different nodes, which communicate and coordinate together to provide a common service to other systems, services or apps to end users. Distributed systems range from multi-agent systems to large-scale systems for parallel data processing. A computing unit may be a process, an actor, a physical or virtual machines, devices, containers and so on. As opposed to a parallel system, a distributed system is not necessarily homogeneous, composed of loosely coupled units, has important inter-unit latency and suffers from unavoidable asynchrony and failures.

Distributed systems are at the core of our society's critical system, such as telecommunication or financial transactions. For instance, we are witnessing a technological convergence between distributed systems and telecommunication (e.g., 5G core network). The COVID-19 pandemic has further accelerated the digital transformation of our societies and has reinforced Cloud integration in our daily lives. Our appetite for remote human interaction (e.g., social network), data processing and collaborative work rely on complex distributed system, capable of ingesting data, storing and processing them, while guaranteeing privacy and consistency to some extent.

Such distributed systems are complex to design, develop, deploy, maintain and debug. They can fail in many ways, and they are exposed to many threats: crashes, asynchrony, network outages, etc.

Their intrinsic complexity is one of the major issues of distributed systems. This complexity is mainly due to the fact that they are composed of many elements, each having its own behaviour, (informal) specification, protocol, state, failure model, security model, etc. Moreover, the interactions between these elements are often under-specified, asynchronous and non-deterministic. Combined with the fast growth of distributed systems and their messy integration, the intrinsic complexity of distributed systems makes them hard to understand, reason and debug.

For instance, according to the 2021 Facebook Papers leak, the tangle of systems processing user data is such that it was impossible to know what personal data is processed by what systems and when. Untangling this situation was a major engineering challenge for the company. This implies to redesign and rearchitect the whole data processing pipeline, composed of several distinct systems. Internally, the company estimated the time cost associated with this redesign around 750 years of engineering work[1].

[1] `https://www.documentcloud.org/documents/21716382-facebook-data-lineage-internal-document`

A popular approach to decrease complexity is to re-use and assemble existing components (e.g., libraries, services, processes, etc.) often off-the-shelf or developed independently [60]. Actors, services (or microservices), virtual machines and containers are all mechanisms for reducing complexity and for composing and reusing components. Each component is dedicated to providing crucial services such as data processing, storage, synchronisation primitives and resource management.

In sequential programming, reusing libraries and frameworks is well established. Strongly typed, narrow procedural interfaces and encapsulation limit the opportunities for errors. However the situation darkens for distribution because existing approaches for managing this assemblage are less rigorous and disciplined than for sequential code. Typically, a component in a distributed system runs as an independent process or actor, which communicates through a message-based API, e.g., REST. This process lacks structure, is time-consuming, and error-prone. Extensive manual effort is required, including repetitive and standard code implementation. Concurrent interactions introduce a high level of complexity due to the exponential growth of possible combinations. Moreover, the lack of formal constrains hampers the verification of the concurrent interactions.

Ad-hoc composition comes with its set of harmful bugs. Those bugs are hard to detect and to fix: they are often non-deterministic, intermittent and hidden in some corner cases of integration. Recall that the size of mainstream distributed systems is measured in millions of lines of codes doing asynchronous and concurrent interactions with each other. For instance, a recent study reviews 120 cross-system interaction failures in production-ready and mainstream systems [168], indiscriminately in private services (e.g., AWS, Azure and Google Cloud) or in large open-source projects (e.g., Spark, Flink). Therefore there is a pressing need for tools to support the developers of distributed systems, at all stages: design, development, testing, debugging, deployment, monitoring, etc.

Programming languages could help to prevent bugs and help programmers building systems. Morever, a high-level programming language could improve programmer productivity by providing well-defined primitives and abstractions to manipulate different aspects of distribution. For instance, Charles, Grothoff, Saraswat, Donawa, Kielstra, Ebcioglu, Von Praun, and Sarkar [41] ensures deadlock freedom by construction using specialised parallel combiners. It can prevent subtle bugs, by either using static analysis (e.g., ownership), or by high-level abstraction of consistency or fault-tolerance policies. Most programming languages does not provide a global vision of the system and hide the complexity of distribution. They tend to get ride of the non-functional aspects of the system that are delegated to external tools

or configuration files. For instance, deployment, consistency, security and fault tolerance are assumed to be addressed by a separate system, not programmable with the same first-class abstractions.

In practice, programmers use programming language to build the different subsystem, then the rely on external tools, as orchestration engines, to mechanise the handling of non-functional properties. Unfortunately, these tools are not aware of the application semantics. Moreover, they are not subject to the same rigorous design, and analysis typically accorded to a traditional programming language [162].

At the other end of the spectrum, a specification language captures the functional aspect of the architecture. However, this has four main drawbacks: real software components have incomplete, evolving, heterogeneous and undocumented assumptions [162]; a specification language aims at expressing functional correctness, and leaves out non-functional aspects; the system architecture evolves [79], [162]; and, it adds an extra layer of indirection.

One needs a *pragmatic* high level approach for distributed programming interweaving specification language and programming language in order to easily compose heterogeneous distributed pieces in a sound way.

Our ambition is to raise the level of abstraction, thanks to a high-level language, called Varda, with simple top-level requirements: provide strong guarantees, have good performance, and automate common tasks.

Let us first consider the guarantees. To deal with off-the-shelf (OTS) components that might misbehave in arbitrary ways, we require that a component interact with its environment through a strongly typed interface, called its *shield*. A shield specifies:

- The signatures of invocations that the component may emit or accept.
- Its *protocol*, a state machine formalising the pacing of invocations.
- *Contracts*, predicates over invocation arguments and component state.

Components can be logically nested, to provide encapsulation boundaries. A higher-level component *orchestrates* the life-cycle of its inner components, spawning and killing component instances, interconnecting them through communication channels, *intercepting* and manipulating invocations and replies, and more generally computing over components and messages. The protocol, hierarchy and interception constraints are enforced by Varda.

Varda natively allows safe incremental development using interception. *Interception* make it possible to interpose adaptor code, in order to evolve a component interface

or to substitute a component with a similar one. Since interception is a first-class object in Varda, the compiler ensures that the functional guarantees of the composition are preserved.

Varda provides control over a few non-functional properties that are important for efficient system programming. First, *elasticity* make it possible to dynamically create or destroy component instances at run-time. Second, the developer may specify where to *place* a component instance, e.g., on a specific node or co-located with, or away from some other specific component[2]. Finally, Varda provides supervision mechanisms for fault-tolerance and non-stop execution.

To automate common tasks, the Varda compiler generates the interconnection *glue* code from the architecture. This includes supervising and responding to run-time error conditions, creating and linking sockets, marshalling/unmarshalling language-level data into messages and dynamic checking of safety conditions.

Varda provides built-in support for architecture optimisations. Our *inlining* mechanism can compile away expensive inter-process communication between co-located components. The idea is to group multiple logical units into a single execution unit (e.g., a process, a container, etc.). The developer may write safe, modular code and leverage interception without paying the overhead of crossing protection boundaries. Therefore, programmers can split the business logic at fine grain in multiple independent components - thereby leveraging modularity and incremental programming - while preserving the locality of data and computation.

Publications

Some of the results presented in this thesis have been published as follows:

- L. Prosperi, M. Shapiro, and A. Bouajjani, “Varda: An architectural framework for compositional distributed programming,” M. Mezini and M.-A. Koulali, Eds., vol. LNCS 13464, Online, May 2022, pp. 16–30

During my thesis, I collaborate on other research topics. These efforts have led me to contribute to the following publications:

[2]Note that, by design, placement is orthogonal to the encapsulation hierarchy. The developer may co-locate components that are at different levels of encapsulation but are closely coupled; conversely, she may place components that are at the same encapsulation level on different nodes, for instance, for fault tolerance.

- B. Martin, L. Prosperi, and M. Shapiro, “Transactional-turn causal consistency.,” in *29th International European Conference on Parallel and Distributed Computing (Euro-Par)*, Cyprus, 2023
- B. Martin, L. Prosperi, and M. Shapiro, “A new environment for composable and dependable distributed computing,” in *EuroSys Doctoral Workshop*, 2020

Organization of this thesis

This thesis is divided into three parts. This introduction is Part I.

In Part II, we introduce the background of our work, formulate the problem, present existing approaches and discuss requirements. This part is divided into two chapters: Chapter 1 discusses the problem statement and the top-level requirements. From this, we derive our requirements for composition-based programming. Then, Chapter 2 reviews the state of the art and positions Varda in the ecosystem.

In Part III, we detail our language, Varda, for safe component-based programming: its expressiveness, how it works, design choices and applications to system programming. This part is divided into four chapters. Chapters 4-6 describe different aspects of Varda: respectively, the core programming model; extensions to support fine-grain system programming with optimisations and provided guarantees. We conclude this part with the Chapter 7, which discusses the compiler and the run-time system.

In Part IV presents an empirical evaluation to demonstrate the usability of Varda for distributed system programming. We show that Varda successfully expresses safety and improves programmer productivity with reasonable performance overhead. Chapter 8 shows how to represent some common communication (e.g., streams, pub/sub mechanism) and distribution patterns (e.g., two-phase commit). Chapter 9 presents a full-scale example: a geo-distributed database. It concludes with a preliminary experimental analysis.

Part V, summarises our contributions; we discuss the pros, the cons and the scope of our approach; and, we present our vision for future development of Varda and research directions.

Part II

Background

1 Problem statement and requirements

Distributed systems suffers from an overwhelming intrinsic complexity that comes from failure, asynchrony, network costs and system size. A popular approach to decrease their complexity is to re-use and assemble existing components [60]. Moreover, when design distributed systems by composition, a developer has to balance between the two high-level objectives [113], [114]: *efficiency* and *dependability*. In the following, we discuss these objectives and identify the requirements to build a programming language dedicated to system programmer. Last but not least, to ease the adoption of such a language, we need to make the language *yummy* for system programmers.

However all these objectives conflict (Section 1.5). There is no single right solution for these trade-offs: it is application specific and it may evolve over time. Therefore, we conclude this chapter by proposing pragmatic requirements to let the programmer explicitly balance these trade-offs. Figure 1.1 summarise all the requirements identified in this chapter. In the next Part, Chapter 3 derives and explains the design of our language from these identified high-level requirements.

Methodology To evaluate the current state of practice, needs, frictions for the developers, we conduct interviews with developers and architects of distributed systems. They covered the following topics: platform for collaborative development (Plateform.sh, XWiki and the DiverSE team at IRISA/INRIA), Edge and IoT computing (AdLink and Concordant), storage and data management (AntidoteDB and Scality), and blockchain (Nomadic Labs).

1.1 Composition

The existing ecosystem of languages and tools already covers the construction of individual components very well, as well as network interaction technologies between components (below the application layer of the OSI model). To take advantage of the maturity of these technologies, we shall abstract them and focus on composition and distribution-related properties (Requirement 1). One of the best abstraction levels to describe systemwide features is *architecture* [82] since it focuses on interconnecting components and it describes the overall system structure and features. An architecture integrates all the blocks with their orchestration and interconnection logic.

Programmers have a fierce appetite for composition and modularity since they ease code re-use, written in various languages and possibly managed by multiple tenants. Modularity enforces a separation of concerns between the different building blocks such that each of them can be specified and built in isolation using dedicated tooling. Therefore, the developer should be able to import arbitrary black box components, built using any kind of technologies, in the architecture (Requirement 2).

Additionally, composition enables the incremental building of distributed systems. This helps keep the system easier to understand and to get it right [113], [114]. The idea is to enrich the system step by step by either adding new elements without changing the existing architecture or, by modifying the behaviour of the existing architecture. To automate that and to ease the work of the developers, a language must provide safe and built-in evolution mechanisms (Requirement 3).

Requirements Writing distributed systems, by composition : (1) program composition while abstracting away non-distribution related functionalities; (2) reuse existing code base and services written with heterogeneous languages; (3) incrementally add new components and new features without having to update the whole system.

1.2 Dependability

Building system by composition simplifies the design, the build and the verification process of individual components [66], [178]. Each module has a limited complex-

ity, which eases writing module specification[1]. This feature of composition is not widespread enough. On the one hand, the literature extensively covers the verification and testing of modules in isolation. On the other, in practice, according to our interviews, existing code base is loosely specified. Therefore, one need a common ground in between specification and system implementation. For this we enrich the architecture such that it should formalise the individual components [59] or at least their observable (Requirements 4); and specify how they communicate [@74], [109], [128] (Requirements 5).

Reasoning about the dependability independently for each module is not enough. The composition of modules may introduce new behaviours and some interesting properties do not compose [60], [109] (e.g., linearisability in the general case) . As a result, building systems by composition uncovers a new source of bugs: the *cross-system interaction* (CSI) failures [168]. Programmers introduce those bugs when they interconnect systems that do not enforce strict functional specifications. Indeed, the current manual, ad hoc approach to composing OTS components cannot ensure safety properties since interconnecting informal API ,thanks to network layer and configuration files, do not provide strong guarantees [@74], [128]. For instance, a classic stream processing infrastructure is made up of Apache Kafka (0.7 million LoC) for ingestion, Apache Flink (2 million LoC) for processing and Apache Zookeeper for synchronisation (170,000 LoC). Under the hood, Flink leverages systems like Hive (2.2 million LoC) and HDFS (700,000 LoC) to manage the data. Tang, Bhandari, Zhang, Karanika, Ji, Gupta, and Xu [168] counts 23 bugs in the previous stream processing architecture: 12 CSI failures between Flink and Kafka, 8 CSI failures between Flink and Hive, 3 CSI failures between Flink and HDFS. Note that having access to the sources does not change significantly the situation due to the complexity of each component. To address this, programmers should formally specify the orchestration and the communication of the composed object (Requirements 6).

Beyond the functional correctness of the composition, the *dependability* of a distributed system encompasses various non-functional properties [113], [114] since they directly affect the capacity of a system to provide the services for which it was deployed. Often non-functional correctness encompasses[2]: *availability*, the capacity to keep functioning, especially under load or failures; *consistency*, defined as the absence of contradictions between the observable; *fault tolerance* is the system ability to behave in a well-defined manner once a fault occurs. Therefore, the

[1] Specification tells you what a system do whereas the code tells you how.
[2] In this paper, we omit a major system requirement : security.

specification language should be able to express some non-functional properties (Requirement 7).

Having a specification separated from the implementation could ease the introduction of subtle errors when programmers fill the gaps in between [39], [60]. For instance, the Nomadic Labs team encodes the blockchain specification in Coq and writes its implementation in Ocaml. At the time of the interviews, the Coq specification is orthogonal to the Ocaml implementation: engineers bridge the gap by hand. Therefore, there is a lot of back and forth between specification and implementation to evolve the design of the system. To avoid this, a mechanised process should ensure that implementation of the architecture follows its specification (Requirement 8).

The former separation makes it difficult to maintain a specification that keeps pace with the evolution of systems. Indeed, programmers adapt systems during their lifetime to meet new requirements, to scale up or down, or even to preserve compliance when the regulation updates. To address both issues, programmers should write the architecture and its specification in a single place to reason about the global behaviour of the system (Requirements 7). In addition, this helps fight against the *lack of global vision*, which identified as one major practical limitation when building large-scale systems by composition. System designers often struggle with the cartography of their systems and lack a global and up-to-date view of the system to reason on the overall properties.

Requirements Ensuring safety implies the programmer to be able to: (4) formalize the individual components; (5) specify how they communicate; (6) specify the orchestration and the communication of the composed object; (7) formally write the whole architecture with additional specification in a single place to reason about the global behaviour of the system. (8) ensure that the composed implementation follows those specifications.

1.3 Efficiency

The individual efficiency[3] of components and network links is outside the scope of this work since existing tools are mature and performed well. Leaving aside

[3] *Efficiency* encompasses traditional performance metrics (such as CPU, memory or energy consumption).

individuals, the overall performance of a distributed composition is mainly correlated with *scalability*, *elasticity*[4] and its ability to *run non-stop* [113], [114]. During our interviews, system programmers require to be able to programme and control at fine grain these three features to correctly design and build high-performance systems. Since the features depends on the semantics of the application and directly impact the capabilities of the system, the architecture should model this logic as orchestration code (Requirements 9-11). Under the hood, non-stop execution often requires asynchronous communication between components (Requirement 9). Furthermore, elasticity requires that the orchestration is aware of the underlying resources, e.g., nodes location, and can reason about them (Requirements 10- 11) .

Programmers often have to finely tune the network layer that interconnects the components according to the nature of the system. For instance, to programme efficient distributed storage in a cluster, they need efficient serialisation and asynchronous communication. Conversely, they leverage brokers (e.g., Kafka or RabbitMQ) to implement multi-cloud communication to have a single logical exchange point that provides interesting properties like adaptable routing logic, back-pressure and message persistency in case of a network outage. Hence, programmers should have on-demand access to low-level primitives when this matter for performance (Requirement 12), for instance, to customise the network.

A side effect of the quest for modularity is scattering the logic across a large number of isolated execution units (e.g., containers or processes). This dispersion induces a significant performance overhead due to context switching and remote communication (latency, message marshalling and unmarshalling). For instance, a recent feedback on the monitoring platform of Prime Video shows that moving from a distributed microservice architecture to a monolithic architecture significantly improves performance[5]. Their initial serverless architecture suffers from two main problems: the cost of managing orchestration and the cost of sending video images between the different components. Engineers re-architected the system by collocating the orchestration and the different analysers in a single service.

Addressing this issue implies to preserve the locality of data (resp. computation) either in the same execution unit or in close enough units to reduce communication cost. On the one hand, programmers should be able to co-locate components in the same nodes (Requirements 11). On the other hand, they should be able to group

[4] *Scalability* is the ability of a system to handle a varying demand for work by consuming a proportional amount of resources (e.g., nodes). *Elasticity* denotes the capacity of a system to *dynamically* enlarge (or shrink) to sustain the load.

[5] `https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90`

logical units (i.e., components) in one execution unit to avoid context switching. Moreover, both optimisations must be easy to apply throughout a project's life cycle as performance bottlenecks often appear on system use and vary according to its maturity. To enhance productivity and development efforts, a programmer should apply these transformations orthogonally to the logical structure of the architecture without rewriting the system (Requirement 13).

Requirements Writing distributed systems, by composition, with good performance, requires the developer to be able to: (9) build non-stop and asynchronous systems; (10) support a form of elasticity; (11) control non-functional and performance-related properties, such as component co-location; (12) on-demand access low-level primitives when this matters for performance, for instance to customize the network; (13) apply architecture optimizations, for instance compiling away expensive inter-component communication.

1.4 The ergonomic problem

Fulfilling the previous requirements is not enough, system developers should accept the proposed language. This is a difficult and non-specific to distribution languages. It often exists a gap between what the programmers use and what are the available mature tools. For instance, the C (resp. C++) language is still widely used in system programming despite its lack of safety. During the last decade, memory safety remains the most common vulnerability class [133].

Transitioning to safer language and tools requires that their design limits the *frictions with the developers* [133]. Otherwise, programmers will not adopt them whatever their quality and functionalities. Therefore, our language should be *ergonomic* for system developers, it should:

- make unsafe things hard to do, however, the programmer must be able to do them intentionally since system programming often requires to do so (Requirement 15);
- increase programmer productivity in order to give incentive to use it (Requirement 14);
- simple enough and *familiar* for system developers (Requirement 16).

During our interviews, we cartography the technologies and processes that the system builder community use to programme distributed system. This cartography highlights that 1. programmers are reluctant to use high-level distribution languages because they fear to lose control on their system; 2. system specification is mostly informal; 3. and system validation relies on manually writing tests.

Programmers tend to avoid distributed programming language since they fear loss of control on the system and to struggle to find the origin of a bug or a performance bottleneck. They use general-purpose languages such as C, C++ and Rust for low-level backend and Python, Typescript for the exposed interfaces.

At first glance, this trend seems to go against the idea of relying on distribution-specific abstractions to ease programming and to improve the dependability of the system. However, programmers express the needs for new tools to handle the inherent complexity of distribution and component integration. Tackle this reluctance, while providing safety guarantees, demands to provide easily explainable generated code (Requirement 17) and to let programmers specialise the building blocks of the languages according to there needs (Requirement 18). For instance, they should be able to update a memory allocator for a given component or to embed their own implementation for a given network link.

Similarly, our interviews show that programmers rarely use existing solutions to specify and verify distributed programs. Most of the time, system architects write specifications in an informal language (e.g., English). From time to time, they formalise small critical parts of the system. For this, programmers write the specification using static verification tools, such as model checkers, to detect errors. For instance, the Scality team use TLA+ [111] to model some synchronisation protocols. Therefore, to reduce the developers' frictions the core specification language of the architecture should rely on well-known and used formalism: types and predicates without fancy high-level logic.

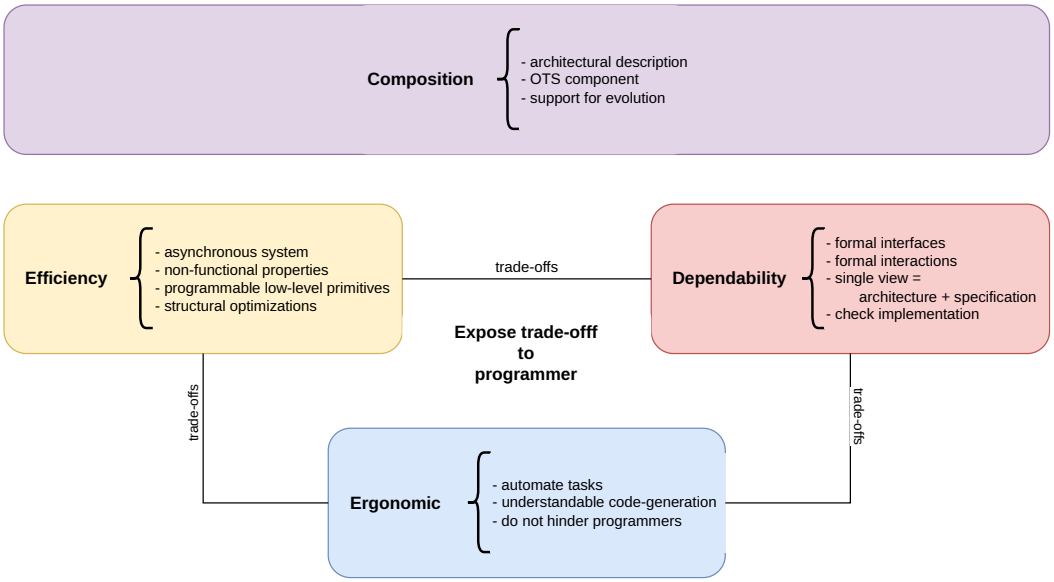


Fig. 1.1.: Our top-level objectives and the resulting requirements.

Requirements

Requirements To reduce the *frictions with the developers*, a new language should: (14) relieve the programmers from repetitive programming tasks; (15) do not hinder programmers, i.e., make unsafe things hard to do but not impossible; (16) expose a familiar language entities; (17) make the generated implementation understandable;(18) let programmers specialize the building blocks of the languages.

1.5 Sumup

Each of these aspects (modularity, efficiency and dependability) have inherent trade-offs and are conflicting with each other. They lead unavoidably to trade-offs when designing systems (e.g., CAP [25], FLP [77]). For instance, efficiency and dependability conflict with each other. On the one hand, dependability often requires strong component isolation to dynamically ensures some safety properties. On the other, high performance requires zero-cost abstraction and to breach isolation to avoid paying an extra overhead.

There is no one-size-fits-all solution as they depend on the application requirements, the expected environment and workloads, the available resources, etc. To be usable in a large range of distributed systems, a high-language should provide a *pragmatic approach to let the programmer explicitly balance* those trade-offs according to its needs (Requirement 19). For instance, this means making costly verification and dynamic checking optional. In addition, the language should guarantee strong isolation by default, and on explicit programmer demand breach isolation for performance. The key idea is to avoid forcing developers who do not need the expressive power of an operation to have to pay for it (in terms of performance or of cognitive cost). Therefore, one needs to expose an adaptable programming model composed of simple and specialised building blocks [113], [114].

Requirement To let a programmers balance between those requirements, our proposed language should (19) offer a wide range of simple and specialized building blocks.

2 Related Work

We see three categories of prior work: *distributed programming languages*, *tools for composing* OTS modules, and tools that aim for reaching *high-level guarantees* on systems. Accordingly, we structure the following state of the art along those three categories. We discuss our three top-level objectives (i.e., dependability, efficiency and ergonomic) through the following sections for each sub-category of prior work.

2.1 Languages for distributed programming

In one form or another, programming languages are the main tools that developers use to create their systems. On the one hand, using a programming language specialised for distribution programming improve programmer productivity by providing well-defined primitives and abstractions to manipulate different aspects of distribution. On the other hand, it helps to prevent bugs.

This section is structured according to the main ways to abstract distribution in languages. For each category, we present the expressiveness of the underlying programming model and we discuss if it is suitable for system programming in the light of the previous chapter. In particular, we discuss the following aspects: 1. does it ease composition of OTS modules? And, how easy it is to evolve a system? 2. what guarantees does it provide on the resulting system? 3. what degree of performance control does it offer programmers? We conclude this section by a synthetic comparison of these different categories and we highlight the main borrowings of Varda. We defer all non-programming languages (e.g., specification or interface language) to the following sections (Sections 2.2-2.3).

2.1.1 The actor model

The actor model [97] enables to program a modular system at fine grain. A system built using the actor model is composed of a set of "atomic" execution units, called actors, that interact by exchanging messages. An actor is a "process" that responds

to messages it receives by making a local decision, creating other actors, sending messages and determining how to respond to the next message. An actor may modify its own private state, but cannot affect another actor except through messages.

An actor alternates between two states: ready to accept a message or busy processing a message. A *turn* is the processing of a single message by an actor until completion [55]. An actor's turn terminates without interruption. Within one actor, turns do not interleave.

There are a lot of actors frameworks implemented from scratch (e.g., Erlang [16]) or on top of mainstream languages (e.g., Akka [@8], Orleans [30] or Pony [@47]).

Composition Actor programming model requires that the entire system is rewritten using actors. It does not provide any mechanism to embedded existing OTS components. Programmers have to write their own actors to wrap existing components. Moreover, it prescribes using other forms of concurrency control (e.g., threads or classical futures) in conjunction with actors [92], [167]. There exists a few experimental actor languages that mix actor with futures [84], [136].

To compose actors, or group of actors, a developer programs the sending of messages and their reception, using callbacks. There is no specific combiners for composition except method calls and native streams.

Adding new features to a system means creating new actors and interconnecting them with the previous system by updating the existing code. There is no built-in mechanism to evolve the actor-based architecture of a system. Programmers have to manually update the source code. However, some actor runtimes (e.g., Erlang) provide hot-swapping mechanisms to apply the update on a running system without downtime.

Dependability

The model actor model provides strong isolation between actors. By nature, an actor cannot directly alter the memory of another actor. It can only communicate through messages. This is a useful property as it prevents memory-based deadlocks. Note that, message-based deadlocks are still possible.

Most of the existing framework guarantees extend the actor isolation to failure: the failure of an actor does not make other actors crash. In addition, most of the frameworks provide, to the programmer, building blocks to react to failure. The most

common mechanism is to provide a supervision tree, parent actors can monitor the execution of their children, and can react to their failure [16]. Bykov, Geller, Kliot, Larus, Pandya, and Thelin [30] discharge the programmer from writing the reaction logic to a failure. It leverages actors state persistency and optimistic transactions to automatically recover for an actor crash. Each incoming request starts a transaction. On transaction success, the actor persists its new state. Otherwise, it discards its transient state and restart the transaction.

Actor models do not come with specific static analysis nor verification tools. Most of the framework relies on the type system of the host language. For instance, Akka [@8] uses the Scala (resp. Java) type systems to detect illegal sends of message: a message can only be sent to an actor that has a callback that can handle its type.

Regarding our Requirement 7 for dependability, the actor model suffers from a major issue: it tangles the representation of the system, especially for large systems. Programmers have to invest a lot of efforts to understand, to maintain the code, and to track bugs or performance bottlenecks. According to Boer, Serbanescu, Hähnle, Henrio, Rochas, Din, Johnsen, Sirjani, Khamespanah, Fernandez-Reyes, and others [23] this is intrinsic to the actor model since its usage tends to facilitate the appearance of the callback hell [70], which denotes the situation where callbacks are nested within other callbacks several levels deep.

Efficiency

Most actor frameworks leverage the underlying runtime to dynamically optimise the network communication between runtime node or to replace messages with shared-memory communication when the communicating components are on the same runtime node [@8]. Moreover, the Orleans [30] runtime can dynamically optimise the system by controlling the activation placement. It uses a load balancing and load shedding policy with the ability to migrate actors between servers to balance the load at run time.

Conversely, the compilers often do not consider optimising the architecture. On the one hand, host compilers are unaware of the distribution semantics of the program. On the other, built in compilers focus on generating efficient local code [155].

Non-functional properties In a traditional actor system, the developer remains responsible for the creation, the placement, the discovery, the recovery, the scaling

and the load-balancing of actors. Even with actors, this remains complex and tedious. Two kinds of works address this issue, the first one by exposing high-level primitives to programmers [157], the second one by automating the management of actors [30].

Sang, Roman, Eugster, Lu, Ravi, and Petri [157] augments the actor model with programmable scalability overlay. Programmers specify high-level bottleneck conditions (e.g., an excessively high volume of messages exchanged among specific actors), with their corresponding mitigation actions (e.g., co-locate actors).

To automates all of these aspects, Bykov, Geller, Kliot, Larus, Pandya, and Thelin [30] introduces virtual actors. A virtual actor is a stateful actor that supports multiple instances, called activations. An activation is the base execution unit. It has the same structure as the virtual actor. Upon creation, its state is a copy of the virtual actor one. During its life, the activation has a dedicated local state that evolves independently from the virtual actor state and from the other activations state. Upon termination, the runtime merges the state of the activation into the state of the virtual actor.

For instance, Orleans can automatically and transparently scale up or down according to the load with a small overhead. When the load increase, the runtime spawns new activations. Each activation state is hydrated using the persistent state of the virtual actor. When the load decrease, the runtime terminates some activations to free resources. The runtime merges the state of the terminated activations into the persistent state of the virtual actor.

2.1.2 Dataflow programming

Dataflow programming provides a simple programming model to design the distributed pipeline of computation. A dataflow program is a directed graph with nodes representing operations and edges representing data dependencies between computations [17].

Traditionally, dataflow implies a complex engine to schedule the execution of the graph, to dispatch the operators on nodes, and to manage the data at run time. Most of the time, this engine automatically provides the following non-functional properties: elasticity, placement optimisations, replication, and fault tolerance.

Large-scale data processing widely use dataflow platforms [7], [19], [33], [37], [56], [138], [175], [176]. Indeed, the dataflow paradigm is well suited to describe

the pipeline of the data. Moreover, dataflow frameworks integrate well with external services that emit or store data. A dataflow program can natively ingest (e.g., from a Kafka) or sink data to external services (e.g., a database).

Composition As we said, the dataflow paradigm is well suited to embedded OTS components that consume or produce data. For this, a programmer warps the OTS inside an operator. Mainstream dataflow frameworks provide out of the box integration with classical OTS (e.g., brokers and data stores).

To compose two operators, a programmer simply needs to interconnect an output "port", i.e., an output data stream, of the first operator to an input "port" of the other operators. The number of output (resp. input) data streams of an operator depends on its type. For instance, a filter operator has one input and one output data stream whereas a join operator has two input data streams and one output data stream.

This composition mechanism suffers from a major limitation: it does not support the composition of subgraphs. A programmer has to manually list the operators of the first subgraph and bind their output streams with the correct operators of he other subgraph. This limitation comes from the fact that the dataflow graph does not support nested operators. Indeed, a node cannot abstract a whole subgraph.

This hinders the evolution of the dataflow program: programmers have to manually update the whole program. Adding new features to a dataflow program means creating new operators and interconnecting them with the previous system by updating the existing code. For instance, to replace a map operator by a subprogram, the programmer has to explicitly add all the subprogram operators to the main program as top-level operators and to correctly bind them with the pre-existing operators.

Dependability

Most of the dataflow engines [19], [33], [56], [138], [176] automatically provide fault tolerance, high-availability, and a limited form of consistency guarantees. In this case, consistency often refers to guarantees on message delivery between two operators (e.g., at most once, at least once or exactly once delivery).

Interestingly, the graph structure eases a global view of the program, with the appropriate visualisation tool. Moreover, the structure eases static analysis and helps the engine to manage fault tolerance and high-availability, and to optimise the program, as we shall discuss later.

However, the graph structure often support a limited level of detail. Operator interfaces only define their input (resp. output) streams along with the type of the events they carry. The internal of operators are completely hidden and the programmer cannot specify their observables. Often, the compiler only checks that the interconnected operators are compatible, i.e., that the output type of an operator is compatible (i.e., a subtype) with the input type of the other operator.

Efficiency

Thanks to the graph representation, dataflow programs support a wide range of optimisations: *graph rewriting* at compile time and *graph scheduling* at run time. Most of the dataflow frameworks apply them in sequence [10].

At compile time, graph rewriting provides nonlocal optimisations. A graph rewriting optimisation transforms an input dataflow graph in an equivalent[1] one to improve the performance of the computation [98], [102]. For instance, one common optimisation goal is to reduce the network cost of the execution. The compiler can easily optimise the program structure. The literature explores various kinds of rewriting. For instance, operator permutation can reduce the network cost by moving light weight operators first [98]. Operator grouping aims at reducing latency by increasing data locality [102]. It merges multiple operators in a logical one to avoid splitting them on distinct nodes.

At run time, the graph scheduling maps the operators to the underlying infrastructure (i.e., nodes) while avoiding overloading nodes and minimising various (dynamic) metrics [150], e.g., the execution time or network cost. Moreover, some works take the heterogeneity of the infrastructure into account to allow data processing on a hybrid Edge-Cloud infrastructure [33]. Most of the engine scheduling relies on an external declarative configuration. Developers cannot program their behaviours inside the dataflow application. In some cases, programmers can refine operators with declaratively annotations to locally specialise the scheduling. For instance, with Flink [33], they can specify an upper bound on the number of replicas per operator in order to guide operator replications, that is use to improve throughput.

The same limit holds for the management of other non-functional properties (e.g., fault tolerance, elasiticy, etc.). They are most of the time non-programmable in the dataflow model, but they are often configurable using external configuration files. Moreover, the programming model cannot represent resources as places or network links.

[1] In terms of observable.

There is a variant of dataflow programming, namely synchronous dataflow programming [91], [116], [117], that get rid of the runtime engine. This variant takes advantage of the compile time knowledge of the dataflow graph and of the underlying infrastructure. Therefore, the compiler can statically convert the dataflow program into sequential programs and statically schedule them on the underlying infrastructure. However, it does not support dynamic evolution of the scheduling when the load changes.

2.1.3 Distributed reactive programming

Distributed reactive programming generalises the dataflow paradigm [18], [72], [123]. It provides high-order nodes, i.e., an operator can encapsulate a set (or a hierarchy) of operators. Moreover, programmers do not have to explicitly define dataflow graphs. They can write classical programs where the different remote entities are interconnected by shared reactive variables.

A reactive variable is a special type of variable that automatically updates its value in response to changes in the system or other variables it depends on [17]. For instance, let us consider the following variable declaration: `x = y + 1`. Whenever the value of `y` changes (on a remote node), the value of `x` is automatically updated. From the underlying dataflow graph perspective, this declaration adds a directed edge from node `x` to node `y`.

Under the hood, the reactive runtime ensures the propagation of changes [73]. They are various flavours of propagation (e.g., a push or pull model) [18], [123]. Some of them specialise the semantics of the reactive variables. For instance, propagation changes can either happen in one direction or in either direction. Bidirectional means on our previous example, that the engine 1. updates `x` when `y` changes; and, 2. updates `y` when `x` changes.

As a result, the dataflow graph cannot be known statically anymore. The runtime engine has to dynamically maintain it.

Composition With reactive programming, sharing variables is the core mechanism to compose remote entities. Therefore programmers can reuse classical programming entities for modularity [101], [123] (e.g., functions, modules, objects, etc). For instance, compared to traditional programming, actors and distributed objects, reactive programming leads to code that is more composable and more compact than with the actor model [156].

Likewise, embedding OTS components is external to reactive paradigm, programmers should use the host language mechanisms, if any, to do so.

Dependability

Compared to the actor model, reactive programming get ride of the callback hell since programmers can program linearly their application. As a result, the code is easier to understand and to maintain [129], [156].

Compared to dataflow, the dynamicity of the graph limits the benefits of static analysis and hinders the ability to automate some dependability features. Often the runtime only provides guarantees (e.g., consistency and fault tolerance) that covers the propagation. For instance, Drechsler, Salvaneschi, Mogk, and Mezini [68], [69] provides different level of consistency guarantees for variable propagation to avoid glitches[2].

To cover the logical components, programmers often have to write their own logic. Mogk, Baumgärtner, Salvaneschi, Freisleben, and Mezini [134] propose an extension that automatically stores and recovers program states from crashes while preserving weak consistency. The runtime automatically propagates errors using the graph of dependencies. Moreover, developers can integrate their own fault-tolerance logic leveraging an `onError` guard on reactive variable .

In addition, dynamically maintaining the dependency graph complicate the handling of faults. Indeed, this mechanism leads to a tighter coupling between the dependent components of the application, making them less resilient to network failures and may reduce overall scalability [68], [69].

Efficiency

The reactive manifesto [@24] 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. On the one hand, the asynchronous and non-blocking nature allow the programmer to easily design their system to match their requirements in terms of scalability. On the other hand, likewise for dependability, it limits what the runtime can do to optimise the execution. Moreover,

[2] Glitches are updating inconsistencies that may occur during the propagation of changes. They can come from the order of instruction (eliminate glitches by arranging expressions in a topologically sorted graph - i.e., decency) or by distribution (network failures, delays, and lack of a global clock).

reactive programming is not designed to expose resources (e.g., places, links, etc.) which make it difficult to program with fine grain control on the performance execution. This is exacerbated by the complexity of the runtime execution model, which makes it difficult to predict performance.

2.1.4 Multitier programming

Multitier languages, also known as tierless programming, seek to make developing distributed systems closer to programming single-host applications by making logical entities orthogonal from the distribution unit and their location.

Traditionally, programmers split their application in multiple logical tiers (i.e., module) according to the running location of the code [170]. For instance, a web application is often split in two tiers: the client and the server. With multitier programming, the programmer can combine functionalities from different tiers in the same logical unit (e.g., an object). For instance, the developer can modularise the web application per features. Then, write linearly a function for each feature that embedded the client and server code. It is up to the compiler to automatically splits the compilation unit into artefacts, one for each tier, and to generate the necessary communication code in between [141], [158].

Composition Multitier paradigm enables the programmer to changes the boundaries of its logical entities to fit the system features and not the placement of the code. This paradigm does not impose a particular composition mechanism. It depends on the nature of the logical entities (i.e., function, object, etc.), which are often imported from the host language. Likewise, support for OTS components is external to tierless programming.

Various works hybridise the multitier paradigm with the reactive one [154], [173]. Their goal is to use the reactive variable to interconnect logical units while leveraging the ability of mulititier to avoid modularising the program according to tiers. In practice, this reduces the complexity of a distributed system since programmers do not need to scatter the logic of the same feature in multiple code entities.

Dependability

Since programmers can program linearly, this paradigm reduces the callback hell [104], [130]. However, apart from limiting the complexity of the application[3], tierless does not provide additional guarantees.

As a negative side effect, programming without modularising according to tiers scatters the communication code in the application since tiers often model the communication boundaries. Weisenburger, Köhler, and Salvaneschi [173] addresses this issue by adding a layer of static checking to ensure properties about the placement of data and computations. For this, programmers annotate the application with types that represents tiers and their legal static communication topology. For instance, they can specify that a client cannot communicate directly with a storage backend. Then, programmers type their reactive variable with the appropriate tier type, i.e., placement. Based on this additional information, the type system enforces that the programmer cannot access remote data without explicitly asking for it and that the dynamic communication respects the static topology.

Efficiency

Tierless main purpose is ergonomic, not efficiency. For instance, the compiler can generate communication code between tiers (e.g., data marshalling, network communication). However, it does not enable specific optimisations nor control non-functional properties.

Most of the works on tierless programming focus on web orchestration [43], [153], [158], [171]. As a result, they do not take system programming into account. By mixing the reactive and multitier paradigms, ScalaLoci [173] proposes a general programming paradigm and apply it to program a stream processing platform engine. However, when it comes to controlling over performance, this work suffers from the same drawbacks as those of reactive programming.

2.1.5 Serverless programming

Serverless computing shifts the focus towards writing and deploying code, instead of dealing with infrastructure management [93]. Developers program with known abstraction, functions, and focus on the application logic. They do not need to

[3]Which indirectly limits the likelihood introducing bugs.

bother with the distribution aspects of the application. The paradigm abstracts the infrastructure and the underlying resources: the concept of the server is not visible at the application logic. The underlying platform transparently manages the infrastructure [71], [159]: it automatically handles parallelism, placement, elasticity and fault tolerance.

Serverless computing is an approach where the program runs short-term function in response to events [27]. The quantum of computation is a serverless function. Functions are asynchronously triggered by external events. The source of an event could be a database updates, an incoming API request, a scheduled event (by the runtime) or a call from another function. When a function is triggered by an event, the serverless platform automatically creates an instance of that function to handle the event. The instance is "destroyed" when the function has finished executing and outputs a response.

Composition The core composition mechanism is the function calling.

Very often serverless application orchestrates various OTS components. Furthermore, these applications rely on external services to persist data (e.g., databases, object storage). To embed them, programmers manually write serverless functions that call the exposed OTS API (e.g., a REST API).

Dependability

Most of the serverless platforms automatically provide fault-tolerance and guarantee failure isolation between independent functions.

Conversely, serverless complicates the modelling, verification and overall reasoning on applications. On the one hand, the cognitive complexity increase due to return of the callbacks hell. Functions can be triggered by an external event, breaking away from the more natural sequential reasoning. They also often interact with various remote services which makes difficult any semantic reasoning. On the other hand, the programmers move the state to external data stores because most of the mainstream platforms propose stateless functions or have arbitrary limitations on the size of messages that can be sent between functions (e.g., AWS EventBridge is limited to 256KB per message). This requires programmers to reason upon parallel and concurrent access to these states, and to handle partial execution failures [27]. This is notoriously difficult.

Efficiency

Serverless is designed to build applications on top of existing distributed systems, rather than the other way around. Hence, the nature of serverless goes against providing fine grain control on the execution by system developers.

2.1.6 Summary

Table 2.1 compares the previous languages according to the requirements we identified in Chapter 1. As a result, we observe that none of these languages offers mechanisms to ease transparent composition of remote components. In addition, they offer limited expressiveness and checking (either static or dynamic) capabilities to increase the developer's confidence in the correctness of the composition. Conversely, for efficiency, the trend is to automate non-function properties (e.g., fault tolerance, elasiticy) and to make it transparent for the programmer. This work well for developers that build high-level applications. However, it is not suitable for system developers since they are seeking control (Chapter 1).

Position of Varda From actors, we pick the overall execution model of a component (i.e., send/receive messages and spawn children).

We follow the core principles of the reactive manifesto: loosely coupled components, asynchronous and non-blocking interaction. However, to provide enough control over performance to programmers, we avoid introducing dataflow graph nor variable dependencies. We use a programming model closer to the execution model.

Unlike tierless, we do not try to bring distributed programming closer to programming single host to avoid restricting the control of programmers on the distribution behaviours. As tierless, our compiler can automatically generate specific artefacts for specific targets without having to split the architecture in terms of tiers.

Varda follows a completely different approach than serverless. Serverless abstracts away too many infrastructure details that matter for performance. Indeed, both approaches have distinct objectives: the former one target system developers whereas the latter focuses on making it easier to build distributed applications, built on top of existing distributed systems.

	Actor model	Dataflow	Reactive	Mulititier	Serverless
Expressiveness					
Execution unit	actor	operator	any execution entities		function
Communication	message	stream	shared variable	variable	function call
Composition					
Built-in OTS	✗	✓	✗	✗	✓
Composition mechanism	✗	✗	✗	✗	✗
Program evolution	manual	manual	manual	manual	manual
Hot swapping	✓	(✓)	✗	✗	(✓)
Guarantees					
Isolation	✓	✓	✗	✗	(✓)
Additional specification	✗	limited	limited	✗	✗
Verification	✗	limited	✗	✗	✗
Global vision	✗	✓	✗	✗	✗
Efficiency					
Non-functional properties	manual or automatic	automatic	manual	✗	automatic
Expose resources (e.g., place, network)	✗	✗	✗	✗	✗
Runtime optimisations	✓	✓	✗	✗	-
Architecture optimisations	✗	✓	✗	✗	-
Ergonomy					
Code generation	✗	✗	✗	communication plumbing	✗
Varda's borrowings	✓	✗	✗	✓	✗

Tab. 2.1.: Comparison of the four main language classes for distributed programming according to the requirements we identified in Chapter 1.

2.2 Flexible system composition

There exists to our knowledge four main (and non-exclusive) ways to tackle the composition problem at the scale of a large architecture: API interconnection, orchestration engines, interception mechanism and composition languages.

2.2.1 API and Interface Description Languages

An Interface Description Language (IDL) formalises the API of services. Then, the IDL compiler generates invocation skeletons, which automates marshalling and unmarshalling arguments into messages. Popular examples include Google's Protocol Buffers [@86] (Protobuf), Apache's Thrift [@78] and OpenAPI [@3], [@165], which permits to model an HTTP-based API.

Sharing a common IDL enables interoperability between components written in various languages. For instance, to make a Go client interoperable with a Java server, a programmer describes a Protobuf client interface in Go (resp. server interface in Java). Then, the IDL compiler generates the client and server stubs that ensures the translation between Go (resp. Java) data structures to Protobuf messages. Finally, to interconnect both components, the programmer has to manually configure the Python client to connect to the Java server address.

IDLs offer limited support to specify properties about components. They only specify simple signatures. For instance, they do not specify the legal (resp. illegal) observable of components they encapsulate. Furthermore, they are often limited to RPC-style protocols which prevent them to model complex interactions. As a result, mainstream IDLs do not prevent cross-system interaction failures [@74], [168]

Similarly, IDLs do match our efficiency requirement. Most of the mainstream IDLs lacks orchestration and deployment logic. As a result, they cannot express non-functional properties about the composition. Corba [172] is an exception, however, it is not suitable for system programming as we will explain below. In addition to its IDL, Corba models the orchestration logic. For this, Corba exposes distributed objects that can invoke methods on remote objects, pass parameters, and exchange data transparently. Under the hood, Corba uses middleware and gateways to run the orchestration logic independently of the composed components. However, this additional layer of abstraction induces a performance overhead and tends to complicate the overall architecture [85].

2.2.2 Orchestration and composition engines

An *Orchestration engine* is responsible for automating the management and the coordination of components (e.g., containers or VMs), ensuring their elasticity, availability, fault-tolerance and efficient utilisation of resources engines such as Docker Swarm [@63], Kubernetes [@2] or OpenStack [@4]. Moreover, they are powerful tools to automate deployment and to control network topology.

By design, they compose and orchestrate OTS components by interconnecting their network interfaces [@5], [28], [@62]. They are business logic agnostic. They only requires that developers packages each component in their format (e.g., Docker image or VM image). Often, programmers rely on IDL to make components interoperable. Then, they rely on orchestration engines to manage them.

One of the major interests of this engine is that they transparently handle the deployment and various non-functional properties of the system. For instance, Kubernetes [@2] can provide persistency at the file-system granularity. Moreover, programmers can express simple placement policy using containers' annotations, called labels. For instance, the programmer can specify that a container must be collocated with another one or should be replicated on every node of the cluster. Programmers can also use the engine to scale the system. One common way is to replicate containers [@2], [@63] while using under the hood distributed system (e.g., DB or file system) to share state when needed. However, this is completely orthogonal with the specification of the system: scalability properties are not included in the system specification and, conversely, the engine is unaware of the semantics of the business logic.

In addition, these engines support rolling updates of application deployments, allowing new versions to be gradually rolled out while maintaining the availability of the application. In case of failures, the orchestration engine can perform rollbacks to the previous stable version. However, there is no built-in mechanism to evolve the description of a system architecture. Programmers have to do it manually.

Orchestration engines cannot enforce interesting composition safety properties [@74], [128] since they are run-time tools that interconnect network APIs, but are unaware of application semantics.

2.2.3 Interception mechanisms

To support incremental building of systems, programmers often indirect communication through a proxy. The proxy enables evolving part of the architecture without modifying the other components. Indeed, the proxy interposition is often transparent for the pre-existing components. It integrates well with orchestration engines. For instance, network proxy interposition is the mainstream solution to add access control or load-balancing to containerised HTTP services.

Under the hood, transparently adding a proxy requires a form of interception. Interception denotes the addition of an indirection layer between a set of communicating entities without altering the code (resp. configuration) of these entities.

Interception is a common problem, which has prompted many creative approaches. For instance, firewall features can be used for redirection and interception at the network layer, such as iptables [@1] or mesh services [@26], [@103] in containerised environment. Reverse proxies (e.g., Nginx or HAProxy) intercept and modify the communication between two communicating components to handle encryption or load-balancing for instance. Service workers [@137] redirect requests within a web browser. This enables, for instance, to interpose a persistent cache transparently. These mechanisms are very flexible, but they do not provide any semantics and provide no correctness guarantees since they are intercepting APIs calls and network packets.

Conversely, aspect-oriented programming language (AOP) [106] leverages a built-in interception mechanism to increase modularity by allowing the separation of cross-cutting concerns (e.g., logging). It does so by adding behaviour to the existing code without modifying the code itself. Instead of separately specifying which code is modified, it provides a global entity call "cross-cutting concerns" that permits updating globally the code structure. The programmer encapsulates the crosscutting logic into an additional piece of code called *advice*. Then, the advice is applied at a remote *pointcut* [110], [140], [145], [147]. A remote pointcut is a set of execution points on remote execution units. For instance, a pointcut can be the entry point of a function. Then, the logging advice is applied at the entry point each function. Applying an aspect, i.e., an advice on a pointcut, leverages a form of interception: the advice transparently intercept the flow of the local program.

Even if AOP is aware of the application semantics, it does not match our requirements. It limits the ability to develop programs independently [166]. Indeed, its interception mechanism is not designed to intercept network communication. As a result, it requires that all the intercepted components are written in the AOP language.

Moreover, it increases the cognitive load of the programmer since it complicates the control flow [49], [166].

2.2.4 Coordination and choreography languages

A coordination/choreography language tackles the composition problem from a head-on perspective. It models the interactions, coordination and synchronisation between multiple remote components. Such a language aims at writing coordination plans from a global point of view. Then, the compiler generates a decentralised implementation of the coordination logic [31], [32].

Choreography languages handle OTS components by design. Perez De Rosso, Jackson, Archie, Lao, and McNamara III [148] goes one step further and explores how to build a web application by configuring and composing concepts drawn from a given catalogue. However, they do not provide built-in evolution mechanisms. The classical method is to manually update the high-level choreography description and then recompile it to generate a new implementation.

Various works [29], [89], [135] specifies the semantics of choreography languages using various variants of process calculi. However, they provide a limited way to express additional constraints on the behaviours of the embedded components.

Although appealing for their global vision they propose, those languages lack fine-grained control on low-level implementation details that matters for performance. Often, they are not designed to program non-functional properties since they hide resource management logic and since they do not model the non-functional behaviours of independent components. Indeed, they focus on capturing the behaviour, communication patterns, and dependencies among these components while ignoring the internals of the components. As a result, most of the choreography languages are designed to orchestrate existing web components [13], [89], [@105], [135]. To the best of our knowledge, there is no choreography language for system programming.

2.2.5 Summary

Table 2.2 compares the previous approaches (e.g., IDLs, orchestration engines, choreography langauge and interception) according to the requirements we identified in Chapter 1. As expected, all this tools provide a solid support to compose OTS components. They differs on their evolution supports: high-level tools (e.g., distributed

AOP and choreography) ease the updating of the program whereas orchestration engines provide hot-swapping to deploy a new version of the program without down time. Interception helps on system evolution by providing a way to transparently add new features to the program.

All of them increase programmer productivity either by generating part of the intre-connection code (e.g., IDLs and choreography) or by automatising the deployment of the program (e.g., orchestration engines).

Nevertheless, none of these tools satisfactorily meets our requirements in terms of efficiency and dependability. They do not provide a way to specify the behaviour of the systems. Moreover, IDls, orchestration engines and interception are often not aware of the semantics of the components they compose. In the same way, IDLs and interception cannot express and program non-function properties. Although orchestration engines automatically handle scalability or fault tolerance, they go against requirements (Chapter 1) since programers have to declaratively configure it outside the application. Finally, choreography languages are not designed to program non-functional properties since they hide resource management logic.

2.3 Building dependable distributed systems

There are two main and non-excluding approaches to build correct systems. The first one relies on the usage of carefully design constructs provided by the programming language to eliminates whole classes of bugs (Section 2.3.1). The second focuses on validating the system by checking that it follows its specification. First, this requires to formalise the specification of the system (Section 2.3.2). Then, to validate the implementation against its specification (Section 2.3.3). Existing validation approaches can be split into two groups: proof-based and (systematic) testing.The gold standard is to directly generates, or extracts, the implementation from the specification to avoid subtle glitches in between [39], [60].

2.3.1 Dependability in programming languages

Two of the major specific issues with distributed programming is to correctly handle failures and consistency which denotes the absence of contradictions between the (concurrent) observable. To address these issues, some specialised programming languages provide higher abstraction to provide a safer distribution programming. Some works [66], [@119] explores the use of synchronous models to ease fault

	IDLs	Orchestration engines	Interception	Choreography languages
Composition				
Built-in OTS	✓	✓	✓	✓
Composition mechanism	(✓)	✓	✓	✓
Program evolution	✗	✗	✓	✓
Hot swapping	✓	✓	✗	✗
Guarantees				
Isolation	-	✓	✓	✗
Additional specification	limited	✗	✗	limited
Verification	limited	✗	✗	limited
Global vision	(✓)	✗	✗	✓
Efficiency				
Non-functional properties	✗	automatic	✗	(✗)
Expose resources (e.g., place, network)	✗	✗	✗	✗
Runtime optimisations	✗	(✓)	✗	✗
Architecture optimisations	✗	✗	✗	✗
Ergonomy				
Code generation	(✓)	✗	✗	✓

Tab. 2.2.: Comparison of the four main ways to tackle the composition problem according to the requirements we identified in Chapter 1.

tolerance and to simplify the design of distributed systems. Others, like serverless languages or Bykov, Geller, Kliot, Larus, Pandya, and Thelin [30], offer a runtime mechanism that automatically and transparently handles failures.

Various languages provide consistency abstraction to easily program a system with the desired consistency level (e.g., weak, causal or strong). Language-level consistency [12] allows programmers to write the system with additional annotations to materialise the dependencies and the appropriate consistency properties. Then the compiler can automatically analyse the consistency properties of entire applications. Most of those languages rely on type-checking and guards (pre/post conditions) on methods to specify the consistency. Type checking can be sufficient to achieve flow-level consistency [118], [131]. Embedding a flow-level consistency in language implies adding consistency annotation or leveraging an external specification. Object-level consistency provides fine-grain specification of consistency that describes the behaviour of the objects and not just a label in a lattice. Therefore, they rely on guards [132], [164]. Last but not least, Milano and Myers [131] enables to compose safely different OTS storage systems that provide different consistency levels.

The main interest of having high-level programming entities in a language is that programmers cannot anymore make mistakes that are not expressible in the language. However, in the context of programming systems, developers still have to be able to program unsafe things as in Rust [@6] for performance. As we shall discuss later, Varda design is a balance between providings carefully design constructs and the ability to use arbitrary code.

2.3.2 Specification language and formal methods

A specification language describes *what* should be done by a system whereas a programming language expresses *how* the system will perform a task. Existing specification languages for distributed systems fall into three categories: *general-purpose specification language*, *architecture description languages* and, *domain-specific specification languages*[4]. For the latter, we focus on *protocol specification languages* since supporting other domains in Varda is future work. There are additional

[4]By domain-specific we mean that the language is designed to express properties of a specific domain (e.g., security, consistency, etc.).

works that cover consistency and synchronisation specification [100], [139]; or, that models deployment steps of microservices architectures to express both qualitative and quantitative properties related to both safety and efficiency [40], [53].

Note that, without additional tooling (e.g., code generation/extraction or applying theorem prover on the implementation) a specification language manipulates a model decoupled from the system implementation. The following section discusses how to bridge the gap between the specification and the implementation.

General purpose specification language

Most of the time, system programmers use them to model small critical parts of a system since it requires a tremendous effort by programmers [142]. As you can see in Figure 2.1, the models are often very small compared to the size of the systems. Often programmers use them to explore the effects of concurrency or failures in their program. For instance, one common use case is to model and verify the correctness of consensus primitives or synchronisation protocols [38], [142].

TLA+ [111] is one of the most used[5] language by system programmers. It expresses the semantics of systems in terms of Temporal Logic of Action. TLA+ logic specifies system properties over time. It could express complex behaviours, safety and liveness properties, fairness, and temporal ordering constraints. Moreover, it includes a model checker that checks systematically that the desired properties hold on the model. PlusCal [112] provides an imperative programming language that looks like pseudo-code on top of TLA+. PlusCal eases the specification of sequential algorithms since it is closest to the programmer habits [38], [142]. Recent work [11] extends PlusCal to work well on modular and distributed systems. Namely, it provides built-in entities to model communication channels and message passing.

According to Newcombe, Rath, Zhang, Munteanu, Brooker, and Deardeuff [142], using formal specification when building systems is particularly relevant to find subtle bugs and to make aggressive performance optimisations. Indeed, reasoning on a formal model could give enough understanding to get ride of corner checks/-conditions since programmers could achieve strong confidence that execution flow would never enter this case [177].

specification are also important to better shape systems. Programmers can use them to explore the design space on a simplified model before starting implementation.

[5] Based on our interviews and the work of Newcombe, Rath, Zhang, Munteanu, Brooker, and Deardeuff [142].

Applying TLA+ to some of Amazon's more complex systems.

System	Components	Line Count (Excluding Comments)	Benefit
S3	Fault-tolerant, low-level network algorithm	804 PlusCal	Found two bugs, then others in proposed optimizations
	Background redistribution of data	645 PlusCal	Found one bug, then another in the first proposed fix
DynamoDB	Replication and group-membership system	939 TLA+	Found three bugs requiring traces of up to 35 steps
EBS	Volume management	102 PlusCal	Found three bugs
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence though failed to find a liveness bug, as liveness not checked
	Fault-tolerant replication-and-reconfiguration algorithm	318 TLA+	Found one bug and verified an aggressive optimization

Fig. 2.1.: Usage of TLA+ on AWS services, extracted from Newcombe, Rath, Zhang, Munteanu, Brooker, and Deardeuff [142]. To have an idea of the size of the systems, the number of lines of code of an open-source lock manager is about to 170 KLoC (Apache Zookeeper) and databases are one order of magnitude greater.

For instance, Maude [22], [46] explores how to formally prototype distributed systems. A Maude program models a system in terms of rewriting logic. Then the core Maude model checker verifies safety properties. In addition, a probabilistic model checker estimates performance of the design.

Architecture description languages (ADLs)

ADLs follow a head-on approach to build a system by composition [48]. First, programmer models the architecture. Then, they often provide code generation to generate the final implementation [80], [81], [122], [126]. ADLs tend to be independent of the implementation language(s) in order to build systems by assembling pre-existing components and connectors [127].

We call architecture a description of the overall structure of a system [82]. In particular, architectural issues include interaction between components, either the protocols for communication or data access. Architecture also encompasses non-functional structural properties such as scalability. An ADL models the system's architecture using the following building blocks [81]: *components*, *connectors*, and *architectural configurations* (i.e., topology). Most of the ADLs add those building blocks on top of existing semantic theory (e.g., CSP, Petri nets, finite state machines).

A component is a unit of computation, or of data storage. It has a formally defined interface that specifies its possible interactions with the external world in terms of

messages, operations and shared variables. An interface also includes additional constraints on the observable behaviours of a component.

To communicate, two components should be connected through a connector. A connector represents the underlying communication channel. ADLs exposes connectors as first-class entities [58], [151]. Moreover, to provide efficient communication, a developer may reuse external program connectors in the ADL (e.g., shared variables, SQL links, or sockets). Such a connector may not correspond to a compilation unit in the implementation.

In general, an ADL supports a limited form of evolution [48], [127]. A programmer can modify a component interface as long as its type signature remains compatible with the previous one, based on inheritance or subtyping. However, an ADL cannot express an architecture transformation, i.e., changing the topology of a subset of components without updating each component one by one. For instance, to impose access control to a set of components, a programmer has to divert the connectors to the controller and has to create new connectors between the controller and the components.

Protocol specification languages

A communication protocol is a set of rules that governs interaction between agents. It is a formal description of the messages, in what order and under what exchange conditions.

A protocol specification language is a formal language for describing protocols. There are protocol specification languages [44] based on three main formalisms: on execution traces, on information objects, and on session types.

Trace expression Castagna, Dezani-Ciancaglini, and Padovani [34] and Ferrando, Winikoff, Cranefield, Dignum, and Mascardi [76] specify protocols using execution traces, i.e., sequences of messages. For instance, Castagna, Dezani-Ciancaglini, and Padovani [34] give the semantics of each language expression as a set of admissible traces. It provides three expression operators to build protocols: sequential composition (concatenation of traces), choice (union of traces), and shuffle (interleaving of traces).

These languages can express multiparty protocols, i.e., protocols that involves agents with distinct roles: e.g., a buyer, a seller and compliance officer. In a session, i.e., an instance of a protocol, each agent has a role. Then, for each agent, the compiler

projects the protocol to a local perspective, such that the agent is only aware of the part of the protocol that involves it, i.e., messages from it or to it.

Information object Chopra, Singh, *et al.* [45] and Singh [163] declaratively specify protocols using information objects. An information object describes a set of transition rules. A rule specifies the sending of a message between the two roles. Each message is associated with a set of input or output parameters, i.e., a set of named variables. The order of messages is implicit. It derives from the dependencies between the parameters of the rules. For instance, can send a given message type only if it has a binding for the input parameters, i.e., if it knows a value for the given variables. Output parameters mean that the reception of the message binds those output parameters.

In this approach, there is no overall description of a protocol. Rather it describes the perspective of each agent separately, and the protocol emerges from their respective behaviours. Therefore, the projection of a "global" protocol to an agent perspective is trivial. Conversely, this could undermine confidence in the modelling of the protocol. Indeed, programmers could forget to model some interactions between agents since it tangles the representation.

Session types A session type encodes a protocol definition as a type [61]. Moreover, the programming model provides well-typed operations on sessions: sending a message, receiving a message, selecting a labelled branch of a protocol or receiving the chosen branch. The following Chapter 4 details examples of what session types can express.

A session type describes the current type of the message that can be received or sent[6], and the type of the continuation, i.e., the type of the session after receiving or sending this message. Moreover, session types have been extended in various ways to support recursive protocols multiparty [51], [99] broadcast [57], [108], or timing constraints [143].

The type system guarantees [54] that

- the exchanged data has the expected type,
- the structure of session types matches the structure of communication (i.e., the operations on a given session),
- the session channel has the expected structure, and

[6]It can also specify that the session is over.

- the session channel is private, i.e., it is visible only by the communicating parties.

Comparison Table 2.3 compares the different kinds of protocol specification languages. This work is inspired by Chopra, Singh, *et al.* [44]. It explores the following dimensions:

Concurrency Does the language support specifying protocols in which agents may emit and receive messages concurrently?

Extensibility Is the protocol language such that an agent may participate in multiple, potentially unrelated protocols?

Asynchronous Does the language support asynchronous communication over protocol?

Since developers often need to specify additional constraints on messages, we extend these dimensions with the following criteria:

Dependent Does the language support predicates on the value of the message. For instance, can we specify that two subsequent messages in a protocol are increasing for some order?

Time-aware Does the language supports time delivery upper bounds on messages?

Compared to others, session types with extensions could express interesting constraints on the execution of protocols, i.e., time-bounds and constraints on the value of messages. However, a session type protocol cannot model that *distinct* agents should concurrently send (resp. receive) messages. It is not as restrictive as it seems since an agent may abstract multiple running components. Moreover, Kouzapas, Gutkovas, and Gay [108] extends session types with a parallel composition of session. Last but not least, one of the main advantages of session types when designing a new language is that session types are easy to integrate with existing type systems [115], [144], [146].

For those reasons, we use session types as the basis of our protocol specification language in the following part (Part III).

2.3.3 Bridging the gap between specification and code

Having a specification separated from the implementation could ease the introduction of subtle errors when programmers fill the gap in between [39], [60]. There are two main approaches to bridge the gap: certification by proving (part) of the implementation or by extracting the code from a formal proof (e.g., Coq extraction);

	Trace expression	Information object	Session types (with extensions)
Concurrency	No	Yes	(No)
Extensibility	No	Yes	Yes
Asynchronous	Yes	Yes	Yes
Dependent	No	No	Yes
Time-aware	No	No	Yes

Tab. 2.3.: Comparison of protocol specification languages.

or, generating the code from a specification while embedding external (untrusted) code.

Certified distributed system

There are two contrasting approaches to a certified distributed system: *top-down*, by formally writing the system in a high-level proof language (e.g., Coq, F* or Why3) then extracting the machine code from the formal model. For instance, code in Ocaml, Haskell or Scheme can be extracted from Coq or Why3. *Bottom-up*, by certifying an existing implementation in a mainstream programming language. Often, this means importing a model of the system in a proof language and manually proving properties about it.

Top-down Verdi [174] provides building blocks to implement and formally verify distributed systems. Verdi is developed using the Coq proof assistant, and systems are extracted to OCaml for execution. For instance, Verdi's verified system transformers (VSTs) encapsulate common fault tolerance techniques and offer a refinement method to build systems. The developer can verify an application in an idealised fault model. Then, he applies a VST to obtain an equivalent application in a more adversarial environment.

IronFleet [95] stratifies the certification work in three layers following a top-down approach: 1. First, developers formalise the overall system's behaviour (specification layer). 2. Then, they write an abstract distributed protocol layer and prove that the protocol refines the specification layer. 3. Finally, they write an imperative implementation layer and must prove that the implementation refines the protocol layer. The first proof uses a TLA+-style technic whereas the second one leverages Hoar logic reasoning. They successfully apply this approach to a Paxos-based replication library and to a key-value store. The performance of their key-value store

remains competitive, achieving 75% of the peak throughput of Redis, a popular but unverified key-value store.

Certifying a distributed system requires tremendous effort of an order of magnitude greater than the implementation. For instance, Gu, Shao, Chen, Wu, Kim, Sjöberg, and Costanzo [88] provides a proof of functional correctness of a general-purpose concurrent OS kernel. The certifier kernel is 6500 lines of C and x86 assembly. In contrast, the proof script is 100KLoC.

Bottom-up A different path is to certify an existing code base written in a mainstream programming language. The canonical approach is to translate the code in a proof language then to prove its correctness manually. The translation is not always possible, at least it requires that the programming language has a formal and sound semantics. There are various works that aims at importing programs: Appel [15] translates an existing C program into a formal Coq model. Chajed, Tassarotti, Kaashoek, and Zeldovich [35] takes a subset of Go code with a strong semantics and translates it into a Coq model. Moreover, Chajed, Tassarotti, Kaashoek, and Zeldovich [35] provides tools to reason a bout Go concurrency, data structure and file system with faults. For instance, using this tool, Chajed, Tassarotti, Theng, Jung, Kaashoek, and Zeldovich [36] implements a verified concurrent and fault-tolerant journalling system. The ratio between the code and the proof (about 1:19) is similar to the top-down approach ratio (about 1:15).

Building certified distributed systems is a major success. However, these frameworks require substantial effort and are often too complex to be used by non-researchers. Moreover, the certification process is often too monolithic: a huge challenge on complex system is to keep the proof up to date with upates and bug fixes. Note that certification does not prevents bugs that comes from an ill-defined specification.

Partially verified system

Recent specification languages can generate a correct[7] implementation directly from a formal specification [52], [59], [67], [90], [121]. This is the gold standard of

[7] Assuming that the compiler itself is correct.

programming, as it ensures that the system's specification is rigorously checked and that the system's behaviours does not deviate from the specification.

Compared with the previous approach, this approach is much lighter at the cost of less confidence in the system correctness. The amount of effort to write the specification is similar to that of Section 2.3.2. Moreover, the effort involved in "verifying" the desired properties of the model is limited since, most of the time, these languages relieve the programmer from this burden and automate it using model checkers. In addition, the current approach gives programmers more control: they could often import unsafe code snippets and specialise the code generation in various mainstream programming languages according to the system needs. For instance, Lingua Franca [@119] could generate TypeScript implementation for the sub-system running in the web-browser and C++ implementation for the backend running on a data center.

PGo PGo [90] generates a Go implementation from a Modular PlusCal specification and it delegates the verification of the specification to the TLA+ model checker. PGo suffers, as the subsequent top-down approaches, from a lack of control over performance. For instance, a PGo program cannot define placement (Requirement 11) nor the nature of the network links (Requirement 12). Conversely, the PGo compiler generates efficient implementation. However, they do not provide any architecture optimisations (Requirement 13).

Lingua Franca Lingua Franca [@119] focuses cyber-physical systems. The programmer writes a deterministic specification with holes to fill with unsafe code snippets in various target languages (C, C++, Java or Typescript). Then, the compiler generates the implementation, which embeds the arbitrary unsafe snippets. Therefore, Lingua Franca supports OTS natively. However, it does not provide composition primitives to help interconnecting remote components (Requirement 1-3).

The main goal is to prevent accidental non-determinism behaviours. To write non-deterministic code, the programmer must explicitly express it [121]. Forcing determinism implies strong assumptions on network delay, clock synchronisation and a turn-by-turn execution. This conflict with our asynchrony requirement (Requirement 9).

Beyond the guarantees provided by the execution model, LinguaFranca does not offer additional tools to specify and checks a system.

P P [59] unifies modelling and programming of asynchronous code into a single language using state machines. A state machine represents each system component. State machines communicate through events and callbacks. Then the P compiler generates C code from the specification.

Programmers could model the behaviours of a state machine by defining safety and liveness properties based on machine observables. For this they write a monitor, i.e., another state machine. A monitor observes a set of events during the execution of the system and checks that various assertions and predicates hold on the observable events. Moreover, the P compiler could erase monitors during code source generation to increase performance.

Once the programmer has written the specification with monitors, P leverages type-checking then model checking to ensure that the following properties hold. The type checking ensures that the code is wellformed, i.e., that the compiler can erase monitors and that statements are deterministic. Then, the model checker ensures that safety and some liveness properties hold (up to some predefined depth) . Moreover, the model checking pass ensures that each machine has a handler for any kind of events it may receive.

To scale the size of a system that P can check, ModP [60] proposes a compositional verification using modules. A module is a collection of state machines. The idea is to decompose the system-level testing problem into a collection of simpler module-level testing problems. Moreover, ModP offers a composition operator to compose the specification of the modules. However, this ModP operator does not ease the composition of components, i.e., state machine. As in P, developers must write the communication logic manually in the internal code of the state machines (e.g., callbacks and sending messages). Conversely, P supports natively C OTS since it can import arbitrary C building blocks (types, methods) during code generation.

Eventually, P suffers from the same limitations as PGo and Lingua Franca concerning our requirements about efficiency and control. Developers cannot program placement, nor model the network link nor use architecture optimisations to get ride of certain communication and context-switching overheads.

2.4 Summary

Neither API description languages nor orchestration engines can express rich composition semantics that depends on the behaviour of the components. Conversely,

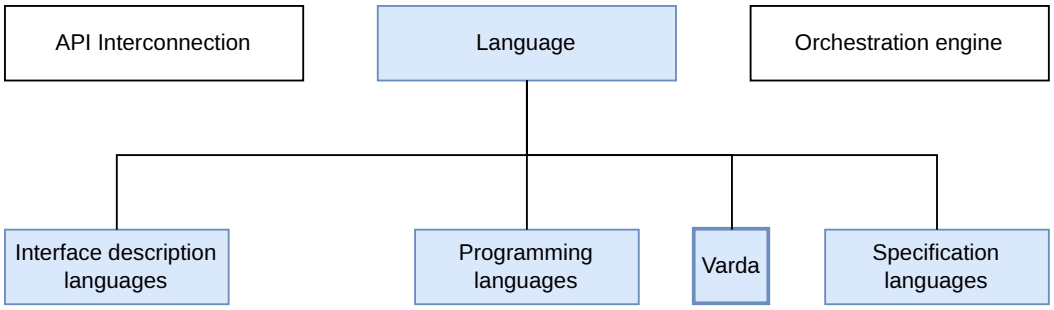


Fig. 2.2.: Overview of prior work and positioning of Varda.

current specification and programming languages for distributed programming restrict programmer control on the system and do not give enough power to express low-level and non-functional properties while keeping the system safe.

We believe that there is a sweet pot between specification languages and programming languages for a high-level architecture language that provides enough control over performance for system programmers while reducing the occurrence of bugs. Figure 2.2 illustrates the positioning of Varda. Next Chapter discusses how we design Varda, our language, to take advantage of this sweet pot to address the requirements of Chapter 1.

Part III

Contributions

3 Design and overview

Like most “new” languages, little about Varda is actually novel. We build Varda upon prior language features. It tackles the composition problem from a new perspective: trying to balance the requirements for system programming by composition with the objective of providing a safe distributed programming model[1]. Its novel contributions include an interception-based programming model to ease composition and performance optimisations to avoid paying the cost of modularity.

Classically Varda is composed of a language - subdivided in one sublanguage for each top-level requirements -, a compiler that checks properties and generates glue code. To support the generated code we provide a runtime and a library for each target. A target denotes the *target* programming language of the code generation. Figure 3.1 presents an overview of the Varda environment.

In Section 3.1, we discuss the design choices of Varda according to the requirements of Chapter 1. Then we illustrate the usage of Varda from the developer’s perspective while discussing the compiler work to build and deploy a system, in Section 3.2. Note that, this chapter focuses on the main concepts of Varda. The details are deferred to Chapters 4-7.

3.1 Design

As explained by Cheung, Crooks, Hellerstein, and Milano [42] language and compiler for distributed programming has four main objectives: (1) to express a program simply;(2) to provide guarantees on the behaviours of a program; (3) to generate code; and (4) to perform optimisations. Accordingly, we structure the following discussion along those four axes. Section 3.1.1 presents the *expressiveness* of Varda. Section 3.1.2 presents the guarantees provided by Varda. Section 3.1.3 discusses how Varda compiler eases the work of the programmer using code generation. Section 3.1.4 shows how Varda it helps improve the *efficiency* of a system.

[1] Chapters 4-7 details our contributions.

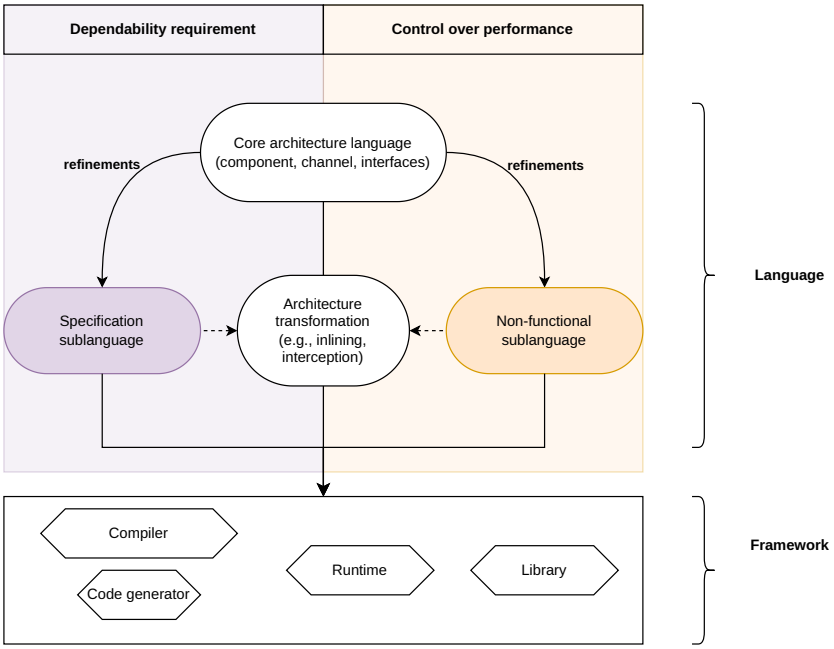


Fig. 3.1.: Varda overview.

3.1.1 Expressiveness

To express properties related to our three main categories of requirements, a Varda program provides a global vision of the system that integrates the following abstraction levels:

The *architecture* describes the topology of the system in terms of components, encapsulation and their interconnection [82]. An architecture is a set of components, which communicates with asynchronous message-passing, with their interconnection and orchestration logic. We extend this core minimalistic language with architecture transformation to *ease composition and evolution*

The *specification sub-language* adds constrains over the architecture in order to explicitly defines the admissible behaviour of the system. With this sublanguage, a programmer may add constraints on communication, interfaces, and components and orchestration behaviours.

The *non-fonctional sub-language* makes programmable non-functional properties that are relevant for performance and fault tolerance (e.g., network, placement). Moreover, this sublanguage exposes annotations to guide optimisations (e.g., component inlining and co-location).

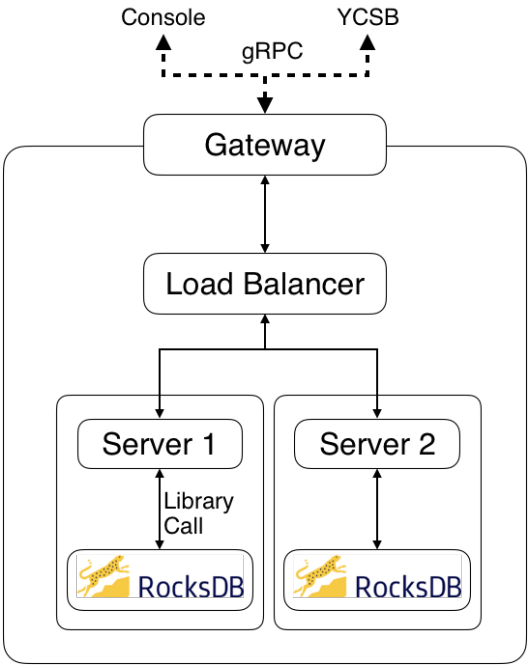


Fig. 3.2.: A storage service with two OTS servers and load balancing

Architecture

We introduce the main concepts of Varda, based on a running example. Suppose that you want to create a distributed storage service by assembling off-the-shelf components. You might instantiate several copies of an existing storage server, for instance RocksDB [@75], behind a load balancer, as illustrated in Figure 3.2. A server executes `get` and `put` invocations. The load balancer is configured to accept the same interface from clients, and to route a `get` to any server, and each `put` to all of them; it awaits the response and sends it back to the caller. We defer the details and the code snippets to the next chapter. The following sections focus on the overall design.

The corresponding Varda program formalises a system *architecture* as a composition of *components*. A component either describes a computation unit or a data store. Each component is an independent execution, asynchronous. As actors, the execution of Varda a component instance is composed of turns. Each turn runs without interruption. On completion of a turn, the component suspends and waits for an external event. On reception of an external event, the suspend component start a new turn and process the event.

The developer can program a component entirely in Varda, which is Turing-complete. Alternatively, it is possible to import, into the Varda architecture, foreign components,

developed independently (and quite possibly in a different programming language), for which we use the term "off-the-shelf (OTS)." An OTS component may, for instance, be a library, an independent process or a REST service. The behaviour of an OTS component may be completely arbitrary, and even buggy.

To embed arbitrary OTS and ensure safety, the programmer isolates an OTS component behind a Varda shield. The shield restricts the component's behaviour by specifying its interface, its protocol (i.e., what it may send or receive, and in what order), and pre- and post-conditions. The developer also provides stub methods, called adaptors, that enable Varda to communicate with the OTS component.

In our example, the developer wraps the OTS component (RocksDB) behind a shield that constrains its API and interaction protocol. For instance, the shield could restrict the storage service to `put` and `get` invocations, passing a strictly increasing counter, and keys structured as the following pattern `"name.ext"`.

To protect separate parts of a program from unwanted interactions, Varda relies on encapsulation to restrict what a component is aware of (e.g., communication channels, identities of other components). Moreover, in Varda encapsulation most often implies communication and failure isolation (see Chapter 6). In Varda, the unit of encapsulation is the component. To provide fine-grain protection, components may be nested[2]. For instance, to be able to update the we group storage backend in a generic component that exposes the `get/put` interface since the Gateway does not need to see the internal behaviours of the store. An outer component orchestrates its inner components, spawning or killing component instances, interconnecting them, and supervising error conditions; it can intercept and manipulate communication, and more generally compute over components and messages.

Communication and interactions Components communicate with one other. To ensure safety, communication passes through a strictly defined interface. This interface is composed of ports. A port drills an explicit hole in the boundary of a component. It ensures the binding between the internal logic and the external communication. A Varda port interface specifies both the syntax of messages, and their *protocol* in the style of session types [61]. A protocol specifies the type and the ordering of messages between communicating parties. For instance, the store protocol specifies that a client opens a session with the storage service, sends any number of get or put messages, receives a response for each request, and finally ends the session.

[2]However, strict encapsulation can get in the way. Accordingly, Varda supports escape mechanisms such as component inlining (Section 5.3) and direct communication channels (Section 4.2.2).

To interconnect two Varda components, the developer binds one active port with one passive port using a network channel. We introduce network channels to model the underlying network in charge of the transmission of messages. We defer the discussion of channels to Section 3.1.1.

Interaction with non-Varda code A distributed system often provide services to other programs. For instance, a messaging application may use our key-value store to share and persist data between different users. Moreover, for interoperability, the external caller (e.g., the messaging application) should not need to be aware of the existence of the Varda framework. Hence, to serve external requests, Varda enables the programmer to expose a generic network interface. Then, the compiler generates RPC stubs (either gRPC or REST). For instance, the `Gateway` exposes a `get/put` interface that checks the wellformdness of the request (e.g., the shape of the key) before delegating the query processing to the backend.

Conversely, a Varda program may receive asynchronous notification from the outside or either from the code an OTS component. For instance, to react to failure, the runtime should notify the crash of a storage node to the load balancer (Section 5.2.1). To receive those notifications, a component exposes a supervision port and a callback for each notification type it can receive.

Architecture evolution To support incremental building of systems, programmers often indirect communication through a proxy. It may help to evolve the system without modifying the other components. In Section 4.2, we use proxy interposition to transform a single-node key-value store into a multi-node store with sharding. Unfortunately, a common practice is to use network-level proxy tricks, but this is quite specific, ad hoc and somewhat awkward. Those tricks are unaware of the semantics of the components they intercept.

To address this common problem, Varda has a generic interception mechanism. A component that encapsulates inner components may impose their communication to redirect messages to an interposition component without any changes to the intercepted components. Interception is completely transparent for the intercepted parties. If the interposer specifies the same protocol and contract as the interposed component, interception guarantees that impersonation is correct.

Specification sub-language

To improve safety, programmers can refine the architecture with additional constrains on communication, interfaces, and components and orchestration behaviours. We choose to integrate these specifications inside the architecture to avoid the classical gap between high-level descriptions and their concrete implementations, discussed in Section 2.3.3.

Mixing correctness requirements with our performance and ergonomic (i.e., acceptability for the developers) requirements, requires a pragmatic approach. For this, we design the specification sublanguage in order to let the programmer balance between performance and checking overhead (i.e., specification expressiveness). For each kind of specification (communication, components and orchestration behaviours), Varda provides various specialised building blocks with increasing expressiveness and overhead. For instance, the programmer can specify the communication between components at different grains from component interface to properties on the observables of a subset of the architecture. Furthermore, to mitigate cognitive overhead, we made most of the constraints optional except for mandatory type annotations since programmers are used to it.

To ensure that a Varda component internally behaves as expected, the programmer can optionally write contracts, i.e., predicates over the arguments, return values of a method and over local variables. A contract pre (resp. post) condition can call any method of the component, any function or any Varda primitives. Moreover, programmers can add contracts to the OTS shield to contain the misbehaviour of OTS components by checking its observable. Since an OTS could be an arbitrary black box Varda cannot express properties about its internal. For instance, using a contract on a storage node, we specify that the value returned by `get(key)` is the one associated with the most recent `put` for the same key (Section 6.1.2).

Non-fonctional sublanguage

To make non-functional properties programmable in Varda, this sub-language embeds resources (e.g., failures, places and network links) as first-class objects in the languages. Moreover, it provides various primitives and annotations to easily program classical distribution properties as fault supervision or elasticity. For instance, we leverage this to extend the `LoadBalancer` such that it spawns a new storage component when a new node join the infrastructure (Section 5.2.2).

Placement impacts the properties of a distributed system. For instance, close-by components tend to fail together, negating the benefits of redundancy and replication. Placement also has security implications, related to trust in the execution environment of a component. Therefore, we integrate placement into the architecture. Varda exposes a *place* abstraction, which summarises information about the network topology. A place represents a specific location, such as a virtual machine, compute nodes, compute clusters, etc. Programmers can refine the architecture with placement annotations in order to colocate components, pin components to a place or specify that two components must run on distinct nodes. For instance, we colocate the gateway and load balancer in the same node to eliminate network overhead.

The nature of the component interconnection matters for various functional (e.g., message order and delivery guarantees) and non-functional properties (e.g., security). For this, Varda offers channels to finely control the network between components. In a Varda architecture, channels interconnect communicating ports. Different classes provide specific guarantees, e.g., a FIFO TCP socket channel, a RabbitMQ persistent channel or a secure TLS channel. We choose to model the network at the channel level since the network is often non-uniform and heterogeneous.

3.1.2 Guarantees

Anytime, anywhere programming with Varda provides a minimal set of guarantees: component isolation and communication safety (i.e., communications are abiding by their protocol).

To avoid unplanned interactions and to constraint failure propagation, Varda guarantees isolation between distinct component instances. Components' state and internal logic are encapsulated from external components, children and OTS. Components can only interact with each other by exchanging messages through channels and ports, and by spawning children. However, isolation guarantees do not take hidden side channels into account since Varda design allows embedding arbitrary OTS. Note that those side-channels can either result from a malfunctioning or a malicious OTS.

To ensure that the sending component provides at least what the receiving side expects, Varda type-checking ensures that the interconnected ports are compatible. Moreover, the type-checking ensures that the following properties hold for protocol-based communication code inside a component [54]: the exchanged data has the

expected type, the structure of communication follows the protocol (e.g., its order), and the communication is private, i.e., a third component cannot see messages except when using the interception mechanism.

The compiler checks that there is no glitch between the architecture, the (additional) specification and the (additional) non-functional behaviours by taking advantage of the integration of those three abstraction layers. Moreover, the code generator ensures that the generated implementation is a faithful and up-to-date representation of the architecture. In addition, the compiler ensures that the architecture transformations (e.g., interception, inlining) are legal and that the resulting architecture preserves the guarantees of Varda and the specification.

Vardac can instrument the glue code, i.e., the shield of components, with dynamic checks. The coverage of checks, and their overhead, depends on the level of detail of the specification.

3.1.3 Ergonomy

To automate common tasks, the Varda compiler generates the interconnection glue code between components. This includes supervising and responding to runtime error conditions, creating and linking sockets, marshalling/unmarshalling language-level data into messages and dynamic checking of safety condition.

For programmers, top-down approaches that generate code may result in a loss of control on the development workflow and in difficulties to track bugs or performance bottlenecks. To address this, we design Vardac to generates human-readable code with additional compilation provenance information. This help programmers to easily blame an architecture piece for a system behaviour.

Last but not least, programmers may specialise the implementation according to their deployment use case without altering the architecture since the logical units (i.e., components) are orthogonal to the physical location (using placement annotation), to the computation units (using inlining), and to compilation units (using code-generation configuration).

3.1.4 Efficiency

To enhance efficiency, we follow a three steps approach. First, programmers can control low-level details and non-functional properties using the non-functional

sublanguage. Moreover, since a component can embed arbitrary code, programmers could leverage specialised and optimised implementation of various parts. For instance, they can import their custom data structures.

Second, to balance between correctness and performance requirements, all the specification annotations are optional, except for the types. We take care to provide simple and specialise Varda specification building blocks such that the programmer choose the trade-offs they want. Moreover, at compile time, programmers can disable the injection of dynamic checks in the implementation to avoid run-time overheads.

Third, the compiler optimises the architecture (e.g., component inlining) and the generated code. To generate an efficient implementation, the compiler performs traditional local optimisations such as ghost elimination, constant-propagation, partial evaluation, unaliasing and dead-code elimination.

Additionally, crossing component isolation boundaries can be costly, due to context switching, marshalling/unmarshalling, and network overhead.

- To mitigate network overhead, programmers can co-locate components.
- To avoid marshalling, shared-memory communication replaces local message-passing at run time, i.e., between components hosted by a same place. This optimisation depends on the implementation of the execution units[3].
- To eliminate context switching, Varda can statically inline components, independently of the nature of the execution units. Inlining merges the implementation of two or more instances into a single component before code generation. Currently, programmers have to annotate the architecture to trigger component inlining.

3.2 Usage

Before diving into the details of the language, the Figure 3.3 presents an overview of the usage of Varda from the developer's perspective while discussing the compiler work to build and deploy a system. Suppose that you want to create a distributed storage service by assembling off-the-shelf components, developer follows the following method:

[3]It works well for processes, threads and actors. Our prototype leverages the Akka actor model. It may be tricky for containers. It should be impossible for VMs.

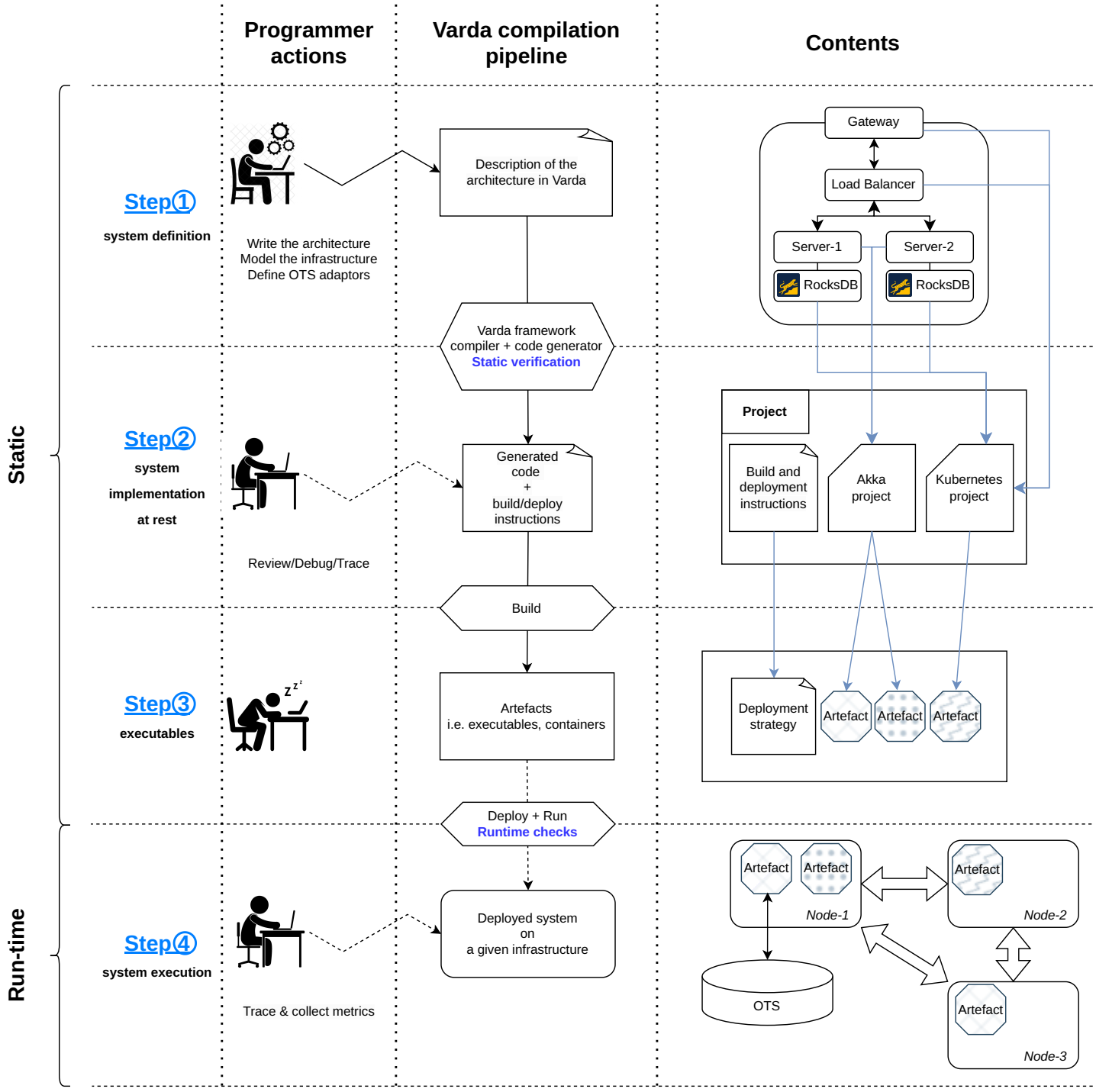


Fig. 3.3.: General overview of Varda usage to build distributed systems.

Step 1 System definition The developer starts by defining the system in Varda as follows

- Describe the architecture of the system in terms of components, interfaces, and interconnections. Incrementally add new system features thanks to interception.
- Define OTS adaptors to bridge the gap between a shield interface (in Varda) and the OTS. An adaptor is a manually written piece of implementation code that maps the OTS API to the shield interface.
- Model the infrastructure, i.e., the kinds of nodes (e.g., servers and VMs) and the group relationship between them. Then annotate the architecture with placement annotations. For instance, to model a multi-region cloud, specify that there are data centers that contain compute nodes in each. Then, place the component on the infrastructure. For instance, the developer can place at exactly one store replica per data center to achieve geo-distribution (Chapter 9).

1 —compiler→ 2 Use the compiler to check the architecture for safety issues, generate the glue code with an optional deployment strategy, and inject dynamic checks. Moreover, the compiler can (optionally) instrument the generated code with basic tracing and metrics functionalities.

The generated code is a set of code projects one per target (e.g., Akka target, Kubernetes target or Typescript target) with additional build instructions to automate the generation of the artefacts. An artefact is a compilation unit. Its nature depends on the target (e.g., Docker images, binaries and libraries, etc.).

The deployment strategy (currently expressed as a DockerCompose file) specifies how to package and deploy the artefacts, and how to start external OTS.

Step 2 System implementation at rest (Optionally) Review, profile and manually improve the generated sources code. The Varda compiler generates readable source code, with provenance information, to ease debugging and profiling.

2 —Build instructions→ 3 Use the build instruction to generate the artefacts.

Step 3 Executables Specify how to group logical units into compilation units, according to the given deployment strategy. Then, use the build instructions to generate the artefacts.

2 —Deployment strategy→ 3 Use the (optional) generated deployment strategy instruction to deploy the artefacts on the infrastructure and to bootstrap the system.

Step 4 System execution Bootstrap the system by starting the deployed artefacts on each node. Each artefact starts an entry point component that is responsible for spawning a subset of the architecture on any nodes enrolled in the runtime. Note that, a new node can be added or removed dynamically.

Trace, collect metrics and monitor the running system either by using the generated tracing and metrics functionalities or by using external tools.

4 Core Varda language

In this chapter, we present the core Varda language. We explain the concepts of the core minimalistic language that describes the architecture in terms of components and interconnections (Section 4.1). Then we extend the language with *interception* to ease the composition and the evolution of a distributed system (Section 4.2). We discuss these concepts based on our key-value store running example introduced in the previous chapter (Figure 3.2).

We use these minimal primitives as base building blocks for all the other complex features of the language. For instance, we define interception as a rewrite mechanism that manipulates only this minimalistic language. Moreover, in the following chapters, we extend them with additional primitives to meet our dependability and our efficiency requirements.

4.1 Varda concepts

4.1.1 Component

Modularising a system into components is a well-known approach to ease the development and the maintenance of a system. In Varda, is the core building block to describe a computation unit or a data store. It interacts with other components by message passing and with the external world through events. Those interactions are formally defined and identified/tracked by its declarative interface. Additionally, programmers can specify the internal logic. Note that the developer must explicitly define the integration between the internal logic and the interface holes to allow a component to communicate.

In Figure 4.1, we zoom on the server component of our running key-value store example. This component is composed of three parts: a declarative interface that defines and constrains the allowed communication with the load balancer; an OTS interface that specifies the interaction with the external RocksDB component; and, the shield logic that handles the request from the Load balancer, performs (optional)

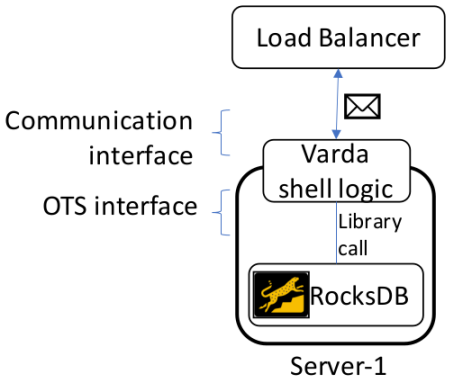


Fig. 4.1.: The three parts of a Server component: the communication interface, the OTS interface and the shield logic.

additional checks and query the RocksDB instance to response to the incoming requests through the OTS interface.

Compared with some other approaches, a Varda's component is a logical unit, it is neither a compilation unit nor a deployment unit. Using inlining (Section 5.3), a programmer can merge multiple components into a single compilation unit. By configuring the code generation targets (Section 7.2), a programmer can also group multiple components into a single deployment unit (e.g., a container, a binary).

Declarative interface of a component

We first examine a component's interface. It specifies how the component communicates with other components in a declarative style. The Varda compiler and run-time enforce the interface specification. Component interfaces type check if they are compatible, i.e., if the sending side provides at least what the receiving side expects, as detailed in Chapter 6.

Message types Abstractly, a component sends and receives messages.[1]

Varda provides classical built-in types (e.g., integer, boolean, list, tuple, dictionary and optional types). Moreover, programmers can define their own type aliases. To simplify the code generation when targeting languages with basic type system, Varda does not provide a rich type system for user-defined types. However, programmers can embed richer types available in the code-generation target language (Section 5.1.4).

[1] Co-location and inlining eliminate message overhead, as we will explain in Section 5.3.

A message type is a classical Varda type such that the type of its payload (i.e., its type parameters are serialisable). For instance, a "`list<int>`" can be used as a message type and "`list<non_serialisable>`" cannot.

In our running example, the following declaration (Figure 1, line 16)

```
type key of string;
type value of string;
type put_request of tuple<integer,key,value>;
```

specifies the type of a "`put_request`" message, whose payload is a tuple of `counter` to represent versions, `key` and `value`.

Protocols Component instances communicate within a *session,* itself an instance of a *protocol* type. A protocol describes a kind of a state machine, in the style of session types [61]. Protocol primitives create a session, send or receive a message within the session, and close the session.

Some session states are branching states, i.e., they have several possible outgoing transitions: each one corresponds to a unique branch label, i.e., a unique message.

In our example, the following declaration (Listing 1 from lines 18–25)

```
(&{ (* choice *)
  l_get: !get_request?value.;
  l_put: !put_request?bool.;
  })*;
```

describes a protocol where the client opens a session with the storage service, sends any number of `get` or `put` messages, and finally ends the session. The "`&`" type constructor indicates a branching state, with branches labelled "`l_get`" or "`l_put`". The get branch "`!get_request?value.`" states that the client starts the communication by sending a "`get_request`". Then, it expects to receive a value.

The Kleene star "`*`" operator states that the protocol can either terminate or iterate any number of times. The Kleene star protocol is just a handy syntactic sugar on top of Varda recursive protocol. Recursive protocol is directly inspired by recursive session types [54]. A recursive session types explicitly define the type of its continuations. Compared to the Kleene star, a recursive session protocol can specify different continuations for each branch of a protocol whereas with the Kleene star implies the same continuation for all the branches.

For instance, the following snippet defined a protocol where the "`Client`"does an unbounded number of "`put_request`"then finishes by doing exactly one "`get_request`".

```
type p_rec = μ x: (* x denotes the session types define as follows *)
    &{ (* choice *)
        l_get: !get_request?value.;
        (* x is the type of the continuation of the put branch *)
        (* i.e., the whole p_rec *)
        l_put: !put_request?bool - x.;
  };
```

The protocol of one party is the dual of the other party, i.e., one's send is the other's receives, and vice versa. We indicate a reversed protocol with the keyword "`dual`", as in "`dual kv_protocol`" (Listing 2, Line 39).

Ports A component has *ports*; a component instance connects to another one by pairing their ports over a *channel*.[2] There are different kinds of ports, notably: An *active* port is the initiator of sessions over its channels; a *passive* port waits to be notified that a session was created by its active counterpart; a *supervision* port receives error conditions (see Section 5.2.1).

The Listing 4 shows the active port of the "`Loadbalancer`" (Line 62) component and the passive port of the "`Server`" component (Line 39). The load balancer has is communication interface split into two sub-interfaces: an interface that describes the allowed communication with the gateway another one that describes the communication with the backend. Both the active and the passive ports are typed by the `p_kv` protocol either directly or through its `dual`.

Ports are typed by admissible protocol. The set of ports' signatures, and the type of the "`onStartup`" method[3], defines the functional signature of a component.

Imperative aspect: inside a component

A component can perform arbitrary computation, as long as it satisfies the declarative interface. Varda provides an imperative Turing-complete language, enriched with primitives related to protocols and non-functional properties, described in Chapter 5.

Classically, in Figure 4.2, a component has methods (i.e., local procedures), and per-instance local variables with the usual types (integer, string, array, tuple, etc.). A method may call another method explicitly. A method may also be invoked by a *callback* when some specific event occurs. In particular, when the component is

[2]The definition of channel is deferred to Section 4.1.2.
[3]This method is invoked on component instantiation.

```
15  component KVStore {
16    type put_request of tuple<integer,key,value>;
17    type get_request of tuple<integer,key>;
18    protocol kv_protocol = (&{ (* choice *)
19        l_get: !get_request?value.; (* send get_request, receive value *)
20        l_put: !put_request{msg -> (* predicates on the value of put_request *)
21            predicate_key(msg.1) &&
22            last_c < msg.0 &&
23            store_meta(last_c,msg.0)
24          }?bool.; (* or send put_request, receive ack *)
25    })*; (* do any number of get or put request *)
26
27    onStartup (){ (* On KVStore bootstraping *)
28      (* Create a FIFO communication channel *)
29      channel<Gateway, Server, kv_protocol> chan = channel();
30      (* Start new component at a given location *)
31      activation_ref<Server> kv_a = spawn Server(chan);
32      (* Start and connect a client *)
33      activation_ref<Gateway> c = spawn Gateway(chan, kv_a);
34    }
35  }
```

Listing 1: Code for the load-balanced storage service (simplified)

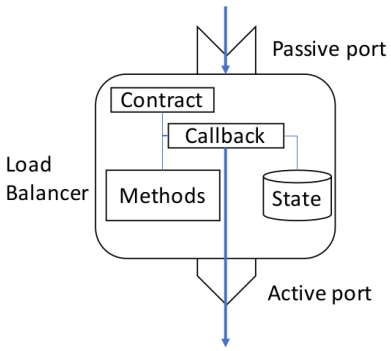


Fig. 4.2.: Inside the load balancer

instantiated, this invokes the method associated with the keyword "`onStartup`". When a passive port suffers the opening of a session, or when a supervision port receives a notification, this invokes the associated callback.

Execution model

The component is the smallest observable grain of concurrency and distribution in Varda[4]. It runs on a given node (e.g., container, virtual machine), specified or not by the programmer (Section 5.1.1). Then the orchestration code manages the component life cycle.

[4]It could encapsulate a multithreaded OTS. However, this will totally hidden from the Varda perspective.

```
component Server {
    (*  Listen for session with type (dual kv_protocol).
        Upon reception, message is handled by [this.kv_callback]. *)
    passiveport p_p expecting (dual kv_protocol) = this.kv_callback;
    onStartup (channel<pk_protocol> chan){ (* On Server creation *)
        bind(this.p_p, chan); (* Dynamically bound the channel [chan] with the [p_p] port *)
    }

    (* Bindings between interaction interface and procedural interface *)
    void kv_callback (blabel msg, kv_protocol s) {
        branch s on msg { (* choice*)
            | "get" => s -> {
                tuple<tuple<key,int>, ?value.> tmp = receive(s); (* async wait for key
                ↪  message *)
                (* return the value bound to the received (key,counter) *)
                (* the backend store expected a key of type string *)
                fire(tmp.1, get(tmp.0.0 +":"+str(tmp.0.1)));
            }
            | "put" => s -> { ... }
        }}

    (* Procedural interface - will bound to an OTS adaptor (Section 5.1.4) *)
    value get(string k);
    bool put(string k, string v);
}

component Gateway {
    activeport p_a expecting kv_protocol;
    activation_ref<Server> kv;
    onStartup (channel<kv_protocol> chan, activation_ref<Server> kv){(* On [Gateway] creation
    ↪  *)
        bind(this.p_a, chan); (* Dynamically bound the channel [chan] with the [p_a] port *)
        this.kv = kv;
    }

    (*  [api_put] is exposed as a gRPC (or a REST) interface to external client
        Therefore, it can onyl takes as arguments and returns primitive types (e.g. int)*)
    @exposed result<bool, error> api_put(string key, int counter, string value){
        (* Creation of a session with [this.kv] through the activeport [p_a] *)
        session<kv_protocol> s = initiate_session_with(this.p_a, this.kv);
        (* Select the [put] branch of the protocol*)
        !put_request?bool. s = select(s, l_put)?;
        (* Craft and send the request then wait for the response *)
        ?bool. s = fire(s, put_request(counter, key, v))?;
        tuple<bool, .> res = receive(s);
        return ok(res.0); (* Return the received boolean *)
    }

    @exposed result<string, error> api_put(string key, int counter){...}
}
```

Listing 2: Code for the load-balanced storage service (simplified)

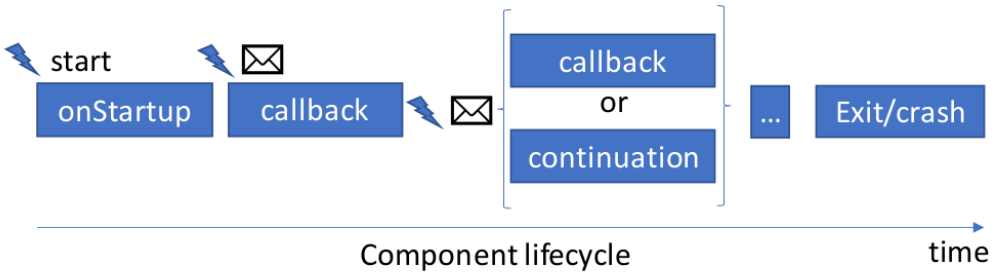


Fig. 4.3.: Varda turns: upon instantiation, a component bootstraps by running the "`onStartup`" method. Then, it expects requests from external components (i.e., session establishment). On notification of session creation, i.e., first message, it runs the callback bound to the receptionpassive port. At the end of the callback or on an asynchronous receive primitive on a session, the component suspend and returns to the waiting state. At this point, the component can be resuming either by receiving a new message on a passive (resp. supervision port); or, by receiving a message of an asynchronous receive which will trigger the execution of the continuation of the receive expression.

A component responds to messages or events (e.g., timers, errors) it receives by making a local decision, creating other components, sending messages and updating its local state. This is inspired by the actor model [97]. A component instance may arbitrarily modify its own private state, but cannot affect the state of another component instance with Varda primitives. However, a component can alter stealthily the state of another one through their unsafe OTS code. For instance, two OTS components can use the same database entry to share information. In this case, Varda it is not aware of these dependencies cannot guarantee any isolation.

A component alternates between two states, like actors [55]: ready to accept a message or busy processing a message or an event. Figure 4.3 shows the life cycle of components. A *turn* is the processing of a single message (resp. event) by a component instance until completion. The component executes the message (resp. event) callback until completion or until the execution flow reaches an explicit asynchronous "`receive`" primitive (resp. "`branch`").

Mixing the explicit asynchronous "`receive`" primitive (resp. `branch`) of the session types paradigm with the callbacks of ports may seem odd at first glance. It permits to explicitly control all the initialisation of communication thanks to ports, and, at the same times to program linearly the processing of a session which greatly mitigate the callback hell. For the sake of availability, the processing of two distinct sessions can be interleaved. On an asynchronous "`receive`" (resp. `branch`), the suspension only affect the current session (i.e., the one passed in argument to the "`receive`"

primitive). While the session processing is suspended, messages of other sessions (or external events) can be processed.

Orchestration

Orchestration code has to do with managing the life cycle of a component instance and its interactions. As explained above, method callbacks are associated with component startup, passive session creation, supervision events, and message receives. Conversely, a component can actively *spawn* an instance (Listing 1, Line 31), connect a component's active port with another's passive port, create a session (Listing 2, Line 73), or send a message (Listing 2, Line 77).

When a component is in a branching state, it may actively *choose* the next state by sending an appropriate message. Conversely, it may passively suffer the next state transition by receiving a message.[5] The "`Gateway`"(Listing 2, Line 75) selects the put branch, then it can send the "`put_request`"and waits for a `value`. Conversely, the "`Server`" wait for the choice (Listing 2, Lines 46–54) using a form of pattern matching on the labels. When the server receives the "`put`" label, it enters the corresponding branch (Listing 2, Lines 47–51). In this branch, the session "`s`" has the type of the "`put`" branch of the protocol, i.e., expects to send a "`put_request`" and, then, to receive an "`ack`".

Orchestration code may also use the so-called non-functional directives. These do not change the semantics, but influence run-time properties that may be important for performance or fault tolerance. For instance, the code may spawn a component at (or away from) some specific "place" (a virtual machine or container, a compute node, or a cluster of compute nodes, for instance) or class of places (e.g., "a node located in Europe"). Conversely, there are discovery primitives for querying the system topology. We discuss placement and discovery later, in Section 5.2.

4.1.2 Communication

To make two component instances communicate, the programmer specifies the underlying network in charge of the transmission of messages using channels. A *channel* is the abstraction of a network connection. Then, the imperative shield manages the communications through sessions that flows over the previously defined channels (Figure 4.4).

[5] In the vocabulary of session types, active choice is called "select," and suffered choice is called "branch."

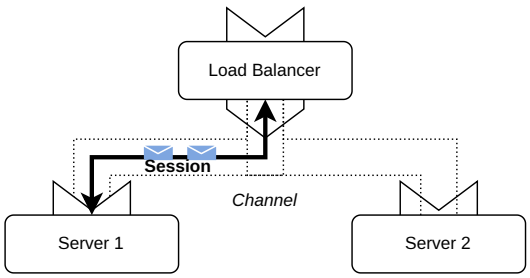


Fig. 4.4.: Channel "chan" ensures the network interconnection between the "Loadbalancer" and the Servers. Messages between both groups are flowing through "chan" and are logically grouped into sessions.

A channel has two endpoints, an initiator charged with actively initiating sessions, and *listeners* that passively await session creation. The syntax "`channel<A, B, some_protocol>`" denotes a channel that connects initiator ports of type "A" with listener ports of type "B", under protocol "`some_protocol`". The Section 5.1.2 details the system and network aspects of channels.

To interconnect two components with a channel, the developer binds one active port and one passive port to the selected channel. Varda provides a port binding primitive : "`bind(this.port, some_channels);`". Each component is responsible for binding its ports with channels.

For instance, the Listing 1 shows the interconnection between the "Gateway" and the "Server" using the FIFO channel "chan" (Line 31). The "Gateway" instance binds its active port with "chan" (Line 65), received as a "onStartup" parameter. Then (Line 73) , it starts a session over the previously bounded channel and begins its communication.

Being interconnected by the same channel enables communication. However it does not make components exchange messages. The internal logic of the interconnected components manages the communications. Conversely, the absence of a channel between components disallows any direct interaction between them. More specifically, they must both declare type-compatible channel types, and share a channel instance at run-time.

Vardac ensures that the protocol of the port and of the channel are compatible[6]. By transitivity, it ensures that the interconnected ports are compatible. Moreover, it also ensures that the orchestration logic that interacts with the port (either the callback at reception or the expression emitting a message) is well typed according to the port protocol.

[6]We discuss compatibility in Section 6.1.1.

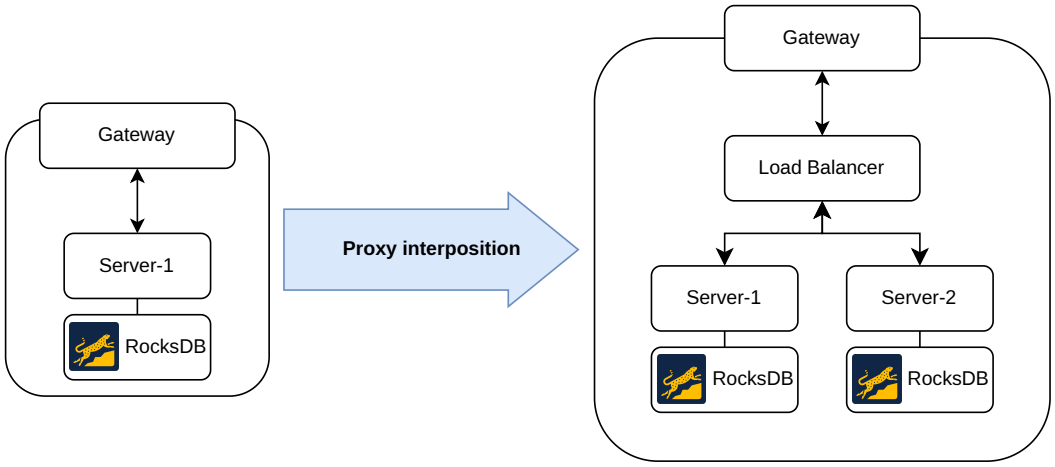


Fig. 4.5.: Interposing the "`LoadBalancer`" in between an existing "`Gateway`"-"`Server`" architecture.

4.2 Interception

It is often useful to indirect communication through a *proxy* [160]. For instance, in our running example, the load balancer serves as a proxy for the backend storage servers (Figure 4.5). A proxy might filter, redirect, and/or modify messages. The purpose may be to mask internal interfaces, for instance, hiding internal servers behind a single external one; to sanitise messages; to add provenance or authorisation information; to improve performance, for instance, by interposing a cache. It may help to evolve the system, for instance, replacing a key-value storage server with a file server.

A proxy, between some client and some service, is semantically transparent if it *impersonates* the client for the service, and the service for the client. Unfortunately, enforcing indirection through the proxy is often technically non-transparent, requiring changes to one or the other of the endpoints. A common example is setting up a web proxy or a service worker, where it is the responsibility of the browser to redirect traffic. Alternatively, a common practice is to use network-level DNS tricks, but this is quite specific, ad hoc and somewhat awkward. Neither mechanism ensures any correctness guarantees.

4.2.1 Interception mechanism

To address this common problem, Varda has a generic *interception* mechanism. A component that encapsulates inner components may impose their communication

to redirect to an interposition component, without any changes to the intercepted components. Interception is completely transparent for the intercepted parties. If the interposer specifies the same protocol and contract as the interposed component, these guarantees that impersonation is correct; furthermore, the interposer may constrain the interposed one, by using a restricted sub-protocol (a subtype of the original protocol) and/or a stronger contract (see Section 6.1.1).

Interception depends only on the interface of the intercepted components. Therefore interception remains transparent, and orthogonal to non-functional properties such as placement or inlining.

Interception redirects the communication between components to a proxy, the *interposition* component. The proxy may be inlined to avoid any networking overhead (Section 5.3). The intercepted component is a first-class piece of Varda architecture. The interception logic is responsible for processing (alteration, delaying and forwarding) session establishments and messages between internal and external activations.

Programmers can control each stage of the life cycle of an intercepted session: session creation, sending/receiving of messages and branch selection. Varda provides a method decorator for each stage. "`@sessioninterceptor`" methods are triggered when a session is established, conversely "`@msginterceptor`" methods are triggered when a message (or a branch label) crosses the interception border.

The steps to automate interception are as follows. 1. Code the interceptor according to the specific interception requirements. 2. Enclose the intercepted components inside an interception scope with the "`intercept<Interceptor> { stmts }`" block statement.

Let's apply this to the storage service example. To simplify exposition, we write the load balancer purely in Varda instead of using an OTS balancer such as HAproxy.

The code, illustrated in Listing 4, specifies an interception scope (Lines 155–158) containing two "`Server`" instances. The interceptor, "`LoadBalancer`", implements the load-balancing algorithm (Line 84–149).

The "`LoadBalancer`" establishes a session between the interceptor and a "`Server`" once the client has provided the key, which is required to select the right "`Server`" using a user-defined function `pick` (Lines 137–147).

Varda uses the signature of "`intercept_get`", i.e., the type of the arguments, to select the correct communication. Here, the session of type "`!get_requet?bool.`".

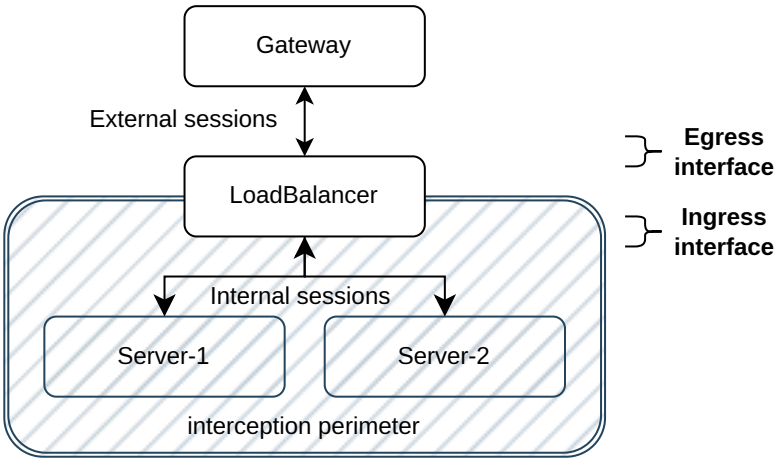


Fig. 4.6.: The interception mechanism applies to add a load balancer in between a `Gateway` and a `KVServer`

For compactness, the "`LoadBalancer`" does not intercept the branch selection (`l_get`, `l_put`).

Delaying messages can be tricky, since arbitrary long delay between messages of the same session could be triggering a timeout depending on the session implementation. And, a fortiori, dropping messages of a running session leads to a timeout on the other side of the session. Dropping session establishment is correct.

An interceptor has three interfaces (See. Figure 4.6). The ingress interface handles communication between the interceptor and the external components. The egress interface handles the communication between the interceptor and the intercepted components. The onboarding interface handles onboarding request, i.e., communication between the parent(s) and interceptor(s).

Programmers can write "`@messageinterceptor`" logic (resp. "`@sessioninterceptor`") specialised for messages received on the egress interface, on the ingress interface or, indifferently, on both. Varda provides an optional interface parameter for "`messageinterceptor`" (resp. `sessioninterceptor`) which can be either `ingress`, "`egress`" or "`both`", which is the default value.

```
component LoadBalancer {
    list<activation_ref<Server>> replicas = []; (* List of managed replicas *)
    (*  Mapping between ingress session and egress sessions
        when broadcasting the put order to all replicas *)
    dict<session_id, !key?bool.> s_inner_put = dict();

    (************** Onboarding **************)
    (*  @param [a] is the activation to onboard
        @param [p_of_a] is the place (i.e., the node) on which [a] has been spawned
    *)
    @onboard([Server])
    bool onboard_A(activation_ref<Server> a, place p_of_a){
        (* initialize the list of replicas managed by this load balancer instances *)
        append(this.replicas, a);

        (* [is_safe_node] is a predicate that returns
           - [true] if the place belongs to the internal network
           - [false] otherwise *)
        return is_safe_node(p_of_a);
    }

    (************** Session interception **************)
    @sessioninterceptor(true, both)
    result<option<activation_ref<KVServer>>,error> p_kv_interceptor(
        activation_ref<Client> from, string requested_to_schema,
        blabel msg (* triggered when receiving the first message *)
    ){
        return ok(none()); (* Do nothing, wait for key *)
    }

    (************** Msg interception **************)
    @msginterceptor(both)
    result<!key?value.,error> intercept_branch(
        (dual p_kv) s_client, p_kv s_replica,
        blabel msg
    ){
        branch s_client on msg {
            | l_get => s -> { (* nothing to do, suspend and wait for the key *) }
            | l_put => s -> { (* create one session per replica then suspend and wait *)
                for(activation_ref<Server> replica in this.replicas){
                    session<p_kv> s = initiate_session_with(this.custom_p_out, replica);
                    !key?bool. s_a = select(s, l_put)?;
                    add2dict(this.s_inner_put, sessionid(s_client), s_a);
                }
            }
        }

        !key?value. s_out = select(s_replica, msg)?;
        return ok(s_out);
    }

    (* Intercept all the get_request *)
    @msginterceptor result<.,error> intercept_get(
        ?bool. s_client, p_kv s_replica, get_request m){
        (* [onboarded] denotes the set of all intercepted components *)
        activation_ref<Server> dest = pick(this.onboarded, m);

        (* Select the get branch of the protocol *)
        p_kv s = initiate_session_with(dest);
        !get_request?value. s_a = select(s, l_get)?;

        (* Propagate the put request *)
        fire(s, m)?;
    }

}
```

Listing 3: Code of the "LoadBalancer" written as an interceptor (simplified syntax)

```
150 (* Extending the simple KVStore with load balancing *)
151 component KVStore {
152     onStartup (){
153         (* All the instances spawned inside the [intercept] scope
154         are intercepted by the LoadBalancer *)
155         intercept<LoadBalancer> {
156             activation_ref<Server> kv = spawn Server(chan);
157             activation_ref<Server> kv = spawn Server(chan);
158         }
159
160         (* Outside the intercept scope, the [kv] variable is the
161         identity of the interceptor *)
162         activation_ref<Gateway> c = spawn Gateway(chan, kv);
163     }
164 }
```

Listing 4: Intercepting "`Server`" for load balancing (simplified syntax)

4.2.2 Properties and guarantees

Transparency

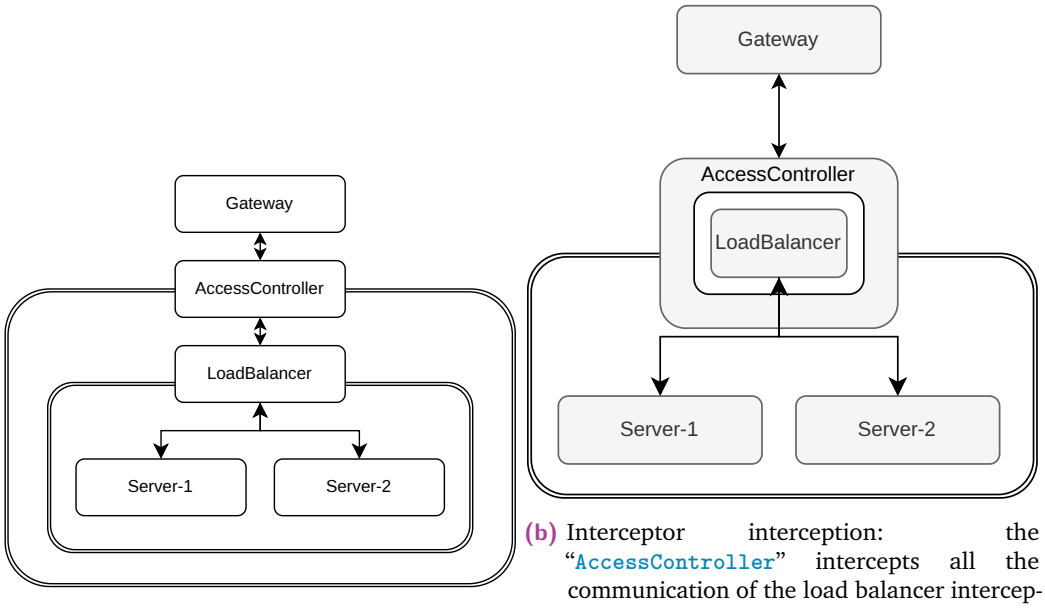
Interception is transparent for both the intercepted component instances and the external ones. Both are unaware that the communication is intercepted. Therefore, their code does not need to be altered to support interception.

To achieve this, Vardac guarantees, after rewriting that the interceptor *egress interface* is compatible with each interface of the intercepted components. The *ingress interface* is compatible with each interface of the external components that may communicate with an intercepted component.

Composition

Intercept blocks can be nested, like any other statements. For instance, the following snippet shows how to add access control on top of the load balancer by enclosing the intercept block of Listing 4 by another intercept block parametrised by a user-defined "`AccessController`" (Figure 4.7a). The Section 8.3.1 details how to build access control using interception.

Programmers can also intercept all the communications of an interceptor, i.e., both the external components (as in Figure 4.7a) and the internal ones. This behaviour is useful to implement an access control mechanism that does not trust the clients nor the servers. For instance, in the Figure 4.7b, the `AccessController` intercepts all the communication between the `Gateway` and the `LoadBalancer`; and, all the communication between the `LoadBalancer` and the `Servers`.



(a) Nested composition: the outer `AccessControl` intercept block intercepts all the communication between the `Gateway` and the inner `LoadBalancer` intercept block.

(b) Interceptor interception: the "`AccessController`" intercepts all the communication of the load balancer interceptor. Compared with the figure on the left, in this setup the "`AccessController`" also intercepts the communication between the load balancer and the shards.

Fig. 4.7.: Interception composition

Interception isolation

An intercepted component cannot communicate directly (i.e., send and receive Varda messages through session) with an external component. Moreover, interception does not provide communication isolation between two external components, and, neither between two intercepted components if they share the same interceptor instance.

Interception isolation could either be *bypassed from below* by using non-Varda communication primitive or *breached by above* by establishing a side channel after interception.

Bypassing from below is intrinsic to Varda design. The programmers can embed any kind of arbitrary and unsafe code behind a shield. For instance, two components unsafe code can create unseen side channels by using raw sockets, filesystem-based communication or a third-party service (e.g., a shared database). Preventing such side channels exceeds the scope of Varda: it implies to strictly contains arbitrary code (e.g., network, syscalls and file system).

Breaching by above is programmable in plain Varda. Conversely, programmers can filter it out. Breaching interception enable communication optimisation by removing

unnecessary redirection. A programmer can breach the isolation by establishing a direct side channel between an external and an internal instance. This requires that one of the intercepted protocols supports exchanging channels (recall that channels are first-class value) and that both the external and internal instances agree to use this channel to communicate.

Conversely, developers can prevent the *breaching from above* either by ensuring that intercepted protocols cannot carry channel value; or, by writing a dedicated "`@msginterceptor`" in charge of changing the nature of the side channel.

4.2.3 Interception workflow

Figure 4.8 shows the workflow of the interception of a "`Server`" by a "`LoadBalancer`". The programmer writes the interceptor (①) and defines the interception context (②). Then, Vardac prepares the system architecture (③–④) at compile time, i.e., it generates the interception boilerplate composed of new ports, channels, states and communication logic.

At run-time, while executing the "`intercept`" block, Varda runtime performs[7] the interception setup (⑤–⑦), i.e., it starts the interception logic, it onboards intercepted instances and it interconnects the intercepted components with the interceptor.

Once the interception is set up, the intercepted components can communicate with the external world[8] through their interceptors (⑧.a–⑧.e details the workflow of a "`get_request`").

Static interception workflow (See. Figure 4.8)

① The programmer codes the "`LoadBalancer`" logic (Listing 3). In particular, the developer specifies the interception logic using decorated methods: "`@sessioninterceptor`" and "`@msginterceptor`".

[7] The compiler cannot perform the interception statically since the exact perimeter of the interception depends on the execution path of the "`intercept`" block. For instance, a given shard component could be spawned outside the interception perimeter to model an additional metadata server. Moreover, the interception perimeter can contain a variable number of shards.

[8] The communication initiator can be either an intercepted or an external component, depending on the exposed ports.

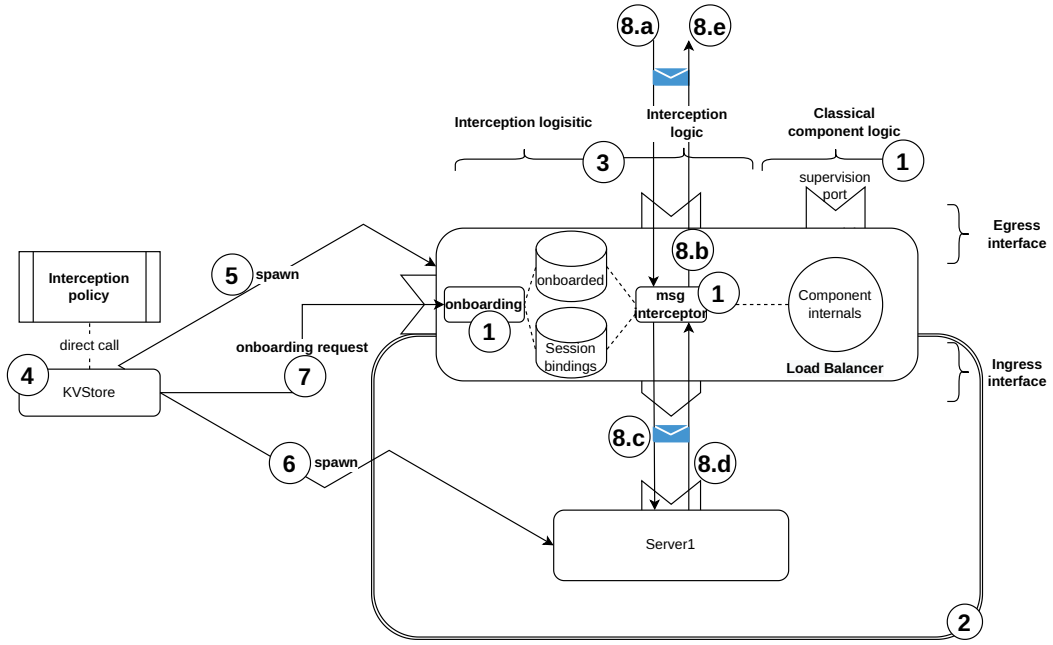


Fig. 4.8.: Intercepting a "`Server`" with a "`Load Balancer`". The compiler performs steps 1-4. The generated code is responsible for steps 5-8.

This user-defined interception logic can coexist with non-interception logic written in classical Varda. For instance, the "`LoadBalancer`" may expose a supervision port to monitor the addition of a new place to achieve elasticity[9].

(2) The programmer defines the interception scope which defines the static perimeter of the interception (Listing 4, Lines 155–158). Note that the "`intercept`" block is defined inside the orchestration code of "`KVStore`". Note that the "`KVStore`" instance is the *parent* of the intercepted instances and of the interceptors.

(3) From (1) and (2) Vardac generates the logistics for the interceptor (3). The compiler adds one interceptor's port for each port of each intercepted component[10]. Then, Vardac binds these intercepted ports with their user-defined interception logic based on the signature of the decorated methods ((1)). The behaviour of this binding is detailed at the end of this section.

Furthermore, Vardac adds some helper local states to manage the intercepted communication : the "`onboarded`" set of intercepted component instances and

[9]The details of supervision are deferred to Section 5.2.2.
[10]Multiple instances of the same component connected to the same channel share the same interceptor ports.

the "session_bindings" map, which translates and external sessions[11] into and internal ones.

(4) Vardac rewrites the "intercept" block[12] ((2)) as follows:

- The compiler generates the logic in charge of selecting (or spawning) the interceptor for the block, based on the *interception policy*.
- Then, it adds the code in charge of triggering the onboarding procedure at each spawn of an intercepted component. For this, it adds an active port to the "KVStore" and binds it to the "onboarding port" of the interceptor.
- It correctly binds the ports of each intercepted component with those of the ingress interface (generated during (3)). Conversely, it binds the intercepted bridges to the ports of the ingress interface.
- Outside the intercept block, it updates the identity of all the intercepted instances with the interceptor one, without altering their type thanks to subtyping[13].

Dynamic setup This stage begins when the parent of the intercepted components (here the "KVStore" instance) starts executing the "intercept" block. Vardac performs the *dynamic setup* (i.e., steps (5)–(7)) at each spawn of an intercepted component. The *dynamic setup* of a new intercepted instance can run in parallel with the *interception of communication* of already onboarded instances.

(5) When the execution flow enters the "intercept" block, the "KVStore" spawns an interceptor (here a "LoadBalancer" instance) using the interception_policy[14].

(6) When the execution flow reaches the following spawn expression

```
server1 = spawn Server(...);
```

inside the "intercept" block, the "KVStore" spawns a "server-1" such that all its ports are bound to the interceptor ones. The interceptor blocks all the communication of "server-1" until its onboarding.

[11] An external session denotes a session between a component external to the intercept block and the interceptor. Conversely, an internal session denotes a session between the interceptor and an intercepted instance.
[12] It is part of the parent instance (e.g. KVStore) orchestration logic.
[13] The interceptor egress interface is a superset of the union of the interfaces of the intercepted components.
[14] In practice, Varda provides a more expressive workflow where the "interception_policy" is used to assign an interceptor, possibly distinct, per intercepted components.

(7) Then, the "KVStore" instance sends an onboarding request to the "LoadBalancer". The onboarding allows the interceptor to distinguish intercepted component instances from external ones.

On acceptation, the interceptor registers the intercepted activation identity into its local "onboarded" set. The "LoadBalancer" notifies the "KVStore" with the acceptation (or the rejection) of the onboarding[15].

Intercepting communication This stage begins when the onboarding of the first intercepted component instance is over and successful. The following describes the route of a "get_request" between the "gateway" and the "server-1".

(8.a) The "gateway" sends a "get_request" to what it thinks to be the server, i.e., the "kv" variable (Listing 4 at line 158). Outside the interception scope, the "kv" points toward the "LoadBalancer".

(8.b) The "LoadBalancer" receives the "get_request" and run the corresponding "sessioninterceptor" followed by the "msginterceptor" logic, both defined in (1), to decide what is next.

- The "@sessioninterceptor" is triggered upon reception of a session creation notification (Listing 3, Lines 107-112), i.e., when the interceptor receives the first message of the session. Here, it is the branch label.

  On success, it triggers the "intercept_branch". Its logic stores the newly created external session in the interceptor store and terminates. The creation of the internal session is delayed until the reception of the key, managed by a "intercept_get", for a "get_request". Since the "LoadBalancer" chooses the server according to the key.

- Each message reception triggers its related "msginterceptor" callback. Upon the reception of a "get_request", the interceptor executes the "intercept_get" method. It selects a server among the intercepted replicas and creates an internal session.

(8.c) The method "intercept_get" initiates a session with the intercepted "Server" and sends the "get_request", unmodified. The interceptor stores the pair of the internal session (i.e., between the interceptor and the intercepted components) and the external session (i.e., between the interceptor and the

[15] Currently, the interception mechanism does not support handlers. A promising approach, inspired from exception handling, would be to add an "except cases" to the "intercept" block in order to handle onboarding failure (resp. rejection).

external component) inside its local state session_bindings. This binding is mandatory to be able to correctly route the subsequent messages exchanged on both internal and external sessions[16].

8.d The "`server`" processes the request without noticing the interception. It sends back its "`get_response`" to the "`LoadBalancer`".

8.e The response is processed by either the continuation[17] of the "`intercept_get`" logic, by another user-defined "`@msginterceptor`" specifically written protocol step or by the default message interceptor (generated by Vardac).

The default logic propagates the messages using the internal session (resp. external). The interceptor computes the destination session thanks to the `session_bindings`.

4.2.4 Expressing the interception (details)

Controlling the dynamic perimeter of the interception with onboarding

The onboarding allows the interceptor to distinguish intercepted component instances from external ones. This process is triggered each time a new intercepted component start.

Interception scope defines *statically* an over-approximation of the perimeter of the interception, whereas, *onboarding* defines the run-time (and dynamically evolving) perimeter of each interceptor instance.

The onboarding is programmable. A developer can specialise the onboarding process to initialise some local state of the interceptor or to reject intercepted component instances based on run-time information (e.g., accept only instances that know a given security token or instances that run on safe nodes). To do so, a programmer defines a method decorated by `@onboard`. Furthermore, the programmer can define multiple onboarding functions, each one managing a disjoint set of component types using `@onboard[component_types]`.

[16] For the put request, the interceptor binds the external session with a set of internal sessions, one per intercepted `Server`.
[17] If there is an asynchronous "`receive`" inside (resp. `branch`).

```
166 (*Scope 1)
167 intercept<Interceptor, modifiers> interception_policy {
168     (*Scope 2*)
169     (* stmts  such that *)
170     kv = spawn Component(...);
171     (* is intercepted *)
172 }
173 (* Scope 3*)
```

Listing 5: General interception block.

For instance, Listing 4 defines an onboarding function (Lines 95–103) that register new intercepted `Server`, i.e. "`Server`" spawned inside the "`intercept`" code block, into the "`LoadBalancer`" local state. Moreover, this function reject onboarding request of untrusted instances, i.e., running outside some internal network.

The interception block

The "`intercept`" block syntactically defines the frontier between intercepted component instances and external ones: components spawned inside the "`intercept`" scope are intercepted, the others are not.

To make the interception fully transparent, the interception scope exposes its binders contrary to the classical syntactic scopes. The Listing 5 presents the different syntactic scopes introduced by an "`intercept`" block. The variables bounded in "`scope 2`" are bound in `scope 3`.

The scope exposition semantics depends on the type of the variable: non-component and non-channel variables are exposed as is; component instances and channels must be processed with special care not to break interception. For instance, in Listing 4, the "`kv`" variable is bound inside the intercept block (Lines 155–158) and is used (Line 162) outside the interception context as an argument for `Gateway`. The semantic of "`kv`" (Listing 4) differs according to the scope. Inside the `intercept` scope it points either on "`server-1`" or "`server-2`" according to the execution flow. Outside the `intercept` scope, it denotes the interceptor.

Exposing component identity There are two different use cases: for load-balancing, the "`Gateway`" does not need to (and should not) known the number of the "`Server`" nor should not be able to distinguish their identity. Whereas to achieve access control with interception (see Listing 22), the intercepted activation identity must be exposed since sending a request to "`server-1`" differs from sending a request to "`server-2`" even if they share the same component type.

Varda interception provides additional mechanisms (Listing 5), called)modifiers, to control component identity exposition. The "anonymous" modifier erases the identity of intercepted instances whereas its absence exposes the identity intercepted instances. To avoid leaking internal information, and possibly breaching interception isolation, the exposed identity is aliases of the real ones.

Exposing channels Contrary to components, channels defined inside an "intercept" block are not exposed to avoid creating unseen side channels. Those internal channels are dedicated to non-intercepted communication between intercepted components.

Channels defined before the block (i.e., in the scope 1) and used inside the intercepted scope are intercepted. Vardac creates a copy of those channels. The original channel interconnects an external components (either defined in "scope-1" or in scope-3) with the interceptor. The copy interconnects the interceptor with an intercepted component (defined in scope-2). Furthermore, it can interconnect two intercepted components.

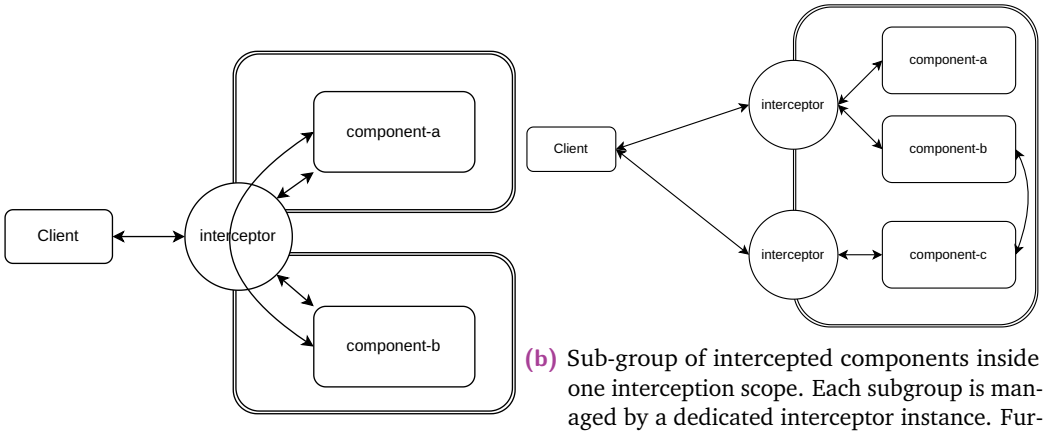
User-defined interception policy

Programmer can control the choice of the interceptor instance in charge of an "intercept" block or even, at finer grain, the programmer can control the choice of the interceptor instance in charge of a given intercepted activation. To achieve this, the "intercept" statement can be parametrised by a user-defined function called the *interception policy*[18].

For a given intercepted component, this policy computes (resp. spawns) the interceptor instance in charge of it. It can either create a new instance while customising its placement and the value of its parameters, or lookup some state to select an existing one.

The interception policy can specify complex relationship between *intercepted instances* and *interceptor instances*. This range from specifying multiple interceptors for a given interception scope to define exactly one interceptor for multiple context. Whatever the relationship, to avoid non-determinism, Varda guarantees that there is exactly one interceptor in charge of a given intercepted activation.

[18] Neither the interception logic nor the interception scope can express *how and where* interceptors are spawned and *what* is the relation between intercepted activation and interceptor activation (e.g., one to one or many to one).



(a) Reusing the same interceptor between two distinct contexts

(b) Sub-group of intercepted components inside one interception scope. Each subgroup is managed by a dedicated interceptor instance. Furthermore, the "`interceptor-2`"is collocated with "`component-c`"to increase performance.

Fig. 4.9.: Using the interception policy.

To achieve this flexible behaviour, Varda calls the given interception policy before each "`spawn`" of an intercepted activation. It computes (resp. creates) the interceptor instance in charge of this specific instance based on the type and the place of the intercepted components. It cannot use the identity of the instance yet since the interceptor assignation must precede the spawn of the instance not to lose any message send by the "`onStartup`" method on the intercepted component[19].

One interceptor multiple contexts One can reuse the same interceptor between interception scopes to create logical interception scope that aggregates scattered interception scope. Such a logical interception is not constrained by syntactic scope boundaries limitations. Figure 4.9a illustrates this example. Listing 6 shows how to write a `singleton_policy` that reuse the same singleton interceptor instance (Listing 6, lines 189-190) for all the "`intercept`" blocks parametrised by the `singleton_policy`.

One context multiple interceptors Conversely, one can spawn multiple interceptors for a given interception scope to improve performances by replicating the interception logic (Figure 4.9b). In that case, the policy assigns an interceptor instance for a (dynamic) set of intercepted instances. The perimeter of those sets is controlled by either the place or the type of the intercepted instances. Filter groups based on policy parameters, i.e., place and type of the intercepted activation.

[19] One possible Varda extension could be to provide access to the spawn arguments in the policy function.

```
174 (** Singleton policy logics **)
175
176 option<activation_ref<LoadBalancer> singleton_interceptor = none();
177
178 (*
179     [intercepted_component_schema] denotes the schema of the intercepted component
180     [p_of_intercepted] denotes its place
181     [factory] is the function that instantiate a new interceptor, the compiler provides it
182     Moreover, a policy takes the parameters of the onStartup method of the interceptor, if
    ↪  any.
183 *)
184 activation_ref<LoadBalancer> singleton_policy(
185     place -> activation_ref<Server> factory,
186     string intercepted_component_schema,
187     place p_of_intercepted
188 ){
189     if(this.singleton_interceptor == none()){
190         this.singleton_interceptor = some(factory(current_place()));
191     }
192
193     return option_get(this.singleton_interceptor);
194 }
```

Listing 6: Interception policy for S-KV

For instance, for stateless interceptors, an interesting strategy could be to initiate one interceptor replica per node where an intercepted component runs. We provide an implementation of such a `replication_policy` in Listing 7.

Of course, programmers can hybridise and tangle the two approaches to control precisely the interception.

General notes on interception implementation

Binding user-defined interceptor logic with generated ports Vardac *specializes the interceptor component* (3) for each interception block, in order to create the needed ports according to the intercepted ports and bridges. Vardac binds the annotated methods with generated ports of the interceptor based on methods signature which includes and the intercepted session type and the current message type. Note that "`@interceptsession`" is triggered when the first message of the session reaches the interceptor.

For instance, in Listing 4, Vardac binds "`intercept_get`" (Lines 137–147) with the generated interceptor's port expecting a "`get_request`" since the signature of

```
195 dict<place, activation_ref<KVServer>> interceptors = dict();
196
197 (*
198     [intercepted_component_schema] denotes the schema of the intercepted component
199     [p_of_intercepted] denotes its place
200     [factory] is the function that instantiate a new interceptor, the compiler provides it
201     Moreover, a policy takes the parameters of the onStartup method of the interceptor, if
    ↪  any.
202 *)
203 activation_ref<SomeInterceptor> node_replication_policy(
204     place -> activation_ref<Server> factory,
205     string intercepted_component_schema,
206     place p_of_intercepted
207 ){
208     if(exist2dict(this.singleton_interceptor, p_of_intercepted)){
209         return get2dict(this.singleton_interceptor, p_of_intercepted);
210     } else {
211         activation_ref<SomeInterceptor> interceptor = factory(p_of_intercepted);
212         add2dict(this.singleton_interceptor, p_of_intercepted, interceptor);
213         return interceptor;
214     }
215 }
```

Listing 7: Replication policy. It instantiates at most one interceptor per node.

"`intercept_get`" specifies that it is responsible for the messages of type "`get_request`" with a continuation of type "`?bool.`".

Interceptor logic uses method annotations and not classical component definition to reduce the programming efforts and to remain as general as possible. In this manner, programmers do not have to take care of creating the communication interface of the interceptor which depends on the nature of the intercepted components. Moreover, an interceptor component can be re-used for various "`intercept`" block that may contain different set of intercepted components. Vardac generates all this boilerplate.

As a side effect, two intercepted channels with the same protocol and the "same" left (resp. right) hand-side component type are intercepted by the same methods. However, inside the interception logic, programmers can distinguish between both, by accessing the unique identifier of the channel instance.

4.3 Sumup

We propose a new language, Varda, at the intersection between programming and specification languages. A Varda program describes how to compose components into a coherent architecture. To ensure safety, the programmer isolates an OTS component behind a Varda shield. The shield restricts the component's behaviour by specifying its interface, its protocol (i.e., what it may send or receive, and in what order), and pre- and post-conditions. Components can be logically nested, to provide encapsulation. An outer component orchestrates its inner components, spawning or killing component instances, interconnecting them, and supervising error conditions; and more generally compute over components and messages.

We extend this core language with an interception mechanism to support incremental building of systems. Varda interception is a safe language abstraction of proxy interposition. A component that encapsulates inner components may impose their communication in order to redirect messages to a proxy component without any changes to the intercepted components.

5 Support for systems programming

Supporting the development of real, distributed systems, has conflicting requirements: on the one hand, to reduce complexity and opportunities for errors, by abstracting away low-level detail; on the other, to pay attention to features that affect performance and fault tolerance. It is a trade-off: parallelism is good for performance but bad for complexity; co-locating components is good for communication cost, but bad for parallelism, resource contention, and fault tolerance (since faults are not decoupled); some system algorithms require low-level but unsafe code, etc. Clearly, there is no right answer; Varda follows pragmatic approach: it lets the developer choose the trade-off for its specific use case.

This chapter explores what are the entities that a programmer can leverage to have low-level control over the system and its performance. First, we explore the representation of the network and placement in Varda and how they integrate with components. Then, we review how to combine the existing building blocks to program classical distribution features as scalability and fault supervision. Finally, we introduce the component inlining optimisation to compile away expensive inter-process communication. In particular, it eliminates the extra overhead induced by the usage interception.

5.1 Primitive features for system programming

5.1.1 Placement

Placement is an important trade-off in a distributed system. On the one hand, placing together two components that communicate may reduce network overhead; on the other, placing too many components together may overload the underlying node (e.g., VM or server). Furthermore, close-by components tend to fail together, negating the benefits of redundancy and replication. Placement also has security implications, related to trust in the execution environment of a component.

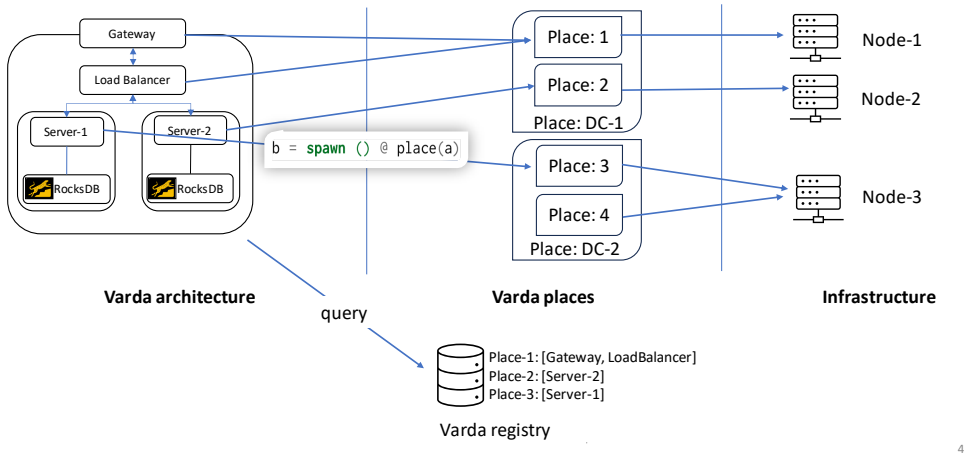


Fig. 5.1.: Placement of the load balance key-value store on a three node infrastructure.

To program placement as a first-class feature, Varda embeds an abstract representation of the infrastructure: the places. For this, Varda provides a two-step flexible placement mechanism, illustrated in Figure 5.1.

The first step, the left-hand step on the figure, corresponds to the placement of the component instance on a logical representation of the infrastructure. The architecture specifies this step. Furthermore, places are Varda logical values, which can be manipulated by the program. However, they cannot be created or destroyed by the core Varda's programming model. The runtime manages their life and death.

The second step, the right-hand step in the figure, corresponds to the mapping of the logical infrastructure to the physical infrastructure. This mapping is external to the architecture and is defined thanks to an external configuration file and deployment phase.

Step 1: Component placement

Varda supports a *place* abstraction, which summarises information about the network topology. A place represents a specific location, such as a virtual machine, compute nodes, compute clusters, etc.

A Varda component may specify, imperatively, where to place a new instantiation of a component, near to, or away from, some specified node, or specified components. To this effect, the directive "`spawn`" instantiates a component with an optional "`place`" parameter. A component instance is placed when it is instantiated and does not move thereafter. Dynamic migration is not supported directly, but can be manually programmed above, by spawning a new instance, synchronising the state between the old and new instance.[1] As an example, writing "`b = spawn () @ place(a);`" co-locates instance b with a.

To ease the programming of placement, Varda runtime provides a placement and discovery service that answers questions such as: What places currently exist? What is the place of this specific instance? What instances are running at this given place? This service maintains the mapping between abstract places and physical nodes (set up during deployment, as specified in a configuration file), and the locations of component instances.

To monitor the evolution of the infrastructure, a component can subscribe to notification triggered by this service by exposing a supervision port. For instance, a component can be notified when the set of places grows or shrinks dynamically, by subscribing the supervision port to events "`new_place`" and "`del_place`".

Step 2: Mapping places with the infrastructure

The mapping of places to infrastructure is inherently dynamic. The running infrastructure changes for each deployment and evolves dynamically (e.g., node failure, commissioning, decommissioning or network partition). Therefore, it is the Varda runtime that onboard places in the registry.

Even if the infrastructure is dynamic, the placement of distributed components is often related to a static infrastructure schema. For instance, a geo-distributed key-value store expects to have nodes grouped in several DCs. Each DC should host its own frontend, a replica of the backend store and an inter-DC replication strategy.

[1] To make migration transparent, the idiomatic way would be to use interception to redirect communication to the new instance.

In this situation, the hierachisation of the infrastructure per DC is known in advance, even if the number and composition of the DCs remain dependent on deployment.

To models this static part, Varda uses an *infrastructure schema* which represents the architecture a priori knowledge of the infrastructure. This schema models the hierarchy and ownership of places. Hence it is composed of place schemas. A place schema represents a group of places, for instance a DC. Moreover, to model the hierarchical infrastructure, place schemas can be nested.

The infrastructure schema is immutable whereas actual infrastructure (composed of places) is dynamic.

To set up this infrastructure schema, a programmer defines the schema in a separate file, which describes the name of each place schema, their properties and, optionally, their children. In addition, the developer could optionally specify additional properties (as a list of arbitrary key-value tuples) for each place schema. For instance, programmers can set a secure flag when a node runs in a trusted network.

To bind the infrastructure schema with the running infrastructure, the programmer should bind each place with a place schema during place onboarding. For this, the easy way is to tag the generated deployment strategy with the names of place schemas.

Finally, in the architecture, the programmer can leverage the knowledge of the infrastructure schema to select places based on their tags or on their properties. For instance, in the geo-distributed key-value store, the orchestration logic can distinguish between DCs and spawn a replica in each. In Chapter 9, we use this feature to build a clone of AntidoteDB.

Discussion

Performance considerations Mapping places with the infrastructure is done during deployment, it does not induce runtime overhead. On the other hand, placement reflexivity is intrinsically dynamic, since the infrastructure may evolve and components' placement changes. Therefore, placement reflexivity induces an extra runtime cost when an instance is created or deleted and when a place joins or leaves. This overhead comes from the need to register the modification into a shared registry. To

avoid those runtime-cost, programmers can disable placement reflexivity : globally (Vardac parameters) or per component, by annotating the component schema with `@disablePlacementReflexivity`.

Deployment Once, Vardac has generated and build the glue, programmers package the glue code for their deployment tool. First, they need to deploy the underlying Varda runtime on each node. During this phase, they onboard the node and tag it. This exposes the node as a place. Programmers (or failures) can remove (or add) places at runtime with the same mechanism. Then, they start the system entry points. Each entry point start spawning activations on places.

To ease this step, Varda can generate a deployment plan. The programmer just needs to write the docker-compose file template. Then Varda can automatically generate a Dockerfile for each target. To use Kubernetes instead of docker compose, the developer has to replace the docker compose template file with a Kubernetes one.

5.1.2 Networking

A channel is an abstract object supporting message-passing between ports and encapsulating low-level OS or network primitives. In contrast to many high-level systems, a channel is a first-class, programmable object. As explained earlier, channels are subject to type checking.

A channel is a serialisable type, i.e., a message can send a channel to another component. This enables establishing a direct connection between components, bypassing encapsulation and interception boundaries. For instance, when a client first connects to a service, its interceptor might return a direct connection to some internal server for the service, minimising indirection overhead. This does not break safety, because it must be allowed both by the message type and by the channel class.

Channel classes are implemented in a trusted library. Different classes provide specific guarantees, e.g., a TCP socket channel, or a RabbitMQ persistent channel. Additionally, we provide channels wrappers that alter the non-functional properties of the channel, i.e., it does not change the communication pattern. A channel wrapper is a function that takes a channel as input and returns a channel as output. Wrappers can be chained. For instance, a channel wrapper can add encryption to a channel as follows:

Channel class	Guarantees	Comments
`default_channel`	FIFO per sender-receiver tuple	Support a *many-to-many* pattern, which means that it can interconnect multiple initiator components with multiple listener components.
`amqp_channel`	FIFO per sender + at most one delivery	many-to-many + AMQP is the mainstream protocol for brokers. Our prototype uses RabbitMQ under the hood.
Channel wrappers		
`aes_wrapper`	the properties of the wrapped channel + AES encryption	We assume that the key are known by all the tenants of such a channel.

Tab. 5.1.: Currently available Varda channel classes. Guarantees cover all the messages exchanged on the channel even if they belongs to distinct sessions.

```
channel<A, B, my_protocol> chan = amqp_channel(my_protocol, "amqp://localhost:5672",
↪  "my_topic");
channel<A, B, my_protocol> secure_chan = aes_wrapper(chan);
```

The management and distribution of shared secrets are out of the scope of the "`aes_wrapper`" and of this work. We assume that there is an external mechanism that provides the key to the wrapper. For instance, we can embed a HashiCorp Vault[2] client in a Varda component and use it to distribute secrets. HashiCorp Vault provides a secure and centralised solution for storing, accessing, and distributing secrets such as API keys, passwords, encryption keys, and certificates.

As in ADLs languages, we dissociate channels from components and make channel internals opaque. Conceptually, channels (like places) are the reprensentatives of the lower layers of the OSI model. This permits to provide taylored and efficient implementation of channels whitout extra overhead. This separation is needed to finely control the network interconnection between components since the network is often non-uniform and heterogeneous at the scale of an infrastructure. Moreover, this give the ability to the programmer to use high-level system to propagate messages between components (e.g., Kafka, RabbitMQ, etc.).

[2]https://www.vaultproject.io/

```
218 for(activation_ref<_> b in rightactivations(c)?){
219     if(some_predicate(b)) {
220         session<st> s0 = initiate_session_with(b);
221         (* starts communication *)
222     }
223 }
```

Listing 8: Discovering the activations listening on channel c

5.1.3 Instance discovery

Recall that in Varda an activation, i.e., a component instance, a can start a communication with activation b only if a knows the identity of b, i.e., the "`activation_ref`". Conversely, b does not need to know a before receiving the first message. Activation a knows the identity of b if is the parent of b. Otherwise, a needs to discover the identity of b before starting a communication. a can gain this knowledge from its parent, by receiving it thanks to its `onStartup` arguments. Otherwise, a can discover it by receiving a message containing b with another activation c.

Manually propagating the activation identities can be time-consuming and error-prone. Each Varda channel offers reflexivity on its endpoints. Figure 8 shows how to get the activations listening on channel c. Channel-based activation discovery only work if both activations a and b have a port bound with c, and therefore knows c. Varda placement reflexivity primitives give the ability to gain access to activation running on given place (Section 5.1.1). For performance or security, programmers can choose to make activations, of given component, invisible for reflexivity primitives.

Channel discovery behaves like the activation discovery, except that an activation cannot discover a channel thanks to place-based discovery. Therefore, channel discovery mechanism works only if activations a and b share a common ancestor. If they don't, two different entry points have spawned them[3]. In this case, channels cannot be shared without prior common knowledge. For instance, a traditional common knowledge is the tuple of an IP address and a port number. Varda represents this common knowledge using static channels. As its name suggests, a static channel is fully statically determined. Thus, it can be defined at top-level outside any component and pre-exists the system. Note that, a static channel requires that its arguments, if any, are known at compile time and that its instantiation is deterministic. All of the current channels kind can be used statically.

[3] A Varda system can have multiple "main functions", i.e., entry points. Each entry point spawns a different part of the system.

5.1.4 Interaction with non-Varda system

Exposition

To serve external requests, a system built using Varda must expose an interface that can be queried using mainstream technologies. Such that the external caller does not need to be aware of the existence of the Varda framework. For instance, in Figure 3.2, a YCSB client[4] (resp. a console client) can fire requests to the gateway using its REST benchmarking plugin.

To ease this step, Varda enables the programmer to expose a generic network interface. This requires to open up the corresponding encapsulation boundary. Varda provides keyword "`exposed`." It instructs the compiler to generate RPC stubs (either gRPC or REST). The stubs do some minimal type-checking, and include run-time conformance checks that enforce the protocol and contract.

In our running example, the "`KVStore`"component both encapsulates the storage service (see Lines 15–34). Its "`Gateway`"component exposes the method `api_put` as a gRPC interface (Lines 71–80). Different clients, e.g., a console or a YCSB driver, can address this interface.

OTS adaptor

To wrap an OTS module inside a component often requires an unsafe language, for instance, Java. Varda structures this into three parts: safe component code, unsafe *adaptor*, and the OTS module itself. The component declares the interface of methods that call into the OTS module, but not their implementation, which is delegated to the adaptor[5].

As the OTS module can be completely arbitrary, this requires flexibility in adaptors. For instance, if the OTS is a Java library, it requires Java code to use Java calling conventions; if it is a web service, it wants to receive and send REST messages. Typically, the adaptor is written as short Varda snippets embedding unsafe code, in Java or some other appropriate language. The Varda compiler inlines the adaptor code, generates glue code, does type-checking, and generate (optional) run-time conformance checks.

[4]YCSB is mainstream a benchmarking tool for databases [50].
[5]This is inspired from the P language [59].

```
224 impl component Server {
225     impl raw java:{=
226         /* Use a unique DB connection per Server */
227         RocksDB _db;
228         RocksDB getDB (){
229         if(this._db == null){
230             // Open the backend
231             String db_path = System.getenv("ROCKSDB_PATH");
232             org.rocksdb.Options options = new org.rocksdb.Options();
233             this._db = RocksDB.open(options, db_path);
234         }
235         return this._db;
236         }
237     =}
238
239     (* binding for the get method *)
240     impl method get java:{=
241         // Perform the GET request on key [k]
242         // {{% Varda expression %}} is compile to Java code
243         byte[] value = this.getDB().get({{%k%}}.getBytes());
244         return new String(value);
245     =}
246
247     impl method put java:{= ... =}
248     }
249 }
```

Listing 9: A "`Server`"to RocksDB adaptor

By design, adaptor code is kept in a separate file. This makes it easy to substitute an OTS module with another. For instance, with suitable adaptor files, our storage service supports multiple backends, an in-memory table, Unix files, or an advanced storage engine such as RocksDB [@75].

Adaptor syntax is as follows. Notation "`{= ... =}`"embeds a code snippet written in the unsafe programming language. The compiler interpolates templates containing Varda expressions, signalled by "`{{% ... %}}`".

In our example, Figure 9 illustrates a Java adaptor for "`Server`". The "`get`"method is a wrapper around the RocksDB Java library (Lines 240–245). It uses opaque Java (Lines 225–237), and reuses a single RocksDB connection for all the requests to a given Server. Our compiler translates "`{{%key%}}`" (Line 243), where "`key`" is an Varda variable, into the corresponding Java code.

Programers can embed existing data structures with their helping primitives. For this, the developer has to embed the data type as an abstract type and the primitives as abstract functions, with their code-generation adaptors. For instance, we embed a CRDT LastWriterWin register when we build a geo-distributed datastore (see Chapter 9).

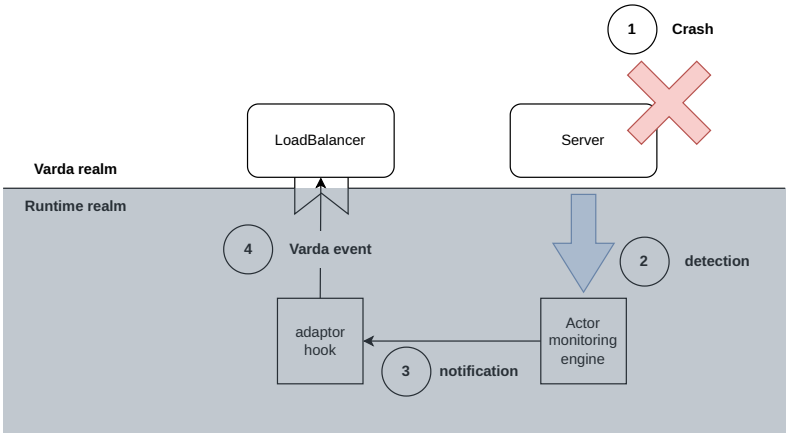


Fig. 5.2.: Supervision of a storage server by leveraging the existing supervision mechanism of the underlying actor runtime.

This mechanism avoid to complefixify the language and, at the same time, to rely and reuse existing libraries that implements interesting data structures.

5.2 Building features for distributed systems

This section shows how to combine the available foundational elements to attain traditional distributed system capabilities, such as scalability and fault monitoring. This demonstrates that the Varda expressiveness is sufficient to program those functionalities and that it can easily leverage the existing implementation provided by the underlying runtime or by the infrastructure management system.

5.2.1 Supervision

Varda embeds existing tools provided by the underlying runtime (e.g., Akka supervision) in its programming model, thanks to OTS adaptors and supervision ports, Whereas other environments, such as Erlang [16], provide built-in crash supervision. Indeed, reusing tools has two major advantages: leveraging performant and well-tested tools, and controlling underlying OSI layers that would otherwise be abstracted.

```
250 component LoadBalancer {
251     (* Supervision port expecting a [child_failed] event *)
252     supervisionport child_failed =
253       x : child_failed -> print("Backend failure");
254
255     onStartup (){
256       (* The activation to suppervise *)
257       activation_ref<A> a = ...;
258       (* Start the supervision of [a] *)
259       watch(a);
260     }
261 }
```

Listing 10: Supervision of a storage server.

Figure 5.2 shows how to leverage an existing supervision mechanism to built component supervision. In this example, the load balancer supervises the backend servers it manages. We built this example on top of an actor model: the component is compiled toward actors and the actor runtime already provides an actor supervision mechanism.

A component may subscribe to events regarding its subcomponents, for instance, an indication that a sub-component is not responsive. These events are received on the component's supervision port. With this information, the developer can, for instance, recover from a failed instance.

The runtime environment can be set up to deliver error events to a component's supervision port.

Listing 10 shows how this works. Component "`LoadBalancer`" supervises storage server a. Supervision starts with "`watch(a);`". If a fails, the runtime triggers a signal of type "`child_death`" on the supervision port "`p_monitor`", the "`LoadBalancer`."

The approach works as is for other fault-detection mechanism and for non-actor components as long as the authors can provide an adaptor for the `watch` primitive and a runtime hook to transform the notification of the fault-detection into a `child_failed` event.

5.2.2 Elasticity

Elasticity is the capability of a system to adapt dynamically to the load and to the available resources, by provisioning and deprovisioning [96]. In Varda the developer

```
262 component KVStore {
263     (* suppervision port listening for a [new_place] event,
264     sent by the runtime *)
265     supervisionport new_place =
266     (* On reception trigger the [add_server] callback *)
267     e : new_place -> this.add_server(e.place);
268
269     (* Spawn a server on each new place *)
270     void add_server(place p){
271     (* Add the newly created server to the the
272     replicas pool of the loadbalancer *)
273     intercept<LoadBalancer> {
274         spawn Server(chan) @ p;
275     }
276     }
277 }
```

Listing 11: Adding elasticity to Listing 4.

controls computation needs by spawning and terminating component instances. Moreover it can discover the available resources (e.g., additional places) thanks to the discovery service.

Listing 11 shows how to add elasticity to the example storage service. When "`ElasticManager`" receives a "`new_place`" notification (Line 265), spawns a new storage server using the "`add_server`" method. The interceptor "`LoadBalancer`"receives a notification when the new server has started, and adds it to its intercepted pool (Line 273).

Last but not least, if the underlying runtime or infrastructure management system provides elasticity/scalability features, the developer can embed them in the Varda realm by following the same pattern as for the previous section.

5.3 Component inlining

Crossing component isolation boundaries can be costly, due to context switching, marshalling/unmarshalling, and network overhead. Co-location avoids part of this overhead. Furthermore, Varda can *inline* co-located components, to eliminate context switching and marshalling overhead. Inlining merges the implementation of two or more instances into a single process or virtual machine, depending on the compilation target of a component.

To take a simple example, consider our key-value store with load balancing: the gateway listen to external gRPC requests, validate them and forward them to the load

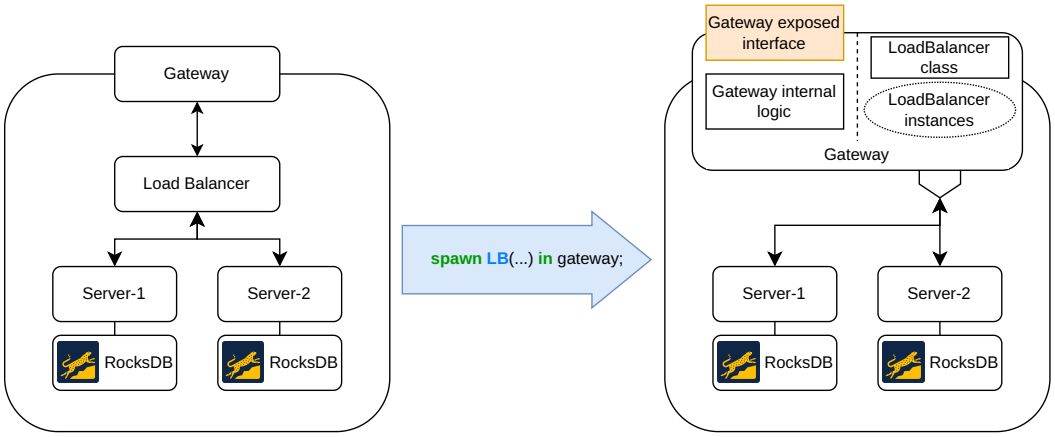


Fig. 5.3.: Inlining the load balancer (LB) into the gateway.

balancer. An interesting optimisation is to eliminate the message overhead between the gateway and the load balancer by inlining the loadable into the gateway[6].

Figure 5.3 illustrates the transformation. After inlining, the gateway acts as a component host for the loadblancing logic. The load balancer component is encoded in a "single host/sequential" object such that the gateway can directly call its methods. We use object representation to keep data encapsulation: a component could host multiple inlined instances of the same components.

Inlining occurs on request, at run time (in contrast with previous work [64], [65] where it is automatic and at compile time). The execution of "`lb = spawn LoadBalancer(...) in gateway`" triggers inlining of lb into $gateway$. Vardac cannot do it statically, since the identity of s is not known in advance.

5.3.1 Properties and guarantees

Using Varda inlining

Induces a lost of parallelism Inlining unavoidably affects liveness, as it results in a loss of parallelism between the host and the inlined components.

Affects isolation between the host and the inlined components Inlining relaxes the isolation between the host and the inlined component: the compiler preserves OTS and communication isolation. However, if the host crashes, the

[6] Inling the gateway in the load balancer is conceptuallyu possible. However, our prototype does not support it: Vardac does yet handle exposed interface when inlining.

inlined components crashes as well. The reverse is not true, an inlined component can fail without crashing the host.

Note that, the host and the inlined component can communicate if and only if they could communicate without inlining: i.e., they are interconnected by a channel.

It also leads to an unavoidable host identity leakage since external components needs to send messages to the host to reach the inlined component.

Suffers from channel limitation A component cannot host two instances of the same component type that uses two different channels for the same-named port. This comes from the fact that the Varda programming model does not support to dynamically add new ports to a component. On the other hand, having static ports greatly simplifies the reasoning on a Varda architecture and the code generation.

Preserves other Varda guarantees The other safety guarantees are preserved: components inlining preserves interfaces and encapsulation of the components. Moreover, it inlining mixes well with all the building blocks, provided by Varda, to strengthen system dependability (see Chapter 6).

Generates a correct-by-construction architecture

Our prototype does not yet support nested inlining.

5.3.2 Inlining workflow

In this section, we present a generic inlining workflow which is fully transparent for code-generation since it compiles down to core Varda. One could specilize the inlining compilation for a specific code-generation target to leverage existing underlying inlining mechanism.

Figure 5.4 shows the workflow of the inlining of a "`LoadBalancer`" into a "`Gateway`". The programmer writes the different components (1) and specifies that the load balancer should be inlined into the gateway instance using the spawn instruction.

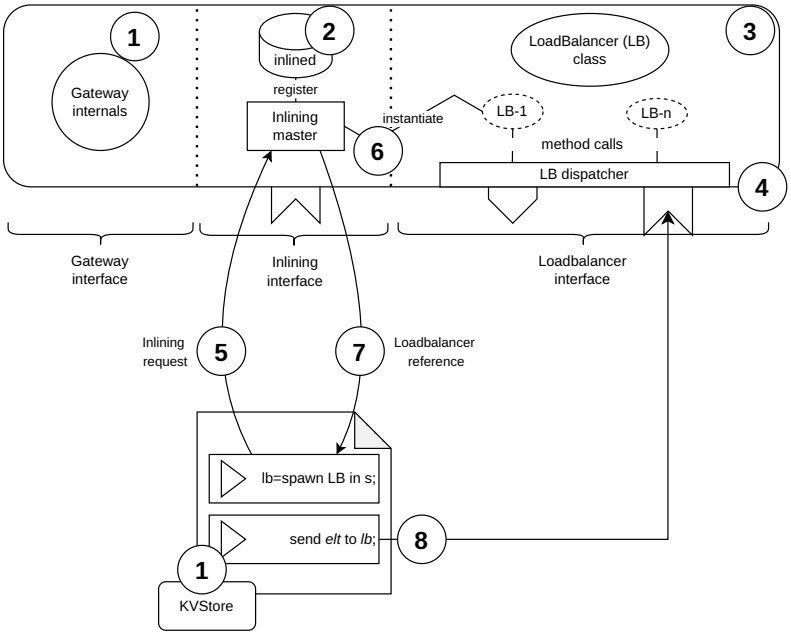


Fig. 5.4.: Inlining general workflow. "`Filters`" $f_1, \ldots, f_n$ are inlined inside the Source. Filter g is a regular Varda component.

Vardac prepares the system architecture at compile time (2-3), i.e., it generates the inlining boilerplate composed of new ports, states and communication logic.

At runtime, while executing the "spawn with inlining", Varda runtime performs the dynamic inlining setup (5-7), i.e., it starts an object in the host and correctly set the communication links.

Once this is set up, the inlined component can communicate with the external world through their host (8 details the workflow of a message).

Static inlining workflow (See. Figure 5.4)

(2) From (1), Vardac generates the generic inlining logistics inside "`Gateway`". This logistics permits to receive inlining requests and to process them. It is composed of a *dedicated port* listing for inlining request; a *local state* that registers the running inlined component; and, the *inlining master logic* that spawns a local component instance upon request reception and which attributes to him a unique identity.

(6) For each type of component that might be inlined inside `Gateway`, here `LoadBalancer`. Our compiler generates a "`LoadBalancer`" class that embed-

ded all the internals of its respective component, i.e., its methods and its local state.

7 Vardac generates adds a load balancer interface to the gateway, since the load balancer class and logic are hidden inside the host internals.

There is one `LoadBalancer` interface shared by all the objects ($lb_1, \ldots, lb_n$) whereas each classical instance of a component has its one dedicated interface. Vardac generates the "`lb_dispatcher`" logic to correctly paired the incoming message with the related load balancer object based on session metadata.

Dynamic setup

The dynamic setup stage starts when the execution flow of the parent reach the "`sapwn LoadBalancer(...) in ...`" instruction and terminates at the end of this instruction execution.

5 When the "`KVStore`" component reach this instruction, it sends an inlining request to the "`Gateway`" containing the type of the "`LoadBalancer`" and its "`onStartup`"arguments.

6 Upon the reception of the request, the inlining master logic starts a "`LoadBalancer`" object, here `lb-1`, with a unique global id for future communication. The gateway stores the id in its local state for further routing and sends it back to the parent component.

7 Upon reception of this id, the parent craft a component reference using the s reference (the host of the inlined component) and the id of the internal object.

Vardac generates all the code specific for this dynamic setup during the static phase.

Communication with an inlined component

This the steady state after inlining. The following describes the route of an element sent by another component to lb_1.

(8) The other component sends a message elt to "`lb_1`" which points to "`gateway`" with an additional metadata that uniquely identifies the lb_1 object. This metadata is piggybacked on the message by the generated logic.

The "`Gateway`" receives it on its "`LoadBalancer`" interface. The dispatcher routes the message to lb_1 based on metadata. Which process it and can optionally response or create new session through the "`LoadBalancer`" interface of the "`Gateway`".

5.3.3 Interaction between inlining and interception

On the one hand, inlining can be used orthogonally to interception enables tailored optimisations: inlining can be used without restriction inside an interception scope and, in some conditions/restrictions, can be orthogonal to an interception scope. On the other hand, inlining is a powerful feature that can be used in conjunction with interception to eliminate the communication overhead of the interception. This allows programmers to use interception to incrementally build distributed systems by leveraging interception without paying an unacceptable overhead.

The current prototype does not yet support all these behaviours.

To discuss and to illustrate these behaviours, we use the following toy example:

```
intercept<I> policy_i {
    activation_ref<A> a =  spawn A();
    intercept<J> policy_j {
        b = spawn B();
        c = spawn C() in a;
        d = spawn D() in b;
    }
    e = spawn E() in b;
}
```

Inlining inside an interception scope Inside an interceptor context, programmers can use inlining directive without restriction. For instance, `d = spawn D() in b;` works as we discussed in the previous section.

Inlining $\perp$ interception By nature, the Varda inlining instruction can cross the interception boundaries (i.e., interception scope). In the following, we discuss the two cases where inlining could interfere with interception: either the

host is outside the interception scope or the inlined component is outside the interception scope.

To discuss the first case, without loss of generality, let us consider the inlining of `c` in `a` such that there is an interceptor `j` in between. In this setup, `j` should intercept both the communication induces by the inlined setup and the actual communication of `c` with external components. This implies that the interceptor `j` will intercept the `C` interface of the host. As a result `a` cannot host intercepted and non-intercepted instances of `C` at the same time.

To discuss the first case, without loss of generality, let us consider the inlining of `e` in `b` such that there is an interceptor `j` in between. In this setup, `i` should intercept both the communication induces by the inlined setup and the actual communication of `b`. However, it should not intercept the communication of the inlined `e` with external components. This implies that the interceptor `i` should intercept the `C` interface of the host. As a result, with the current inlining mechanism, `a` cannot host intercepted and non-intercepted instances of `C` at the same time.

Mixing inlining and interception induces a restriction in the interception capabilities: the communication betwen the host and the inlined components are not intercepted. Even if they are split by an interception scope.

Inlining an intercepted component into its interceptor This case is observably equivalent to classical interception. The inlining takes over interception, without modifying the functional behaviour of the interception. Note that inlining nested interceptors (e.g., `j` in `i`) is a special case.

Inlining must respect the spawn order. For instance, inlining `i` in `j` (resp. `j` in `b`) is impossible due to the interception spawn order: the interceptor is always spawned before the intercepted component.

Inlining an interceptor The idea is to compile away communication indirection introduced by interceptors using inlining. In our example, the idea is to inline the inner interceptor (e.g., `J`) into the outer interceptor (e.g., `I`). To support this, an interception policy should specify interceptor inlining. We introduce a variant of the interception factory function in order to allow either

inlining or classical placement of the interceptor according to the execution flow. According to the usage, the compiler generates the either a simple factory function of the previous chapter (with placement specification only) or this inlining factory function. For instance, `policy_j` can inline `j` in `i` as follows:

```
activation_ref<J> policy_j (
    option<place> -> option<activation_ref<any>> -> activation_ref<any> factory,
    string intercepted_component_schema,
    place p_of_intercepted
){
    (* Where this.interceptor_i is the first interceptor *)
    return factory(none, some(this.interceptor_i));
}
```

5.4 Summary

To make non-functional properties fine-grain programmable (e.g., fault tolerance, elasticity, placement), Varda provides resources (e.g., failures, places and network links) that models the underlying infrastructure or the underlying OSI layers. To ease programming Varda exposes these resources as first-class objects. In the sense that components can copy them, store them or send them to other components, with some restrictions depending on the nature of the resources. For instance, a component cannot create a place out of thin air.

Using these core resources, Varda provides various primitives and annotations to easily program classical distribution properties as component placement, fault supervision or elasticity. Moreover, programmers could easily extend the Varda to control additional non-function properties (e.g., security). For this, a developer could expose custom resources and primitives by leveraging OTS adaptors and supervision port. We demonstrate this approach by extending the Varda programming model with fault supervision by abstracting the underlying supervision mechanism of an actor framework.

Last but not least, Varda provides optimisations to reduce the overhead inherent to the composition of independent components (e.g., due to context switching, marshalling/unmarshalling, or network communication):

- To mitigate network overhead, programmers can co-locate components using placement annotations independently of the encapsulation boundaries.
- To eliminate context switching, programmers inline components at the cost of a loss of parallelism.

6 Towards verified distributed programs

Introduction

Modern mainstream programming languages provide safety guarantees for non-distributed programs, ranging from memory safety to type systems, to predicates on types [41]. Additionally, distributed systems must deal with issues specific to distributions and composition. This chapter explores how Varda can help to improve correctness. We focus on the *interaction between components* and the *adequation between code and specification* with a special focus on the OTS.

We first present the Varda's safety building blocks (Section 6.1). Then, we discuss how to combine those blocks to achieve the following objectives (Section 6.2): formalising component behaviour; specifying the interaction between components; constraining OTS behaviours; and, bridging the gap between specification and code. Note that, this work does not address liveness properties since Vardac checking mechanisms (e.g., type checking and dynamic check injection) are not sufficient to detect a liveness violation.

A pragmatic approach is required to be able to reconcile correctness requirements with performance and ergonomic requirements. Therefore, Varda toolbox often provides various specialised building blocks for a given property. Each block refines the previous one by adding more expressiveness at the cost of additional cognitive cost and system overhead.

6.1 Safety toolbox

This section details the building blocks to strengthen system dependability. Varda provides two distinct kinds of tools: (mandatory) declarative tools that hold by construction when using Varda entities (e.g., interception or isolation) and (optional) imperative tools that a programmer can use to additionally constraints on a system. Figure 6.1 shows a classification of those blocks. On the x axis, we classify the

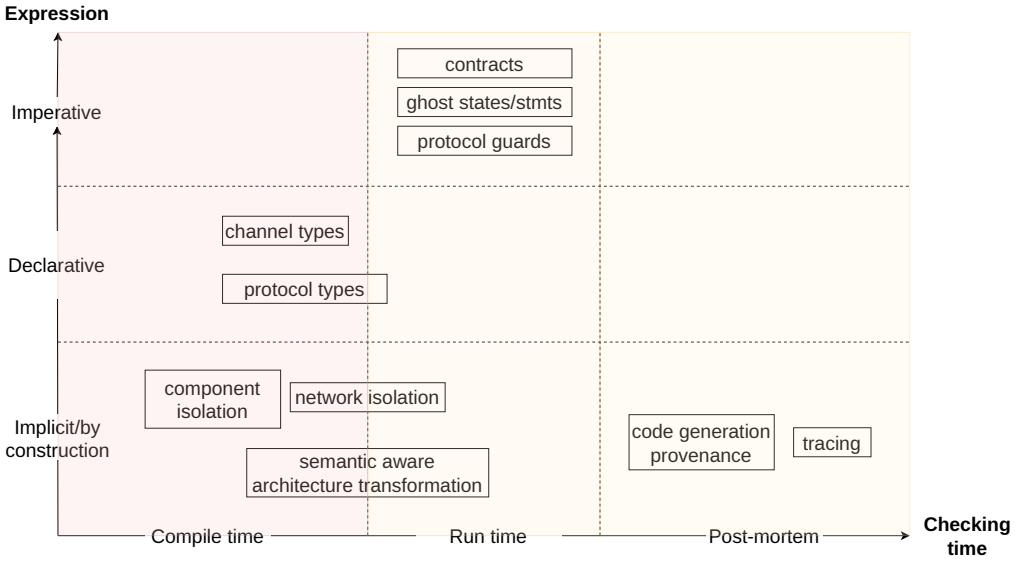


Fig. 6.1.: Overview of the Varda verification toolbox.

blocks according to *when and where they are checked*: at *compile time*, at *run time*, or *post-mortem*[1]. On the *y* axis, we categorise the blocks according to the *their expressiveness*[2]: whether a guarantee is provided *by construction*, or the whether a programmer must state them explicitly by *by declaratively annotating* the architecture or *by programming them*.

6.1.1 Declarative specification

Component compatibility

Component compatibility is the property that two components are mutually compatible based on their interfaces, i.e., they can be interconnected and can communicate.

To ensure this, Varda leverages type-checking since programmers are used to annotate programs with types and since types are sufficient to formally specify component interfaces, while ensuring component reuse and evolution [128]. The type system works at three layers: classical types 1. describe the internal logic of a component. Additionally, 2. each communication between components is typed by a protocol.

[1] We denote by "post-mortem" the fact that information is available, at the end of the system execution, for inspection either inside the logs or inside the generated code.
[2] Note that the control provided by a block is often proportional to its cognitive cost.

Furthermore, 3. the type system ensures that the internal logic processes messages according to these protocols.

Varda type system provides built-in types for the most common data structures such as lists, tuples or dictionaries. Programmers can define new types by composing any of the existing Varda types as named tuples. Moreover, they can embed external types (see Section 5.1.4) as abstract type [59]. These types are defined in an external language and use in Varda as opaque data type, i.e., a Varda component can only pass by argument or (optionally) send a value with an opaque type. To manipulate those value, programmers can import external libraries using adaptors. For instance, using embedded types, a developer can import existing data structure libraries (i.e., the data structure representation and its associated primitives) to write the business logic of a component.

Communication is described by a protocol specified as a binary session type [61]. Binary, in contrast to multiparty, means that the protocol only involves two components[3]. The type-system guaranteeing that the communications performed through session primitives (i.e., fire, select, receive, branch) follow the protocol. More precisely, the following properties hold [54]: the exchanged data has the expected type, the structure of session types matches the structure of communication at run time, the session channel has the expected structure and the session channel is visible only by the communicating parties.

A component type is classically described by a signature of its internal logic (i.e., methods, states and subcomponents) and by its interface. The interface denotes the set of the signatures of its ports (i.e., the protocols), and the type of the "`onStartup`" method.

To ease code reuse and architecture evolution, Varda type system supports both parametric polymorphism and subtyping [128]. Parametric polymorphism allows developers to parameterise types based on a generic type. For instance, it this enables the programmer to write code generic code using Varda collections (e.g., set or list) or to manipulate components independently of their interfaces. Conversely, subtyping represents a notion of substitutability between data types and between components [120]. This allows replacing (or update) a component by another one with additional observable features without having to change the code of the other components which are already connected to it.

[3]We disscuss adding multiparty session types as futur work, Chapter 11.

Interface substitution The interface of a component can evolve independently of the others as long as the interconnected ports remain compatible. A programmer can extend the interface of a component by adding arbitrary new ports. Moreover, the developer can update the protocol (and the related inner logic) of existing ports according to port substitutability. Note that interconnected ports cannot be deleted without updating other components.

The Varda type system captures these properties through interface subtyping which leverages session type subtyping [83]. Without loss of generality, component "`A`" is *substitutable* by component "`B`" (i.e., "`B`" is a subtype of "`A`") if

- for each port of "`A`" there is a substitutable port of "`B`";
- the "`onStartup`" method of "`B`" is a subtype of the "`onStartup`" method of "`A`"

Port "`p_1`" is *substitutable* by port "`p_2`" if

- both ports have the same kind (i.e., passive, active or supervision);
- port "`p_2`" is a subtype of port "`p_1`".
- Port subtyping is covariant for passive ports and contravariant for active port. Therefore,
  - for passive port, the protocol of "`p_2`" is a subtype of the protocol of "`p_1`".
  - for active port, the protocol of "`p_1`" is a subtype of the protocol of "`p_2`".

The protocol subtyping relation allows the protocol to evolve by changing message types covariantly[4] in input positions or contravariantly[5] in output positions; changing the set of labels covariantly in branch types or contravariantly in choice types; and changing the continuation types covariantly in input, output, branch and choice types.

For instance, a straightforward evolution of our key-value store will be to replace one replica at a time by a new replica that supports deletion and that exposes a monitoring port to provide load information to guide the "`LoadBalancer`" when it selects the replica responsible for an incoming "`get_request`".

The new protocol "`kv_protocol_2`" with the additional delete operation is defined as follows

```
type delete_request of key;
protocol kv_protocol_2 = (&{
    l_get: !get_request?value.;
```

[4] A typing rule or a type constructor is covariant if it preserves the ordering of types, which orders types from more specific to more generic

[5] A typing rule or a type constructor is contravariant if it reverses this ordering.

```
    l_put: !put_request?bool.;
    l_delete: !delete_request?bool.;
})*;
```

This requires the following changes to Listing 2: 1. add a passive monitoring port; 2. update the protocol of the passive port "p_p" from "kv_protocol" to "kv_protocol_2".

Compatibility and architectural transformation The architectural transformations presented above (i.e., interception and inlining) preserve functional compatibility. Technically, such a transformation is a source-to-source rewriting. Therefore, the output architecture passes the compiler standard checks.

Component isolation guarantees

Component's internals are encapsulated from other components, children and OTS. Components can interact only by exchanging messages through channels and ports, or by spawning children. Moreover, a parent component is isolated from its children, and vice versa. For instance, a parent cannot access the state of a child nor terminate it. Of course a parent could send a termination request message to a child if the protocol supports it.

Varda ensures strong isolation between components in order to mitigate the effects that a component can have on another one. Namely, Varda provides: *communication isolation* at both the network layer and the application layer, *failure isolation* and *OTS sandboxing*.

Communication isolation By default, components cannot communicate with each other. The programmer explicitly drills holes in interfaces to interconnect them through channels and ports, or to receive events from the environment (i.e., the runtime) through supervision ports.

At the network layer, Varda ensures that components can communicate only if their ports are connected through a channel and are compatible. Moreover, at runtime, a message sent on a channel cannot be intercepted by another component, even using interception. Remember that the interception adds a new component that act as a proxy between the sender and the receiver. Interception is performed at compile time, at run time the architecture is composed of traditional channels and components. (Section 4.2).

At the application layer, Varda ensures that sessions are isolated from each other even over the same channel. A component can receive only a message of which it is an explicit destination.

Crash and failure isolation Varda follows the "Let It Crash" ([16]) principle since faults and crash are unavoidable in a distributed system. The runtime guarantees that if a component instance crashes, the other components are not directly impacted. Furthermore, a faulty component cannot block another's execution since all message reception is asynchronous in Varda. However, a crash still indirectly affects other components since communication with the faulty component may trigger a timeout or a crash notification to fault supervisors. As a result the business logic of a component can stop progressing.

OTS sandboxing A Varda shield isolates an OTS component. The compiler ensures that an OTS can interact with Varda components only through its shield. The shield restricts the component's behaviour to its specified interface, protocol, and pre- and post-conditions.

However, Varda cannot detect nor protected against hidden side channels. Such side channels may result either from a malfunctioning or a malicious OTS. In this context, such an OTS could bypass the Varda guarantees. For instance, a side channel may result from malfunctioning components using the same external key-value store with an overlapping key space. Detecting such a channel implies to be able to reason about the semantics of the OTS implementation. It is out of the scope of Varda and it is the responsibility of the programmer. In the same way, security isolation is out of the scope of the current prototype since a malicious component can escape the Varda safe programming model, by accessing low-level resources (e.g., raw sockets).

Isolation and architectural transformation Interception preserves the isolation guarantees. Conversely, inlining relaxes the isolation between the host and the inlined component. The compiler preserves OTS and communication isolation[6], , if the host fails, the inlined component crashes as well. The reverse is not always true: an inlined component could fail without crashing the host depending of the nature of the failure and of the coge-generation target.

[6]The host and the inlined component can communicate if and only if they could communicate without inlining: i.e., they are interconnected by a channel.

```
301 contract m
302     (* pre-contract constant binders *)
303     with
304         int x = store_a_copy_of_some_local_state
305         int y = ...
306     (* precondition *)
307     ensures <expr> (* expr can contain local state and pre-contract variables *)
308     (* post-condition *)
309     (* the post-condition is a predicate that takes the return value of m as arguments *)
310     returns x : m_ret_type -> <expr> (* expr can contain local state and pre-contract
        ↪ variables *)
311
312     (* the invariant is an optional predicate that is added to the pre and to the post
        ↪ condition *)
313     invariant <expr>
```

Listing 12: Detailed syntax of a contract over method `m`.

6.1.2 Imperative specification

Component contract

To verify that a component behaves as expected, the programmer may optionally attach *contracts* to internal methods. A contract is a set of predicates over the arguments, return values of a method and over local variables. Listing 12 shows the general syntax of a contract for a method `m`. It has three optional predicates: precondition (`ensures`), postcondition (`returns`) and invariant (`invariant`). The invariant is a syntactic sugar such that the compiler adds it to the pre- and post-conditions. A predicate may include any legal Varda code but should not have side effects. Indeed, using communication primitives in contracts may be tricky and dangerous: the architecture with and without contracts may not be observably equivalent. Therefore, erasing (resp. injecting) the dynamic checks in the generated code could potentially change the semantics of a component.

To express complex properties over the execution of a component and to only pay the extra overhead when needed, a predicate may refer to *ghost variables*. A ghost variable is only involved in expression and statement that concerns the verification logic. They can be compiled away if the contracts are not injected in the final implementation. More generally, Varda supports *ghost methods*, *ghost state* and *ghost statement*. A ghost statement might be nested in non-ghost logic.

For instance, the following contract (Listing 13) specifies that the value returned by "`get(key)`" must be the one associated with the previous "`put`" the highest "`counter`". It uses *ghost* state "`last_value`" to remember the value of the highest counter, for each key. The `get` contract checks the condition[7] whereas the `put`

[7]For simplicity, it omits the case where ghost state does not contain the key.

```
314 (* For each key [last_value] stores the value with the highest counter *)
315 ghost dict<key,value> last_value = dict();
316 (* Checks that the get(key) returns the value with the highest counter *)
317 (* [key] is the argument of [get]; [res] is the return value of the [get] call *)
318 (* [store] is a user defined function that wraps dictionary insertion into a predicate *)
319 contract get returns res -> this.last_value[key] == res
320                              && store(this.last_value, key, res)
321 contract put returns res -> store(this.last_value, key, value)
```

Listing 13: Contract for the `get` method of the key-value store. It ensures that each read value corresponds to the value with the higest counter.

contract registers value using the "`store`" function which wraps dictionary insertion into a predicate.

Varda enforces contracts by injecting dynamic checks and ghost variable in the generated code. Programmers can disable dynamic checks injection by setting a compilation flag `--erase-dyncheks`. The evaluation on method call follows as is[8]: 1. bindings of local `wit` binders, if any; 2. runs the precondition and the invariant if any; 3. runs the function body and store result; 4. runs the post-condition on the result and the invariant.

Protocol guards

Session types cannot express properties based on the content of the exchanged messages, nor on the history of the protocol (previous exchanges/rounds). To circumvent these limitations, programmers can optionally specify the observable of communication by adding dynamic guards to protocols. A protocol guard is a predicate over messages. Those predicates can express constrains on message value, message timeout delivery or on session history.

To support predicate over session history, a session has a context, i.e., a side state used to remember values during the session lifetime. A programmer can add a new variable in the context to store a piece of session history (e.g., to remember a message value), then they can access it. Programmers can update a context value with `store_meta(name, value)`. Currently, to simplify the compilation in the case of recursive protocols, they cannot delete it, nor update it with a different type.

For instance, one can specify that, the key of a "`put`" contains two alphanumeric substrings separated by a dot (e.g., follows a "*name.extension*" pattern), and that its counter increases monotonically. This example is shown in Listing 4 (Lines 20–23). Here "`predicate_key`" is a procedure that returns true if its argument satisfies the

[8] We adapt the evaluation strategy of the D language: `https://dlang.org/spec/contracts.html`.

```
322 (* msg is the value of the current put_request *)
323 (!put_request{msg ->
324     (* check that the key follows a pattern defined by [predicate_key] *)
325     predicate_key(msg.1) &&
326     (* Ensure that the counter of the message is strictly greater
327     than the one of the previous message, if any. *)
328     last_c < msg.0 &&
329     (* Register the current counter in the session metadata *)
330     store_meta(last_c,msg.0)
331 (* Define a session metadata counter *)
332 }?value.)*{metadata int last_c = 0}
```

Listing 14: Example message predicate, extracted from Listing 4

required pattern[9]. Directive "`metadata int last_c`" piggy-backs on the message the metadata variable "`last_c`", which contains the counter of the last "`put`" request, as directed by "`store_meta(last_c, msg.0)`".

Furthermore, one can specify times bound on sessions using ad hoc timers. Programmers can set timers at the beginning of the session step (e.g., sending or receiving a message) and specify upper (resp. lower) bound on those timer inside guards of subsequent session step. For instance, the protocol "`!key{timer t|}?value{|(t<100)}.`" triggers a notification if the sender of "`key`" does not receive "`value`" within 100 ms.

Guards are *session specific* and *component independent*. *Component independence* means that the guard is agnostic of the identity and type of the component that uses the protocol. *Session specificity* means that a guard cannot ensure communication properties that involve multiple sessions, and, especially, not properties that involve distinct protocols. Hence, a guard context is specific to a given session. It cannot be shared between two sessions. We add this limitation to avoid synchronisation and consistency issues between arbitrary distant components. However, programmers can, at their own risk, use an external backend to share information between sessions guards[10]. This behaviour is outside the scope of the Varda guarantees.

[9] For brevity, we omit the definition of the "`predicate_key`". It can be either written in plain Varda or as an external library using a component stub.

[10] Synchronization overhead and consistency may depend on the selected backend.

A component contracts constraints the behavior of a specific component whereas a protocol guard applies to all instances of protocol independently of the components instances that use it. Therefore, a guard cannot reference a local state of a component. We introduce this distinction because this avoid to annotate manually all the reception logic with contracts and to centralize the definition of communication properties.

Evaluation strategy Guards are a kind of syntactic sugar on top of core Varda. Vardac either injects the guards at the reception or at the emission of a message involved in a guarded protocol. An alternative approach could be to add guards in the middle of a communication by leveraging an interceptor in charge of applying them. However, this would add an extract communication indirection with its resulting overhead. Therefore, we do not use this approach for point-to-point communication.

Guard context Varda adds metadata to the session to store its context and to propagate it between communicating components. That metadata is piggy-backed to session messages. Therefore, this does not imply extra communication, nor external backends, nor synchronisation. However, the advantages of this technique fade when the context becomes too large with respect to the original message size.

Extending guards to nonlocal properties In the following snippet, we show how a programmer can incorporate an external backend, at its own risk, to express nonlocal guards. We transform the previous guarded protocol of Listing 14 into the following protocol such that it ensures that the counter (i.e., the version) of each "`put_request`" is increasing, independently of the session.

```
(*  Abstract methods bound to OTS implementations
    using the same datastore backend
    (e.g., redis://shared_redis_host:6379)
*)
int get_shared_counter(string key);
bool set_shared_counter(string key, int amount);
protocol p_b = μx: !msg{get_shared_couter("last_c") < msg.0 && set_shared_counter("last_c",
↪  msg.0)}-x;
```

To achieve this, we use an external backend to store the shared counter using our OTS adaptor. In such a case, Vardac only ensures that the expressions in the predicate are well-defined and type check.

Performance

To mitigate the system overhead, the usage of most of the block is optional and the Vardac can compile away/eliminate most of the constraints in production mode[11]. This eliminates the protocol guards, the contracts, the ghost states and the ghost statements. Furthermore, Varda building blocks does not require synchronisation and have a limited network cost extra-communication[12].

6.2 Using the primitives

6.2.1 Constraining the OTS behaviours

Varda is designed to reuse existing code bases and services. Therefore, a common pattern is to reuse and compose unsafe off-the-shelf (OTS) components. Such a component may misbehave in arbitrary ways.

To contain that risk[13], Varda encapsulates OTS components in a *shield*. All the interaction between the shield and the OTS must be initiated by thformerld. The OTS does not have the initiative except if the shield explicitly allows it by exposing a supervision port to collect effects from the environment. In that case, the programmer has to write an adaptor code to translate the OTS actions into environment events.

Moreover, programmers can leverage core Varda safety building blocks (e.g., contract) to add extra constraints on the observable of an OTS.

6.2.2 Constraining interactions between (two) components

A programmer should be able to *specify how components communicate* [@74], [109], [128] and *reason over the composition logic* [109], i.e., the orchestration code running in the Varda shields, either handwritten or generated thanks to the interception mechanism.

This section presents a summary of what levers do Varda propose for constraining the effects between components. Running components can have both direct and

[11] Using the `-erase-dynchecks` compilation option.
[12] It only attaches additional metadata (controlled by the programmer) to messages for some usages.
[13] Note that the current Varda implementation does not mitigate security risk. It only focuses on safety.

indirect effects on each other and can suffer from effects from the environment, more precisely:

Direct effects come from message passing exchanged through ports and sessions.

Indirect side effects Components can alter the knowledge of each other thanks to reflexivity primitives (either on channels or on placement). They can also use the environment to deliver specific messages using supervision ports. Last but not least, components can establish a hidden side channel using OTS, either by design, by malfunction, or, even, by malfeasance. For instance, two components that use the same data store might share the same key space.

Environment The runtime can send arbitrary events on supervision ports[14] as long as they are well typed. Those events could be triggered by other components, by runtime internal events, or by the external services if the runtime is instrumented with hooks to convert unsafe notification from services to a safe environment event.

This section focuses on the effects that are visible by Varda's runtime: effects that result from message passing and event delivery through supervision ports. The other effects are out of the scope of this work.

Programmers can specify the *component topology*, i.e., who can communicate with whom ?; the *communication semantics*, i.e., how the communication must behave; and, *some non-functional effects* (e.g., write contracts on placement).

Topology

Topology in Varda covers three abstraction levels: static topology (domain of types), network dynamic topology and the session dynamic topology, which represents the effective communication.

Let us define the relationship between those topologies: the static topology is a superset of both the dynamic topology and the network dynamic topology; the latter is also a superset of the session dynamic topology. This means that if two components cannot communicate in the static topology, they cannot communicate in the dynamic topologies. Conversely, if two components can communicate in the

[14] For instance, it cannotify a component that one of its children has been killed.

static topology, they *might* communicate in the dynamic topologies depending on the coordination logic and of the execution flow.

Static topology The static topology is a multigraph, where edges are annotated by protocols, which represents the allowed communication between component type. It is derived from channel types. If there is a channel of the form `channel<U,V, some_protocol>` then the static topology contains the following edges: `A` $\xrightarrow{some\_protocol}$ `B` for all $A <: U$, $V <: B$ where $<:$ denotes the subtyping relation.

In practice programmers do not have to write the whole static topology as a monolith but only need to type each channel. From this, the compiler infers the static topology: it statically ensures that the network dynamic topology matches the static topology. Moreover, it can optionally provide simple visualisation capabilities of the inferred graph.

Dynamic channel topology The dynamic channel topology is a dynamic multigraph that represents the run time allowed communication between component instances[15]. Programmers explicitly define the edges by binding channels to ports.

A communication channel must be established prior to any communication. Note that two components that can communicate according to the topology may not exchange any messages during an execution due to the execution flow.

Varda runtime does not maintain a global view of such topology to avoid overhead and consistency issues. Instead, components knows only their neighbours.

Programmers can constrain this topology by restraining the channels that can bind to ports. To protect the binding logic, they can write contracts using Varda's primitives that can answer questions such as: What are the components already bound to a given channel ? What is the channel already bound to a given port ?

[15] Currently Varda is agnostic of the infrastructure topology, i.e., network between nodes. To specify/-collect the topology of the infrastructure, programmers have to write their own API and expose it thanks to abstract methods.

Dynamic session topology This represents the effective exchanged message. This topology is managed by session initialization and communication primitives. Programmers can write nonlocal form of topology constraints thanks to interception. In Section 8.3.1, we discuss how to implement access control at the session layer leveraging interception.

Constraining message passing

Allowed *interactions* between two activations are abiding by a formal protocol guaranteeing the type, the order of messages, optional predicate on the content of the session and optional time delivery upper bounds. However all the involved building blocks provided by Varda toolbox are designed for peer-to-peer communication. To express property about groups of components, programmers either needs to manually inject their logic inside the orchestration logic, or, they can use a monitor pattern [59] leveraging interception.

The role of a monitor is to check properties on the global view of a subsystem. For this, monitors introduce a kind of global state for a region of the system. Note that, a monitor cannot constrain properties about the internal states of the observed components. For instance, the following properties cannot be expressed: *the sum of the internal counter of a component type is lower than ten*.

In Varda, the idiomatic way to express a monitor is to define it as an interceptor which intercepts all the component instances that should be observed. Monitors can check complex properties with the history of a set of components that aggregates multiple sessions with different components types, exchanged between a (dynamic) set of components. Programmers can craft a monitor by defining a ghost interceptor such that the compiler can transparently interpose it in this architecture, and, optionally, remove it at compile time when building the production release.

The Listing 6.2 shows the definition of a monitor, called "`MyMonitor`", that checks that the number of sessions created by a set of components is lower than 100. "`MyMonitor`" intercepts all the session creation and increment an internal counter (Lines 348-351). If the counter reach 100, the contract of the "`incr`" method is violated and the execution is aborted (Lines 344-345).

```
340 component MyMonitor{
341     int number_of_sessions = 0;
342     int max_number_of_session = 100;
343
344     contract incr
345     ensures this.number_of_session < this.max_number_of_session
346
347     (* For compactness, we do not write the return type nor the arguments of the method *)
348     @sessionintercept(both)
349     ... incr(...){
350         this.number_of_session = this.number_of_session + 1;
351     }
352 }
353
354 (* Check that the inner component cannot create more than 100 sessions *)
355 ghost intercept<MyMonitor> my_monitor_policy {
356     (* The monitor monitors all the components spawned inside this scope *)
357 }
```

Fig. 6.2.: `MyMonitor` ensures that the number of session created by the intercepted components are less than 100 without relying on an external backend.

Constraining the environment effects

Contracts are the idiomatic way to check properties on environment notification delivered on supervision ports. As above, they can only express component-local properties, unless the contract (or the component) relies on an external backend to access a shared state. Additionally, using contracts, programmers can constrain non-functional interactions related to the placement thanks to the *placement reflexivity* primitives (Section 5.1.1). Note that, the current prototype of Varda does not provide any consistency guarantees on the placement view obtain by using those primitives[16].

For instance, the following example (Listing 15) checks that there is at most one *KVServer* per nodes in both `DC-` and `DC-2`. This check is performed when a `Loadbalance` start (Line 381). It calls the `check_d` function (Line 366-375) on the places of the two DC. `check_d` iterates over the nodes attached to the current DC and marks those that hosts a `KVServer`. The `place_selecto` function (Line 358-364) selects the places where `KVServe` they are deployed.

Since the placement registry is asynchronously updated without any consistency guarantees, this example can trigger false positive. For instance, if there is a node a such that it hosts a `KVServer`, this instance dies and a new one starts again with a distinct identity. During a short period of time, depending on the propagation mechanism, the `LoadBalance` can observe two `KVServe` on the same node.

[16] There is a delay between the addition/deletion of a node (resp. component) from the global index and its visibility by others.

```
358     bool place_selector(place p){
359         for(activation_ref<any> a in activationsat(place)){
360             if(schemaof(a) == "KVServer")
361                 return true;
362         }
363         return false;
364     }
365
366     bool check_dc(vplace vp_dc){
367         set<place> marked_place = {};
368         for(place p in select_places(vpa, place_selector)){
369             if(exists2set(vpb, place))
370                 return false;
371             else
372                 add2set(vpb, place);
373         }
374         return setlength(vpb) >= k;
375     }
376
377     component LoadBalancer{
378         vplacedef vp_dc1_backend_nodes of "DC-1::nodes";
379         vplacedef vp_dc2_backend_nodes of "DC-2::nodes"; (* TODO check syntax to access a
            ↪ child *)
380
381         contract onStartup
382         invariant check_dc(vp_dc1_backend_nodes) && check_dc(vp_dc2_backend_nodes);
383     }
```

Listing 15: Checks that there is at most one instance of `KVServer` per node and that there is least `K` nodes per DC that host a `KVServer`.

6.2.3 Bridging the gap between code and specification

Combining a high-level description of the system architecture and a low-level implementation may lead to introduce subtle discrepancies between the two, and thus to introduce unwanted behaviours and bugs [39], [60].

To ensure that the system's behaviour does not deviate from the specification. Vardac generates[17] a glue code from the architecture written in Varda, given the implementation/adaptor with the OTS. Moreover, Vardac can instrument the glue code, i.e., the shield, with dynamic checks to detect run time violations of the architecture specification.

To cope with OTS and arbitrary legacy code adaptor, Varda isolate them behind a shield that exposes a well-defined Varda interface[18]. Programmers can write contracts to check properties on the component-local observables of an OTS. Those conditions can use all the tools that we present earlier in the toolbox. Vardac instruments the shield with those dynamic checks.

[17] The code generator is part of the *trusted computing base* (TCB).

[18] If the target language, used to encode the glue code, is statically typed, the compilation ensures that the adaptor/OTS interfaces can be hidden by the shield interface.

Conversely, the outside world can call a Varda system thanks to its expose API (see Section 5.1.4), which is using non-Varda representation as gRPC or REST. The Vardac ensures that the API is consistent with the architecture specification, since it derives from annotated methods. If the external caller is not trusted, the programmer should check the entries inside the exposed method. Otherwise, when the caller is trusted, for instance, when the Varda system is API-composed with other existing building blocks to form a new system, the programmer can add contracts on exposed methods in order to check that the caller match its "specification" and remove it in production.

6.3 Summary

By construction, Varda ensures strong isolation between components (e.g., failure isolation, memory isolation and OTS sandboxing). Moreover, each component has formal communication interfaces such that an inter-component communication using Varda primitives follows the following properties:

- it is abiding by a formal protocol which guarantees the type and the order of the messages;
- it can only occur between components that are explicitly connected by channels;
- it is private, i.e., messages are visible only by the communicating parties.

Additionally, Varda provides a specification sublanguage such that the developer could enrich the architecture with custom constraints. With them it can specify:

- component's behaviours and OTS's observables using contracts (i.e., pre/post conditions) and ghost component state;
- run time protocol behaviours using message predicates, delivery upper bounds or protocol history predicates. The use of these additional constraints prevents programmers from scattering communication specifications within the component logic.
- the behaviours of a set of components using monitors, by combining the interception with the previous building blocks (e.g., contracts).

Varda toolbox often provides various building blocks for a given property kind. Each building refines the previous one by adding more expressiveness at the cost of additional cognitive cost and system overhead. We specialise these building blocks in order to avoid forcing developers who do not need the expressive power of a block to have to pay for it. Table 6.1 sum up these properties with their trade-offs in terms of expressiveness, cognitive cost and system overhead.

Constraints	Perimeter	Executors	Cognitive cost	System overhead
General constraints				
Functional compatibility	point-to-point	compiler	low	low
Bridging the gap	system-wide	compiler	medium[19]	-
Component internals				
Isolation-base	local	compiler + runtime	low	low
Isolation-interception	components-group	compiler + runtime	medium	medium
Contracts	component-instance	component instance	low	arbitrary
Interaction				
Static topology	group-to-group	component instance	low	-
Network topology	point-to-point/channel-wide	component-instance	medium	low
Session topology	point-to-point	component-instance	medium	low
Session-specific guard	session	component-instance	medium	arbitrary
Monitors	component-group	component-instance	medium	arbitrary
Environment				
Sandboxing/shield	local	compiler + runtime	medium	low
Observable	local	compiler-instance	medium	variable

Tab. 6.1.: Summary of the trade-offs of dependable programming with Varda. Perimeter denotes the scope of the constraint: a local constraint is only checked at the component level and cannot express properties involving multiple components. A point-to-point constraint can express properties about the communication between (exactly) two components. A component group constraint can express properties about the communication between a group of components and the remaining part of the components. Note that programmer control is inversely proportional to the runtime and cognitive overhead.

Under the hood, to ensure all this, Vardac performs static checks, mostly type checking, and injects dynamic checks to detect constraint violations. Moreover, Vardac can erase all the dynamic checks to avoid system overhead.

[19] On one hand, programmers do not have to write the glue code to compose the OTS. On the other hand, they have one more language to learn and one extra layer of compilation.

7 Compiler

The development workflow is described in Figure 7.1. It is designed to be multi-target, where a target is a programming language and a support runtime. An input Varda program describes a system's architecture. The Varda compiler first applies target-independent transformations. Then it links the result together with OTS adaptor code that the developer has provided separately. Finally, it generates code in the target language; it also generates a deployment plan, from a template provided by the developer. Here the target-specific compiler and deployment system take over.

To help with debugging, we strive to make generated code readable by humans. On demand, it is annotated with provenance information.

Vardac is composed of a large target-independent part[1] and some small target-specific plugins (e.g., the Java/Akka code generation plugin). The current version of Vardac, supports one target based on the Java version of Akka with DockerCompose for deployment.

The Table 7.1 the development effort in terms of LoC. The core compiler consists of 38 KLoC[2] of OCaml 4.12. The Akka target-specific part adds 4 KLoC of Java[3]. Benchmarks and tests contribute another 15 KLoC of Varda, Java and OCaml. At the

[1] In terms of LoC, 80% of the Varda compiler is independent of the code generation target.
[2] LoC are given without comment and blank lines.
[3] The OCaml Akka plugin represents 8 KLoC of the main compiler.

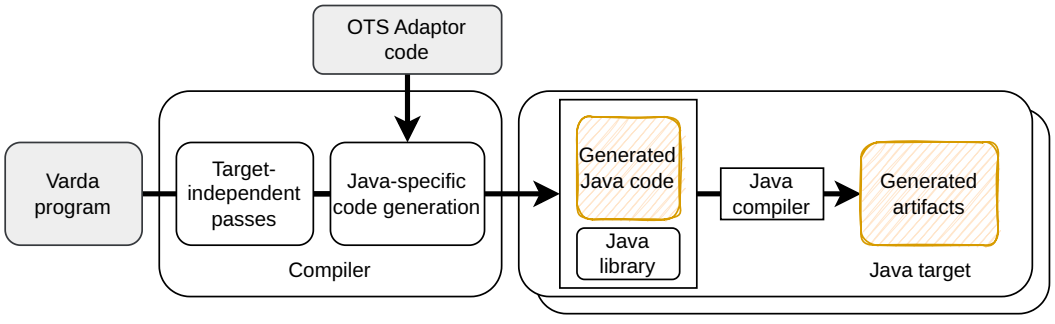


Fig. 7.1.: Varda workflow. The developer of a distributed system provides the grey parts. The Varda compiler generates the dashed (orange) blocks.

Parts	KLoC
Core compiler target independent	Ocaml: 30 KLoC
Code generation	
Generic code generation	800 LoC
Akka plugin	OCaml: 8 KLoC
Akka library	Java: 4 KLoC
Vardac	**OCaml: 38 KLoC, Java: 4 KLoC**
Tests	6 KLoC
Benchmarks[4]	14 KLoC

Tab. 7.1.: Summary of the Varda development effort in terms of LoC.

end of this chapter, we provide a detailed overview of the support of Varda entities by the compiler.

The Varda framework is open-source under an Apache2 license. The code source is available as an open source repository at the following location: `https://gitlab.lip6.fr/lprosperi/Lg4DC`.

7.1 Target-independent compilation

Vardac consists of a succession of Abstract syntax tree (AST) rewriting passes. These compilation passes are generic, they do not depend on the target choice. Figure 7.2 shows the overview of the AST transformation. To simplify the code generation and to avoid duplicating logic for each target plugin, Vardac performs a series of transformations that simplify the AST. Then, Vardac enriches the *Simple IR* with the code of the OTS adaptors and the description of the infrastructure.

Figure 7.3 shows the detailed flow of the compilation passes. Early passes perform static analysis, including standard type-checking and protocol compatibility verification. Then, it performs target-independent optimisations such as ghost elimination, constant-propagation, partial evaluation, unaliasing and dead-code elimination.

Vardac interposes message marshalling/unmarshalling as required (EventAutoBoxing). Then, it rewrites the initial AST to a simpler subset of the Varda language, by compiling away *complex features* such as branching and asynchronous receive

[4]This includes the code of the baseline implementation of the microbenchmarks in Akka.

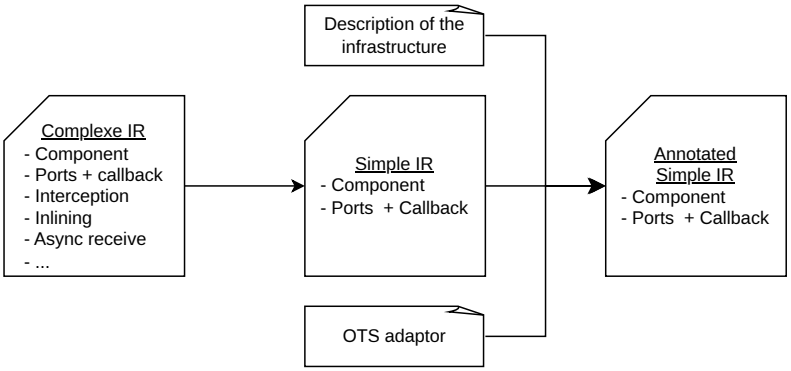


Fig. 7.2.: Overview of the Intermediate Representation (IR) transformations.

(CommSimplification), and *architecture transformations*, such as inlining (InlineElimination), interception (InterceptElimination) and RPC derivation (DerivationElimination).

In practice, Vardac execute those passes multiple times. For instance, the *InterceptionElimination* pass introduces session communication that should be compiled away by the *CommSimplification* transformation.

7.2 Code generation

Code generation is target specific: the compiler splits the architecture into one distinct (sub)architecture per target. Then, for each sub-architecture, it delegates the code generation to the corresponding plugin.

A target defines a set of compilation units, each of them grouping a set of components. Target configuration and target assignment are orthogonal to Varda architecture. This enables and facilitates two following properties:

- Varda architecture is polyglot [@119] which means that for a given target can generate an implementation in various target languages without having to alter the architecture.
- developers can finely adapt the implementation according to the underlying infrastructure. Often, the infrastructure is heterogeneous: i.e., mixing well defined and homogeneously infrastructure in the Cloud (either public or private) with more heterogeneous and less controlled infrastructure at the edge. Having the compilation units orthogonal to the logical units (i.e., components) simplifies specialising the final deployment according to a given infrastructure without having to alter the architecture.

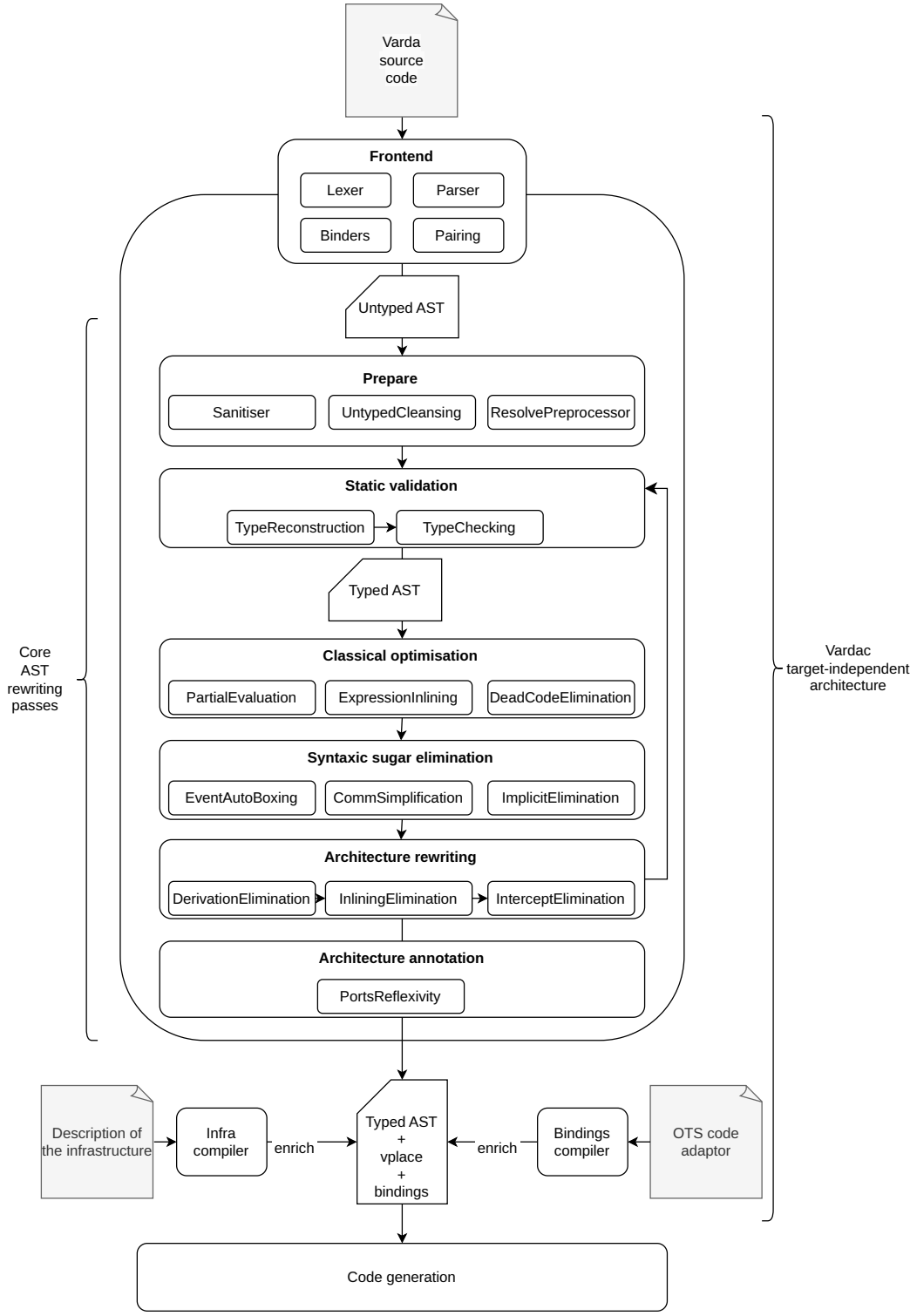


Fig. 7.3.: Target-independent structure of Vardac. The developer of a distributed system provides the grey parts. Note that the names of the passes are derived from those of the source code.

To increase the programming flexibility, programmers can inject custom source files to customise the generated code for a given target. They just need to add those files into an external (resp. templates) directory of the Varda project. The compiler automatically processes those files and injects them inside the final generated code. Moreover, those files can be templates[5] that will be rendered by the code generation plugin according to the environment exposed by the plugin (e.g., list of component names).

7.2.1 Target configuration

The *Target config file* describes the different targets of a given system. Each target entry specifies a code-generation plugin which defines the underlying technology (e.g., the Akka framework in Java) and a custom configuration that range from the choice of the expose interface implementation (e.g., gRPC or REST), the definitions of the generated artefacts (e.g., how to package system in JAR files); and, the target compiler options (e.g., the JAVA compiler options).

The Listing 16 shows the simplified target configuration file that we use for our running key-value store example. It defines one target named "`akka`" configured to use the Akka runtime plugin (line 389); to output the code in Java (line 388); and, to use a gRPC[6] implementation for the expose interface[7] (line 390).

This target specifies three distinct artefacts: 1. one for the *gateway* (Lines 394-399), 2. one for the *store* (Lines 400-405) 3. and one for the *console* (Lines 411-423). The architecture is split between the first two artefacts. The *gateway* bootstraps the `Gateway` component. The *store* bootstraps the `KVStore` component which spawns a load-balancer instance and the backend servers. Both the *gateway* and the *store* are packaged in a single JAR file named according to the artefact name. Note that, these artefacts do not define custom main function[8]

The *console* artefact defines a user-defined artefact (lines 411-423), out-of-the scope of Varda architecture. We us the Vardac's template mechanism to inject a user-defined *Console.java* file that exposes CLI console leveraging the gRPC client generated to query the exposed interface of the key-value store.

[5] We use the Jinja2 template engine to resolve them.
[6] The Akka plugin supports two interface generation plugins: gRPC and REST.
[7] The exposed interface is defined by methods annotated with "`exposed`" (see. Section 5.1.4).
[8] Any Varda top-level function can be defined as a main function.

```
- target: akka

  # configuration for the code-generator plugin
  codegen:
    language: Java # select output language plugin
    runtime: Akka # select the code-generator plugin
    interface: gRPC # the interface extension used (if any) for @exposed method

    # [artefacts] defines the set of artefacts generated for this target
    artefacts:
      gateway:
        # [no_main] is a dedicated key word
        entrypoint: Gateway
        # the main function used to process the cmdline option
        # its output is the arguments of the onStartup of the root
        entrypoint: no_main
      kvstore:
        # [no_main] is a dedicated key word
        entrypoint: KVStore
        # the main function used to process the cmdline option
        # its output is the arguments of the onStartup of the root
        main: no_main

  # arbitrary user-defined string that will be appended to the build dir
  # here it defines a Gradle task that compiles a console
  # from a ConsoleClient.java file provide by the programmer
  user_defined: |
    task jarConsole(type: ShadowJar) {
      archiveBaseName.set('console')
      archiveClassifier.set('')
      archiveVersion.set('')
      configurations = [project.configurations.compileClasspath]
      manifest {
          attributes('Main-Class': '{{author}}.{{project_name}}.ConsoleClient')
      }
      transform(AppendingTransformer) {
          resource = 'reference.conf'
      }
      with jar
    }

  # user-defined k/v map defining the options for the target compiler (e.g., Java compiler)
  compiler:
    loglevel: INFO
```

Listing 16: Targets definition for our running key-value store.

7.2.2 Code-generation plugin

A code-generator plugin translates a piece of Varda architecture into a final implementation in a given target technology (i.e., language and runtime and runtime library).

Additionally, a Vardac plugin comes with a runtime library, written in the target language, that adapt the underlying programming model to the needs of the code generation. For instance, the library for the AkkaJava plugin provides java classes for sessions, places, or errors. It adds a virtual layer of ports on top of actors, and it implements the placement registry leveraging the Akka Distributed Data.

Figure 7.4 shows the three parts architecture of the *AkkaJava* code-generator plugin: a *runtime plugin* that encodes the Varda into the programming model of the plugin; a *language plugin* which is independent of the *runtime* and provides general utilities to generate the code in a given language (e.g., Java); and, a *translator* that encodes the runtime AST into the language AST. Note that, the Java plugin could be reused by another code-generation plugin.

The code-generator structure is common to all kind plugins. Plugins differ by the implementation of the internal building blocks: each plugin defines its internal ASTs, their intermediate compilation passes and the pretty printing of the language AST.

Vardac provides all the pipework and utilities (e.g., type constraints and functors) to easily build a plugin. Moreover, it also provides general compilation passes to optionally simplify the intermediate Varda before code generation by removing some non-trivial features, if the underlying programming model does not support them.

For instance, the Akka plugin, encode components as actors. However, mixing actors with other concurrency control entities is tricky [167]. Hence, we use the core *FutureElimination* compilation pass to eliminate futures introduced by the asynchronous receive elimination transformation, applied by the core compiler. This completely gets rid of futures. It encodes them using ports, states and callbacks (using a form continuation-passing style).

7.3 Starting a Varda system

Deployment is orthogonal to the work of Varda. It is the responsibility of the programmers to deploy the artefacts on a running infrastructure. The programmers

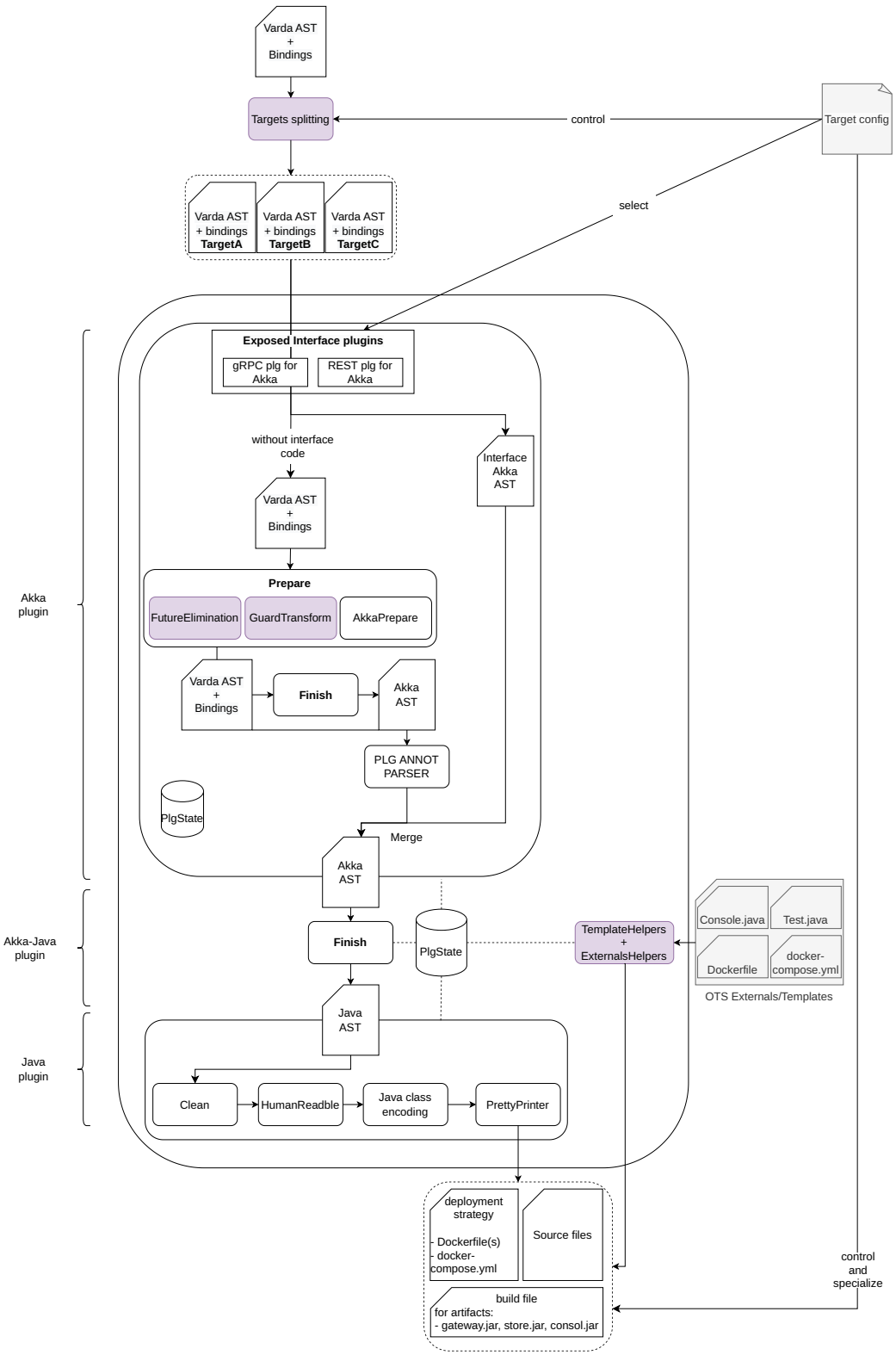


Fig. 7.4.: Architecture of the AkkaJava code-generator plugin. Purple blocks are target-independent.

can use any deployment tool (e.g., Docker, Kubernetes, Ansible, etc.) to manage the artefacts.

Varda can partially automate the generation of the deployment instructions leveraging the template mechanism. For all the examples of this document, we generate Docker Compose instructions[9]. Note that, the deployment templates can contain instruction to set up external services. For instance, we add the deployment of OTS services (i.e., Redis, Kafka and RabbitMQ) inside those templates.

7.4 Understandability

Top-down approaches that generate code, like Varda does, may result for programmers in a loss of control on the development workflow and in difficulties to track bugs or performance bottlenecks. According to the interviews, we conducted (see. Chapter 1), it is one of the main reasons why system programmers do not use the code generation.

To address this, we design Vardac to help them to easily blame an architecture piece (or an adaptor) for a generated code behaviour. Vardac generates human-readable code with additional compilation provenance information. Vardac maintains provenance information for each piece of the ASTs throughout the compilation pipeline. This information reflects the origin of the language entities, ranging from component definition to expression and types. The provenance information is injected in the generated code as comments. The provenance definition contains the source[10] of the entities and (optionally) the major transformations applied to them.

The remaining burden of a developer is to bind the system observables with the generated code to track bugs or performance bottlenecks. Currently, Varda only provides rustic and limited tools to tackle these issues. Note that, by design, Varda allows programmers to use their favourite tools to track bugs and profiles artefacts since the generated code can be loaded as a classical code project. They could also inject arbitrary debugging and profiling codes in the Varda architecture by leveraging abstract ghost methods. Moreover, to avoid drowning programmers in the analysis of the mass of the generated code, Vardac provides a basic tracing

[9] To use Kubernetes instead of docker compose, the developer has to replace the docker-compose.yml template file with a Kubernetes one.

[10] The source is either location into a Varda source file (resp. in a usage adaptor definition) or a compilation pass that creates the entities.

Feature	Main compiler	Akka plugin	Varda library	Comments
Component				
Communication ports	✓	✓	-	
Supervision ports	✓	✓	-	
Other component entities	✓	-	-	
Communication	2 KLoC	600 LoC		
Receive	✓	-	-	
Branch	✓	-	-	
Recursive protocol	✓	-	-	
Multicast (low/weak)	✓	-	-	
Event auto-boxing	✓	-	-	
Expressions				
Local error propagation	✓	-	-	
Standard library	✓	✓	✓	

Tab. 7.2.: Status of Vardac support for core Varda features.

mechanism that highlights the execution of the Varda building blocks[11]. Eventually, Vardac instrument the code to collect some metrics[12].

7.5 Vardac support

The Tables 7.2-7.5 show the status of the support of the Varda features by the compiler. They distinguish between the *main compiler* that is targeting independent, the *code generation plugin* (e.g., AkkaJava) and the *Varda standard library*[13]. Additionally, we indicate a subapproximation of the core development efforts in terms of LoC[14].

[11] Currently, those blocks are: components, methods and protocol branching.
[12] Currently, we use it to collect session performance metrics.
[13] A plugin can add new functions and change the implementation of existing ones. However, it cannot alter, nor delete, the existing interface of the standard library.
[14] LoC exclude comments and blank lines. We omit quantification when the support is transverse and scattered to too many parts of the compiler. We also omit all the helper/utility code (e.g., lexer, parser, etc.) and the cost of translating from an AST (e.g., Simple Varda) to another (e.g., Akka).
[15] This only counts the cost of implementing the type checker and the type inference with some helper functions.

Feature	Main compiler	Akka plugin	Varda library	Comments
Network				
Classical FIFO channel	✓	✓	-	
External service-base channel	-	✓	-	
Encrypted channel	-	✓	-	
Discovery primitives	✓	✓	-	
Placement	300 LoC	518 LoC (runtime)		
Remote spawn	✓	✓	-	
Placement index	-	✓	-	
Infrastructure representation	✓	-	-	
Reflexivity primitives	✓	✓	-	
Integration with OTS	1,2KLoC			
Adaptor	✓	-	-	
Abstract/hidden type and state	✓	-	-	
Exposed API		3 KLoC		
Expose annotations	✓	-	-	
REST/gRPC generation	-	✓	-	

Tab. 7.3.: Status of Vardac support for system entities and features.

Feature	Main compiler	Akka plugin	Varda library	Comments
Interception	3,7 KLoC			
Base interception	✓	-	-	
Nested interception	✓	-	-	
Interception of an interceptor	✓	-	-	Compared to nested interception, the communications between the interceptor and the instances it manages are also intercepted
Explicit un-intercepted channel	✓	-	-	
Inlining	1,6 KLoC			
Base inlining	✓	-	-	
Nested inlining	✗	-	-	Not tested
Inlining an interceptor	✗	-	-	Not yet implemented
Interception of inlining	✓	-	-	Interception is transparent to the existing components
Derivation	700 LoC	-	-	
RPC	✓	-	-	

Tab. 7.4.: Status of Vardac support for architecture transformation.

Feature	Main compiler	Akka plugin	Varda library	Comments
Encapsulation	✓	✗	-	The code generation does not handle the method visibility yet, i.e., arbitrary injected Java code could breach the encapsulation.
Isolation	✓	(✓)	-	Currently, the Akka code-generator plugin does not provide unforgeability of component reference since Akka implementations does not provide actors reference unforgeability yet. It should be released in Akka 3.
Type system	2,5 KLoC[15]			
Base type system	✓	✓	-	This includes polymorphism and subtyping.
Interface subtyping	✓	✗	-	Encoding in the typed Akka is not implemented yet.
Type reconstruction	✓	-	-	
Type checking	✓	-	-	
Constraints				
Contracts	✓	-	-	
Ghost elimination	✓	-	-	
Guards	1 KLoC	300 LoC		
Protocol guards	✓	✓	-	
Timers	✓	✓	-	
Metadata guard-context	✓	✓	-	

Tab. 7.5.: Status of Vardac support for verification toolbox.

Part IV

Validation

This last part of the thesis presents an empirical evaluation that demonstrates the usability of Varda for distributed system programming. Our main focus is to provide a proof of existence of Varda the ecosystem, how it can be used in practice, and what are its benefits and limitations.

First, we perform a qualitative analysis of Varda the framework in the Chapters 8-9. The Chapter 8 shows the gain in expressiveness and conciseness obtained through the use of Varda. On the one hand, we encode common communication (e.g., streams, pub/sub mechanism) and distribution patterns (e.g., two-phase commit, access control or metadata piggybacking). On the other, we show how to transparently apply those patterns to the existing system to make them evolve. Chapter 9 illustrates how to incrementally build a distributed system by composition on a real use case. We implement a geo-distributed database [@14]: starting from a single node database, made of various OTS component plugged together, to geo-distribution.

Secondly, we perform a light quantitative analysis in Section 9.2 to evaluate gain in conciseness and performance overhead of using Varda. Note that, we do not aim to provide a comprehensive performance evaluation of Varda, since the compilation pipeline is not optimised yet.

Both chapters demonstrate the integration of Varda in a real ecosystem. We integrate five different OTS into an Varda architecture (Kafka, RabbitMQ, Redis, RocksDB and a Java CRDT library). Conversely, we illustrate how a code-generated system can be integrated in an existing ecosystem not aware of Varda: for the different use cases we generate either REST or gRPC API with additionally external benchmarks (e.g., YCSB), observable testing or console clients that are written in Java without any knowledge of the existence of Varda.

The examples used in the evaluation, the scripts to launch the evaluation and the scripts to parse the results are all available at the following url: `https://gitlab.lip6.fr/lprosperi/Lg4DC`

8 Varda at work: Classical communication and distribution patterns

This chapter shows how to encode classical communication and distribution patterns in Varda. We first present the patterns that can be achieved with core Varda (i.e., with protocols, ports and channels) either for point-to-point (Section 8.1) or group communication (Section 8.2). Moreover, we introduce a syntactic sugar which allows programs to perform remote call of a method belonging to another component without having to handle the communication by hand (Section 8.1.2). Then, we show how to use interception to transparently impose an arbitrary communication pattern in between a set of components (Section 8.3). We denote this new pattern as a virtual network layer since it belongs to the Varda realm and not on the network layer.

8.1 Point to point communication patterns in Varda

8.1.1 Streams

Streams are common point-to-point communication primitives to model a possibly unbounded sequence of data. Streams are often used in data processing scenarios where large volumes of data need to be processed or transformed. They interconnect the different operators that process the data incrementally [33].

Varda provides native support for streams using recursive protocols. Moreover, it offers a simple and linear way to handle streams by making the asynchronous `receive` on a stream iterable.

The Listing 17 shows a scenario where a component `A` sends a `ping` message to a component `B`, which in returns replies with a random number of `pong` messages. We model communication by "`protocol pingpong = !ping (?pong)*;`". Recall that

```
428 protocol stream_pong = (?pong)*;
429 protocol pingpong = !ping stream_pong;
430
431 component A {
432     port p_pong expecting (dual pingpong) = handle_pong;
433
434     result<void, error> start( channel<A, B, pingpong> chan, activation_ref<B> b) {
435         pingpong s = initiate_session_with(chan, b);
436
437         stream_pong s1 = fire(s, ping())?;
438
439         (* Receive the first pong *)
440         for(tuple<pong, stream_pong> tmp in receive(s1)){
441             (* Custom logic unrolling one round of the protocol *)
442
443             (* Set the session for the next round *)
444             s1 = tmp.1;
445         }
446
447         return ok(());
448     }
449 }
```

Listing 17

Kleene star "$*$" operator states that the protocol can either terminate or iterate any number of times.

After sending the initial `ping` message, the code of **A** (lines 440-440) processes each round of the stream in order[1]. To avoid having to use callbacks and to lost the interest of the stream, **A** iterates over the **receive**.

At the end of the loop body (line 444), the programmer should explicitly set the value of the session for the next round, if any. Computing the continuation cannot be fully automated since the body of the loop may use arbitrary logic to process a round. For instance, such logic can unroll an arbitrary number of rounds or delegates the execution to an external function. The following snippet illustrates this:

```
void process(tmp){
    (* Expects to receive two consecutive pongs *)
    tmp = receive(tmp._1);
    tuple<pong, stream_pong> another_var = receive(tmp._1);
}

for(tuple<pong, stream_pong> tmp in receive(s1)){
    process(tmp);
}
```

[1]Remember that Varda guarantees that messages of a same session are always processed in the order. To avoid synchronizing the rounds, programmers have to emulate the stream through supervision ports. In that case, the communication logic is defined outside the Varda perimeter thanks to OTSs and adaptors.

where the process function expects to receive two consecutive `pong` messages before handing over to the main loop. Remember that a session can be consumed at most once by a session primitive (or a listening port). Therefore, the value of `s1` updates to value of `another_var` (Line 453) before continuing iterating on the stream.

Our prototype does not yet trigger session termination notifications when iterating on streams. For non-recursive protocol, the termination is inferred without notification when the protocol of the continuation is the empty protocol "`.`".

Varda streams can have strict bounds. For instance, a stream of `pong` messages can be bounded to a given number of rounds. This is done by defining a protocol guards that are evaluated in each round of the stream. Section 6.1.2 discusses how to write that kind of guards. If the guard predicate fails, the stream is terminated and the session triggers a runtime error.

8.1.2 Transparent RPC

Varda can relieve programmers from writing the communication logic when they are equivalent of remotely calling a method of another component. To motivate this feature, we compare the work of using a shared counter in Varda without (Listing 18) and with transparent RPC (Listing 19).

In core Varda, all the communication use message passing through sessions. Therefore, to use a shared counter between components, the developer needs to write the following communication logic. It has to represent the counter as a stateful component `Counter` and defines a protocol that supports both increment and read operations. Furthermore, for each component `C`, which might want to use the counter: it has to bind a channel between the `Counter` and `C`, and write the corresponding communication logic.

To avoid writing this cumbersome code, Varda provides a transparent RPC mechanism. A component can do remote method calls by using the same syntax as local calls. Remote calls are asynchronous, the caller can process incoming events in between the remote call and the return of the call.

```
protocol p_shared_counter = +{
    l_incr: .;
    l_read: ?int.;
};

component Counter {
    (* Listen for either an increment or a read request *)
    passiveport p_in expecting (dual p_shared_counter) = this.onOperation;
    int counter = 0;

    (*  Note that the logic of both [incr] and [read]
        depends on the nature of the [counter].
        It can be a local state, as in this example, or
        an entry in a remote DB, for instance.
    *)
    void incr(){
        this.counter = 1 + this.counter;
    }

    int read(){
        return this.counter;
    }

    result<void, error> onOperation (blabel op, p_shared_counter continuation) {
        branch continuation on op{
            | l_incr => s -> {
                (*  Increment the counter
                    The continuation [s] has type [.]
                *)
                this.incr();
            }
            | l_read => s -> {
                (*  Get the value from the internal logic of the [Counter]
                    Send the value to the caller
                    The continuation [s] has type [!int.]
                *)
                fire(s, this.value());
            }
        }
    }
}

component C{
    (* Port to communicate with the counter *)
    activeport p_counter expecting p_shared_counter;

    onStartup(channel<Counter, C, p_shared_counter> chan, activation_ref<Counter> counter){
        (* [chan] is the channel interconnecting [C] and [Counter] *)
        bind(this.p_counter, chan);

        (* Read and print the value of the counter *)
        p_shared_counter s = initiate_session_with(chan, counter);
        ?int. s = select(s, l_read)?;
        print(receive(s)._0);
    }
}
```

Listing 18: Manually sharing a counter between all the instances of the `C` component. Note that the `+{...}` is the dual of the notation of `...` which means that the first notation denotes the selection of a non-deterministic branch whereas the latter denotes the non-deterministic wait for the notification of the selection of a branch.

```
component Counter {
    int counter = 0;

    void incr(){
        this.counter = 1 + this.counter;
    }

    int read(){
        return this.counter;
    }
}

component C {
    onStartup (activation_ref<Counter> c){
        (* Remote call *)
        c.incr();
        print(c.read());
    }
}

(* Instruct the compiler to generate the RPC boilerplate code and
to rewrite all the remote calls to [Counter]. *)
@@derive rpc<Counter><>();
```

Listing 19: Counter exposing RPC primitives

Varda supports this feature by providing an optional and parametric architecture rewriting transformation. The `@@derive rpc<Counter><>();` statement instructs the compiler to generate all the boilerplates to support RPC calls on `Counter` methods and to rewrite all the remote method calls, performed on a `Counter` instance, to message-passing. Vardac generates the ports, the channels, the session, and the communication logic. Note that, the compiler will reject all the remote call on non-rpc methods.

Our prototype lacks an annotation to specify the visibility of component methods. A classical component, with no RPC support, has all its methods private and encapsulated. A RPC component, on the other hand, have all its methods exposed to remote calls.

Listing 19 shows the RPC version of our shared counterexample. In this version, message-passing communication is completely hidden from the programmer. In this example, using RPC derivation instead of manual communication logic decrease the number of codes to write by a factor of two.

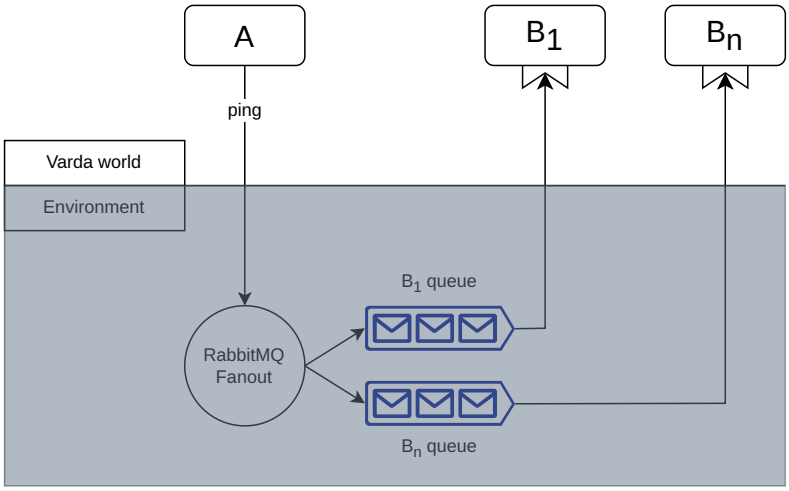


Fig. 8.1.: The low-level multicasting is implemented using the Varda external communication primitives.

8.2 Group communication patterns in Varda

8.2.1 Broadcast and multicast

Compared to point-to-point method, multicasting and broadcasting[2] avoid sending the data to each recipient one by one. The sender sends, exactly once, the data to a special multicast (resp. broadcast) address. Different layers of the OSI model provides multicasting (resp. broadcasting) primitives. For instance, this ranges from application-level multicast to the IP protocol broadcast address (local network) to wireless links broadcast.

Varda could embed multicasting (resp. broadcasting) at two abstraction layers, depending on the application requirements: low-level multicasting using external communication primitives (Figure 8.1) or high-level multicasting using sessions. The former approach seeks to maximise performance whereas the latter focus on providing additional safety guarantees by encoding the pattern in the protocol. Currently, Varda protocol only supports binary session types therefore we defer the discussion of the latter to future work.

Varda components can also leverage low-level multicasting primitives, external to the Varda programming model, to take advantage of the performance or of the scalability they offer. For instance, the former can leverage an external broker or the IP broadcast address at the cost of losing all Varda communication guarantees.

[2]Broadcasting is a method of transferring a message to all recipients simultaneously. Multicasting is a method of transferring a message to a group of recipients simultaneously.

```
(* Abstract method *)
result<void, error> my_broadcast_enroll(activation_ref<any> a);
result<void, error> my_broadcast(string msg);

component A {
    onStartup() {
        (* Broadcast "Hello" to all enrolled instances of B *)
        my_broadcast("Hello");
        return ok(());
    }
}

component B{
    onStartup (){
        (* Join the broadcasting group *)
        my_broadcast_enroll(current_activation());
    }

    supervisionport broadcast_msg = e : broadcast_msg ->
        print("Received broadcast message");
}
```

Listing 20: A broadcast `"Hello"` to all running `B` components.

This multicast provides the same behaviour as classical multicast: it uses one multicast address and it is performed in one operation without session management. However, it is not covered by Varda communication guarantees.

The idiomatic way of proceeding is as follows:

- The source calls a custom multicasting primitive, injected in Varda thanks to implementation bindings.
- The external code transports the multicasted message.
- In each endpoint, another piece of external code receives the message and converts it to a runtime notification.
- The runtime delivers it to the adequate supervision port of the component.

The Listing 20 and the Figure 8.1 show an example of multicasting using a hidden RabbitMQ broker. In this scenario, `A` broadcasts `"Hello"` to all the running instances of components `B`. This example leverages two abstract methods to import the broadcast logic: `my_broadcast_enroll` and `mq_broadcast`. On startup, an instance of `B` runs `my_broadcast_enroll` to join the broadcast group. Then, `A` runs `mq_broadcast` to send a message to all the enrolled instances of `B`.

Under the hood, enrolling a new instance of `B` means to creating a new queue and binding it to the RabbitMQ fanout exchange which represents the multicast address. Broadcasting a message is publishing to the RabbitMQ fanout exchange.

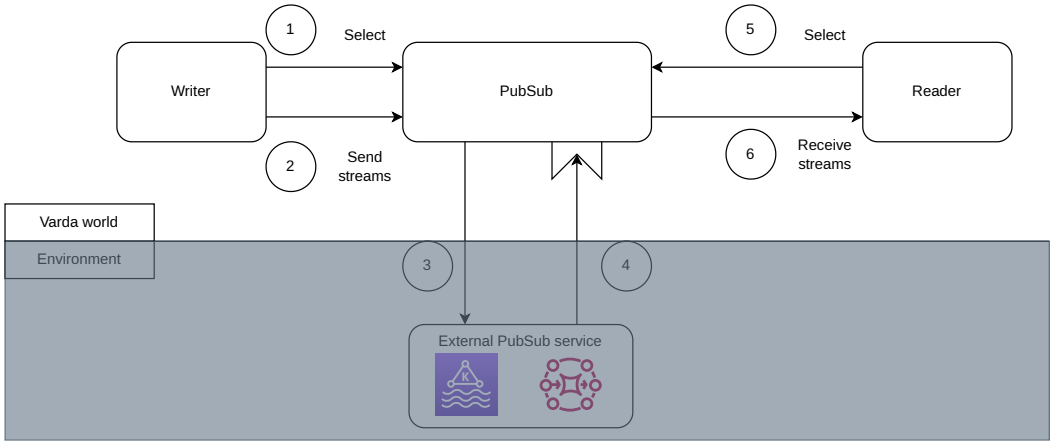


Fig. 8.2.: A Pub/Sub component built on top of an external broker.

8.2.2 Pub/sub

In the *publish-subscribe* pattern, publishers group messages in categories and publish them on a *pub/sub* infrastructure without knowing the subscribers[3]. Usually, the class of the message is either defined by the publisher (topic-based) or by the content of the message. Once the middleware receives a message, it delivers it to the group of subscribers interested in the class of the message. The semantics of delivery is often as follows: the message is delivered to at most (resp. exactly) one member of the group.

Pub/sub brokers are widely used in distributed systems. They act as intermediaries between services and decouple them. Moreover, they provide out-of-the box interesting properties as scalability, fault tolerance, and back pressure. They ease the building of a scalable, loosely coupled, and resilient distributed system architecture.

The idiomatic way to use *publish-subscribe* communication in Varda is to use an OTS *PubSub* component that models the broker infrastructure (Figure 8.2). Other Varda components publish and subscribe using session communication primitives.

With respect to the broker, the `PubSub` component acts as a subscriber for each Varda subscriber and as virtual publisher for each Varda publisher. Its implementation uses an external broker thanks to Varda adaptors. For instance, it can be a RabbitMQ or Kafka based implementations.

In Listing 21, we propose a generic topic-based `PubSub` component. It supports a `pubsub_protocol` (Line 561) that encompasses both the subscription and the publi-

[3]Note that publisher does not send messages directly to subscribers.

cation protocols. Hence, a subscriber component (resp. publisher) is a component that has established a session with the PubSub component and that has selected the sub branch (resp. pub).

Let's follow the path of a message from a subscriber to a publisher.

1. When a publisher wants to publish a message, it creates a session, then it selects the pub branch and it sends the message.
2. The PubSub component receives the topic and the message (Lines 583-589). Then, it publishes the message on the broker using the abstract OTS method pubsub_publish[4].
3. **At this point, the message leaves the realm of Varda.** The broker delivers the message to one of its internal queue.
4. Since the PubSub component acts as a virtual subscriber with respect to the broker. This queue sends the message to the brocker-subscriber logic of the component. This logic is brocker-specific, therefore it is implemented in an adaptor and not exposed in the shield interface.
5. On reception, this adaptor transforms the message to a Varda supervision notification.
6. The runtime delivers it to the supervision port of the PubSub component.
7. **The execution comes back into Varda realm.** On notification on the supervision port, the PubSub component forwards the message to the corresponding component subscriber (lines 616-623). It computes the retrieves the subscription stream toward the corresponding subscribers thanks to its internal state active_subscribers, which map the topic name to the active set of subscribers.

We use a centrealized PubSub component. Programmers can replicate this components as long all instance use the same broker, as depicted by Figure 8.3. Furthermore, since this PubSub pattern adds an extra message-passing layer between components, an interesting optimization is to inline an instance of the PubSub in each component that use it.

[4]In our implementation, we use a RabbitMQ adaptor.

```
559 event topic of string;
560 event msg;
561 protocol pubsub_protocol = +{
562     (*  Publish a message on a topic
563         - Step1: the publisher selects the pub branch
564         - Step2: the publisher sends the message to publish *)
565     pub: !topic!msg.;
566     (*  Subscribe to a topic
567         - Step1: the subscriber selects the sub branch
568         - Step2: the subscriber listen for a stream the messages *)
569     sub: ?topic (μx. ?msg-x);
570 }
571
572 component PubSub {
573     (*
574         The [PubSub] component maintains a map between
575         the [topic] and the set of the actual subscribers.
576         The set contains subscription streams. *)
577     dict<topic, set<μx. !msg-x.>> active_subscribers = dict();
578
579     (*** Interface with the other components ***)
580     passiveport pub expecting pubsub_protocol = this.callback;
581     result<void, error> callback(blabel label, pubsub_continuation continuation) {
582         branch continuation on label {
583             | pub => continuation -> {
584                 (* Receive the topic and the message to publish *)
585                 topic t, stream_msg stream = receive(continuation);
586                 (msg, stream) = receive(stream);
587                 (* Call the OTS interface to publish it *)
588                 pubsub_publish(t, msg);
589             }
590             | sub => continuation -> {
591                 (* Receive the topic to subscribe for the sender of [s]. *)
592                 topic t, (dual stream_msg) stream = receive(continuation);
593                 (* Register the output stream *)
594                 if(exist2dict(this.active_subscribers, t)){
595                     add2set(get2dict(this.active_subscribers, t),stream);
596                 } else {
597                     add2dict(this.active_subscribers, t, set(stream));
598                 }
599                 (* Register the current [PubSub] instance as a subscriber of topic [t] *)
600                 pubsub_add_suscriber(t, vid);
601             }
602         }
603         return ok(());
604     }
605
606     (*** Interface with the OTS broker ***)
607     result<void, error> pubsub_publish(topic t, msg m);
608     result<void, error> pubsub_add_suscriber(topic t, virtual_id vid);
609
610     (*  This supervision port listens for messages from the broker.
611         Those messages correspond to message delivered by the broker
612         to a topic watched by the current [PubSub] instance.
613
614         On reception, the onIntermalMsg compute the stream session associated
615         to the subscriber and forwards the message.  *)
616     supervisionport sub expecting tuple<topic, msg> = this.onInternalMsg;
617     supervision onInternalMsg(tuple<virtual_id, msg> tmp) {
618         (*  Get the stream session associated to the subscriber
619             Pick a random session among [active_subscribers] for the topic [t] *)
620         stream_msg stream = pick(get2dict(this.active_subscribers, tmp.0));
621         (* Send m to the subscriber and update the internal state *)
622         add2dict(this.active_subscribers, tmp.0, send(stream, tmp.1)?);
623     }
624 }
```

Listing 21: Topic-based `PubSub` component.

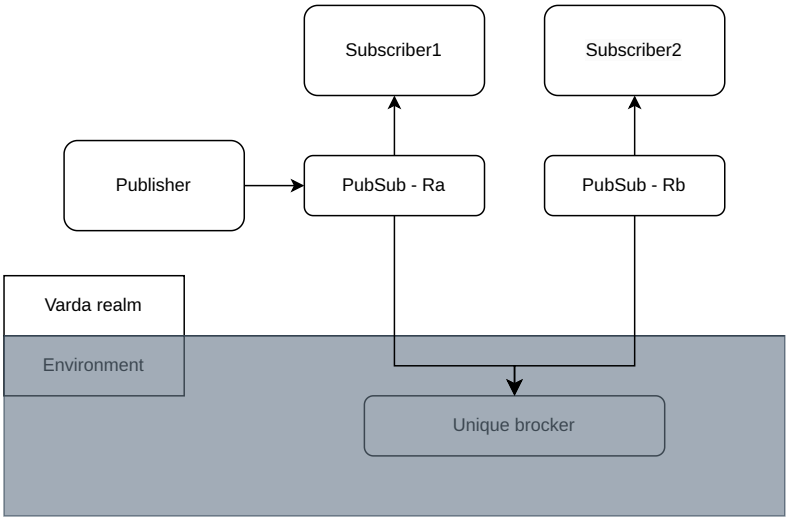


Fig. 8.3.: `PubSub` replication.

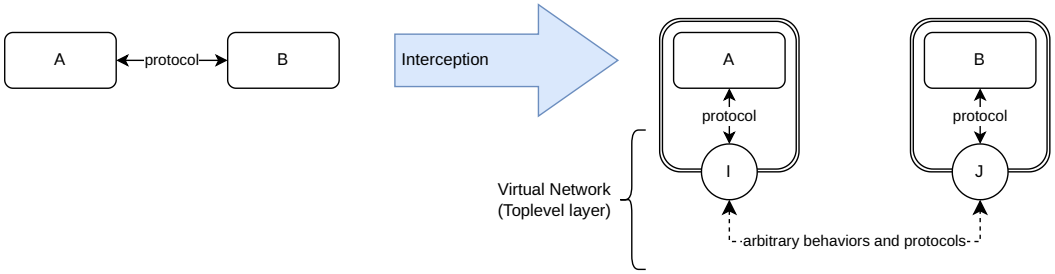


Fig. 8.4.: Building virtual network with interception.

8.3 Updating the communication topology with virtual networks

In the previous sections, we show how to onboard/encode classical communication patterns in Varda. Now, we review how to transparently transform an existing communication topology. The interception mechanism can transparently impose a virtual network layer in between a set of components. to arbitrarily modify the communication pattern. Figure 8.4 shows the general transformation consisting of adding a virtual network between two components `A` and `B`. The first step is to intercept `A` and `B` with, respectively, `I` and `J`. Then, the communication between `A` and `B` is routed through `I` and `J`. The virtual network denotes the network between `I` and `J`.

The programmer can implement any communication pattern between `I` and `J` using Varda communication primitives. Note that, a virtual layer may use other virtual networks under the hood leveraging nested interception. Conversely, a virtual network layer could use an external system to transport communication thanks to OTS and adaptors.

In the following, we review two applications of virtual networks: access control and metadata piggy packing.

8.3.1 Dynamic access control

Access control refers to the mechanisms that regulate and manages access to services and resources within a distributed system. It involves authentication and authorisation, which determines the level of access of the authenticated applicant. Access control can be enforced at different layers of the OSI model. In this section, we show how to enforce simple access control at the application layer such that the access control is aware of the Varda semantics.

Dynamic access control in Varda means that the programmer can prevent communication (i.e., by disallowing session creation) or can filter the protocol (e.g., by disallowing some branch or restricting message values) according to the identity of the communicating components or based on the history of the communication.

A programmer can transparently impose access control to a set of components by intercepting those components by a user-defined `AccessController` component (Figure 8.5). This is a simple form of virtual network where there is a single interceptor since the protocol remains unmodified. Optionally, each component can have its dedicated `AccessController`, possibly inlined, to enforce access control locally and independently of other components.

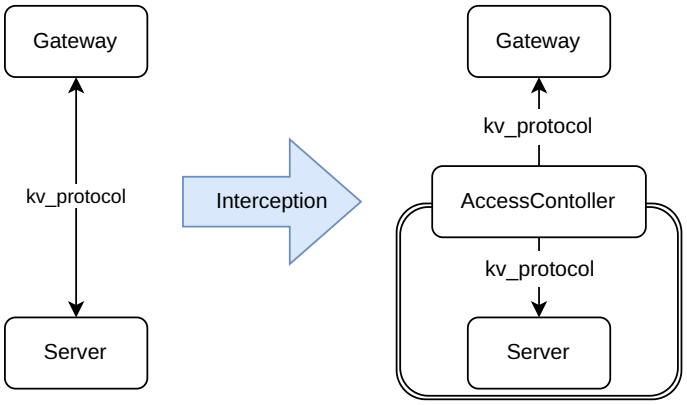


Fig. 8.5.: Adding access control when accessing `Server`.

Network-layer access control cannot be achieved by virtual networks. To work at network layer (resp. syscall layer), programmers must use existing and external tools like host firewall (e.g. iptables) or service-mesh (e.g. Istio or Linkerd) in a containerzed environment. Those tools will also constrained the runtime and loose the knowledge of the Varda semantics.

Let us take a version of our running key-value store example with a protocol supporting *get*, *put* and *delete* operations without streams. The Listing 22 specifies access control for a "`Server`" such that

- Only gateways running in a private network can perform *delete* operations;
- *put* operations are restricted for gateways running on public nodes. Those gateways cannot update security-related keys;
- *get* operations are unrestricted.

To impose access control to a set of `Server`, the programmer encloses them into an interception scope (Lines 632-635). Since the access control logic does not require shared state, the controller is replicated for each server to avoid introducing a bottleneck. The `one_controller_per_server` policy starts a new controller for each intercepted components[5]. The interception block exposes the identity of the intercepted components[6]. Hence, the gateway can distinguish between the identity of both servers.

[5]Another strategy is to start a single controller per place.
[6]Since we do not use the *anonymous* modifier to parametrize the intercept block.

```
protocol p_protocol = &{
    l_get: !key?value.;
    l_delete: !key?bool.;
    l_put: !tuple<key,value>?bool.;
};

(* Protect two servers *)
intercept<AccessController> one_controller_per_server {
    spawn Server(...) @ ...;
    spawn Server(...) @ ...;
}

component AccessController {
    set<session_id> unauthorized_sessions   = set();

    (* This predicate checks if the activation [a], i.e. a gateway,
       is runing either in the local or on the public network. *)
    bool is_public(activation_ref<any> a){
        (*  For simplicity, we checked that the ip of the node runing [a] is
            in a certain range of IP addresses [public_ips].
        *)
        return exist2set(ip(place(a)), public_ips);
    }

    (* Intercept each session creation *)
    @msginterceptor(both)
    result<.,error> intercept_choice(
        activation_ref<A> from, activation_ref<B> to,
        (dual p_protocol) continuation_in, p_protocol continuation_out,
        blabel msg
    ){
        branch continuation_in on msg {
            | l_get => s -> { (* Nothing to do, allowed for all *) }
            | l_delete => s -> {
                if(set2exists(this.public_instances, activationid(from)))
                {
                    (*
                        The error will be logged by the interceptor.
                        The Client will be notified when it will send the key,
                        to preserve session guarantees.
                    *)
                    return err("Message can only be deleted by trusted Client");
                }
            }
            | l_put => s -> { (* Nothing to do at this point since the key is not
            ↪  known *) }
        }
    }

    (* Intercept each put request and check if the key can be updated *)
    @msginterceptor(both)
    result<?put_response., error> intercept_put_request(
        activation_ref<A> from, activation_ref<B> to,
        !put_response. continuation_in,
        !put_request?put_response. continuation_out,
        put_request msg
    ){
        key k = msg._0;
        (* "private" key can only be updated by trusted Client *)
        if( key in {"private"} && this.is_public(from) )
            return ok(fire(continuation_out, err("Unauthorized update from
            ↪  outside"))?);
    }
}
```

Listing 22: Adding access control for "`Server`"

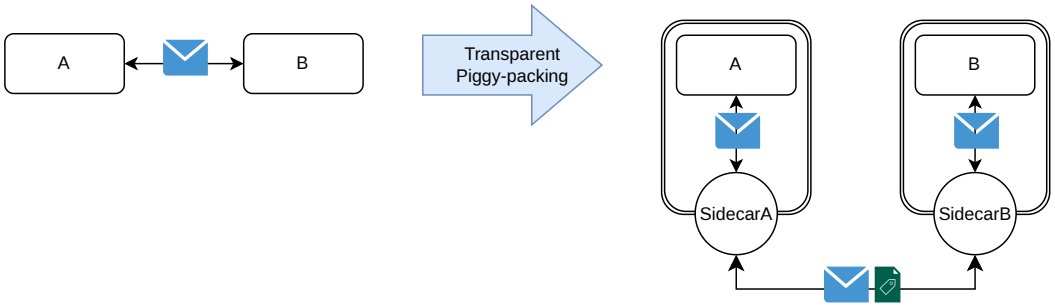


Fig. 8.6.: Piggy-backing metadata

The access control logic is embedded into the "`AccessController`" (Lines 637-686). The controller performs the filtering in two steps. First, it prevents unauthorised deletion when the gateway selects a branch of the protocol (Lines 651-671). Then, it filters update on private keys upon reception of a `put_request` (Lines 675-685).

This `AccessController` can protect any kind of components that use the key-value stores protocol. Namely, it can either protect the `LoadBalancer` or protect each `Server` when a `LoadBalancer` calls them.

8.3.2 Encapsulating messages and piggy-backing metadata

Using the virtual network, programmers can piggy-back metadata to messages (resp. encapsulate messages) without updating the components. Piggy-back metadata can carry consistency, provenance or debugging information.

For instance, let us take a ping-pong application (at the left of Figure 8.6) where component "`A`" sends a "`ping`" message to component `B`, which replies with a "`pong`". The protocol between `A` and `B` is as follows "`!ping?pong.`". The use case is to piggy back the metadata of user-defined type `meta` to the "`ping`" (resp. `pong`) message without modifying neither `A` nor `B`.

To achieve this, the programmer adds a virtual network, illustrated by the right-hand side of Figure 8.6. This virtual network leverages Varda direct communication primitives with a modified protocol that support metadata: "`!tuple<ping, meta>?tuple<pong, meta>.`".

`SidecarA` and `SidecarB` intercept, respectively, `A` and `B`. The two sidecars are connected by a channel supporting the modified protocol. On the one hand, `SidecarA`

intercepts the `ping` message, piggy-backs metadata to it and transfers it to `SidecarB`. Then `SidecarB` unwraps the message, processes the metadata and delivers the `ping` message to `B`. The `pong` message is processed in the same way.

The Listing 23 shows the implementation of the two sidecars and explains how to set up the inter-sidecar communication. To illustrate how it works, let's follow the path of the `ping` message from `A` to `B`.

Phase 0 `A` sends a `ping` message to `B`.

Phase 1 - `ping` interception `SidecarA` intercepts the message and processes it using the `intercept_ping` method (Lines 690-696). This method delegates the redirection to the `route_ping` method.

Phase 2 - inter-sidecars routing This routing logic compute the destination of the message (i.e., the identity of `SidecarB`) and it creates a session, with metadata support, with the destination sidecar. Then, it wraps the `ping` message in a `tuple` to carry the metadata and it sends it through the session.

Phase 3 - reception `SidecarB` receives the message through the `p_in_side` port and it processes it with `handle_side_ping` method. For simplicity, this toy method discards the metadata and delivers the message to `B`.

Phase 4 - delivery to `B` The final delivery is quite tricky since there is no established session between `SidecarB` and `B` at this point. It is up to `SidecarB` to initiate the session.

To initiate a session with `B`, `SidecarB` uses the virtual `p_inner_out` port. Compared to classical ports, its up to Vardac to bind it with the correct channel. Here `p_inner_out` is bound to the interception channel that interconnects `SidecarB` with `B`.

With the current prototype, a programmer cannot yet write an arbitrary piggy-backing interceptor that works for any kind of components since there is no meta-programmation primitives that permit to rewrite protocols: i.e., to take a protocol in input (e.g. `!ping?pong.`) and to output a new protocol (e.g. `!tuple<ping, meta>?tuple<pong, meta>.`). Therefore, programmer has to write a piggy-backing interceptor for each protocol. Note that an interceptor can support multiple protocols.

```
component SidecarA {
    (** Interception of A communication **)
    @msginterceptor(both)
    result<?pong.,error> intercept_ping(
        activation_ref<A> from, activation_ref<B> to,
        !pong. continuation_in, !ping?pong. continuation_out,
        ping msg
    ){
        this.route_ping(continuation_in, msg);
    }


    (* Side channel that interconnects both sidecars with the modified protocol *)
    activeport p_out_side expecting p_intermediate;
    onStartup (channel<SidecarA, SidecarB, p_intermediate> chan) {
        bind(this.p_out_side, chan);
    }

    (*  Inter-sidecar logic
        It piggy pack a metadata to the ping message.
        Then, it sends this new message to the other sidecar
        through the inter-sidecars channel.
    *)
    result<void, error> route_ping(!pong. inner_s1, ping msg){
        set<activation_ref<SidecarB>> sidecars =
        ↪  rightactivations(channel_of(this.p_out))?;
        activation_ref<SidecarB> dest = set_pick(sidecars);
        session<p_intermediate> outer_s0 = initiate_session_with(
            this.p_out, dest);
        ?pong. outer_s1 = fire(outer_s0, (msg, meta(1)))?;
        tuple<tuple<pong,meta>, .> res = receive(outer_s1);
        . outer_s_end = fire(inner_s1, (res._0)._0)?;
    }
}

component SidecardB {
    (** Interception of B communication **)
    @msginterceptor(both)
    result<?pong.,error> intercept_pong(
        activation_ref<A> from, activation_ref<B> to,
        !pong. continuation_in, !ping?pong. continuation_out,
        ping msg
    ){
        ?pong. s_out = fire(continuation_out, msg)?;
        return ok(s_out);
    }

    (* Side channel that interconnects both sidecars with the modified protocol *)
    passiveport p_in_side expecting (dual p_intermediate) = this.handle_side_ping;
    onStartup (channel<A, B, p_intermediate> chan) {
        this.bind(this.p_in_side, chan)
    }

    (* Virtual port that point toward B instances *)
    activeport<ingress:ingress> p_inner_out expecting (inline p_pingpong);

    (*  Inter-sidecar logic
        On reception of wrapped ping:
        it unwraps the message, processes the metadata and
        delivers the ping message to B.
    *)
    result<void, error> handle_side_ping (tuple<ping,meta> msg,
        !tuple<pong,meta>. outer_s1) {
        session<p_pingpong> inner_s0 = initiate_session_with(
            this.p_inner_out, this.replica);
        ?pong. inner_s1 = fire(inner_s0, msg.0)?;
        tuple<pong, .> res = receive(inner_s1);
        fire(outer_s1, (res.0, meta(2)));
    }
}
```

Listing 23: Piggy-packing metadata on a pingpong protocol with virtual network.

9 Use Case and Preliminary Performance Evaluation

To demonstrate the expressiveness of Varda and its ability for real distributed system programming, we model and implement in Varda a simplified version of AntidoteDB [@14], [94], a highly available geo-replicated key-value database (Section 9.1).

We build it *incrementally* starting from a single node database to a geo-distributed database. For this, we leverage Varda interception to transparently extend the system with new features without having to update the existing code base. Moreover, this chapter demonstrates the ability and the ease to compose various OTS ranging from a conflict-free replicated data type (CRDT) library to popular distributed services (e.g., Redis and Kafka).

The chapter then concludes with a brief quantitative analysis in Section 9.2 to evaluate gain in conciseness and performance overhead of using Varda.

9.1 Use Case: Step-by-step implementation

9.1.1 AntidoteDB overview

AntidoteDB supports concurrent operations over a number of data centers (DCs). Clients' operations are grouped into transactions that starts by a begin, contains a various number of data operations (e.g., read/write) and ends by a commit. Moreover, to maintain isolation and asynchrony between transactions, AntidoteDB use a form of multiversion concurrency control (MVCC) [20] to allow a client to (read and) write data without waiting. MVCC creates multiple versions of each data item, and assigning a unique timestamp to each version. A transaction only sees the version of the data that is valid at the start time of the transaction.

AntidoteDB provides horizontal scalability per DC and replication between DCs. Each DC is partitioned into non-overlapping storage servers called shards. Fig-

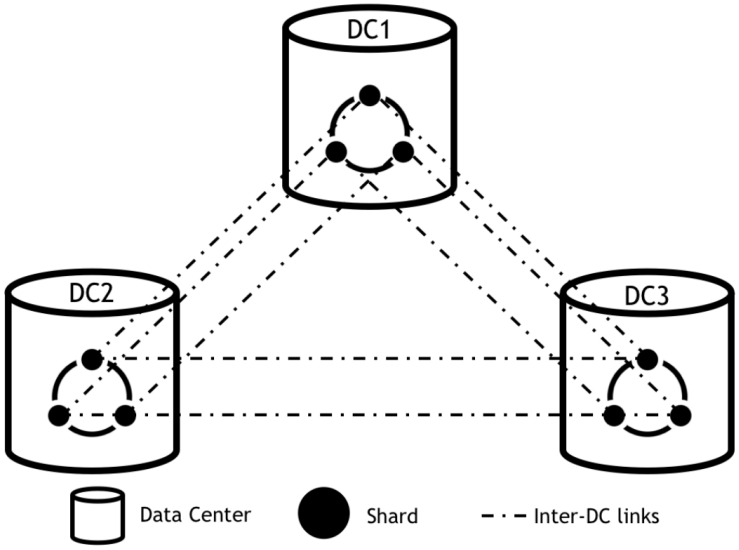


Fig. 9.1.: Overall architecture of AntidoteDB, extracted from Hatia and Shapiro [94].

ure 9.1 illustrates this overall architecture. To ensure consistency in the presence of concurrent transactions, AntidoteDB ensures

- strong consistency for commits inside each DC[1].
- transaction causal consistency between DCs [9] two handle operations that happen concurrently on different DCs. Two clients can concurrently update the database from two distinct DCs. In that case, AntidoteDB orders operations by causal order. To track causal dependencies and to maintain causal order, the authors tags each transaction with a logical clock, implemented by vector timestamp with one entry per DC.

9.1.2 AntidoteDB entities

A data center is composed of different kinds of entities to handle transactions (the *TransactionManager* and the *TransactionCoordinator*), to store the data (the *Journal*), and to support querying the state of an object in a given version (the *Materializer*). Furthermore, the *Inter-DC* entity ensures data replication across the DCs.

Transaction Manager The *TransactionManager* processes transactions from clients. There is one manager per DC. On clients' transaction creation, the *TransactionManager* creates a *TransactionCoordinator* to handle the transaction. The manager supervises its coordinators: it ensures that they terminate correctly and restart them if needed.

[1]For this, the authors use a variant of Snapshot Isolation.

Transaction Coordinator A coordinator manages exactly one transaction identified by a unique identifier and a dependency version (i.e., a vector timestamp). A transaction operation can only see an operation that is in its logical past to preserve causality, i.e., such that its commit version is older or equal to the dependency version.

Upon creation, the coordinator starts the transaction on each involved shard. A shard is involved in a transaction if it contains an object that is read or written by this transaction. Then, the coordinator sends each client's operation to the correct shard.

On commit, to ensure strong consistency inside a DC, *TransactionCoordinator* coordinates the commit of the transaction among all the involved shards using a two-phase commit protocol.

Shard In a shard, the *Journal* persists the writes whereas the *Materializer* response to the reads. Furthermore, the *Inter-DC* asynchronously replicates the persisted data to other DCs.

Journal Inside a shard, the operations that impact the state of the store (i.e., the updates, the begin, and the commit) are persisted in a log, called the *Journal*. A log is a grow-only sequence of records, ordered by timestamps. Each (internal) operation is represented by a given type of record and persisted in the journal.

Materializer On read operation, the *Materializer* computes the state of the read object at a given version from the data stored in the *Journal*. We denote by *materializer logic* the union of the Cache, Fill Daemon and Evict Daemon of Hatia and Shapiro [94]. The Cache speeds up the reads. It avoids unneeded scan of the Journal by storing the most recent versions of the objects in memory. The two daemons manage the cache either filling it form the *Journal* or evicting the least recently used objects.

Inter-DC replication Each DC has the same number of shards and each replica of a shard manages the same key domain. Hence, the *Inter-DC replication* works on a shard-to-shard basis, according to their key domain: all the updates arriving to a shard are sent asynchronously to the corresponding shard in other DCs.

Others We omit some AntidoteDB's features like the checkpoint store and the journal trimming mechanisms.

9.1.3 Method

In the following, we implement a simplified version of AntidoteDB, called VAntidoteDB, such that

- VAntidoteDB only supports simple `put` and `get` operations on objects;
- Objects are Last Writer Wins Registers (LWWRegister) [161]: a read operation on LWWRegister returns the value of the highest version.
- VAntidoteDB embeds a simplified replication process. In AntidoteDB each shared is replicated one-by-one across all DCs, whereas in VAntidoteDB, the replication works at coarse grain: the whole set of operations applies to a DC is replicated to other DCs.
- VAntidoteDB supports multi-versions and strong consistent commit per DC. However it does not support causal consistency between DCs.

We build VAntidoteDB incrementally, step-by-step, as follows:

1. We start with a single shard in a single DC.
2. We add shards without ensuring consistency between them.
3. We ensure strong consistency to commit between shards.
4. We add multi-DC support and we implement replication across them without maintaining causal consistency.

To evolve the system from one step to the next, we leverage interception such that we do not modify existing orchestration logic except when the type of protocols changes.

Figure 9.2 shows the whole logical architecture of VAntidoteDB. The `Gateway` exposes a gRPC interface to the external clients. Like AntidoteDB, VAntidoteDB split the transaction handling logic between a `TxManager` and a set of `TxCoordinators`. We implement both components logic in Varda.

The backend is composed of a `Virtualizer` that shards the operations on a set of backend servers according to the keys. A shard contains a `Materializer` and a `Journal`. We reuse the log of Kafka to implement the `Journal`. Furthermore, we shield the key-value store of `Redis` to build the `Cache`, and we embed the LWWRegister from the Akka CRDT library to represent the VAntidoteDB objects. We implement the sharding logic and the materializer logic in Varda since we did not find ease to use off-the-shelf components for them.

The `Replicator` component asynchronously replicates the operations to the other DCs. We implement the replication logic in Varda. In each DC, the replicated operations are received by the recipient `Replicator` instance which forwards them to the DC backend. We use a RabbitMQ broker to transport inter-DC communication

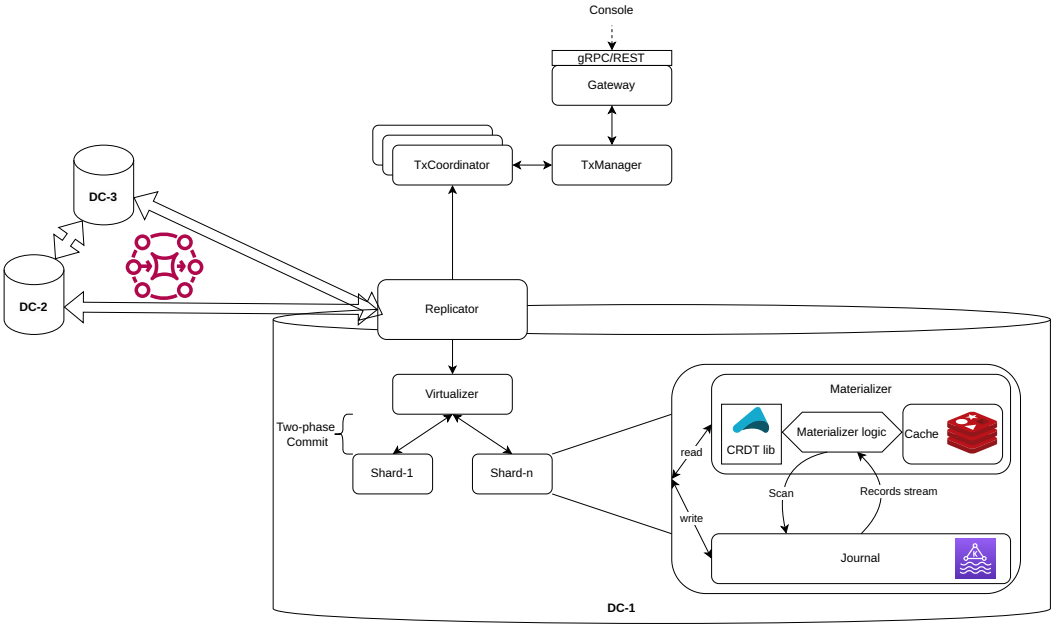


Fig. 9.2.: Logical architecture of VAntidoteDB.

to benefit from the delivery guarantees in case of network partition and the back pressure offered by RabbitMQ. Conversely, we rely on a classical FIFO channel (i.e., Akka inter-actor communication) for intra-DC communication.

9.1.4 Step 1 - Single shard, single DC, no replication, no consistency between shards

For step 1, we built the definitive[2] frontend and transaction management layers. The backend layer is simplified: it contains a single shard. However, we implement the definitive functionalities of a shard which encapsulates both a `Materializer` and a `Journal`. For the journal, we sandbox Kafka in a Docker container and wrap it as an OTS component. We implement the `Materializer` as the composition of a `Cache` and the embedded CRDT Akka library (using OTS adaptors and abstract types). We write in Varda the built-in materializer logic that integrates both the library with the `Cache`. For the cache, we sandbox Redis in a Docker container and wrap it as an OTS component.

Figure 9.3 describes the logical architecture of VAntidoteDB at step 1 with inter-connections annotated by protocols. Before delving into the technical details, we explain the flow of executing a request as follows:

[2]By definitive, we mean that the subsequent steps will not modify them.

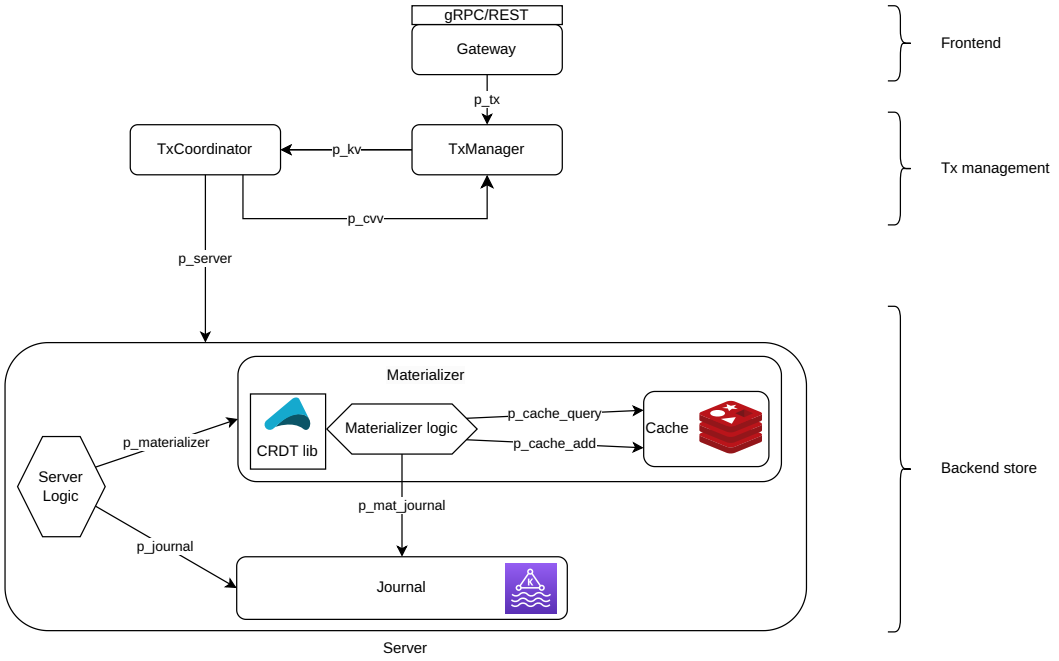


Fig. 9.3.: Logical architecture of VAntidoteDB at step 1 where arrows represent typed communication channels. Both `p_tx` and `p_kv` represent a transaction: the former one models a full transaction whereas the last one models a transaction without the `begin`. `p_cvv` is the protocol that defines the exhange between a `TxCoordinator` and its `TxManager` to select a commit version. The protocol `p_server` is the read/write protocol supported by the backend. Unlike the previous protocol, it does not refer to transaction operation but to reading or writing backend record. A backend record is a generic persistent structure that either represents a begin, an update, a commit or an abort depending of the value of its attribute. The `p_materializer` and `p_journal` protocols are respectively the read, write projection of the `p_server`. `p_mat_journal` represents the scan of the journal by the materializer on cache miss.

Execution flow of a request

The VAntidoteDB pipeline is as follows:

Starting a transaction An external client (e.g., a console) connects to the `Gateway` through a gRPC interface. It starts a transaction by sending a `begin` request to the `Gateway`. The `Gateway` assigns a unique identifier to the transaction and delegates its processing to the `TransactionManager`. The `TransactionManager` starts a `TransactionCoordinator` dedicated to this transaction. The coordinator writes, on the backend, a `begin` record with the following transaction information: the transaction id and the dependency timestamp provided by the client.

Reading an object On read requests, the `Gateway` forwards it to the `TransactionCoordinator`, through the `TransactionManager`. We instruct the compiler to inline the `TransactionManager` into the `Gateway` to avoid inter-component indirection.

There are two different cases for the read:

- *The read operation concerns a key that has not been updated by the current transaction.* In this case, the `TransactionCoordinator` forwards the read request to the backend which in returns delegate it to its internal `Materializer`.

  On reads, the materializer queries the cache. If the value is in the cache, the materializer responses back to the `Shard` with the value. On cache miss, it starts a scan of the log on the `Journal` which replies with an ordered stream of records.

  On record reception, the `Materializer` logic updates the cache. On reception of the commit record corresponding to the requested version, the `Materializer` returns the materialised values of the key.

- *The read operation concerns a key that has been updated by the current transaction.* In this case, the `TransactionCoordinator` returns the value of the last update since objects are LWWRegister. This ensures the *read-your-writes* consistency properties. For this the coordinator, maintains a local cache[3] which materialises the state of the objects updated by the current transaction.

[3] Since it is a local and transient cache, bounded by the number of objects update by a transaction, we implement it as a simple Varda map without involving an OTS.

Updating an object On write, the `Gateway` forwards it to the `TransactionCoordinator`, through the `TransactionManager`. Then the `TransactionCoordinator` updates its local cache, crafts a backend record which describes the update (i.e., the key, the value, the version and the transaction ID), and sends this record to the backend.

On record reception, the `Shard` permanently stores the record in its `Journal` which appends the record at the end of its log. Each Journal's instance uses a dedidcated Kafka topic to implement its log.

Committing a transaction On commit, the `TransactionManager` computes a valid commit timestamp and send it to the `TransactionCoordinator`. Then the coordinator crafts and writes a commit record on the backend.

Component interconnection: protocols and channels

For the sake of brevity, we only detail the `p_kv` protocol which models an AntidoteDB transaction, in between the `TxManager` and `TxCoordinator`, as follows:

```
(* [p_kv] is a recursive protocol
    it executes an arbitrary number of operations before either committing or aborting.
*)
protocol p_kv = μ y. +{
    (*
        [y] denotes the continuation of the branch
        i.e. when the session reaches the "[y] state",
        it loops an execute the protocol again.
    *)
    l_get: !key?option<value> - y;
    l_put: !tuple<key,value>?bool - y;

    (* l_commit and l_abort are terminal branches and are mutually exclusive *)
    l_commit: ?bool.;
    l_abort: .;
};
```

We do not add the `begin` operation in `p_kv` since the `TxManager` spawns the `TxCoordinator` on `begin` and passes it the dependency version and the transaction identity as arguments.

9.1.5 Step 2 - Sharding, single DC, no replication, no consistency between shards

To horizontal shard the backend while minimising the updates of the architecture, we interpose a `Virtualizer` component between the transaction layer and the servers using interception. Currently, we explicitly declare the additional shards when the system starts. This transformation is similar to the sharding of Section 4.2. Most of the work is to adapt the routing logic to handle the various cases of the transaction protocol.

9.1.6 Step 3 - Adding strong consistency between shards

To ensure strong consistent commit between shards, we use the two-phase commit protocol [87] (2PC) as in AntidoteDB. We add the 2PC between the virtualizer and the shards in two steps. On the one hand, we define a generic[4] two-phase commit protocol: `p_server`. For this, we encode 2PC as a virtual network pattern (see Section 8.3). Then, we transparently interpose this virtual network in between the `Virtualizer` and the `Shards` thanks to the interception mechanism.

The two-phase commit protocol

The two-phase commit protocol ensures atomic commitment for a transaction. This protocol uses a central coordinator to handle the synchronisation between the participants. Figure 9.4 illustrates the sequence of the protocol.

In the first phase, the coordinator asks the participants to prepare the commit. The protocol ensures that all the participants, involved in the transaction, have already persisted the transaction's updates to their stable storage.

In the second phase, if all the participants of the first phase answer with an OK, the coordinator asks the participants to commit the transaction. Otherwise, it tells them to abort. After committing, each participant persists the same commit record in the log and sends a success message to the coordinator.

[4] By generic, we mean that the protocol is not specific to the VAntidoteDB use case. Moreover, it is independent of the type, the number, and the identity of the interconnected components. It only depends of the protol on which we add 2PC.

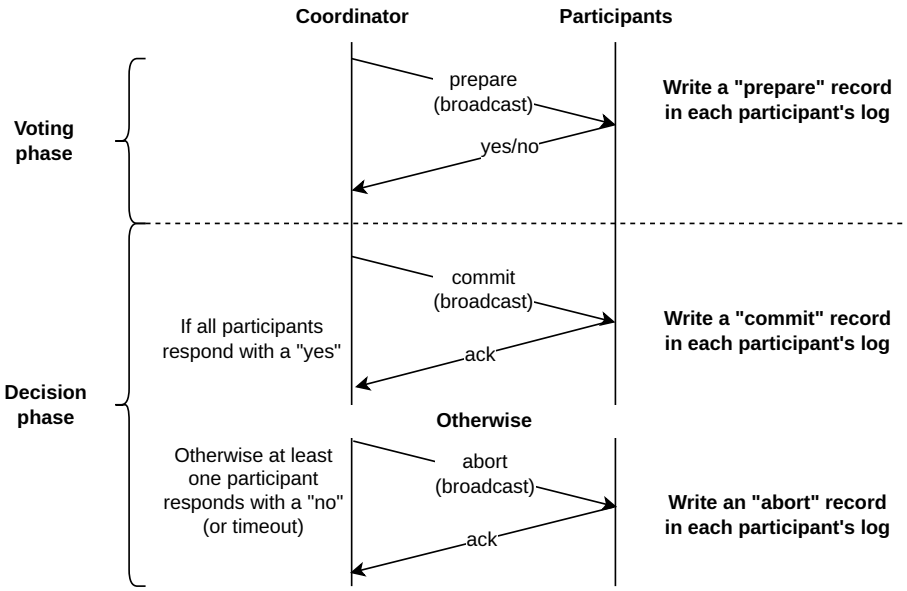


Fig. 9.4.: Two-phase commit protocol between a coordinator and a set of participants.

Two-phase commit protocol as virtual network

To encode the 2PC protocol as a virtual network, we define two new interceptors, detailed in Figure 9.5:

- `CoordinatorTwoPC` which plays the role of the coordinator in the 2PC protocol;
- `ParticipantTwoPC` which plays the role of a participant in the 2PC protocol;
- and, we also define the intermediate Varda protocol, `p_2pc`, which is used to exchange messages between the coordinator and the participants.

The idea is to interpose a `ParticipantTwoPC` in front of each shard and to add the *CoordinatorTwoPC* in front of the *Virtualizer*. Both interceptors are connected thanks to a dedicated channel typed by the `p_2pc` protocol (Listing 24). When the virtualised sends a message to a shard, the `CoordinatorTwoPC` wraps it into the `p_2pc` variant:

- On non-commit message, it behaves like the identity function;
- On commit message, it starts a two-phase commit. It starts by sending `l_pc_prepare` (Listing 24, line 782) message to all the participants. Then, it waits for the answer of all the participants. If all the participants answer with an OK, it sends a `l_pc_commit` message (Listing 24, line 783) to all the participants. Otherwise, it sends a `l_pc_abort` message (Listing 24, line 784) to all the participants.

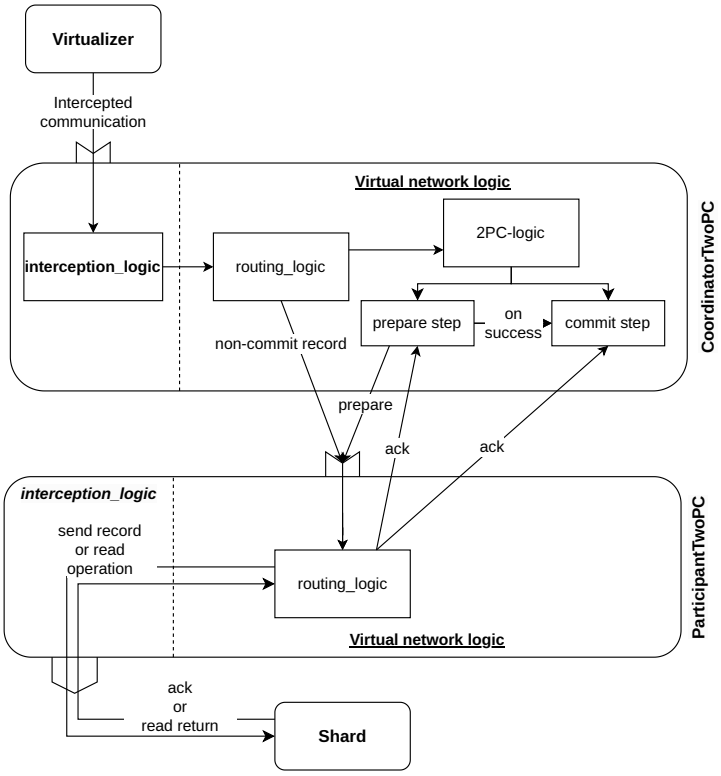


Fig. 9.5.: Logical structure of the two-phase commit interceptors.

```
772 (* dep_ts, commit_ts, tid, type, key, value
    ↪ *)
773 event record of timestamp,
    ↪ option<timestamp>, t_tid, record_type,
    ↪ key, value;
774
775 protocol p_server = +{
776     l_read: !tuple<timestamp,
        ↪ key>?option<value>.;
777     l_write: !record?bool.;
778 };
```

```
779 protocol p_two_pc_server = +{
780     l_pc_read: !tuple<timestamp,
        ↪ key>?option<value>.;
781     l_pc_write: !record?bool.;
782     l_pc_prepare: ?bool.;
783     l_pc_commit: ?bool.;
784     l_pc_abort: .;
785 };
```

Listing 24: On the left, the `p_server` is the protocol between the virtualizer and the shards. On the right, the `p_two_pc_server` is the protocol between the `CoordinatorTwoPC` and the `ParticipantTwoPC`. It is an extended version of `p_server` with the 2PC messages.

The Listing 25 details the implementation of the `CoordinatorTwoPC`. Let us explain the journey of a commit message record from the virtualizer to a shard.

1. First, the classical interception part of the coordinator (Listing 25, Lines 787-798) receives the message. It delegates the routing of the message to a helper function `route_round` (Line 795).
2. Then, the inter-sidecar part[5] of the coordinator takes over (Listing 25, Lines -843).
3. On commit record, the routing function starts a prepare phase, "waits" for the answers, and, triggers the commit phase (resp. abort). For brevity, we only detail the prepare phase. The commit one follows the same encoding.
   a) The prepare phase starts by sending a prepare message to all the participants (Line 829). The local state `this.rights` stores the participants' identity. The coordinator discovers those identities thanks to channel reflexivity: it queries the running listeners of the inter-sidecar channel.
   b) Then, `CoordinatorTwoPC` "waits" for all the answers (Line 831).
   c) Meanwhile, on the shard side, the `ParticipantTwoPC` instance handles both the `prepare` and the `commit` phase. Moreover, it also unpacks messages from the `p_2pc` protocol and encodes them into the `p_server` one then send them to the backend.

Interposing the virtual network

The question that remains is how to transparently interpose this virtual network knowing that the `Virtualizer` is already an interceptor. Transparently means that we do not want to modify either the `Virtualizer` or the `Shards`.

[5] The logic that ensures the communication between the `CoordinatorTwoPC` and the `ParticipantTwoPCs`

```
component CoordinatorTwoPC {
    (********** Interception *********)
    @msginterceptor(both)
    result<p_server, error> intercept_round(
        activation_ref<TxCoordinator> from, activation_ref<KVServer> to,
        (dual p_server) continuation_in, p_server continuation_out,
        blabel msg
    ){
        non_generic_activation_ref to = to;
        this.route_round(to, msg, continuation_in);

        return err(());
    }

    (********** Inter-sidecar routing *********)label{line:2PCcoordinatorinterstart}~
    outport p_out_two_pc expecting p_two_pc_server;
    inport <egress:ingress> p_abstract_in expecting (dual p_server) =
    ↪  this.default_callback;
    result<void, error> default_callback(blabel msg, (dual p_server) continuation){
        return err(error("Vardac replace this callback by the interception one"));
    }

    (* For brevity, we only present the handling on a commit operation *)
    result<void, error> route_round(activation_ref<ParticipantTwoPC> dest, blabel msg,
    ↪  (dual p_server) inner_continuation){
        session<p_two_pc_server> outer_continuation =
        ↪  initiate_session_with(this.p_out_two_pc, dest);

        branch inner_continuation on msg {
            | l_write => inner_continuation -> {
                tuple<record, !bool.> tmp = receive(inner_continuation);
                record r = tmp._0; record_type rtype = r._3_; !bool. inner_continuation =
                ↪  tmp._1;

                if(rtype == t_COMMIT){
                    (* Start 2PC *)
                    bool ack_prepare = this.prepare()?;
                    if(ack_prepare)
                        bool ack_commit = this.commit()?;
                }
            }
        }
    }

    result<bool, error> prepare(){
        list<bool> votes = [];
        for(activation_ref<KVServer> right in this.rights){
            this._prepare(votes, right)?;
        }


        if(listlength(this.rights) == listlength(votes)){
            for(bool vote in votes){
                if(vote == false){
                    return ok(false);
                }
            }
            return ok(true);
        } else {
            return err(error("CoordinatorTwoPC:: prepare - not all votes received"));
        }
    }
}
```

Listing 25: Simplified logic of the coordinator interceptor

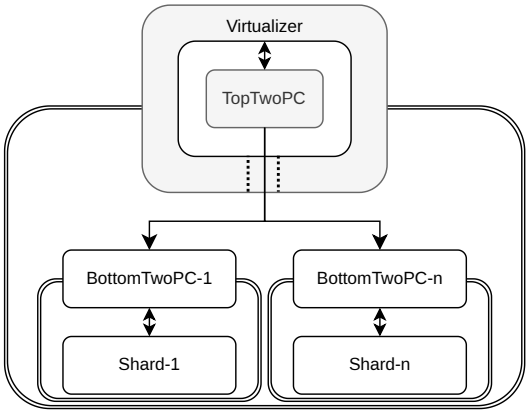


Fig. 9.6.: Varda virtual network providing two-phase commit.

The easiest way (see Figure 9.6), although counter-intuitive, is

1. to intercept each shard with a `ParticipantTwoPC` replica;
2. to intercept the resulting sub-architecture, formed by the shards with their sidecar, with the `CoordinatorTwoPC` thanks to nested interception;
3. to intercept the `CoordinatorTwoPC` with a `Virtualizer` in order to link the coordinator with the virtualizer. Moreover, to avoid a costly and unused indirection, the inter-sidecar channel remains unintercepted.

The first two steps are straightforward, they directly leverage classical interception (Listing 26, Lines 874-878). The third step is trickier. We instrument the interception policy in charge of the `CoordinatorTwoPC` (Listing 26, Lines 855-867) such that there is at most one coordinator (Line 860) and such that the `Virtualizer` intercepts the instantiation of the coordinator (Line 861).

As you may notice in the listing, we annotate the intercept block declaration with "`@1`" (Line 861). This is a compiler instruction that specifies the order of interception elimination. In this case, Vardac waits for the end of the generation of the `Coordinator` interception block before starting the generation of the `Virtualizer` one. More generally, the annotation "`@n`" specifies that the interception block is generated after the generation of the interception blocks annotated within "`@0-@n-1`". "`@0`" is the default priority level associated to unannotated interception blocks. The compiler cannot automatically infer the priority level of an interception block since it depends on the desired behaviours.

9.1.7 Step 4 - Add multi-DCs replication

The transition from one single-DC to multi-DCs implies to add a notion of DC, to manage the placement per DC and to add a replication mechanism between the DCs. In this section, we do not detail the replication since it behaves like the sharding logic with another routing logic. We focus on how to represent and handle a geodistributed infrastructure.

For simplicity, we encapsulate the backend of a DC into a `DCStore` component. This component orchestrates all the backend layer, namely it spawns the sharding logic, the 2PC virtual network and the shards on the nodes of the DC. Conversely, we only spawn one frontend that works with all the DCs.

```
845 (* For simplicity, we use a KVStore attribute since interception does not intercept
    ↪  attributes bridges yet
846 in the future we will need to carrefully establish a non-intercepted bridge between
847 CoordinatorTwoPC and ParticipantTwoPC and Right
848 *)
849 bridge<CoordinatorTwoPC, ParticipantTwoPC, p_two_pc_server> two_pc_bridge =
    ↪  bridge(p_two_pc_server);
850
851 option<activation_ref<Virtualizer>> virtualizer = none;
852 option<activation_ref<CoordinatorTwoPC> left = none;
853
854 (*** Interception policy ***)
855 activation_ref<CoordinatorTwoPC> make_interceptor_left (
856     bridge<CoordinatorTwoPC, ParticipantTwoPC, p_two_pc_server> -> option<place> ->
        ↪  activation_ref factory,
857     string intercepted_component_schema,
858     place p_of_intercepted
859 ){
860     if(is_none(this.left)){
861         with<Virtualizer, anonymous> this.make_interceptor @1{
862             activation_ref<CoordinatorTwoPC> res = factory(this.two_pc_bridge,
                ↪  some(p_of_intercepted));
863             this.left = some(res);
864         }
865     }
866     return option_get(this.left);
867 }
868
869 (*** Interposing the 2PC virutal layer ***)
870
871 bridge<DCStore, KVServer, p_server> b_backend = bridge(p_server);
872 bind(this.p_out_backend, b_backend);
873
874 with<CoordinatorTwoPC> this.make_interceptor_left {
875     with<ParticipantTwoPC, direct_onboarding> this.make_interceptor_right {
876         activation_ref<KVServer> backend = spawn KVServer(b_backend);
877     }
878 }
```

Listing 26: Interposing the `CoordinatorTwoPC` in front of the `Virtualizer` with one shard.

```
- place: dc1
  nbr_instances: "1"
  children:
    - place: nodes
      nbr_instances: "N"
- place: dc2
  nbr_instances: "1"
    - place: nodes
      nbr_instances: "N"
```

Listing 27: Modelling the infrastructures

Modelling and managing the infrastructure

We model the infrastructure as follows, in Listing 27: we define two named DCs (`"dc1"` and `"dc2"`) such that they contain an arbitrary number of nodes.

On startup, the architecture loads the infrastructure model (Listing 28, Lines 10-12). Then, it computes the actual running nodes on both DCs by querying the places that are tagged by `"nodes"` and that belongs to `"dc1"` (resp. `"dc2"`) (Lines 14-16).

Integration between the infrastructure and the architecture

We spawn the `DCStore` component on a random node of each DC (Lines 23 and 28). On creation, the `DCStore` spawns its children (i.e., component instances) on the nodes of the data center for which it is responsible.

To add the replication mechanism, a `DCReplicator` component intercept each `DCStore` component (Listing 28, lines 22 and 27). The replicators are interconnected by the `inter_dc_channel` (Line 19). As we discussed earlier, we implement this channel using a RabbitMQ broker.

The `DCReplicator` component behaves as follows:

On begin, update, commit or abort It asynchronously sends a copy to the other DCs and forwards the message to the `DCStore`.

On read It forwards the message to the `DCStore`.

9.1.8 Summary

The Table 9.1 summarise the development efforts to incrementally build VAntidoteDB. The system remains compact thanks to interception and virtual network. Most of the LoC are related to the implementation of the component individually. More specifically, communication code represents a large part of it since our protocols

```
(* Load the definition from the YAML file *)
vplacedef dc1 of "dc1";
vplacedef dc2 of "dc2";

(* Select all the running nodes of each DC *)
list<place> nodes1 = select_places(select_children(dc1, l'nodes'), x  : place-> true);
list<place> nodes2 = select_places(select_children(dc2, l'nodes'), x  : place -> true);

(* Inter-DC communication channel using a RabbitMQ broker *)
bridge<TxCoordinator, DCStore, p_server> inter_dc_channel = amqp_channel(p_server,
  ↪  "amqp://broker_ip_address:broker_port", "inter_backend");

(* DC-1 *)
with<DCReplicator, anonymous> this.make_interceptor_replicator{
    activation_ref<DCStore> backend = spawn DCStore("dc-1", nodes1, inter_dc_channel) @
    ↪  dc1;
}

(* DC-2 *)
with<DCReplicator, anonymous> this.make_interceptor_replicator{
    activation_ref<DCStore> backend = spawn DCStore("dc-2", nodes2, inter_dc_channel) @
    ↪  dc2;
}

(* One frontend *)
bridge<Gateway, TxManager, p_dep_kv> b_gateway_manager = channel(p_dep_kv);
activation_ref<TxManager> manager = spawn TxManager(b_gateway_manager, inter_dc_channel,
  ↪  backend) @ dc1;
(* Colocate the gateway with the manager *)
activation_ref<Gateway> gateway = spawn Gateway(b_gateway_manager, manager) @
  ↪  placeof(manager);
```

Listing 28: Adding geo-distribution with inter-DC replication.

	Lines of Code		Comments
	archi.	adaptor	
Miscellaneous	219	29	Type, protocols, top-level components and helping functions
Single Shard, Single DC	710	184	
Gateway	84		
TxManager	93		
TxCoordinator	84		
Shard	38		Excluding Materializer, Cache and Journal
Materializer	86	30	
Cache	52	35	
Journal	54	90	
Sharding	67		
Strong consistent commit	159		
CoordinatorTwoPC	115		
ParticipantTwoPC	44		
Multi-DC	20		RabbitMQ does not increase adaptor LoCs since the standard library already provides a generic channel for brokers that use the AMQP protocol.
DCReplicator	60		
DCs orchestration logic	21		
Total	956	184	

Tab. 9.1.: Summary of the programmers' efforts at each stage of VAntidoteDB's implementation. Most of the LoCs comes from the handling of each protocol branch.

has a lot of branches. The composition codes (i.e., children creation, interception scope and policy) only represent a few dozens of LoC. For comparative purposes, the existing AntidoteDB code base, with all its features, is about 7000 LoCs for the transaction layer and 6500 LoC for the backend (i.e., the caching, materialisation and logging layers). However, you should keep in mind that our clone only implements the core features.

We identified the following limitations and future needs for our prototype. As we will see in the next chapter, most of them are not scientific needs but are related to the maturation of the Varda ecosystem and tooling. Namely, our compiler is not mature enough to mix complex interaction between features (e.g., between

nested interception and inlining). Moreover, the most complicated task was to debug the Varda code since there is no debugger, profiler nor decent integration in IDEs[6] yet. In addition, this work highlights the need for some meta-programming to write interceptor factories that take a protocol in input and generate an interceptor component in output such that the interceptor logic is derived according to the structure of the protocol. One application could be to write a generic TwoPC factory that automatically adds the prepare, commit and abort phase to an input protocol, and that generates the corresponding interceptors.

9.2 Experimental Evaluation

Our experimental evaluation addresses the following questions: How difficult is it to develop a system with Varda? How does a system developed with of Varda perform, compared to its manually written counterpart? Note that, we do not aim to provide a comprehensive performance evaluation of Varda, since the compilation pipeline is not optimised yet.

9.2.1 Experimental protocol

We compare systems written in Varda with a baseline written manually in Akka (a Java variant). Both variants incorporate the same OTS component.

We conduct performance benchmarks, using micro-benchmarks to isolate the impact of the abstractions, and the load-balanced storage service using RocksDB. After a warm-up phase (not measured), each experiment runs five times.

We run the performance experiments on an Intel Core i7-10510U, clocked at 1.80–4.9 GHz with 16 GiB of memory. To simulate multiple places, we use Docker containers, each of them embedding a Java Virtual Machine.

Micro-benchmarks

Massive parallel ping-pong (MPP) The MPP benchmark is composed of two components, "`A`" and `B`. "`A`" sends n "`Ping`" asynchronous messages to `B`. "`B`" replies with a "`Pong`" message.

[6] Our prototype only provides syntax coloring right now. Adding them would require a significant engeeering effort.

	Akka		Varda		
	LoC	CB	LoC archi.	LoC adaptor	CB
MPP	310	2	133	10	2
MS	338	3	173	48	3
KVS	681	10	185	104	1

Tab. 9.2.: Programmer effort. LoC = Lines of Code. CB = Number of callbacks

The *MPP-contract* version counts the number of ping-pong rounds, which is added to each message. "`A`"has a contract ensuring that the received counter Pong counters are strictly positive.

Distributed merge sort (MS) The MS benchmark is a distributed merge sort. A "`Runner`" component splits its input array in two parts, sending each half to a newly spawned instance of itself. The runner waits for both children to return a sorted array, merges their results, and sends this back to its own parent.

Storage service with load balancer (KVS) We implement a key-value storage service with load-balancing (KVS). We consider two backends: a simple in-memory map, or an OTS RocksDB server. We stress test the KVS using Workload A (50/50 reads and writes) of the YCSB benchmark [50].

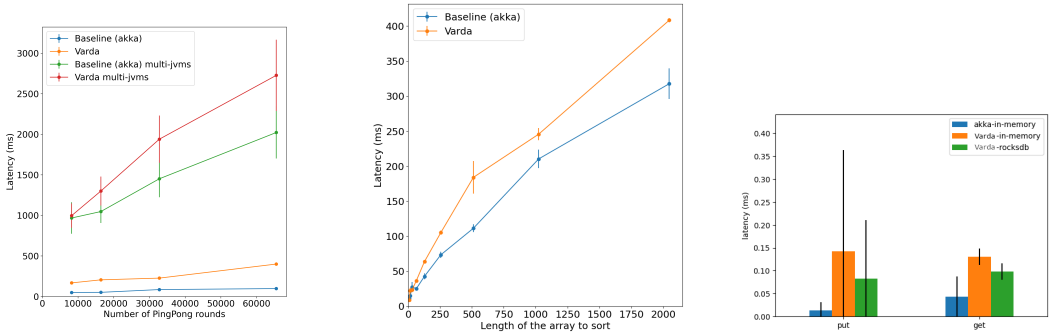
9.2.2 Programmer effort

Programmer effort We measure programmer effort as the number of lines of code (LoC), counted by the *cloc* system.[7] We do not count comments, blank lines, nor lines added for debugging and instrumentation.

The results in Table 9.2 show that Varda is more compact than Akka. On simple micro-benchmarks, the Varda version is twice as short. It is three times shorter for the more complex KVS.

The "CB" columns in Table 9.2 count the number of callbacks, which indicate a complex control flow. Varda drastically reduces this complexity.

[7] https://github.com/AlDanial/cloc



(a) MPP latency **(b)** MS latency **(c)** KVS latency

Fig. 9.7.: Latency experiments

9.2.3 Performance

YCSB YCSB measures latency of requests on the KVS system [50]. It runs on a virtualised infrastructure using Docker [@62].

Figure 9.7c shows latency (average and variance). The Varda version is on average twice slower than Akka. However, as the following micro-benchmark shows, the cost is attributable to our current inefficient message-passing.

Micro-benchmarks The MPP and MS described above are used for micro-benchmarking. We avoid spurious communication overhead by running all components of MS in a single JVM on the same machine; for MPP we compare one and two JVMs.

The MPP benchmark (Figure 9.7a) shows that the session primitives incur a high overhead. Varda creates a new session for each ping-pong exchange, whereas Akka uses asynchronous fire-and-forget. This clearly points to the need for a more efficient approach. Indeed, although our compiler optimises the core Varda code, it does not optimise code generated by the Java plugin.

Note that adding run-time checking of a contract, in benchmark MPP-contract, does not impact the results.

The MS benchmark (Figure 9.7b) shows only a modest overhead over Akka. Indeed, this benchmark evaluates mainly the cost of creating component instances. For a vector of size n, it instantiates approximately n components.

Part V

Conclusion

10 Discussion

10.1 Take-aways

10.1.1 What makes life easier for the programmer ?

Varda eases the incremental design and writing of distributed systems by facilitating safe composition of heterogeneous off-the-shelf components.

Program distribution related behaviours To achieve this, Varda either provides built-in distribution primitives (e.g., placement, error handling, network channels) or embeds existing tools (e.g., supervision, fault tolerance) in its programming model thanks to OTS adaptors and supervision ports. Reusing tools has two advantages: leveraging performant and well-tested tools, and controlling underlying OSI layers that would otherwise be abstracted (e.g., network layers).

Understandable code generation To discharge programmers from the burden of writing boilerplate code, Vardac generates the glue code between components. In addition, to ease integration with existing technologies, the compiler automates the generation of mainstream APIs (e.g., REST and gRPC) from the functional definition of a component interface. Eventually, Vardac generates readable source code with provenance information to ease debugging and profiling.

Incremental design To incrementally and safely extend an existing system, Varda interception transparently interposes proxies between existing components to add new components, modify the behaviours of existing ones or update the communication topology. Then, Vardac automatically propagates the changes to the pre-existing architecture and generates the new implementation. For example, we show that interception allows evolving a transactional database effortlessly from a single-node architecture to a geo-distributed architecture (see Chapter 9).

Explicit tradeoffs between efficiency and dependability To avoid forcing programmers, who do not need the expressive power of an abstraction, to have to

pay for it, Varda offers adaptative control over low-level details, distribution behaviour and safety verification. On the one hand, programmers could explicitly choose to degrade features and guarantees to improve system efficiency. Almost all the features (e.g., using the placement registry) and checks that could impact run time performance are either optional or the compiler can erase them for production releases (e.g., protocol guards and contracts). On the other hand, programmers could extend the programming language with arbitrary unsafe features to fit their needs by leveraging OTS adaptors. For instance, they can embed Cloud providers' primitives to dynamically request identity and security tokens.

Optimization and implementation specialization To optimise and specialise a system without modifying the architecture, Varda programmers can configure the mapping of these components, orthogonally to the logical boundaries (i.e., component boundaries):

- to the physical location using placement annotations,
- to the computation units using component inlining, and
- to compilation units using the code generation configuration.

Combined with the fact that Varda code generation is polyglot, i.e., it can generate the glue code in various languages, programmers can fit a system to infrastructure and deployment constraints without modifying the architecture or its specification.

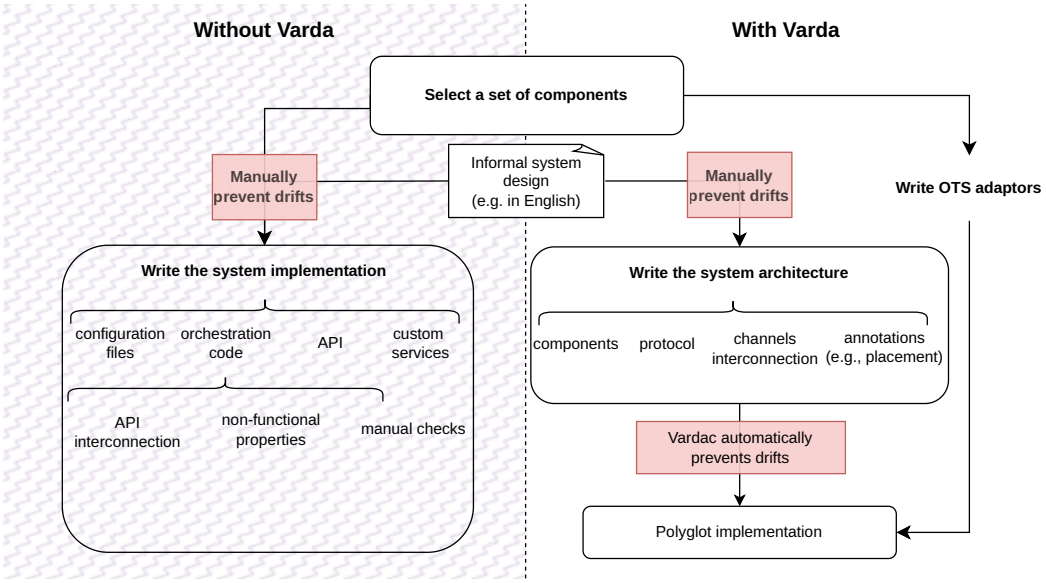


Fig. 10.1.: Comparison of the work of a programmer to build a distributed system with (on the right) and without Varda (on the left).

10.1.2 What would be the work of the programmer with and without Varda?

Figure 10.1 summarise the comparison between the work to build a distributed system without (on the left) and with Varda (on the right). In the following, we review four steps of the lifecycle of a distributed system:

- building the first version of the system;
- adding a new feature to the system;
- optimising the system by leveraging data locality, either by leveraging to the placement or by getting rid of logic boundaries;
- equipping the system with a formal specification. For the last step, we explore how a programmer can gain strong confidence in the fact that the system implementation follows the formal specification.

Without Varda the programmer has to

On first build

1. Write the system design using an informal language (e.g., English);
2. Select (or craft) the different components that compose the system;
3. Write the system implementation
    - Manually write and expose the API of the crafted components;
    - Manually interconnect the different components (e.g., by configuring the network);

- Write the orchestration logic in other components or use an externl system to impose it (e.g., an orchestration engine). For instance, define the placement strategy and synchronize the computation of the different components;
- Build the artefacts and define the deployment strategy.
- Note that the programmer has to ensure manually that there are no drifts between the high-level design and the system implementation.

On architecture update Manually update the system implementation. This means redoing all the steps of *first build* and manually applying changes without introducing regression or bugs.

Optimizing the system depending on the nature of the optimisation

Customizing placement Update the deployment strategy since developers often rely on it to handle placement. In doing so, programmers must manually ensure consistency between the placement strategy, system implementation and high-level design without introducing non-functional issues. For instance, for key-value store, two replicas storing the same content should not be placed on the same physical machine.

Refactoring the execution unit boundaries Re-architect the system. This means updating the component boundaries by moving computation from one component to another. This implies rewriting components, their API, and updating both their interconnection and the orchestration logic.

When the system is equiped with a formal specification Write tests to check the system implementation against the expected behaviours.

With Varda the programmer has to

On first build

1. Steps 1-2 from the non-Varda approach
2. Derive a Varda architecture from the design document. The developer still has to manually ensure that the architecture remains consistent with the design document.
3. Generate the system implementation and (optionally) the deployment strategy:
   - Select the code-generation target;
   - Write (or import) the adaptor for each off-the-shelf component;
   - Note that Vardac automatically guarantees that there is no drift between the architecture and the final implementation.

On architecture update Update the architecture using interception to transparently update the architecture without modifying the existing components.

Then, the compiler automatically propagates the changes through the steps of *First step*.

Optimizing the system Annotate the architecture by adding either place or inlining constraints. Then, use the Vardac to generate the corresponding implementation.

When the system is equipped with a formal specification Embed the semantics's constraints in the architecture by annotating the architecture using Varda verification toolbox, discussed in Chapter 6. Then, the Vardac either statically ensures some guarantees (e.g., communication order) or (on-demand) injects dynamic checks in the implementation.

10.1.3 What safety guarantees would you obtain using Varda?

Global system view By its design, halfway between a specification language and a programming language, Varda provides a centralised high-level view of the system which summarises the architecture of the system, some important non-functional properties for performance (e.g., placement, combining execution units) and some formal specification (e.g., protocol guards, pre/post conditions). This view is useful to:

Provide an up-to-date cartography of the system to the programmers

Reduce drifts between design and architecture Indeed, it should be easier for programmers to convince themselves that a system's architecture follows their design rather than studying the final implementation, since the architecture view strips away unnecessary implementation details.

Eliminate drifts between the architecture and implementation By design, Vardac generates an implementation that is up to date and correct with respect to the architecture. As always, there is no free lunch, this comes at the cost of adding an additional step when building systems. For instance, this could complicate keeping track of bugs. In Section 7.4, we discuss this problem and the mitigation strategy we adopt.

Built-in guarantees and additional specification expressiveness By construction, Varda ensures strong isolation between components (e.g., failure isolation, memory isolation and OTS sandboxing). Moreover, each component has a formal communication interface such that inter-component communication using Varda primitives follows the following properties:

- it abides by a formal protocol which guarantees the type and order of the messages;
- it can only occur between components that are explicitly connected by channels;
- it is private, i.e., messages are visible only by the communicating parties.

Additionally, Varda provides a specification sublanguage that allows the developer to enrich the architecture with custom constraints. The sublanguage can specify:

- component's behaviours and OTS's observables using contracts (i.e., pre/-post conditions) and ghost component state;
- run time protocol behaviours using message predicates, delivery upper bounds or protocol history predicates. The use of these additional constraints prevents programmers from scattering communication specifications within the logic of components.

- The behaviours of a set of components using monitors, by combining the interception mechanism with the previous building blocks (e.g., contracts).

Vardac performs static checks, mostly type checking, under the hood, to ensure all this and injects dynamic checks to detect constraint violations.

10.2 Limitations of Varda

10.2.1 Intrinsic limitations

Limits of the perimeter Varda focuses on system programming. Its perimeter is limited to the system composition and the communication pattern. It covers deployment to a limited extent (i.e., using templates). Moreover, it does not cover side channels or shared state between components. Additionally, programmers cannot, yet, specify synchronisation easily. They have to encode this into protocol and message passing. The current prototype ignores security concerns. We discuss promising approaches to address these problems in Chapter 11.

What is the intrinsic cost of using Varda? Top-down code generation approaches, like Varda, add unavoidable indirection layers in the development workflow. This may complicate the tracking of bugs or performance bottlenecks. Indeed, a programmer has to pair the system observable to the implementation code, then the implementation code to the architecture. We take this into account and provide a mitigation mechanism in Section 7.4.

10.2.2 Technical limitations

Limits of the prototype The prototype (Vardac) suffers from a lack of maturity. It incorporates all the features discussed in this document. However it provides different support for each of them. The Section 7.5 summarise the support of each feature.

Although we built our language and our prototype with the aim of being independent of the implementation, our current prototype has only one code-generation target, i.e., Akka. This prevents us from investigating how to generate a multi-target system, we discuss this in our future work.

Limits of the evaluation The evaluation suffers due to the lack of maturity of the compiler. This precludes a large-scale evaluation with good performance since the compilation does not produce enough optimised code yet. For instance, a low-hanging fruit would be to optimise the Akka library, i.e., the interface between the Akka framework and the generated code. Indeed, Vardac optimisation passes (see Chapter 7) simplify the generated code, however, they do not, yet, optimise the final encoding into Akka programming model.

10.3 Approach discussion

Varda is a preliminary study in exploring the viability of building languages tailored to system programmers according to our requirements (see Chapter 1). Due to limited manpower, going in this direction implies tradeoffs between the expressiveness of the language, the performance of the generated code, and the stability of the framework.

We focus on exploring the design space language (i.e., its expressiveness) and to get feedback on our language design, as quickly as possible, by building distributed systems with it. This approach enables to

- empirically evaluate the ability of the language to build a real-distributed system;
- get feedback on the language design to check that the features covers all the common needs for distributed system programming.
- ensures that the different features of the language are compatible with each other in a system programmer's perspective.

These choices are not harmless. They have a direct impact on the nature of the prototype and its evaluation (see Section 10.2.2). As a result, we had to postpone the writing of a formal semantics since the language design evolves a lot. Now we are confident that the design is stable enough. Hence, equipping Varda with a formal semantics is the next step toward dependability. We discuss this in the following chapter.

Future work depend on logistics, not on science. Before going further and exploring the research directions in Chapter 11, the mandatory work is to stabilize and optimize the existing framework.

Why not start by extending an existing language? We build Varda from scratch because

A different abstraction level Varda proposes an intermediate abstraction level in between specification language and programming language. Therefore, we do not build Varda on top of existing specification languages to ensure that our prototype exposes enough control over performance (i.e., placement, network links or fault tolerance). Conversely, we do not use a programming language as our baseline to provide an implementation-independent architecture stripped away from unnecessary low-level details.

No good candidates The previous point eliminates all the mainstream and mature languages. Some research languages remain, namely those that generates code from a specification (see Section 2.3.3). However, we exclude them because they do not address our composition nor our efficiency requirements. Therefore porting Varda on top of one of them would have required a significant and in-depth transformation of the underlying language. In addition, when we began our work, they were in active development and were not sufficiently mature.

10.3.1 Comparison with Varda competitors

Chapter 2 positions Varda with respect to its competitors, according to our requirements. In this section, we focus on logistics to rough out what it would take to push Varda one step further as proposed in the future work (Chapter 11). We compare our prototype with those of our closest competitors: P, LinguaFranca and PGo. Although imperfect, this comparison is interesting for planning the logistics and governance needed to build an experimental language that goes beyond a first proof of concept. For this, we explore the efforts invested in each prototype (e.g., LoC and number of contributors) in Figure 10.2. Then, we compare the research milestones they reached (e.g., number of papers) with respect to their manpower in Figure 10.3. Eventually, Figure 10.4 summarise the results of the previous figures in a normalised radar chart. We collect the data on April 29, 2023.

Figure 10.2 shows that we produce the same amount (order of magnitude) of LoC as other competitors for the core compiler (excluding tests, benchmarks and comments) with two to ten times less commits. Even if they are imperfect quantitative metrics,

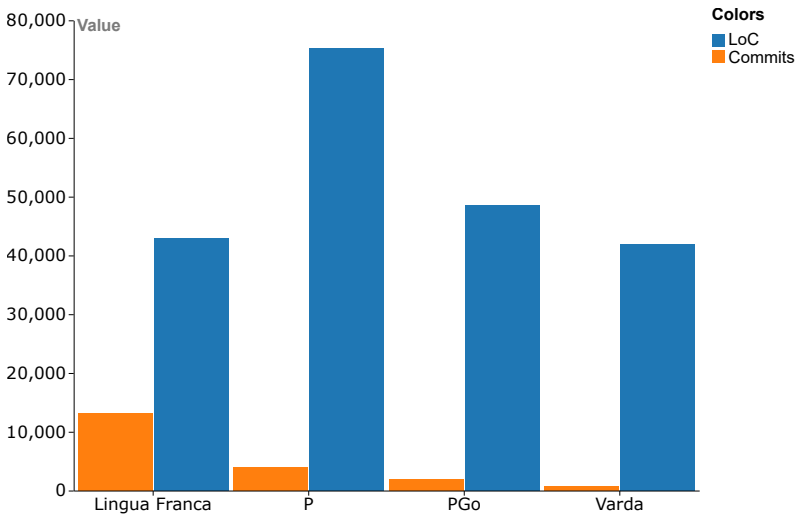


Fig. 10.2.: Comparison with Varda competitors: development efforts

the ratio between the number of commits[1] and the number of LoC is an easy to collect indicators to approximate the polishing and stabilisation effort.

Figure 10.3 illustrates the efforts needed to build a reusable research prototype language. This confirms the intuition of the previous section: future work needs logistics before doing science again. Note that the data we collect does not reflect the coordination overhead of a large number of contributors on a short period.

[1]We do not exclude the merge commits. Their proportion should increase for project with a high number of concurrent contributors (i.e., LinguaFranca).

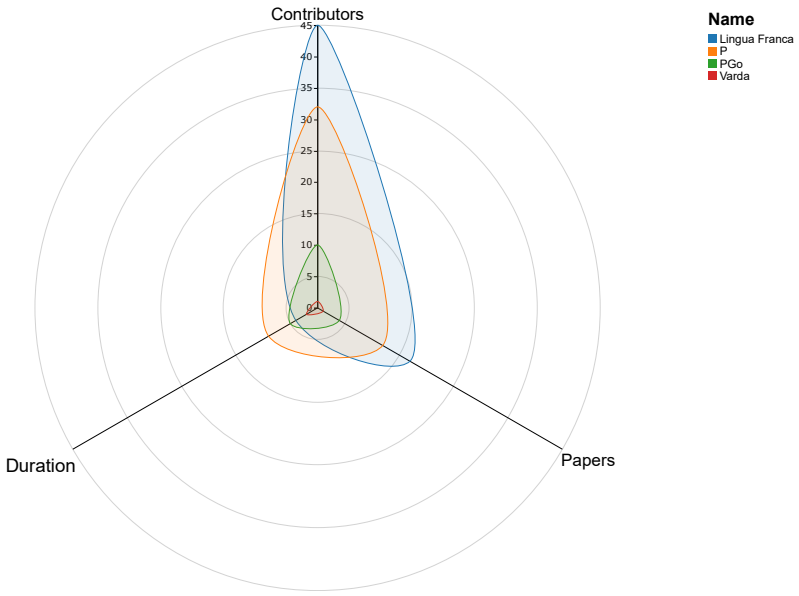


Fig. 10.3.: Comparison with Varda competitors: reasearch milestones with respect to the involved manpower.

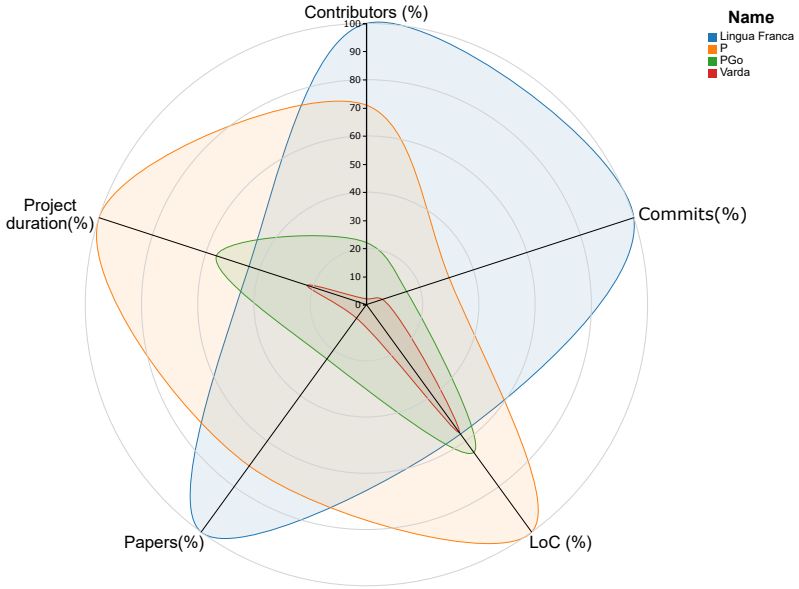


Fig. 10.4.: Comparison with Varda competitors: normalized sumup.

11 Research directions

We have identified three major directions for extending Varda beyond its current scientific limits: 1. *expressiveness* to give programmers greater flexibility and ability to express nonlocal properties (Section 11.1), 2. *system aspects* to cover blind spots that matter for real system programming (e.g., specializing a system to run on a heterogeneous Edge-Cloud infrastructure) (Section 11.2), 3. *dependability* to develop our long-term vision of a language between formal specification and system programming (Section 11.3).

11.1 Extending Varda expressiveness

11.1.1 Smart and reversible inlining

An interesting improvement would be to make inlining reversible by providing a primitive to move away an inlined component. For instance, one use case is to offload a component host in case of an excessive resource consumption. One way to implement it is to provide a `split(host_instance, inner_activation)` keyword, inspired by [169].

The next interesting extension is to build an automatic inlining decision that chooses where and when to inline a component instance based on inlining annotations, topology, and execution traces and metrics. One way would be to provide a control plan modelled by a component that manages a group of components identified either by scope or by annotations. To achieve this, Varda the programming model should support the inlining of a running component. This implies providing a programmable mechanism to migrate the state of a component.

11.1.2 Control global behaviours

To preserve the programmer's ability to control the system behaviour (e.g., placement, elasticity, state, etc.) at the component grain, core Varda only provide

component-local primitives to specify those behaviours. For instance, a component can only specify the placement of its child instances, one-at-a time. Hence the programmer cannot directly specify the global behaviour of a (subset) of the system: for placement, for state. The control is scattered per component, and locally specifies. This complexifies the architecture, increases the cognitive load on the developer and hinders its ability to have a clear situation awareness of the system. As a result, this decentralised control could induce subtle glitches between what the programmers want and what the system does.

To address this, the first research direction is to add a *declarative placement and elasticity policies* layer in the style of PLASMA [157] to centralise control of placement in one logical place. Then, the compiler will either split and scatter the placement constraints on the whole architecture (zero-cost abstraction) or generate a placement engine configuration. Such a placement engine could be written like an ordinary component receiving placement constraints from message passing and using reflexivity to collect knowledge of the current topology.

The second direction is to add *shared states* in Varda. The objective is twofold: On the one hand, to be able to express global properties of the system (using global ghost state) and to ease the expression of some invariant. On the other hand, to be able to model hidden interaction between components that emerges from accessing external sharded state through OTS. Indeed, shared states are massively used in distributed systems in the form of database, key-value store, block or object storage, etc. In this manner, ghost shared state could model a lot of side channels introduced by OTS logic.

11.2 System aspects

11.2.1 Multi-target code generation

Current trend of systems distributed is to build a distributed system on top of a heterogeneous infrastructure ranging from devices to data centers: to take advantage of the Edge-Fog-Cloud continuum for new usage or to improve metrics such as latency or network cost [21], or to regain control over personal data by following a local-first approach [107].

Building systems that span across such a continuum often requires a composition of distinct technologies according to the nature of the infrastructure. Often, the core of the system runs on cloud technologies (e.g., containers and VMs), whereas the

clients run on an edge device (e.g., a web browser) and provides specific logic (e.g., caches).

Generating a system on top a two-side infrastructure requires doing multi-target code generation and correctly interconnecting those targets. One part of the system should be specialised for the cloud technologies, and the other should be adapted to the constraints of edge devices.

The prototype should be extended with new code-generation target languages: on the one hand, a target for the edge (e.g., TypeScript); on the other hand, a better integration with Cloud technology by either providing a Kubernetes or a serverless target.

Note that the current version of Varda can generate the Cloud part and package it as containers. However, it does not integrate in the existing ecosystem or take advantage of the advance of the Cloud platforms in terms of resource management (serverless) and container orchestration (Kubernetes).

Kubernetes target

Having a Kubernetes target would make containers first-class objects and not inert artefacts. Spawning a component should result in spawning a new container, placement primitives should integrate with K8S placement engines and maybe, interception can leverage service-mesh technologies [@26], [@103].

Exploring this direction requires implementing the components using containers and to introduce dynamic interactions between the Varda runtime and the orchestration engine: encoding Varda abstractions into the corresponding Kubernetes entities, and, vice versa, exposing relevant Kubernetes abstractions as first- class values in Varda, thereby adding the capability to program the deployment and elasticity of components.

Multi-target code generation

Doing multi-target code generation means scattering the architecture in different parts, each part being specialised for a specific target. Our prototype already provides support for these features in a per component type basis. Each component type can be compiled down to a distinct target language. The choice of the target is

independent from the architecture[1] to ease porting software to new infrastructure. Programmers specify the target using the target configuration file (see Chapter 7).

Currently, there are no mechanisms that transparently interconnect to separate targets and that maintain all the guarantees of Varda. For instance, a programmer can split a system into two independent variants of the Akka target to avoid sharing the same Akka runtime. In this case, both systems could expose gRPC/REST interfaces. Then the programmer needs to interconnect these interfaces manually. With this setup, sessions are not preserved between two targets and the placement registry is not shared.

A promising direction is to use the Varda code generation to automatically generates the boilerplate to interconnect distinct subsystems built from distinct targets by adding an additional gRPC layer to support session and to optionally interconnect the underlying runtime services (e.g., placement registry), if requested by the programmer.

11.2.2 Code evolution

Distributed system needs to be able to evolve over time without down time[2]. At present, Varda provides built in primitives to allow safe incremental system building, however, it does not support hot swapping yet. Hot swapping refers to the ability to add, remove, or replace components of a system while it is running, without the need to shut down or reboot the system.

The current situation is as follows. At the component level, the inner logic of a component can be altered as long as the new interface remains compatible with the previous one. At the group of component level, interception permits to transparently update the composition and communication pattern while preserving each component. However, leveraging both features to update a system implies recompiling the whole architecture and manually selecting the altered (or new) artefacts to deploy.

An interesting direction is to automate the hot swapping by leveraging these existing building blocks while preserving the guarantees of the Varda. One way could be to adapt the deployment instructions generation to leverage the Kubernetes deployment strategies that already provides various forms of hot swapping and support multiple versions of the same component at the same time.

Doing this implies to address those three complex challenges:

[1]

[2]We discussed this requirement in Chapter 1.

How to handle statefull components ? Indeed, Varda components might be stateful and interact with arbitrary OTS components that can hide any kind of state. Therefore, running multiple versions of the same instance can be tricky in terms of state reconciliation[3].

How to add or remove interception scope at run time ? Recall that the generic and polyglot way of doing interception is to rewrite a subset of the architecture. There are two ways to tackle this problem:

- Either redeploy the whole set of intercepted components and interceptors.
- Or provide a specialised dynamic interception implementation for the Kubernetes target in order to perform interception leveraging service mesh for instance. In this case, the hard part will be to correctly unpack the intercepted network packets and correctly interpret them as high-level Varda communication.

In any case, doing this implies responding to the following question: *What is the semantics of the interception when the migration runs ?* For instance, should the interception migration be atomic ?

How to support component interface evolution ? If the interface update remains in the range of protocol subtyping, then the existing building blocks could handle it transparently[4]. Conversely, *what if a programmer introduces breaking changes in a protocol between two components?* An interesting two-step solution could be to: First, generate for each component a protocol adaptor component that intercepts the origin component such that the adaptor supports both the old and the new protocol. Then, update the running system. Second, once the system is updated, dynamically inline each adaptor in their corresponding component or eliminate the adaptor after use.

[3] In container realm, the classical workaround is to deport the state outside of the container.
[4] This depends on the encoding of the Varda type system in the target language.

11.3 Dependability

Our long-term vision for Varda is to bridge the gap between formal specification and high-performance implementation. We argue that the Varda architecture is a sweet spot that provides the right level of abstraction to act at the common ground between the formal model(s) and the implementation(s). At the same time, programmers can use safe programming primitives and enrich the architecture with safety constraints while expressing the adequate level of control for system programming and mitigating the performance overhead.

To increase dependability, we propose to explore an intermediate approach between extracting an implementation from a formal model and verifying an existing implementation. As for code generation, the idea is to derive a formal model from the architecture and use formal verification tools to reason on it. Moving in that direction implies strengthening what a programmer can specify, equipping Varda with formal semantics, and providing a way to extract a formal model from the architecture.

Formal Varda semantics The first step is to equip Varda with formal semantics that includes a formal core calculus to model system execution, operational semantics and formal type semantics. In addition to traditional properties such as type soundness[5], the semantics should also be able to match the following properties: applying interception preserves the semantics of the intercepted components; and, the semantics of an inlined program is observably equivalent to the semantics of the original program.

Strengthen the specification At present, apart from manually using monitors, programmers have to scatter the safety constraints to either pair of components (protocols) or singleton (contracts). The next step is to provide a way to express *global safety properties* that encompass the whole system (or a subset) in order to model interesting properties about emergent behaviours (e.g., deadlock freedom, consistency or predicate on a set of component state). The first step could be to leverage

[5]Type soundness is the property that well-typed programs do go wrong [149]. This guarantees that: 1. A well-typed program will never get stuck (*Progress*). 2. If a well-typed program takes a step of evaluation, then the resulting program is well-typed (*Preservation*).

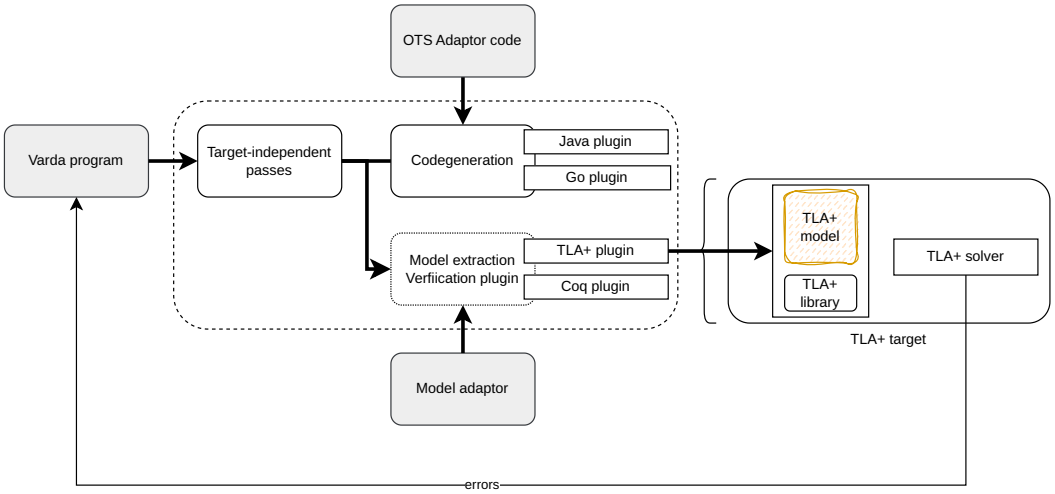


Fig. 11.1.: Expected Varda workflow with model extraction. The developer of a distributed system provides the grey parts. The Varda compiler generates the dashed (orange) blocks.

multiparty session types (with broadcasts) in order to express complex protocols that involve more than two components.

Apart from supervision and timeouts, the current prototype does not formalise liveness requirements; *a fortiori,* it does not help the developer reason about them. Future work should address this limitation.

Another blind spot of the current prototype is the lack of properties on the network channels. Some channel implementations provide interesting properties such as FIFO. However, they are implicit and are not modelled in the architecture. The language could extend the channel types to either represent delivery guarantees as a type annotation (e.g., using type latices for channel properties) or as channel guards. The first solution has the advantage of being easy to check, the second has the advantage of being more expressive. Note that protocol guards cannot express those properties since they reason on sessions and the proposed channel guard reason on network-layer messages (p. ex. TCP paquets).

Formal model extraction The least resistance line to extract models from the architecture is to follow the same architecture as the code generation. The idea is to add a plugin-based extraction system to the compiler. Figure 11.1 depicts the expected workflow. Such that the developer can use the right tool according to the wanted properties and available commitment time. For instance, theorem proover plugins (e.g., Coq, HOL) might allow programmers to prove arbitrary properties about architecture. Model checker based plugins (e.g., TLA+) might automate checking

for general-purpose properties. Finally, specific model tools might be integrated to check distribution specific properties, e.g., consistency (e.g., CISE, Hamzas).

12 General conclusion

Large distributed systems are often built by assembling off-the-shelf (OTS) components, e.g., components, services, processes, etc., developed independently. The current approach is to interconnect their APIs manually. This is *ad hoc*, complex, tedious, and error-prone.

Programming languages offer a promising approach to addressing this problem. First, they could help *reduce the occurrence of bugs*. The programmer specifies the system using well-defined entities and constraints. Then, the compiler a correct-by-construction implementation providing various guarantees (e.g., that communications are correctly ordered). Furthermore, languages could help *improve programmers' productivity*. The code generation offloads the boilerplate plumbing to the compiler. In addition, the compiler might perform optimisations leveraging its knowledge of the system.

In this thesis, we propose a new language, Varda, at the intersection between programming and specification languages. A Varda program describes how to *compose* components into a coherent *architecture*. To ensure safety, the programmer isolates an OTS component behind a Varda *shield*. The shield restricts the component's behaviour by specifying its interface, its *protocol* (i.e., what it may send or receive, and in what order), and pre- and post-conditions. Components can be logically nested, to provide encapsulation. An outer component *orchestrates* its inner components, spawning or killing component instances, interconnecting them, and supervising error conditions; it can *intercept* and manipulate communication, and more generally compute over components and messages. Varda provides strong guarantees (e.g., isolation between components), enforcing the specification by static analysis, by run-time checks, and by sandboxing.

At the same time, to be useful for the development of real, practical distributed systems, Varda takes a pragmatic approach. A shield can contain non-Varda adaptor code (e.g., Java) in order to incorporate black-box OTS components. Varda provides control over non-functional properties that are relevant for performance and fault tolerance, for instance elasticity, placement, and inlining. The Varda compiler automates the generation of boilerplate *glue* code to make components work together. Varda supports non-stop execution, thanks to supervision mechanisms.

We demonstrate the expressiveness and the ergonomic of Varda by encoding classical distributed patterns (e.g., sharding and access control). To illustrate how Varda helps build a real distributed system, we implement a clone of the *AntidoteDB* geo-replicated datastore.

Varda applications are compact, and exhibit modular and reusable design. Our experiments show that the run-time overhead is modest, thanks to compile-time verification and optimisations.

Bibliography

[7] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003 (cit. on p. 24).

[9] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," Nara, Japan, Jun. 2016, pp. 405–414 (cit. on p. 164).

[10] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, 939–964, 2014 (cit. on p. 26).

[11] H. Alkayed, H. Cirstea, and S. Merz, "An Extension of PlusCal for Modeling Distributed Algorithms," in *TLA+ Community Event 2020*, Freiburg (online), Germany, Oct. 2020 (cit. on p. 41).

[12] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein, "Consistency without borders," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–10 (cit. on p. 40).

[13] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, *et al.*, "Business process execution language for web services specification," *BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003 (cit. on p. 37).

[15] A. W. Appel, "Verified software toolchain: (invited talk)," in *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*, Springer, 2011, pp. 1–17 (cit. on p. 47).

[16] J. Armstrong, "Erlang," *Communications of the ACM*, vol. 53, no. 9, pp. 68–75, 2010 (cit. on pp. 22, 23, 100, 116).

[17] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, 2013 (cit. on pp. 24, 27).

[18] ——, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, 2013 (cit. on p. 27).

[19]D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: A programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds., ACM, 2010, pp. 119–130 (cit. on pp. 24, 25).

[20]P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, 465–483, 1983 (cit. on p. 163).

[21]L. F. Bittencourt, R. Immich, R. Sakellariou, N. L. S. da Fonseca, E. R. M. Madeira, M. Curado, L. Villas, L. A. DaSilva, C. Lee, and O. F. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet Things*, vol. 3-4, pp. 134–155, 2018 (cit. on p. 200).

[22]R. Bobba, J. Grov, I. Gupta, S. Liu, J. Meseguer, P. C. Ölveczky, and S. Skeirik, "Survivability: Design, formal modeling, and validation of cloud storage systems using maude," *Assured cloud computing*, pp. 10–48, 2018 (cit. on p. 42).

[23]F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and others, "A survey of active object languages," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–39, 2017 (cit. on p. 23).

[25]E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00, Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7 (cit. on p. 18).

[27]S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021 (cit. on p. 31).

[28]B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*, A. Clements and T. Condie, Eds., USENIX Association, 2016 (cit. on p. 35).

[29]N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, "Choreography and orchestration conformance for system design," in *International Conference on Coordination Languages and Models*, Springer, 2006, pp. 63–81 (cit. on p. 37).

[30]S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ACM, 2011, p. 16 (cit. on pp. 22–24, 40).

[31]M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centered programming for web services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, 8:1–8:78, 2012 (cit. on p. 37).

[32]M. Carbone and F. Montesi, "Deadlock-freedom-by-design: Multiparty asynchronous global programming," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds., ACM, 2013, pp. 263–274 (cit. on p. 37).

[33]P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015 (cit. on pp. 24–26, 145).

[34]G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, "On global types and multi-party sessions," in *Formal Techniques for Distributed Systems: Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 30th IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, Springer, 2011, pp. 1–28 (cit. on p. 43).

[35]T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds., ACM, 2019, pp. 243–258 (cit. on p. 47).

[36]T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich, "Gojournal: A verified, concurrent, crash-safe journaling system.," in *OSDI*, 2021, pp. 423–439 (cit. on p. 47).

[37]C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: Easy, efficient data-parallel pipelines," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, Toronto, Ontario, Canada: Association for Computing Machinery, 2010, 363–375 (cit. on p. 24).

[38]S. Chand, Y. A. Liu, and S. D. Stoller, "Formal verification of multi-paxos for distributed consensus," in *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou, Eds., ser. Lecture Notes in Computer Science, vol. 9995, 2016, pp. 119–136 (cit. on p. 41).

[39]T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, I. Gupta and R. Wattenhofer, Eds., ACM, 2007, pp. 398–407 (cit. on pp. 14, 38, 45, 126).

[40]M. Chardet, H. Coullon, D. Pertin, and C. Pérez, "Madeus: A formal deployment model," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2018, pp. 724–731 (cit. on p. 41).

[41]P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, ACM, 2005, pp. 519–538 (cit. on pp. 4, 111).

[42]A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano, "New directions in cloud programming," in *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*, www.cidrdb.org, 2021 (cit. on p. 53).

[43]A. Chlipala, “Ur/web: A simple model for programming the web,” *SIGPLAN Not.*, vol. 50, no. 1, 153–165, 2015 (cit. on p. 30).

[44]A. K. Chopra, M. P. Singh, *et al.*, “An evaluation of communication protocol languages for engineering multiagent systems,” *Journal of Artificial Intelligence Research*, vol. 69, pp. 1351–1393, 2020 (cit. on pp. 43, 45).

[45]——, “Splee: A declarative information-based language for multiagent interaction protocols,” 2017 (cit. on p. 44).

[46]M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, “Principles of maude,” *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65–89, 1996 (cit. on p. 42).

[48]P. C. Clements, “A survey of architecture description languages,” in *Proceedings of the 8th International Workshop on Software Specification and Design, IWSSD 1996, USA, March 22-23, 1996*, IEEE Computer Society, 1996, pp. 16–25 (cit. on pp. 42, 43).

[49]C. Constantinides, T. Skotiniotis, and M. Stoerzer, “Aop considered harmful,” in *1st European Interactive Workshop on Aspect Systems (EIWAS)*, 2004 (cit. on p. 37).

[50]B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154 (cit. on pp. 98, 183, 184).

[51]M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida, “A gentle introduction to multiparty asynchronous session types,” in *Formal Methods for Multicore Programming*, M. Bernardo and E. B. Johnsen, Eds., vol. 9104, Cham: Springer International Publishing, 2015, pp. 146–178 (cit. on p. 44).

[52]R. M. Costa, “Compiling distributed system specifications into implementations,” M.S. thesis, U. of British Columbia, Vancouver, BC, Canada, May 2019 (cit. on p. 47).

[53]H. Coullon, C. Jard, and D. Lime, “Integrated model-checking for the design of safe and efficient distributed software commissioning,” in *International Conference on Integrated Formal Methods*, Springer, 2019, pp. 120–137 (cit. on p. 41).

[54]O. Dardha, E. Giachino, and D. Sangiorgi, “Session types revisited,” *Information and Computation*, vol. 256, pp. 253–286, Oct. 2017 (cit. on pp. 44, 59, 67, 113).

[55]J. De Koster, T. Van Cutsem, and W. De Meuter, “43 years of actors: A taxonomy of actor models and their key properties,” in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE 2016, Amsterdam, Netherlands: Association for Computing Machinery, 2016, 31–40 (cit. on pp. 22, 71).

[56]J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008 (cit. on pp. 24, 25).

[57]P.-M. Deniélou and N. Yoshida, “Dynamic multirole session types,” in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 435–446 (cit. on p. 44).

[58]F. DeRemer and H. H. Kron, “Programming-in-the-large versus programming-in-the-small,” *IEEE Transactions on Software Engineering*, no. 2, pp. 80–86, 1976 (cit. on p. 43).

[59]A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: Safe asynchronous event-driven programming,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 321–332, 2013 (cit. on pp. 13, 47, 49, 98, 113, 124).

[60]A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, “Compositional programming and testing of dynamic distributed systems,” *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–30, OOPSLA 2018 (cit. on pp. 4, 11, 13, 14, 38, 45, 49, 126).

[61]M. Dezani-Ciancaglini and U. De’Liguoro, “Sessions and session types: An overview,” in *International Workshop on Web Services and Formal Methods*, Springer, 2009, pp. 1–28 (cit. on pp. 44, 56, 67, 113).

[64]J. Dolby, “Automatic inline allocation of objects,” in *Proceedings of the ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*, M. C. Chen, R. K. Cytron, and A. M. Berman, Eds., ACM, 1997, pp. 7–17 (cit. on p. 103).

[65]J. Dolby and A. A. Chien, “An automatic object inlining optimization and its evaluation,” in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, M. S. Lam, Ed., ACM, 2000, pp. 345–357 (cit. on p. 103).

[66]C. Drăgoi, T. A. Henzinger, and D. Zufferey, “Psync: A partially synchronous language for fault-tolerant distributed algorithms,” *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 400–415, 2016 (cit. on pp. 12, 38).

[67]——, “PSync: A partially synchronous language for fault-tolerant distributed algorithms,” *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 400–415, 2016 (cit. on p. 47).

[68]J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini, “Distributed rescala: An update algorithm for distributed reactive programming,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’14, Portland, Oregon, USA: Association for Computing Machinery, 2014, 361–376 (cit. on p. 28).

[69]——, “Distributed rescala: An update algorithm for distributed reactive programming,” *SIGPLAN Not.*, vol. 49, no. 10, 361–376, 2014 (cit. on p. 28).

[70]J. Edwards, “Coherent reaction,” in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, S. Arora and G. T. Leavens, Eds., ACM, 2009, pp. 925–932 (cit. on p. 23).

[71]T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, “Costless: Optimizing cost of serverless computing through function fusion and placement,” *CoRR*, vol. abs/1811.09721, 2018. arXiv: 1811.09721 (cit. on p. 31).

[72]C. Elliott and P. Hudak, “Functional reactive animation,” in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP ’97), Amsterdam, The Netherlands, June 9-11, 1997*, S. L. P. Jones, M. Tofte, and A. M. Berman, Eds., ACM, 1997, pp. 263–273 (cit. on p. 27).

[73]C. M. Elliott, “Push-pull functional reactive programming,” in *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell ’09, Edinburgh, Scotland: Association for Computing Machinery, 2009, 25–36 (cit. on p. 27).

[76]A. Ferrando, M. Winikoff, S. Cranefield, F. Dignum, and V. Mascardi, “On enactability of agent interaction protocols: Towards a unified approach,” in *Engineering Multi-Agent Systems: 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13–14, 2019, Revised Selected Papers 7*, Springer, 2019, pp. 43–64 (cit. on p. 43).

[77]M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985 (cit. on p. 18).

[79]D. Garlan, R. Allen, and J. Ockerbloom, “Architectural mismatch or why it’s hard to build systems out of existing parts,” in *Proceedings of the 17th International Conference on Software Engineering*, ser. ICSE ’95, Seattle, Washington, USA: Association for Computing Machinery, 1995, 179–185 (cit. on p. 5).

[80]——, “Exploiting style in architectural design environments,” in *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1994, New Orleans, Louisiana, USA, December 6-9, 1994*, D. S. Wile, Ed., ACM, 1994, pp. 175–188 (cit. on p. 42).

[81]D. Garlan, R. Monroe, and D. Wile, “Acme: An architecture description interchange language,” in *CASCON First Decade High Impact Papers*, 2010, pp. 159–173 (cit. on p. 42).

[82]D. Garlan and M. Shaw, “An introduction to software architecture,” in *Advances in Software Engineering and Knowledge Engineering*, ser. Series on Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora, Eds., vol. 2, World Scientific, 1993, pp. 1–39 (cit. on pp. 12, 42, 54).

[83]S. Gay and M. Hole, “Subtyping for session types in the pi calculus,” *Acta Informatica*, vol. 42, no. 2-3, pp. 191–225, Nov. 2005 (cit. on p. 114).

[84]G. Germain, “Concurrency oriented programming in termite scheme,” in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, M. Feeley and P. W. Trinder, Eds., ACM, 2006, p. 20 (cit. on p. 22).

[85]A. S. Gokhale and D. C. Schmidt, “Evaluating corba latency and scalability over high-speed atm networks,” in *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS ’97)*, ser. ICDCS ’97, USA: IEEE Computer Society, 1997, p. 401 (cit. on p. 34).

[87]J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco CA, USA: Morgan Kaufmann, 1993 (cit. on p. 171).

[88]R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent os kernels," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., USENIX Association, 2016, pp. 653–669 (cit. on p. 47).

[89]C. Guidi, R. Lucchi, and M. Mazzara, "A formal framework for web services coordination," *Electronic Notes in Theoretical Computer Science*, vol. 180, no. 2, pp. 55–70, 2007 (cit. on p. 37).

[90]F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh, "Compiling distributed system models with pgo," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 159–175 (cit. on pp. 47, 48).

[91]N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991 (cit. on p. 27).

[92]P. Haller, "On the integration of the actor model in mainstream technologies: The scala perspective," in *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, ser. AGERE! 2012, Tucson, Arizona, USA: Association for Computing Machinery, 2012, 1–6 (cit. on p. 22).

[93]H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *J. Cloud Comput.*, vol. 10, no. 1, p. 39, 2021 (cit. on p. 30).

[94]S. Hatia and M. Shapiro, "Specification of a transactionally and causally-consistent (TCC) database," Paris, France, Tech. Rep. RR-9355, Jul. 2020 (cit. on pp. 163–165).

[95]C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 1–17 (cit. on p. 46).

[96]N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in cloud computing: What it is, and what it is not.," in *ICAC*, vol. 13, 2013, pp. 23–27 (cit. on p. 101).

[97]C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73, event-place: Stanford, USA, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245 (cit. on pp. 21, 71).

[98]M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, 2014 (cit. on p. 26).

[99]K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 273–284 (cit. on p. 44).

[100]F. Houshmand and M. Lesani, “Hamsaz: Replication coordination analysis and synthesis,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 74:1–74:32, 2019 (cit. on p. 41).

[101]P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, “Arrows, robots, and functional reactive programming,” *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures 4*, pp. 159–187, 2003 (cit. on p. 27).

[102]F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, “Opening the black boxes in data flow optimization,” *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1256–1267, 2012 (cit. on p. 26).

[104]K. Kambona, E. G. Boix, and W. De Meuter, “An evaluation of reactive programming and promises for structuring collaborative web applications,” in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, ser. DYLA ’13, Montpellier, France: Association for Computing Machinery, 2013 (cit. on p. 30).

[106]G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *ECOOP’97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, M. Aksit and S. Matsuoka, Eds., ser. Lecture Notes in Computer Science, vol. 1241, Springer, 1997, pp. 220–242 (cit. on p. 36).

[107]M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, “Local-first software: You own your data, in spite of the cloud,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019, Athens, Greece: Association for Computing Machinery, 2019, 154–178 (cit. on p. 200).

[108]D. Kouzapas, R. Gutkovas, and S. J. Gay, “Session types for broadcasting,” *arXiv preprint arXiv:1406.3481*, 2014 (cit. on pp. 44, 45).

[109]J. Kramer, J. Magee, and A. Finkelstein, “A constructive approach to the design of distributed systems,” in *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, IEEE Computer Society, 1990, pp. 580–587 (cit. on pp. 13, 121).

[110]B. Lagaisse and W. Joosen, “True and transparent distributed composition of aspect-components,” in *Middleware 2006, ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006, Proceedings*, M. van Steen and M. Henning, Eds., ser. Lecture Notes in Computer Science, vol. 4290, Springer, 2006, pp. 42–61 (cit. on p. 36).

[111]L. Lamport, “Specifying systems: The tla+ language and tools for hardware and software engineers,” 2002 (cit. on pp. 17, 41).

[112]——, “The pluscal algorithm language,” in *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, M. Leucker and C. Morgan, Eds., ser. Lecture Notes in Computer Science, vol. 5684, Springer, 2009, pp. 36–60 (cit. on p. 41).

[113]B. Lampson, "Hints and principles for computer system design," *arXiv preprint arXiv:2011.02455*, 2020 (cit. on pp. 11–13, 15, 19).

[114]B. W. Lampson, "Hints for computer system design," *SIGOPS Oper. Syst. Rev.*, vol. 17, no. 5, 33–48, 1983 (cit. on pp. 11–13, 15, 19).

[115]J. Lange, N. Ng, B. Toninho, and N. Yoshida, "Fencing off go: Liveness and safety for channel-based programming," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 748–761, 2017 (cit. on p. 45).

[116]E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987 (cit. on p. 27).

[117]E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987 (cit. on p. 27).

[118]N. V. Lewchenko, A. Radhakrishna, A. Gaonkar, and P. Černỳ, "Sequential programming for replicated data stores," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–28, 2019 (cit. on p. 40).

[120]B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, 1811–1841, 1994 (cit. on p. 113).

[121]M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," *Model-Based Design of Cyber Physical Systems (CyPhy'19)*, 2019 (cit. on pp. 47, 48).

[122]J. Magee, N. Dulay, and J. Kramer, "Structuring parallel and distributed programs," *Softw. Eng. J.*, vol. 8, no. 2, pp. 73–82, 1993 (cit. on p. 42).

[123]A. Maglie, "Reactivex and rxjava," in *Reactive Java Programming*. Berkeley, CA: Apress, 2016, pp. 1–9 (cit. on p. 27).

[124]B. Martin, L. Prosperi, and M. Shapiro, "A new environment for composable and dependable distributed computing," in *EuroSys Doctoral Workshop*, 2020 (cit. on p. 7).

[125]B. Martin, L. Prosperi, and M. Shapiro, "Transactional-turn causal consistency.," in *29th International European Conference on Parallel and Distributed Computing (Euro-Par)*, Cyprus, 2023 (cit. on p. 7).

[126]N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT1996, San Francisco, California, USA, October 16-18, 1996*, D. Garlan, Ed., ACM, 1996, pp. 24–32 (cit. on p. 42).

[127]N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000 (cit. on pp. 42, 43).

[128]C. S. Meiklejohn, Z. Lakhani, P. Alvaro, and H. Miller, "Verifying interfaces between container-based components," 2018 (cit. on pp. 13, 35, 112, 113, 121).

[129]L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for ajax applications," *SIGPLAN Not.*, vol. 44, no. 10, 1–20, 2009 (cit. on p. 28).

[130]T. Mikkonen and A. Taivalsaari, "Web applications - spaghetti code for the 21st century," in *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, W. Dosch, R. Y. Lee, P. Tuma, and T. Coupaye, Eds., IEEE Computer Society, 2008, pp. 319–328 (cit. on p. 30).

[131]M. Milano and A. C. Myers, "MixT: A language for mixing consistency in geodistributed transactions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*, Philadelphia, PA, USA: ACM Press, 2018, pp. 226–241 (cit. on p. 40).

[132]M. Milano, R. Recto, T. Magrino, and A. C. Myers, "A tour of gallifrey, a language for geodistributed programming," in *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019 (cit. on p. 40).

[133]M. Miller, "Pursuing durably safe system software," Symposium sur la sécurité des technologies de l'information et des communications (SSTIC), 2020 (cit. on p. 16).

[134]R. Mogk, L. Baumgärtner, G. Salvaneschi, B. Freisleben, and M. Mezini, "Fault-tolerant distributed reactive programming," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, T. D. Millstein, Ed., ser. LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 1:1–1:26 (cit. on p. 28).

[135]F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, "Jolie: A java orchestration language interpreter engine," *Electronic Notes in Theoretical Computer Science*, vol. 181, pp. 19–33, 2007 (cit. on p. 37).

[136]S. Mostinckx, T. Van Cutsem, J. Dedecker, W. De Meuter, and T. D'Hondt, "Ambient-oriented programming in ambienttalk," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, San Diego, CA, USA: Association for Computing Machinery, 2005, 92–93 (cit. on p. 22).

[138]D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds., ACM, 2013, pp. 439–455 (cit. on pp. 24, 25).

[139]M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro, "The CISE tool: Proving weakly-consistent applications correct," in *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, P. Alvaro and A. Bessani, Eds., ACM, 2016, 2:1–2:3 (cit. on p. 41).

[140]L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée, "Explicitly distributed aop using awed," in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, ser. AOSD '06, Bonn, Germany: Association for Computing Machinery, 2006, 51–62 (cit. on p. 36).

[141]M. Neubauer and P. Thiemann, "From sequential programs to multi-tier applications by program transformation," *SIGPLAN Not.*, vol. 40, no. 1, 221–232, 2005 (cit. on p. 29).

[142]C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, 2015 (cit. on pp. 41, 42).

[143]R. Neykova, L. Bocchi, and N. Yoshida, "Timed runtime monitoring for multiparty conversations," *Formal Aspects of Computing*, vol. 29, no. 5, pp. 877–910, 2017 (cit. on p. 44).

[144]N. Ng, N. Yoshida, and K. Honda, "Multiparty session c: Safe parallel programming with message optimisation.," *TOOLS (50)*, vol. 7304, pp. 202–218, 2012 (cit. on p. 45).

[145]M. Nishizawa, S. Chiba, and M. Tatsubori, "Remote pointcut: A language construct for distributed aop," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, ser. AOSD '04, Lancaster, UK: Association for Computing Machinery, 2004, 7–15 (cit. on p. 36).

[146]L. Padovani, "A simple library implementation of binary sessions," *Journal of Functional Programming*, vol. 27, 2017 (cit. on p. 45).

[147]R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli, "JAC: an aspect-based distributed dynamic framework," *Softw. Pract. Exp.*, vol. 34, no. 12, pp. 1119–1148, 2004 (cit. on p. 36).

[148]S. Perez De Rosso, D. Jackson, M. Archie, C. Lao, and B. A. McNamara III, "Declarative assembly of web applications from predefined concepts," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 79–93 (cit. on p. 37).

[149]B. C. Pierce, *Types and programming languages*. MIT Press, 2002 (cit. on p. 204).

[150]P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *22nd International Conference on Data Engineering (ICDE'06)*, 2006, pp. 49–49 (cit. on p. 26).

[151]R. Prieto-Díaz and J. M. Neighbors, "Module interconnection languages," *J. Syst. Softw.*, vol. 6, no. 4, pp. 307–334, 1986 (cit. on p. 43).

[152]L. Prosperi, M. Shapiro, and A. Bouajjani, "Varda: An architectural framework for compositional distributed programming," M. Mezini and M.-A. Koulali, Eds., vol. LNCS 13464, Online, May 2022, pp. 16–30 (cit. on p. 6).

[153]G. Radanne, J. Vouillon, and V. Balat, "Eliom: A core ML language for tierless web programming," in *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, A. Igarashi, Ed., ser. Lecture Notes in Computer Science, vol. 10017, 2016, pp. 377–397 (cit. on p. 30).

[154]B. Reynders, D. Devriese, and F. Piessens, "Multi-tier functional reactive programming for the web," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014, Portland, Oregon, USA: Association for Computing Machinery, 2014, 55–68 (cit. on p. 29).

[155]K. Sagonas, C. Stavrakakis, and Y. Tsiouris, "Erllvm: An llvm backend for erlang," in *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, ser. Erlang '12, Copenhagen, Denmark: Association for Computing Machinery, 2012, 21–32 (cit. on p. 23).

[156]G. Salvaneschi, G. Hintz, and M. Mezini, "Rescala: Bridging between object-oriented and functional style in reactive applications," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14, Lugano, Switzerland: Association for Computing Machinery, 2014, 25–36 (cit. on pp. 27, 28).

[157]B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri, "Plasma: Programmable elasticity for stateful cloud computing applications," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15 (cit. on pp. 24, 200).

[158]M. Serrano, E. Gallesio, and F. Loitsch, "Hop: A language for programming the web 2.0," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, P. L. Tarr and W. R. Cook, Eds., ACM, 2006, pp. 975–985 (cit. on pp. 29, 30).

[159]H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022 (cit. on p. 31).

[160]M. Shapiro, "Structure and encapsulation in distributed systems: The Proxy Principle," IEEE, Cambridge, MA, USA, May 1986, pp. 198–204 (cit. on p. 74).

[161]M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Tech. Rep. 7506, Jan. 2011 (cit. on p. 166).

[162]M. Shaw, "Myths and mythconceptions: What does it mean to be a programming language, anyhow?" *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, 2022 (cit. on p. 5).

[163]M. P. Singh, "Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2011, pp. 491–498 (cit. on p. 44).

[164]K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan, "Declarative programming over eventually consistent data stores," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 413–424, 2015 (cit. on p. 40).

[166]F. Steimann, "The paradoxical success of aspect-oriented programming," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, P. L. Tarr and W. R. Cook, Eds., ACM, 2006, pp. 481–497 (cit. on pp. 36, 37).

[167]J. Swalens, J. D. Koster, and W. D. Meuter, "Chocola: Composable concurrency language," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 4, 17:1–17:56, 2021 (cit. on pp. 22, 135).

[168]L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu, "Fail through the cracks: Cross-system interaction failures in modern cloud systems," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23, Rome, Italy: Association for Computing Machinery, 2023, 433–451 (cit. on pp. 4, 13, 34).

[169]G. Tato, M. Bertier, E. Rivière, and C. Tedeschi, "Split and migrate: Resource-driven placement and discovery of microservices at the edge," in *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, P. Felber, R. Friedman, S. Gilbert, and A. Miller, Eds., ser. LIPIcs, vol. 153, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 9:1–9:16 (cit. on p. 199).

[170]C. A. Thekkath, H. M. Levy, and E. D. Lazowska, "Separating data and control transfer in distributed operating systems," in *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, F. Baskett and D. W. Clark, Eds., ACM Press, 1994, pp. 2–11 (cit. on p. 29).

[171]C. Vidal, G. Berry, and M. Serrano, "Hiphop.js: A language to orchestrate web applications," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, Pau, France: Association for Computing Machinery, 2018, 2193–2195 (cit. on p. 30).

[172]S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Commun. Mag.*, vol. 35, no. 2, pp. 46–55, 1997 (cit. on p. 34).

[173]P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with ScalaLoci," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 129, 2018 (cit. on pp. 29, 30).

[174]J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 357–368 (cit. on p. 46).

[175]Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds., USENIX Association, 2008, pp. 1–14 (cit. on p. 24).

[176]M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016 (cit. on pp. 24, 25).

[177]J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, 1789–1806 (cit. on p. 41).

[178]D. Zufferey, "Verification of message-passing programs," C. Ferreira, P. Haller, and G. Salvaneschi, Eds., 10, Place: Dagstuhl, Germany Publisher: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, vol. 9, 2020, p. 125 (cit. on p. 12).

Webpages

[@1]. "Iptables." (), [Online]. Available: `https://www.netfilter.org/projects/iptables/index.html` (cit. on p. 36).

[@2]. "Kubernetes." (), [Online]. Available: `http://kubernetes.io` (cit. on p. 35).

[@3]. "Openapi." (), [Online]. Available: `https://www.openapis.org/` (cit. on p. 34).

[@4]. "OpenStack." (), [Online]. Available: `https://www.openstack.org/` (cit. on p. 35).

[@5]. "Podman." (), [Online]. Available: `https://podman.io/` (cit. on p. 35).

[@6]. "Rust." (), [Online]. Available: `https://www.rust-lang.org/` (cit. on p. 40).

[@8]Akka. "Akka." (), [Online]. Available: `https://akka.io/` (cit. on pp. 22, 23).

[@14]AntidoteDB. "Antidotedb." (), [Online]. Available: `https://antidotedb.gitbook.io/` (cit. on pp. 143, 163).

[@24]J. Bonér, D. Farley, R. Kuhn, and M. Thompson. "The reactive manifesto." (2014), [Online]. Available: `https://www.reactivemanifesto.org/fr.` (cit. on p. 28).

[@26]Buoyant Inc . "Linkerd." (), [Online]. Available: `https://linkerd.io/` (cit. on pp. 36, 201).

[@47]S Clebsch. "The pony programming language. the pony developers." (2015), [Online]. Available: `https://www.ponylang.io/` (cit. on p. 22).

[@62]Docker Inc. “Docker Engine.” (), [Online]. Available: https://www.docker.com/ (cit. on pp. 35, 184).

[@63]——, “Docker Swarm.” (), [Online]. Available: https://docs.docker.com/engine/swarm/ (cit. on p. 35).

[@74]C. Emerick. “Distributed systems and the end of the API,” The Quilt Project. (May 12, 2014), [Online]. Available: https://writings.quilt.org/2014/05/12/distributed-systems-and-the-end-of-the-api/ (visited on Jan. 14, 2020) (cit. on pp. 13, 34, 35, 121).

[@75]O. Facebook. “Rocksdb, a persistent key-value store for fast storage enviroments.” (2019), [Online]. Available: https://rocksdb.org/ (cit. on pp. 55, 99).

[@78]A. Foundation. “Thrift.” (), [Online]. Available: https://thrift.apache.org/ (cit. on p. 34).

[@86]I. Google. “Protocol buffers.” (), [Online]. Available: https://developers.google.com/protocol-buffers (cit. on p. 34).

[@103]Istio. (), [Online]. Available: https://istio.io/ (cit. on pp. 36, 201).

[@105]N. Kavantzas. “Web services choreography description language (ws-cdf) version 1.0.” (2004), [Online]. Available: http://www.w3.org/TR/ws-cdl-10/ (cit. on p. 37).

[@119]“Lingua franca.” (), [Online]. Available: https://github.com/icyphy/lingua-franca/wiki/Overview#reactors (cit. on pp. 38, 48, 131).

[@137]Mozilla. “Service Worker.” (), [Online]. Available: https://developer.mozilla.org/fr/docs/Web/API/Service_Worker_API (cit. on p. 36).

[@165]I. SmartBear. “Swagger.” (), [Online]. Available: https://swagger.io/ (cit. on p. 34).

List of Figures

List of Tables

Résumé

A

Les grands systèmes distribués sont souvent construits en assemblant des composants prêts à l'emploi (OTS), c'est-à-dire des composants, des services, des processus, etc., développés indépendamment. La pratique actuelle consiste à interconnecter manuellement leurs API. Cette méthode est ad hoc, complexe, fastidieuse et sujette aux erreurs.

Les langages de programmation offrent une approche prometteuse pour résoudre ce problème. Premièrement, ils permettent de *réduire l'apparition de bogues*. Le développeur spécifie formellement le système. Ensuite, le compilateur assure des garanties de correction et génère une implémentation correcte par construction. Deuxièmement, les langages contribuent à *améliorer la productivité des programmeurs*. La génération de code automatise l'interconnexion des composants et peut optimiser l'implémentation générée.

Dans cette thèse, nous proposons un nouveau langage, Varda, à l'intersection entre langages de programmation et de spécification. Un programme Varda décrit l'interconnexion de composants en une architecture cohérente. Le développeur isole un composant OTS derrière un *shield*. Celui-ci restreint le comportement du composant en spécifiant son interface, son protocole (c'est-à-dire ce qu'il peut envoyer ou recevoir, et dans quel ordre), ainsi que des pre/post-conditions. Les composants peuvent être logiquement imbriqués. Un composant externe orchestre ses composants internes, en créant ou en tuant des instances, en les interconnectant et en supervisant leurs erreurs ; il peut intercepter et manipuler leurs communications. Varda fournit des garanties solides, grâce à de l'analyse statique et à de l'injection de tests dynamiques.

Pour être utile en pratique au développement de systèmes distribués, Varda adopte une approche pragmatique. Un *shield* peut contenir du code non-Varda (par ex, Java) afin d'incorporer des composants en boîte noire. Varda permet de contrôler les propriétés non fonctionnelles qui sont importantes pour la performance et la tolérance aux pannes, par exemple l'élasticité, le placement et l'intégration. Le compilateur Varda automatise la génération du code d'interconnexion et d'orchestration. Varda prend en charge l'exécution non-stop, grâce à des mécanismes de supervision.

Nous démontrons l'expressivité et l'ergonomie de Varda en programmant des structures distribuées classiques (par exemple, le contrôle d'accès). Pour illustrer l'emploi en condition réelle de Varda, nous implémentons un clone du datastore géorépliqué AntidoteDB.

Les applications Varda sont compactes et présentent une conception modulaire et réutilisable. Nos expériences montrent que le surcoût d'exécution est modeste, grâce à la vérification et à l'optimisation au moment de la compilation.