Álvaro António Silva Ferreira Vieira dos Santos

Bachelor of Science

Distributed Live Programs as Distributed Live Data

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics

Adviser: João Ricardo Viegas da Costa Seco, Associate Professor, NOVA University of Lisbon

Co-advisers: João Carlos Antunes Leitão, Assistant Professor, NOVA University of Lisbon
Mário José Parreira Pereira, Postdoctoral Researcher, NOVA University of Lisbon

Examination Committee

Chair: Carmen Pires Morgado, Assistant Professor, FCT NOVA
Rapporteur: Hugo Torres Vieira, Researcher, Universidade da Beira Interior
Member: João Costa Seco, Associate Professor, FCT NOVA

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2020

Distributed Live Programs as Distributed Live Data

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

In a distributed system, reasoning about the impact of modifying one of its components is non-trivial and may require a large bookkeeping effort to manually guarantee that the assumptions of its consumers are not broken — in fact, for the extreme case of a system where every component depends on all others, the bookkeeping effort grows quadratically in the number of system components.

As microservice-based architectures become evermore widely adopted, finding ways to manage their evolution also becomes more important. Depending on how critical a microservice is in a system, a single erroneous change leading to it failing may cause a cascade of failures across other microservices. Because of this, there is a need for ways to reliably ascertain the impact that a change to a component may have on the system as a whole. Whereas typical approaches to ensure that microservices are behaving "properly" try to verify this via empirical test-based validation, our approach attempts to ensure this via a type-based relation between specifications (descriptions of microservices' interfaces).

In this dissertation we present Regent, a distributed tool that uses the aforementioned relation to preserve the "proper" behaviour of a distributed system by rejecting additions or modifications that would threaten it. In some cases, Regent may detect that a microservice is not directly compatible with the rest of the system but could be made so if slightly modified. When this happens, Regent uses its knowledge of the microservice's interface to create a lightweight proxy capable of automatically adapting its interface at runtime, which frees up the programmer from having to manually adapt either the existing microservices or the new one.

Keywords: distributed systems, distributed system evolution, programming languages, microservices, microservice architectures

Resumo

Num sistema distribuído, raciocinar sobre o impacto de modificar uma das suas componentes é não-trivial e pode requerer um grande esforço de *bookkeeping* para garantir manualmente que as assunções dos seus consumidores não são quebradas — aliás, para o caso limite de um sistema em que cada micro-serviço depende de todos os outros, o esforço envolvido neste *bookkeeping* cresce quadraticamente com o número de componentes do sistema.

Com a crescente adopção das arquitecturas à base de micro-serviços, encontrar maneiras de lidar com a sua evolução também se torna mais importante. Dependendo de quão crítico um micro-serviço é num sistema, uma única mudança errónea que leve a uma falha pode causar uma propagação catastrófica de falhas por outros micro-serviços. Por esta razão, são necessárias formas de verificar o impacto que uma mudança numa componente pode vir a ter no sistema como um todo. Enquanto que as abordagens típicas para garantir que os micro-serviços se estão a comportar "bem" são à base de validação empírica por via de testes, a nossa abordagem tenta garantir isto através de uma relação à base de tipos entre especificações (descrições das interfaces dos micro-serviços).

Nesta dissertação apresentamos o Regent, uma ferramenta distribuída que usa a relação lógica supramencionada para preservar o "bom" comportamento do sistema distribuído através da rejeição de adições ou modificações que a ponham em causa. Em alguns casos, o Regent consegue detectar que embora um micro-serviço não seja directamente compatível com o resto do sistema seria possível alterá-lo ligeiramente para que ele o passasse a ser. Quando isto acontece, o Regent usa o seu conhecimento das interfaces dos micro-serviços para criar um adaptor *lightweight* capaz de as modificar automaticamente em tempo de execução, eliminando a responsabilidade do programador de adaptar manualmente ou os micro-serviços preexistentes ou os que estão a ser criados.

Palavras-chave: sistemas distribuídos, evolução de sistemas distribuídos, linguagens dde programação, micro-serviços, arquitecturas de micro-serviços

Contents

List of Figures

Listings

Chapter 1

Introduction

In this chapter, we give a brief overview of the problem we're tackling, of why it is a problem worth solving, and of how we intend to solve it. This overview starts with a discussion of programming in general and gradually narrows down to focus on the issue of facilitating the robust evolution of microservices, concluding with a broad description of how our approach works and the principles that guided it.

1.1 Context & Motivation

1.1.1 Programming is Conceptually Easy

Conceptually, programming a computer is no different than writing a recipe or giving a passerby directions to reach their destination: given a set of operations (preparing ingredients, walking in different directions), the programmer must order some of them in a sequence ("first heat the water to a boil, then throw in salt", "walk straight ahead and turn left at the third intersection") which materialises their intent (cooking a meal, reaching a place). The specifics of which operations are available to the programmer vary, but whether they're the primitive bitwise operators of an 8-bit processor or advanced uses of metaprogramming in a high-level language, the underlying process remains one of simply composing steps in a sequence.

And yet, anyone who has ever written a program to solve a non-trivial task will disagree with the implication above: even if the underlying process is "one of simply composing steps in a sequence", this view of programming disregards the issue of discovering which of those steps are relevant and designing a program out of them. The following observation empirically supports the complexity of programming: if it were a simple endeavour, then there'd be no motivation for software design methodologies, type systems, or formal program verification to still be active fields of study — they'd already have been “solved”, or they wouldn’t have been “needed” in the first place. Our argument is thus: programming is a *deceptively* complex activity.

This complexity depends on many factors: the tension between minimising execution speed and memory use while maximising portability and ease of comprehension, the choices in modelling the problem, the difficulty of understanding the problem in the first place, the choice of which combination of {hardware, middleware, compiler, tooling, representation format, (...)} is appropriate under the {temporal, monetary, familiarity, (...)} constraints the developers are under, and so forth. Notice however that none of these factors change the nature of programming from one of arranging instructions in a sequence – hence, why programming may at first seem (deceptively) easy.

1.1.2 Distributed Systems & Microservices

“Distributed systems” is a catch-all term for systems composed of multiple independent processes which communicate by exchanging messages across a network, and it is one of the areas where the disparity between theory and practice is most pronounced. The canonical example of this is the Paxos algorithm [12]: although it can be elegantly described at a high level as three types of nodes exchanging a few types of messages, it is notoriously complex to implement in practice since it has a multitude of variants optimised for different use cases and it is difficult to verify the correctness of the implementation. This theory/practice disparity is present even in simple distributed systems, as their distributed nature inherently forces them to deal very explicitly with issues like the distribution of work among the system’s components, the coherence of data despite concurrent execution flows and failures, and the availability of the system.

One way of designing distributed systems has gained significant traction in recent years: microservice architectures. Microservices emphasise the importance of factorising a system into functional units with well-defined boundaries, akin to objects in an object-oriented software architecture, concentrating as few responsibilities as possible in a single service. The goal of these architectures is to promote the decoupling of service components, which in turn allows for the components to be swapped, modified, or replicated while minimising the disruption to the rest of the system.

In an ideally architected system composed of microservices, the independence of the components enables multiple teams of programmers to work in parallel without interfering with each other. In practice, this ideal state is mostly unachievable: any non-trivial update will likely modify a component’s existing interface, which may in turn affect other parts of the system by breaking their expectations, which may themselves affect other parts of the system, and so on. Worse still is that, by design, the communication patterns of microservices are unpredictable without manually examining their behaviour and configurations, making it difficult to gauge how far a change to a single component might cascade.

Regardless, the wide adoption of microservice architectures vouches for their usefulness and indicates that there must be some way of mitigating these shortcomings without jeopardising the benefits of these architectures. In particular, given that updating a component's interface is an inevitability in any application's lifecycle, developers must have some way of guaranteeing that components will remain intercompatible after being updated.

1.1.3 Safeguarding the Evolution of Distributed Systems

For most developers and across most domains, verifying that a system is behaving as expected after an update is synonymous with empirical testing. Concretely, a typical developer will usually validate their code by writing unit and integration tests[1]. Unit tests verify that a self-contained piece of code works as expected, while integration tests verify that two or more self-contained pieces of code interface as expected. As an example in the context of microservices: a unit test might verify that a microservice's HTTP endpoint returns the expected value for a given input, while an integration test might verify that a caller microservice never violates the assumptions of the callee microservice's API.

These tests can be designed manually by the programmer, using only their intuition and domain knowledge, or semi-automatically, using property-based testing tools such as QuickCheck [11] and its reimplementations; Objects under test may be treated as "opaque", foregoing any knowledge of their internal structure, or as "clear", using the details of the implementation; Tests can be run in an *ad hoc* manner by the programmer, or systematically as a battery managed by a continuous integration platform. The key observation to be made is that empirical tests, regardless of their characteristics, are limited by the programmer's imagination: one cannot write a test for conditions that one cannot think of.

A complementary approach to testing is the use of formal methods. In general this approach is not as popular, which might be related to the unfamiliarty of working programmers with the tools that facilitate it, to the upfront cost of having to learn and fit yet another tool into their development process, to the (real or perceived) lack of scalability of these tools from academic examples to "real world" programs, or to a combination of all these factors and others. The domains where formal verification methods enjoy most use are those where even a simple bug may have catastrophic consequences (medical devices, nuclear reactors, space missions, financial software) or where correctness must be certified (compilers, static analysis tools, computer-assisted proofs). A formal verification method may be as simple as having the programmer annotate their code with what assumptions (pre-conditions) they're making about it, what results (post-conditions) they expect to see after the code runs, and running a compiler-like program on these groups of

[1] Note that this is by no means an extensive list of the types of testing developers perform; We merely believe that they broadly represent the kinds of tests that developers are most concerned with.

annotations + code (more commonly known as "Hoare Triples") to automatically detect mismatches between assumptions and results.

With the notable exception of Amazon Web Services and their publicised usage of TLA+ [16], formal verification methods seem to be much less popular than empirical testing in the context of "real world" distributed systems. Fundamentally, this should seem odd to an impartial observer untainted by the knowledge of real world programming practices: distributed systems are often part of a business's critical infrastructure, but they're hard to reason about due to their many concurrently-moving and decoupled parts and because a single fault has the capacity to cause a cascade of failures throughout the system. While not all distributed systems are equally business-critical, these qualities should place distributed systems closer to the category of "domains where (...) correctness is absolutely critical" than where real world development practices appear to place them.

We posit that this lack of adoption of formal methods in the domain of distributed systems is partially due to the same reasons as their lack of adoption in other domains, and partially due to a mismatch between the nature of these systems — which are highly non-deterministic due to the concurrency and unpredictability of their interaction patterns and heterogenous in the technologies they use — and the way most formal verification tools operate by assuming a closed world of homogenous and fully accessible source code. Further, we posit that this mismatch is even more pronounced in the specific context of microservices, where the number of "moving parts" and degree of decoupling between different components are even higher than in a non-microservice-based distributed system.

1.2 Problem Statement & Approach Overview

1.2.1 The Problem

In simple terms, the problem we're trying to solve is the following: **how can the evolution (update) of a distributed system's components be made more robust, particularly in the context of microservices?**

As we described in the previous section, most developers deal with this problem by writing tests for their programs. While this is a perfectly valid approach with a low entry barrier (i.e., anyone can write a test, moreso if they're writing black box tests which require no knowledge of the code's implementation choices), it is not exhaustive — in the sense that it cannot, in general, check the output of all possible inputs — and is thus brittle in the face of changes to the code. Moreover, even if testing-based approaches being brittle weren't a problem, there's a limit to how automated their construction can be due to requiring the programmer's domain knowledge.

For these reasons, our approach to tackle this problem will instead be to build a tool which draws inspiration from formal verification approaches. As we also mentioned in the previous section, despite being powerful enough to be used in domains where correctness is critical, these approaches do not generally enjoy as much use as testing-based approaches in the distributed systems community. Of the reasons we presented for why this might be the case, we now focus specifically on the "mismatch between the nature of these systems (...) and the way most formal verification tools operate".

1.2.2 The Mismatch

To explain the mismatch we speak of, we must examine the differences between developing non-distributed software and developing a distributed system. In a non-distributed program, calls to other pieces of code typically consist of function calls between modules written in the same language and in the same process, with the exceptions to this usually being quarantined in their own libraries and exposed only through an interface that encapsulates and abstracts away the details of communicating outside the language and/or process.

Individual components/nodes in a distributed system are also usually written in a single language, possibly with a second language employed to execute some database queries or other minor tasks. However, by virtue of being part of a distributed system, each component likely has to extensively communicate with other remote processes. Hiding the implementation details of interprocess communication is harder to do than hiding the details of a function call within a process (and occasionally undesirable, in order to specifically handle different kinds of communication failures). Worse still is that communication between pairs of nodes is not necessarily uniform nor predictable: for example, a pair of nodes may communicate by sending JSONs over a REST interface while another pair may communicate using naked UDP sockets with an *ad hoc* protocol and binary blobs.

The effect of these differences on the development experience is significant, as refactoring any piece of software will typically involve modifying some of the interfaces it exposes, whether they're exposed in the form of function definitions to be called across modules in the same program or as endpoints to be called across a network. Although refactoring a piece of code can be seen as merely editing lines of code, the qualitative experience of finding the invocations of a function in a "regular" program is obviously different from (and better than) the experience of trying to find uses of an endpoint across a distributed system. Concretely, a developer will usually find invocations of a function with the assistance of an IDE or by performing a simple textual search for the function's name using tools like grep; Meanwhile, finding the invocations of an endpoint will typically require either a dynamic analysis of the requests performed by the system or a text search across multiple codebases which are not necessarily uniform in their calling conventions — for example, one codebase may have the endpoint's URL hardcoded while another may build it dynamically through calls to an HTTP library.

In a non-distributed program, refactoring a function's definition and calls can be done either manually, possibly aided by a search-and-replace tool, or automatically using an

IDE's refactoring functionality, while benefitting from tools like compilers and linters to automatically detect remaining mismatches between a function's definition and its usages afterwards. Distributed systems development, in contrast, does not have equivalent compiler/linter-like tools to detect mismatches between endpoints and their consumers, nor are IDEs generally capable of automatically refactoring an endpoint's consumers whenever its definition is changed, leaving developers with the burden of tracking endpoint dependencies and refactoring them manually. While some approaches can mitigate this burden — such as having each consumer maintain a copy of the interfaces they consume (i.e., a stub) to facilitate searches later on —, tracking the interdependencies of a distributed system's components will always involve a trade-off with their "open world" nature of decoupled components and fluid interactions.

There is one way of maintaining the interfaces of distributed systems' components which we have not yet mentioned. Essentially, it consists of describing components' interfaces using a high-level specification language such as OpenAPI Specification [22] or Protocol Buffers [9], and then using a tool like the OpenAPI Generator [23] or the Protocol Buffer Compiler to generate code implementing that interface in a target language. However, this approach has a significant limitation: the code generation process must be repeated every time the interface changes, no matter how small the change. However, there is a clear benefit to the ability to describe components at a high-level without worrying about the details of how they are implemented — and for this reason, our own approach will also employ the use of high-level specifications.

Finally, in order to explain the mismatch motivating this section, we must examine how most formal verification methods operate. While there is a breadth of approaches in the field, and we make no claim of being aware of all of them, nor do we claim to capture their full diversity in a short section whose main goal is not to survey the area, our experience tells us that the qualities that follow apply to most of the formal verification tools that enjoy "real world" use (i.e., that are mature enough for a non-academic to consider using them in a professional setting). In no particular order:

- Most formal verification systems fall into one of two categories: they either force the programmer to write their programs in a language written for the special purpose of interfacing with the verification system, or the system supports only a limited range of programming languages — which are not necessarily the ones the programmer would like to use and/or may restrict the usage of certain libraries and language features. Although some of the tools in the first category allow the extraction of code into other languages (i.e., the programmer writes their code in the special purpose language and the system transpiles it to a "real" language), mechanically generated code runs the risk of becoming hard to interpret past a low threshold of complexity.

- Some systems make no explicit allowances for the existence of more than one flow of execution (thread, process, etc), putting the onus of expressing/modelling the primitives needed to work with concurrency (locks, mutexes, semaphores, thread-safe data structures, etc) on the programmer instead of providing them "out of the box". In a similar vein, most systems only model behavioural aspects and offer no direct tools to capture the time/memory/energy consumption/etc costs of a piece of code — e.g., they do not facilitate capturing the differences between an implementation of bubble sort and an implementation of quicksort, as long as both output a sorted list of items.

- Typically these methods assume that all parts of the system have immediate and global knowledge of all other parts of the system — for example, they assume that if a function's signature is changed, then it's feasible to require all of its callers to be refactored to match the new signature before considering the system to be in a valid state.

The mismatch should now be obvious: even though some of these qualities already pose a significant challenge when writing non-distributed programs, they are especially antithetical to the way distributed systems are developed and how they execute in practice. In particular, we have the assumption of immediate and global knowledge: while this is a relatively benign assumption in a "regular" program, written as a group of modules that produce a single executable artifact to run under the purview of a runtime, it does not scale to systems that promote the independent evolution of its different components — components which will then communicate using weakly defined interfaces (i.e., without a compiler or linter verifying their correct usage), for whom guaranteeing a consensual global vision of these interfaces is often unviable under "normal" assumptions of how networks function (as per the CAP and FLP theorems[2]), and which are managed by multiple disparate runtimes. Obviously, this mismatch is amplified even further in the context of developing microservices, as one of the benefits of microservice architectures is precisely allowing for a great deal of parallel and independent development of the system's components and their reuse.

1.2.3 Our Approach

In achieving our goal to support the robust evolution of microservices, we must consider the issues we have just discussed. The following are the principles which have guided the design of our solution, in no particular order:

- **Being Tool-Agnostic:** one of the great benefits of microservice architectures is not only allowing developers to use whatever technologies they feel to be most adequate

[2] A detailed exposition of these theorems falls outside the scope of this document. However, they can be roughly summarised as stating that it is impossible to provide a hard guarantee of every single component in a distributed system agreeing on the same result in a finite amount of time, if communication is asynchronous and the network may arbitrarily lose or delay messages (i.e., as is the case for IP packets). Merely probabilistic-but-not-certain guarantees are often good enough in real world scenarios, however.

for the task, but also allowing them to later change their choices by making components easily replaceable. Thus, our approach should be as independent as possible from the details of which tools, languages, designs, protocols, and paradigms the programmer chooses to employ.

- **Integrating Into Development Workflows:** in designing a tool, it is important to understand how its users will use it. Although the specific details of each developer's process for building software are unique to them, it is still instructive to to think of how the "common" development process may take place. Our tool tries to integrate into this process as seamlessly as possible, by mimicking the approaches taken in other tools used to develop and deploy microservices. As a concrete example, our approach of separating deployment information from interface descriptions mimicks the approach taken when deploying a system using Kubernetes.

- **Caller-Callee Independence:** as we identified in the previous section, formal verification systems tend to operate under an unspoken assumption of it being reasonable to require that all of a system's information be updated at once. In the case of a distributed system, this would manifest as being unable to update a component's interface without changing all of its consumers to match the new version.

  However, this assumption is neither realistic at design time nor at runtime for distributed systems: at design time, a team of developers may not be aware of the latest changes to the rest of the system (and it can be argued that, even if they could be made immediately aware of all those changes, components should be decoupled enough for their intercommunication to resist some amount of change without breaking); at run time, the distributed nature of microservice-based systems imposes hard limits on how fast and reliably a message can be disseminated among a group of nodes, and requiring that the entire system be stalled until every component can be updated obviously works against one of the core benefits of microservice architectures (the ability to rapidly iterate on and replace parts of the system).

  Thus, we have the principle of caller-callee independence: to the maximum extent possible, callers should be able to evolve without requiring that their callees be modified in tandem, and vice-versa.

- **Lightweightness:** this is a predictable goal, but an important one nonetheless. Resources are not infinite — and even though many microservices are deployed to cloud infrastructure, the elasticity of this infrastructure may have non-negligible monetary costs —, and latency is a particularly important factor in distributed settings. For these reasons, our approach cannot be so complicated that it impacts the system's execution to the point where running our tool is more costly (in time, money, or another metric) than running the system by itself.

Given these principles, all that remains is to describe our approach in detail and how we believe it respects them. Succintly, our approach is a combination of high-level interface descriptions, similar to OpenAPI Specifications and Protocol Buffers, a simple type compatibility mechanism, akin to the typechecking mechanism known to any programmer who has used a mainstream compiled language such as Java or C, and a notion of "adaptation", which can be likened to the notion of type coercion also present in many mainstream languages. The high-level interface descriptions in our approach are of two kinds: specifications, which detail what endpoints a microservice offers (its definitions) and what endpoints of other microservices it consumes (its dependencies), and deployment files, which specify configuration details that are not directly related to microservices' interfaces.

For the remainder of this section, we present our approach as explored and implemented in our prototype ("Regent"). Although this introduces some details that are not strictly needed for our approach on a conceptual level, such as the usage of Docker, it greatly simplifies the presentation of our ideas. This presentation is revisited and expanded upon in chapter 3.

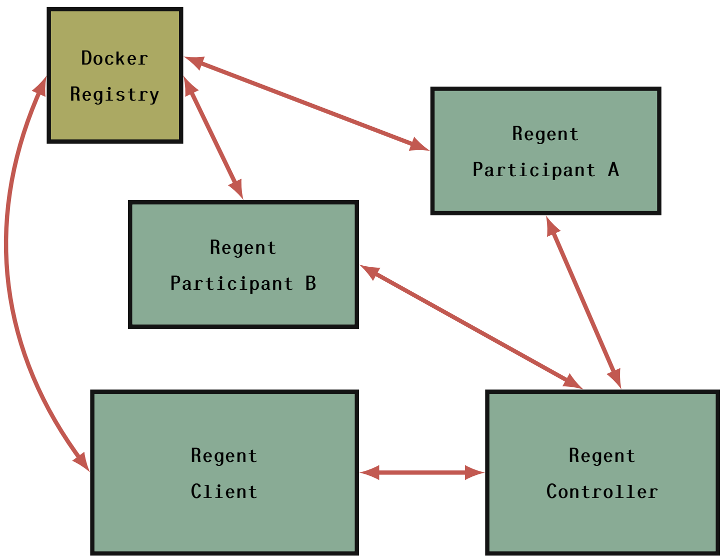

Figure 1.1: Example of a Regent set-up and component interactions during a deployment process *(simplified from figure 3.1)*.

In figure 1.1, we present a simplified version of a Regent architecture and of which components interact with each other during a deployment of microservices. In this example, we assume there's a single client issuing requests to Regent and two servers ("participants") on which microservices may be deployed, but set-ups with different amounts of clients and participants are also possible. This figure is meant to serve only as a visual aid during the explanation that follows; We revisit it in greater detail in chapter 3 (cf. figure 3.1).

Specification and deployment files are sent from the client to a central server (the "Regent Controller"), which decides whether to deploy the microservices or to reject their deployment, whereas microservice artifacts (concretely, Docker images) are uploaded onto a registry outside of Regent. In the case of a microservice exposing an HTTP interface, for instance, its endpoints are composed of an URL path, information about the format in which it expects to receive and return values (e.g., as URL query parameters, as a JSON matching a template, etc), type annotations for the values it will handle (e.g., whether a query parameter must be a string or an integer), and other miscellaneous information (such as which HTTP verb to use). In turn, the deployment file will contain information such as which specification should be associated with each microservice, where the microservice's executable artifact is located, and onto which physical machine the microservice should be loaded.

These specification and deployment files are processed by the central server under an holistic point of view: the correctness of the system as a whole must be guaranteed, and so the server will only allow microservices to join the system or be updated if it can be guaranteed that they will respect all of the system's remaining components. In practice, this means that the central server will evaluate type compatibility between all dependencies and corresponding definitions, the exact details of which we leave for a later chapter. Only if this evaluation succeeds will the central server accept the deployment and proceed to instruct the machines in the system (the "Regent Participant"s) to load the new microservices.

Finally, the central server communicates with the machines where the microservices will be hosted and gives each one instructions on what to deploy — corresponding to the interaction between the "Docker Registry" and the participants in figure 1.1. It is here that the notion of "adaptation" mentioned earlier comes into play: the idea is that the type compatibility check performed by the central server may have succeeded even if the definitions presented by each interface do not exactly match their consumers' expectations, allowing a greater range of permissible deployments at the cost of having to dynamically correct for these differences at run time. As a concrete but simple example, consider an endpoint that accepts a JSON object with a single field `user_id` and a microservice which attempts to supply a JSON object with two fields `user_id` and `username` — obviously, this endpoint call can be automatically corrected by dropping the extraneous `username` field from the object. In our approach, we perform this adaptation by injecting a small proxy into each microservice (prior to starting it up), through which communications are automatically rerouted and where we can drop fields or perform any other relevant modifications.

Although we have elided some finer points of our approach in this description (such as how deployments can be undone), it is sufficiently detailed to see how it enables our goal of supporting a robust evolution of microservices under our stated guiding principles. The **main goal** is achieved because all microservices must be deployed with specifications describing them, which makes it impossible to accidentally break the compatibility between components as long as the specifications are accurate. The **principle of caller-callee independence** is achieved by allowing interfaces to evolve independently and still retain their ability to communicate with one another through the use of automatic proxies. The **principle of tool-agnosticism** is achieved because our approach does not require any specific technologies to work[3]. The **principle of integrating into the typical workflows** of microservice developers is achieved as our approach draws from tools in current use by the industry — such as OpenAPI Specifications and Protocol Buffers, which also require that the programmer describe the interfaces of their programs, and continuous integration platforms like Jenkins, as these also require that the user submit their code to a server which performs some checks (running tests, in this case) and then either accepts it or rejects it. Lastly, the **principle of lightweightness** is achieved because our approach only requires a server which runs a simple typecheck-like verification, and some proxies to rewrite network messages.

1.3 Document Structure

The remainder of this document is organised as follows:

- **Chapter 2: Related Work & Background** – A short discussion of the main concepts that motivated Regent, and of the tools and techniques which it resembles the most.
- **Chapter 3: Design, Architecture, & Implementation** – An in-depth description of how our approach and tool are designed, along with how each part of our system works with one another.
- **Chapter 4: Evaluation** – A discussion of how we evaluated Regent qualitatively.
- **Chapter 5: Future Work** – A discussion of the directions in which we see the potential to further develop Regent.

[3]Naturally, our implementation does make some assumptions about the technologies used by the microservices being deployed — namely, it assumes that microservices are built as docker images, that communication is performed by sending JSONs over HTTP, and other such things. However, this is not intrinsic to our approach, but rather due to the fact that it's prototype written in the context of a Master's Degree dissertation.

Chapter 2

Related Work & Background

In this chapter, we discuss both the prior work from which Regent draws (background) and how the existing approaches to manage microservices relate to ours (related work). Rather than separate our presentation into "Background" and "Related Work" sections, we have chosen to dedicate a section per relevant concept — in which we discuss both kinds of contributions accordingly.

2.1 Microservices

Although microservices are a popular way to design distributed system architectures, an exact definition of the terms "microservice" or "microservice architecture" is elusive and depends on the source used. Despite this, some characteristics that a distributed system's component should have to be considered a microservice seem to be generally agreed upon [5, 17, 26, 28]:

- Microservices are executable units with a small well-defined set of concerns;
- Microservices are stateless in order to be easily replaceable on failure[1];
- Microservices communicate by exchanging messages over a network;
- Microservices should be decoupled from one another and handle failure of their communication partners gracefully.

These characteristics purportedly enable systems designed as microservice architectures to be highly scalable, available, resilient to faults, and modular. In particular, their

[1]Codified in the popular epigram "Cattle, not pets." — i.e., a developer working with an instance of a microservice should be able to "slaughter" it and replace it by an identical copy without any significant disruption to their system's functioning.

modularity and independence should allow developers to better work on independent components in parallel when compared to a monolithic system, making it easier to evolve the system when it gains new requirements or in order to fix bugs.

2.2 Live Programming

"Live Programming" is an overloaded term, which depending on context can describe things as disparate as integrating software into artistic audiovisual live performances [24] or program structure editors (which operate on code as an abstract syntax tree, rather than as text) [18]. Encompassing as it may be, the core meaning of the term seems to revolve around the concept of minimising the latency between a user's actions being presented back to them and the integration of this feedback in useful/intuitive/easily usable ways into the user's experience. For the previous examples, this is respectively done by allowing the user to rewrite a piece of music-generating code as it runs and immediately having it affect the tune without a perceptible pause, and by immediately updating the different subsections of the code the user is writing to reflect the known (and possibly incomplete) type information about them.

Although Live Programming as a paradigm is not overtly present in Regent, the underlying "philosophy" of having a tight loop between programmer action generating feedback which enhances future actions still served as a guiding principle in its design. Specifically, the principles of which we speak in section 1.2.3 were partially motivated by the idea that the system, as a whole, should be able to react quickly to the user's actions. The principle of "caller-callee independence" (from section 1.2.3), for instance, is just a way to codify that the user should be able to quickly change a few microservices and replace them in the system without having to hunt around for and update all of their dependencies (which would increase the latency between their actions and the feedback of "this worked/did not work" provided by our system).

2.3 Type Systems

At various points in this document, we mention a compatibility relation between Regent specifications and liken it to the verifications performed by a typechecker in a "typical" programming language (in particular, section 3.2 is solely dedicated to this relation). In fact, the type system which our compatibility relation most closely resembles is the "Simply Typed Lambda Calculus" extended with records and subtyping ($STLC + rec + <:$) as presented in Benjamin Pierce's "Types and Programming Languages" [19] — a system even *simpler* than that of most mainstream programming languages.

We also mention "adapting" values between communicating microservices when the value one sends is not exactly what the other expects to receive. Without detailing the specifics of this process for now, it suffices to say that the endpoints in Regent's specifications can be likened to function types manipulating integers, strings, and records, and that verifying their compatibility or performing an adaptation from one value to another is done similarly to the way a typechecker for the $STLC + rec + <:$ would verify if a record is a subtype of another. At a high level, the way this verification works is by (1) examining both record types and checking that the candidate supertype contains all the keys as — and possibly more than — the subtype and (2) that the types of matching keys also respect the subtyping rules.

Our compatibility and adaptation relations are heavily inspired by those of Seco et al. in "Robust Contract Evolution in a TypeSafe MicroServices Architecture" [21]. In their work, Seco et al. similarly define a compatibility relation that verifies whether a group of services might "work well" with another, and introduce a concept of adaptation to modify values that could be automatically made to "work well" without impacting the system's correctness. The main difference between our relations is that in their work, the language used to define and compare the services' interfaces is the same as the programming language used to implement them. This allows their relations to have access to a richer set of information than ours — i.e., they can associate an unique immutable key to every value in order to track what happens to it as it's passed around the system —, since in Regent the definition and comparison of interfaces is independent from the programming languages used to write the microservices.

2.4 Microservice Creation & Management Tools

Due to the sheer breadth of tools to manipulate microservices, it is hard to give a complete overview of the tools in this domain. Of all these tools, though, the Docker ecosystem is the *de facto* set of tools for anything and everything microservice-related. Among this ecosystem, Docker [8] itself and Kubernetes [25] were the biggest influences in Regent's development.

Docker has two main functions: packaging software into "images" and running these images as containers. Docker images are essentially bundles of instructions describing how to build an executable artifact — typically, a linux-based environment with some custom processes and configurations. Docker's value proposition is its portability and composability: its portability comes from the fact that an image built on one computer should run in the exact same way on all others by issuing a simple `docker run image_name` command, while its composability comes from the fact that images can be extended/reused to build other images.

However, by themselves these characteristics would also be achievable with a traditional combination of virtual disk image and virtual machine. What sets a Docker container apart from a virtual machine is how it offers this portability/composability in an extremely lightweight manner by sharing more of the virtualisation stack across different containers — i.e., whereas each instance of a virtual machine has its own copy of the kernel/drivers/other utilities, docker shares some of these fundamental resources across multiple containers.

Kubernetes can be understood as being a manager for Docker containers. While Docker solves the problem of packaging up and running software, it does not handle operational or architectural questions. For instance, Kubernetes can automate the dynamic up/downscaling of containers in response to traffic, the process of service discovery via a DNS-like system for microservices to find each other, and the monitorisation of each container's responsiveness, whereas a developer using only Docker would have to perform these tasks manually or with *ad hoc* scripts.

Docker is instrumental in Regent's process. Although there's nothing intrinsic to the Docker platform that no one other container/virtual machine tool could offer, by assuming that microservices being deployed to Regent are running on Docker we can design our system against a common, well-defined, and well documented interface. As we explain in detail in chapter 3, our prototypical implementation of Regent uses its knowledge of Docker to transparently configure and modify the images being deployed onto itself.

Meanwhile, Kubernetes's influence on Regent is subtler than Docker's. Like Kubernetes, Regent also has to handle some concerns related to managing microservices — namely, how to map microservices' deployment details (which ports are open with what purpose, which IP address/URL the microservice is available at, etc) to their interface specifications. Kubernetes's pod templates influenced our decision to separate our microservices' interface specifications from their deployment configurations, increasing the usability of our system by allowing specifications to be reused across different microservice instances (instead of requiring that users wanting to launch multiple copies of a microservice maintain multiple copies of a specification differing only on some IP addresses or ports, as discussed in chapter 3).

Regent also shares some similarities with tools used for service discovery/registration such as Zookeeper, Eureka, or Kubernetes' own DNS-like mechanism. Like these services, Regent dynamically maintains a mapping of microservice instances to the machines they're running at. Although service discovery is not Regent's main purpose, it is needed for Regent to know which requests/responses between microservices must be intercepted and how they must be adapted. Additionally, this service discovery functionality is also exposed in a human-readable format so that developers can dynamically explore their current deployment in terms of which microservices are available at which locations and which endpoints they're offering. While this may seem like a minor detail, it follows the principle of "integration into development workflows" we laid out in section 1.2.3: this way, developers do not need to adjust their workflows by introducing an additional process to track the state of their deployed system.

Lastly, we mention two tools for designing and managing distributed systems that share various similarities with Regent: Jolie [15], a programming language, and CHOReVOLUTION [2, 3], an "Integrated Runtime and Runtime Environment" (IDRE). Although these tools did not directly influence Regent, it is still interesting to compare our approach to theirs.

Jolie is a programming language designed to unify the behavioural, architectural and deployment aspects of building services. Jolie does this by having language-level constructs to support all these aspects seamlessly; A program consists of a series of service interface descriptions, — i.e., "ports", descriptions linking interfaces to deployment information such as URIs and protocols —, and executable code for performing computations. In addition to its capabilities as a general-purpose language, Jolie has mechanisms to support service implementations in other languages, making it possible to integrate existing services into it rather than having to rewrite them from scratch.

The main similarities between Regent and Jolie are that both emphasise the importance of separating services' behaviour from their deployment details, and that both underline the importance of facilitating the integration of heterogeneous services in order to unify their management. One of the key aspects that differentiates Regent from Jolie is that Regent never assumes access to microservices' source code: whereas Jolie requires services written in other languages to be adapted intrusively — e.g., integrating an existing Java service with a Jolie program requires rewriting it to subclass a special class into which the Jolie runtime can hook —, Regent treats microservices as black boxes and only requires that their interfaces be described in terms of the protocols they use to exchange messages — i.e., in the Regent prototype, microservices are integrated into the system by (essentially) describing what JSON values they will send and receive. Thus, unlike Jolie, the Regent approach does not force developers to alter their software artifacts at all after they're finished, making it easier to integrate with their preferred methods for building software.

Another key difference between Regent and Jolie is what they consider to be a well-formed system. While Jolie uses a typical notion of type compatibility to ensure that the interfaces and code fragments it defines are always used "properly" throughout the system, Regent extends this notion with an adaptation mechanism to "repair" types/values in certain cases where an interface's usages don't exactly match its definition. Thus, while in Jolie two different teams of programmers working on separate parts of the system always have to agree exactly on the entire interface between each of their parts, two teams using Regent in the same situation would potentially have to coordinate on fewer non-automatically resolvable incompatibilities, therefore freeing them up to perform more independent and parallel work.

CHOReVOLUTION is an IDRE for designing service choreographies — i.e., systems where services coordinate their communications with each other in a decentralised manner, in contrast to systems with "orchestrator" services responsible for managing most other services. The CHOReVOLUTION IDRE is composed of various tools which cover the design and deployment needs of creating these choreography-based systems of services. Similarly to Regent, CHOReVOLUTION uses a notion of "interface" to decouple service implementations from their behaviour and interactions.

In practical terms, developing a system with the CHOReVOLUTION platform consists of using a visual modelling tool to create a diagram of the interactions between each service interface and then various wizard menus to specify additional details like security policies, how service interfaces map to their actual implementation and the details necessary to deploy each service (credentials, endpoint URIs, etc). Deploying the resulting system consists of using a web console to instruct the IDRE to launch the various services according to their configurations. We note that CHOReVOLUTION, like Regent, assumes that each service already exists and does not provide any facilities for actually implementing them.

Like Regent, CHOReVOLUTION treats components as black boxes whose behaviours are only known through interface descriptions; it is over these interfaces that both CHOReVOLUTION and Regent evaluate/manage the correctness of the system as a whole. The most striking difference between Regent and CHOReVOLUTION is Regent's automatic adaptation mechanism, whereby interfaces may evolve divergently as long as Regent is able to automatically reconcile them (by intercepting the values they exchange).

Another important difference is that Regent places fewer constraints on the developers' design and tool choices: the CHOReVOLUTION IDRE is built on top of the Eclipse IDE and geared towards designing systems orchestrated as choreographies, while the Regent prototype assumes the use of some of the most common technologies in the microservices development community (JSON, REST and Docker) — and only makes these assumptions to ease its development (i.e., at the conceptual level, Regent works equally well under the assumption that XML, SOAP and BSD jails are used).

2.5 API Management Tools

This is the most general of our categories. By "API Management" we mean, broadly, any tools that allow for a systematic development of APIs (in contrast with APIs developed in an *ad hoc* manner to suit a specific need). Examples of this sort of tool are OpenAPI Specifications [22], Microsoft Azure's API Management platform [13], and RPC mechanisms (of which there is a myriad of implementations and derivatives, such as Google's gRPC [10]).

OpenAPI Specifications, by themselves, are simply a standard for describing REST APIs — not unlike Regent's own specification and deployment files (in fact, they inspired our decisions in designing Regent's file formats). Although obviously usable just for the purpose of documenting APIs, OpenAPI Specifications' real power come from the fact that they are machine-intelligible. Because of this intelligibility, tools can automatically act upon the specifications — e.g., code generation tools [23] can transform OpenAPI Specifications into executable code in a target language. Although superficially similar, in that both approaches generate code from interface descriptions, Regent's approach is based on comparing specifications to one another and generating code to "translate" between different interfaces, while OpenAPI's is closer to a generation of code templates to be adapted by the programmer.

Microsoft Azure's API Management platform is comprised of a set of tools to describe, test, analyse, and in general manage APIs. From all the tools presented in this chapter, it is perhaps the one whose functionality is closest to Regent's, as it allows programmers to manually define transformations that should be performed on all replies coming from a microservice [14]. The major difference is that Azure API Management's transformations are intended to be set manually by the programmer, whereas Regent's transformations are meant to be automatically and transparently applied without programmer intervention.

"Remote Procedure Call", or RPC, is a generic term for a type of mechanism used to structure interactions between clients and servers over a network. These mechanisms typically serve a dual purpose of maintaining a versioned description of the server's interface on the client and allowing the client to treat communicating with the server as a simple opaque function call. Typically, the copy of the server's interface (known as a "stub") is kept by the client as a class/group of functions/etc either written manually or generated automatically from an Interface Description Language (such as OpenAPI Specifications or Protocol Buffers [9], for example). The major difference between a proper RPC mechanism and simply generating code from an IDL is that an RPC mechanism allows the client to issue a call using the same syntax and semantics as it would use for calling any other function.

The downsides of RPCs are somewhat obvious from the previous description: insomuch that making remote function calls indistinguishable from local function calls is a benefit, it may also be an hindrance, as it becomes easy to overlook that communicating across a network is inherently different from communicating locally in terms of latency, likelihood of failure, possibility of attacks by malicious actors, etc. Further, there's also the downside that the intercommunication between the client and server becomes constrained by what is possible/easy to express within the given RPC framework, although this applies to almost any framework that has to serialise information across a network.

Chapter 3

Design, Architecture & Implementation

In this chapter, we present the design of "Regent", our tool for supporting the robust evolution of microservices, and describe the choices we've made in implementing our prototype. We dedicate the first half of this chapter to discussing what our system's components do and how they communicate with each other to execute operations, along with some examples of what our specifications and deployment files look like. In turn, the remaining half of the chapter is dedicated to an in-depth discussion of Regent's compatibility relation and adaptation mechanism, both in terms of their design and of their implementation.

3.1 Regent Design, Architecture & Implementation

3.1.1 Design & Architectural Principles

Recall that the problem we are trying to solve is how to make the process of updating microservice-based systems more robust, in the sense that it should be hard (ideally impossible) for the programmer to accidentally break the whole system when trying to modify part of it. To do this, our approach requires that each microservice's interface be described in a high-level language which abstracts away implementation details such as the choice of programming language or runtime. We use these interface specifications to compare each pair of communicating microservices[1], verifying that their communication pattern is in one of two categories: either the consuming microservice expects to find the producing microservice's interface as-is, thus being capable of communicating directly

[1] Note: Although we refer to "producers" and "consumers" in these pairs, we do so only to simplify our discussion. There is no *a priori* requirement that a microservice acting as a producer for another one cannot also act as one of its consumers.

with it, or Regent has the ability to reconcile the differences between the interface the consumer expects and the one actually offered by the producer.

If the communication pattern of the microservices is in neither aforementioned category, then Regent considers them incompatible and stops the completion of the modifications — thereby preventing the system as a whole from entering an incoherent state and breaking. Otherwise, if the communication pattern is in one of the acceptable categories then Regent allows the modifications to be performed. The crux of our approach is defining what interface differences are considered reconcilable and how Regent performs this reconciliation. We discuss these matters throughout the remainder of this chapter; For now, it is sufficient to say that Regent's "reconciliation" consists of injecting a proxy into each microservice and rerouting communications through it, dynamically altering messages to conform to the necessary interfaces.

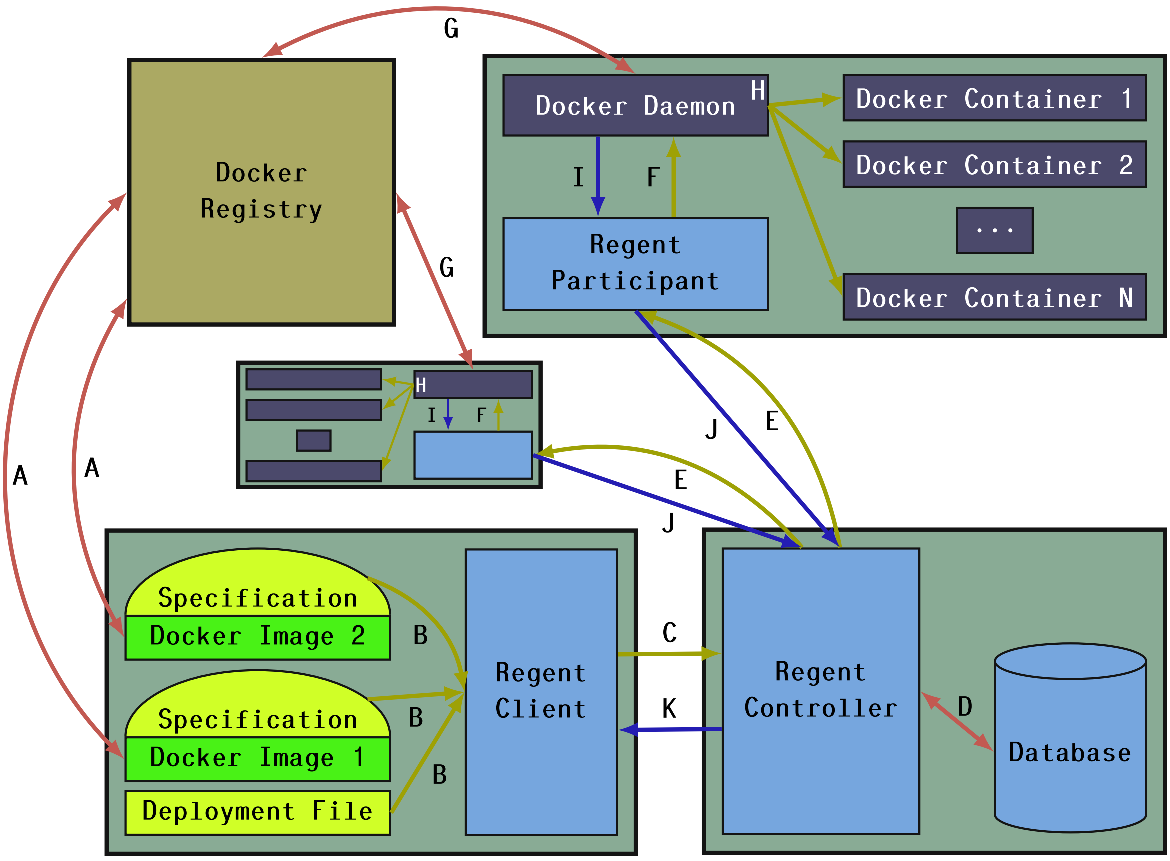

Figure 3.1: Example of a Regent set-up, showcasing how the components interact during a deployment of 2 Docker images.

Figure 3.1 presents a possible set-up of Regent's components while they communicate to perform a deployment and will serve as the motivating example for the remainder of section 3.1. Besides being the most important operation in our system, the deployment process also has the benefit of "touching" all of Regent's components, thus making it the ideal example to use when discussing Regent's architecture.

We depict the machines[2] that make up Regent as greenish grey boxes populated by files and processes, whereas the machine responsible for storing the programmer's microservice executables is differentiated as an empty dark green square since it doesn't need to run any of Regent's components and can be treated as a black box. Regent's components are represented in light blue, while the running microservices and associated managing processes are represented in a dark bluish colour. The executable microservice files are represented in green, and the high-level descriptions required by Regent are represented in a lighter greenish yellow. Finally, we use arrows to represent communication between processes (except for the arrows labelled B, which represent providing files to a process) — mostly realised as HTTP requests and responses in our prototype; We use pairs of green and blue arrows to distinguish requests from their replies (respectively) and bidirectional brown arrows when the distinction is inconsequential.

It is important to note that figure 3.1 is only one of many possible configurations in which to run Regent. Despite our figure showing an instance of Regent composed of 1 Client machine, 1 Controller machine, and 2 Participant machines, all of which communicate with the same Docker Registry when needed, both the number of each kind of machine and the choice of how to group processes per machine are malleable. The only *a priori* assumptions our tool makes are that there is a single Regent Controller, although this is merely due to our prototype being unable to handle concurrent requests to the Controller's Database and not conceptually intrinsic to our approach, and that each Regent Participant only manages services running on the same (physical or virtual) machine as itself, to simplify its communication with the Docker Daemon.

Another detail that must be noted in our presentation of Regent is our reference to Docker. Intrinsically, there's no special functionality provided only by the Docker ecosystem that Regent requires, and it could be implemented to work with opaque executable blobs instead of Docker images & containers. Our choice to use Docker is pragmatic: given that it is the most popular tool to package and run microservices, having our prototype use Docker serves not only to avoid implementing some of the microservice management logic ourselves, but also to show that it is capable of being integrated into a "real world" tool used in the domain we're working in. Our mentions of Docker throughout this document should thus be understood as also being pragmatic rather than indicative of Regent (as a concept) depending on Docker — e.g., the fact that we mention a Docker Registry in figure 3.1 is only because that's how our prototype happens to be implemented.

We now focus our attention on listings 3.1 and 3.2, in which we present an example of the configuration files required to deploy a microservice on Regent. Although the specific details of these files are not immediately important, we believe introducing them now will make it easier to follow along the rest of our discussion.

[2]Note: We use the term "machine" to logically group processes according to their purpose. In practice, there's nothing stopping Regent's components from all being run on the same physical machine — or from being broken up even further than our image suggests (e.g., the Regent Controller's Database could be run on its own separate machine).

```
Defines:
  Endpoints:
    - At: '/login'
      Type: '(user: {id: int, name: str, pass: str}) -> r: str'
      Method: POST
      Body: '{id: {{user.id}},
              username: {{user.name}},
              password: {{user.pass}}}'
      Reply: '{{r}}'

Depends: ~
```

Listing 3.1: A Regent specification of a simple login service.

Listing 3.1 describes the interface of a hypothetical microservice — i.e., what endpoints it `Defines` for others to consume, and which endpoints from others it `Depends` on to function. For simplicity, our microservice has no dependencies upon others (indicated with the special value "`~`") and offers a single endpoint at the `/login` URL path. The remainder of this endpoint's description shows that it will accept HTTP `POST` requests containing a JSON object in the `Body`, that its `Reply` will be sent back as a JSON non-object value, and what is the `Type` of each value involved in this exchange.

```
- Specification-File-Path: "/a/path/to/the/spec.yaml"

  Image-Location: Registry
  Registry-URL: index.docker.io
  Registry-Port: 443
  Registry-Repository: user123/my-login-service:v1

  Hostname: LoginServer
  Service-Port: 8080:80
  Proxy-Port: 12345:19820
  Expects: ~

  Machine:
    URL: 172.17.0.1
    Port: 5353
```

Listing 3.2: The Regent deployment file for the simple login service of listing 3.1.

Listing 3.2 contains the details necessary to actually deploy the microservice described by listing 3.1. Generally, the deployment configuration file might contain several entries describing how to create multiple microservices at once, but our example contains a single one's details for the sake of simplicity. The purpose of this file's contents are as follows:

- The keys prefixed by `Image` and `Registry` locate the Docker image from which to create a microservice as a Docker container;

- The `Specification-File-Path` key locates the specification file for the microservice we're creating — in this case, the file presented in listing 3.1;
- The `Hostname` key uniquely identifies the microservice being created among all others in the system. Despite its name, this key has no bearing on any networking properties of the microservice; Its name is merely a holdover from a previous version of Regent's prototype that made a few different assumptions about the way microservices communicated;
- The keys under `Machine` locate the Regent Participant which will be asked to manage the microservice once it's being deployed;
- The keys suffixed by `Port` are used to map ports from the machine to the Docker container. In our example, `Service-Port` maps the ports for the HTTP server listening for requests to the `/login` URL path and `Proxy-Port` maps the ports for the proxy injected into the microservice by Regent (as we mentioned at the start of this section);
- Lastly, the `Expects` key would normally be used to resolve the names of microservices mentioned in listing 3.1's `Depends` key. As our hypothetical microservice doesn't depend on any others, this key is also left empty here (identified by the "`~`" value).

Although a minor detail, there is a purpose behind our approach of separating the microservices' interface information from their deployment information: by doing so, we facilitate the reuse of these files for different microservices. Consider the hypothetical opposite case where the information from listings 3.1 and 3.2 existed in the same file. In this case, a developer trying to run 2 copies of "the same" microservice would have to maintain 2 mostly equal files differing only in a few deployment keys like `Hostname`, `Machine` or either `-Port`. In this hypothetical situation, it becomes only a matter of time until the files fall out of sync due to a developer forgetting to update one of them when changing the other.

By itself, this minor detail may seem inconsequential. However, this is only one of many seemingly minor decisions which work to ensure Regent respects the four design principles of section 1.2.3: the **principle of tool-agnosticism**, the **principle of integrating into typical workflows**, the **principle of caller-callee independence** and the **principle of lightweightness**.

Of these principles, **caller-callee independence** is the most prominent throughout Regent's design. From the relatively minor separation of interface specifications/deployment information described above to the usage of proxies to allow Regent's "reconciliation" of differing interfaces, almost every architectural design choice in our approach is aimed at increasing the amount of work that can be done independently and in parallel by developers.

The prevalence of this principle becomes especially evident when examining the ideal Regent interaction: in the best case scenario, a developer would simply modify or create a Docker image, describe its interface in terms of what it offers and what other already-existing microservices in the system it depends on, and instruct Regent to create some Docker containers based on that image and interface as well as automatically correct any minor incompatibilities between the new and old containers. In this ideal scenario, the developer is unconstrained by how other developers are using the system and achieves the ultimate goal of being able to robustly evolve it since Regent rejects any problematic updates before applying them.

The remaining three principles are also present throughout Regent's design, but are particularly easy to see in figure 3.1 and our discussion of it. Concretely, both the **principle of lightweightness** and the **principle of integrating into typical workflows** are respected because the only significant differences Regent introduces to a typical Docker architecture are the need to write the configuration files and the Regent Controller's processing of them; In turn, the **principle of tool-agnosticism** is (mostly) respected — the only technological requirements we make are pragmatic (e.g., it would not be feasible to support every communication protocol in use, so we assume services communicate via HTTP) and based on ubiquitous standards and tools such as JSON and Docker.

3.1.2 Detailed Overview of a Deployment

We now revisit figure 3.1 to discuss the deployment process in detail. We start by assuming that the programmer has already produced two Docker images, two Regent interface specifications, and one Regent deployment file[3], with the goal of deploying one microservice built from each of the Docker images they have. In addition to this, we also assume that the Regent Controller, all Regent Participant, and all Docker Daemons are already running.

Initially, both the Docker images and the configuration files only exist on the Client machine. The first step $\boxed{\texttt{A}}$ in the deployment process is simply for the programmer to upload their Docker images onto the Docker Registry, as they would do in the absence of Regent — typically, by using the Docker command line interface's `push` command. Naturally, there is no need to reupload images that already exist in the Registry and this step can be skipped for images in this situation.

Once the images are present in the registry, the programmer launches the Regent Client $\boxed{\texttt{B}}$ by supplying it the internet location of the Regent Controller and the local file path to the deployment configuration file. The client's only responsibilities are to perform some rudimentary preprocessing on the configuration files to replace default values (e.g., it is allowable for a deployment file to provide `~` as the `Registry-URL`, which the client replaces with the default Docker registry at `index.docker.io`), to gather all the

[3]Note: There is no relation between the files mentioned in this example and the ones given in the listings of section 3.1.1.

specification files referred to by the `Specification-File-Path` keys in the deployment file, and to send `C` these configuration files to the Regent Controller.

The Controller is Regent's most important piece, as it is responsible for verifying that prospective containers will integrate well with the ones that are already running. The first thing the Controller does is to perform a validity check on the specifications it receives: for example, to verify that they don't define more than one endpoint at the same path (`At`), or that the types referred to in `Body` really do exist in `Type`.

After performing this validation, the Controller moves on to the typecheck-like verification process we mention in section 1.2.3. This process is what verifies the compatibility of microservices, and is divided into two phases: the ***intra***compatibility check and the ***inter***compatibility check. Both of these phases perform the same basic verification process, with the difference being that the intracompatibility phase verifies the microservices in the deployment file against one another whereas the intercompatibility phase verifies them against the ones that already exist in the system `D`.

To preserve the clarity of our walkthrough of the deployment process, we leave the complete description of Regent's compatibility checks for section 3.2. For now, it suffices to say that the verification process's purpose is to guarantee an invariant required by the system: if an endpoint has type `T` and the consumers of that endpoint *believe* its type to be `S`, then values of type `S` can be automatically coerced into `T` without losing relevant information. As a simple example, consider the case where `T` is the type `{height: int}` and `S` is the type `{height: int, weight: int}` — obviously, coercing values of `S` can to values of `T` by dropping the `weight` field loses no relevant information from `T`'s point of view.

At this stage, one of two scenarios is possible: either both the validity check and the compatibility verifications are successful, or at least one of them fails. In the failure case, the deployment is aborted, all changes within it are discarded, and the user is informed `K` of the list of errors which prevented the deployment.

In the successful case, the Controller's Database is updated `D` to save the new deployment's information and the Regent Participants responsible for hosting the new or updated microservices are contacted `E`. Logically speaking, the process of creating/updating microservices on the Regent Participants should be transactional — that is, either all creations/modifications should succeed across all Participants, or a single one failing should cause all others to fail — in order to guarantee the deployment's coherency.

However, our prototype does not yet provide this guarantee: if a Participant crashes for some reason as it's being contacted, then the Controller will simply abort the deployment without either purging the Database's information about it or ordering the already contacted Participants to undo the creation/modification of microservices.

Although the absence of this feature is mostly due to the difficulty of implementing distributed transactions correctly, it is also partially due to the difficulty of estimating how a straightforward algorithm to solve the distributed transaction problem would perform as the number of microservices increases; Typically, algorithms like 2-Phase

Commit require waiting for all of the transaction's participants to agree on whether to accept or reject the transaction. In the end, we deemed it too likely that the transaction mechanism would have to be scrapped and replaced for it to be worth implementing at this stage.

Assuming that everything has been successful up to this point, the last steps of the deployment process are centered on the Regent Participants. These components have three duties: to fetch new and updated Docker images from the registry, to programatically alter the fetched images so that their communications will be captured by a proxy, and to instantiate those images as Docker containers. In practice, most of these duties amount to directing the Docker Daemon `F` to perform commands it already has support for "out-of-the-box", like pulling images `G` or starting and stopping containers `H`, and waiting to check whether they succeeded `I`.

Despite the ease of implementing most of the Regent Participants' functionality, there are two factors which posed a significant challenge during the course of our prototype's development. Namely, they are Docker's insufficiently powerful mechanisms for programmatically modifying images and the Docker Daemon's inability to execute commands transactionally. Although these are just implementation details and could be left out of a discussion about the deployment process, we feel that presenting them is important to contextualise why our prototype has so many limitations.

The issues related to Docker's mechanisms for modifying images stem from Regent's proxy injection step. Recall that the purpose of injecting the proxies is to be able to transparently intercept the communications of arbitrary Docker images while avoiding impositions on the way they must be built (in keeping with the **principle of tool-agnosticism**). Thus, part of the injection process is to automatically configure the Docker image so that its communications are routed through the proxy without requiring manual developer input — and to do this, Regent uses the iptables utility, since it's a mature tool available on nearly any linux-based environment.

However, the iptables utility has a particularity: its configurations are ephemeral and only last until shutdown. As the docker build cycle works by creating short-lived linux sessions in which to run each build step, this means that we must somehow delay the configuration of iptables until the Docker image is actually being instantiated as a microservice[4].

In order to delay iptables' configuration until the microservice is actually launched as a container, our approach is to copy a script with the necessary `iptables` command (shown in listing 3.3) onto the Docker image and reconfigure the image to run this script at start-up. However, this reveals another problem: Docker offers no way to modify the start-up command of an image without destroying whatever command already exists. Thus,

[4]Technically speaking, it would actually be possible to persist these configurations by using special utilities — which aren't as widely available as iptables itself — or by writing some sort of script to replace them. However, even this workaround only became possible recently (when Regent's development was already underway [4]) and requires more elevated permissions than the normal `docker build` command.

```
#!/bin/sh
iptables -t nat -A OUTPUT -p tcp -m owner ! --uid-owner
        regent_proxy -j REDIRECT --to-port
        ${REGENT_PROXY_PORT_CONTAINER}
su regent_proxy sh -c "python3 /regent/proxy.py &"

exec $@
```

Listing 3.3: Shell script run on boot by the Docker images after being modified by the Regent Participant.

the Regent Participant must inspect the original image to retrieve its start-up command[5] and assemble a new start-up command that works together with the launch script to follow its execution by running whatever the original image ran.

Since the proxy must already be running when the microservice starts communicating with others, we also include the command to run the proxy in the launch script. Although this approach of using scripts to run commands before the start-up commands given in the `CMD` and `ENTRYPOINT` instructions is unusual, it is known in the Docker community [1] and is even used in projects like the Docker Official Image for postgres [27].

To finish this issue's discussion, we briefly present the Dockerfile used as a template to modify the original image (listing 3.4) and explain how it works with the launch script (listing 3.3). Succintly, the Dockerfile starts by creating a user account and directory in which to quarentine Regent's files, and installs the libraries required to run the proxy and configure the microservice's communications[6]. Afterwards, it copies over the launch script, the proxy's files, and the data with which to initialise the proxy, and records the original image's start-up command using the `CMD` instruction. Finally, it finishes off by configuring some environment variables used by the launch script and the proxy. At run-time, the launch script ties all of this together by configuring iptables to reroute all outgoing TCP communications through the proxy, launching said proxy, and executing the contents stored in `CMD` — i.e., the image's original start-up command.

As for the issues related to the Docker Daemon's inability to execute multiple container-management commands transactionally, they arise from our need to manipulate containers in multiple ways when creating or replacing them. As an example, when replacing a container with another one we must at least stop the existing one and start the new one. But as there is no way to transactionally execute groups of instructions, it might be the case that we manage to stop the original container but never manage to start the new one.

Our approach to these issues was to assume that a small set of simple commands (such

[5]Which can be stored using the `CMD` instruction, the `ENTRYPOINT` instruction, or both of them [6, 7]. Note also that these instructions can interfere with one another, and each of them can be written in two different forms for a total of 4 semantically distinct combinations.

[6]Notice how Docker offers no way to query which utilities are available within an image, forcing us to either assume their existence or do without. To handle images whose package manager is not apk, for instance, we'd need a second template differing only in the package manager's name. This is yet another limitation of Docker's mechanisms for programmatically modifying images.

```
FROM {Base image tag} AS modified_base

RUN mkdir /regent
RUN adduser -D -H regent_proxy
RUN chown -R regent_proxy:regent_proxy /regent

RUN apk add iptables
RUN apk add python3
RUN pip3 install flask
RUN pip3 install requests
RUN pip3 install ply

ENTRYPOINT ["/regent/launcher.sh"]
COPY launcher.sh /regent/launcher.sh
RUN ["chmod", "+x", "/regent/launcher.sh"]

COPY __init__.py /regent/__init__.py
COPY proxy.py /regent/proxy.py
COPY database.py /regent/database.py
COPY ast_builder.py /regent/ast_builder.py

FROM modified_base

CMD {Original ENTRYPOINT and CMD joined into one}
COPY {File with initial specification information} /regent/initial_data.json

ENV REGENT_DIR=/regent
ENV REGENT_SPEC_ID="{Specification ID}"
ENV REGENT_SERVICE_PORT_CONTAINER={Service port in container}
ENV REGENT_PROXY_PORT_CONTAINER={Proxy port in container}
```

Listing 3.4: Dockerfile template used for modifying images.

as stopping a container) cannot fail and verify failure for all other commands, but this is not truly safe — what if our assumption is broken and one of those commands does fail? One potential way to improve our process would be to use a technique common in database management systems: we could keep a write-ahead log of which Docker commands we intend to perform and use it to restore our containers' state when failures arise; however, this has the obvious drawback of being prohibitively hard to implement for a prototype.

In practice, microservices are supposed to be designed for quick and seamless replaced on failure, so this challenge does not compromise our approach even if it allows the system to be transiently incoherent.

Moving past these issues, we now finish our detailed discussion of the deployment process with a complete description of what the Regent Participant does once contacted by the Controller `E`:

- First, it pulls all the Docker images `G` from which it is meant to create microservices, as referred by the deployment file's `Registry-URL`, and analyses them using `docker inspect` to retrieve the commands that they perform when booted;

- Next, it derives a new Docker image from each one that it pulled, using the Dockerfile in listing 3.4 with the appropriate values replaced for the `{placeholders}`. Concretely, `{Base image tag}` is replaced with the name of the original image, `{Original ENTRYPOINT and CMD joined into one}` is replaced with the result from the previous step, `{File with initial specification information}` is replaced with a local path to a file containing the microservice's own specification and a copy of the specifications of the microservices on which it depends, `{Specification ID}` is replaced with a unique identifier of the microservice's own specification, and `{Service/Port port in container}` are replaced with the port on which the service/proxy will be listening inside the container;

- Lastly, it builds all the images and tries to deploy them (replacing containers for whom a newer image is available) as reliably as possible (taking into account Docker Daemon's inability to execute multiple commands transactionally).

Once this process is completed successfully, we are left with a group of Docker containers capable of communicating with one another without being aware of Regent's existence. However, whenever the communicating microservices try to exchange incompatible messages, the proxies injected into them transparently intercept their communications and modify them to hide their incompatibility. Although we leave the specifics of how proxies decide which messages need to be altered and of how they do so for section 3.3, we know that this is guaranteed to be possible thanks to the invariant mentioned earlier in this section[7].

At this stage, all that is left to do is for all the Regent Participants to inform the Controller `J` of any — if any — errors encountered while deploying the new containers, and for the Controller to relay that information to the Client `K`.

3.1.3 Other Operations

While deploying and updating microservices is the core purpose of our system, it is not the only operation Regent must support: obviously, Regent must also have some way of listing active microservices and some way to issue their removal. As both of these operations work similarly to the deployment operation discussed in the section 3.1.2, we will discuss reuse figure 3.1 to present them.

[7] In short: if the Controller accepted these microservices, then it must be possible for them to communicate without losing relevant information.

Concretely, listing the active microservices' information is just a matter of contacting the Regent Controller $\boxed{\texttt{C}}$ and having it query the Database $\boxed{\texttt{D}}$ to relay the answer back $\boxed{\texttt{K}}$. The information returned for each microservice is its uniqued hostname, the URL and port at which it's available, the list of endpoint paths it exposes, and the names of the microservices it depends on.

Meanwhile, removing a running microservice is only slightly more involved than the listing operation: the Client sends the Controller $\boxed{\texttt{C}}$ a request to remove a group of microservices, identified by their unique hostnames. Upon receiving the request, the Controller checks $\boxed{\texttt{D}}$ that all of the hostnames correspond to active microservices and that none of them are still being depended upon by anyone else.

If at least one of the hostnames is unknown or at least one other microservice depends on one of the removal candidates, the request is aborted $\boxed{\texttt{K}}$ and no changes are made to the system — preventing requests based on incorrect information (trying to remove inactive servers) or which would break dependencies from going ahead.

Otherwise, the Controller purges their information from the Database $\boxed{\texttt{D}}$, requests that the Regent Participants $\boxed{\texttt{E}}$ responsible for managing the microservices stop them $\boxed{\texttt{F, H}}$, and propagates the Regent Participants' replies $\boxed{\texttt{J}}$ back to the Client $\boxed{\texttt{K}}$.

As per our description, removing a running microservice requires it to not be depended upon by any other components. Thus, the process to remove a microservice on which others still depend is done in two steps: a deployment to safely remove those dependencies by modifying the dependent components so that they no longer try to contact the soon-to-be removed microservice, followed by the regular removal process described above.

3.2 The Compatibility Relation

3.2.1 Types and Endpoints

Throughout the preceeding chapters and sections, we have mentioned compatibility relation between microservices. We have likened this relation to the typechecking performed by compilers to ensure that different parts of a program "fit" together, but without presenting it in full detail. We now explore what we meant by this comparison and how Regent's compatibility relation works.

For the purpose of our explanation, we will start by considering a programming language. As most of the language's details will not be relevant for our explanation, we will consider a simple C-like language. It suffices that our language have a type system with integers and strings, a way to package heterogenous types together (e.g., structs, records, tuples, classes, or an equivalent), and *simple* functions — that is, this language does not need to support closures[8].

[8]Also known as anonymous functions, lambdas, function literals, higher-order functions, etc.

In a programming language like this, there are various degrees to which types are coupled with implementation details. For example, an integer may be able to hold arbitrarily large numbers, or it may have only be able to hold values up capable of fitting in the processor's word size; Meanwhile, a function type will usually not have any special constraints on what code it executes, but the order of its parameters — that is, how the programmer chooses to *write* (implement) the function's parameters — will affect how the function can be called (i.e., for a function type like $f : (int, str) \to int$, $f(123, "abc")$ will typecheck while $f("abc", 123)$ will fail).

Presumably, the extent to which types are coupled to these details "makes sense" for the language's user — "*of course*" that the order of the parameters in a function's definition affects the order by which arguments must be passed to it when it's called. In fact, to call "the order by which things are passed to a function must match its definition" just an implementation borders the incorrect, given how intuitive and widespread the convention is. Obviously, there is no *force majeure* preventing us from stipulating that functions must be provided with their arguments in the reverse order that they appear in its definition, but this convention seems arbitrary and pointless. Regardless, the point we want to make is that the notion of "type" can never be completely divorced from how it is implemented, even if we must be extremely pedantic in our understanding of "implementation detail".

Let us now consider, alongside the language from earlier, a microservice. Assume that this microservice offers an endpoint `/sum` — over HTTP, for the sake of simplicity —, which calculates the sum of two `int`egers. Intuitively, it does not feel senseless to see this endpoint as a "function" that accepts two integers and returns another one. In fact, all endpoints can be likened to functions in this manner, since at their core endpoints just take something (or nothing) and transform it into something else.

However, the parallel between functions and microservice endpoints is not perfect: as we describe above, the order in which arguments are passed to a function must[9] match the order of the parameters in the function's definition — which does not necessarily apply to microservices. Notice how we made no mention of how the `/sum` endpoint in our example receives its arguments — even though we specifically considered an HTTP endpoint, there are multiple mechanisms through which it might receive its values, and each mechanism has its own way to deal with "order". Namely, the endpoint might receive its values via its path (`/sum/1/2`), via a query string (`/sum?x=1&y=2`), via its HTTP Request headers, via its HTTP Request body, or via any combination of these alternatives.

Obviously, the order in which arguments are supplied is irrelevant in the specific case of `/sum` since it's a commutative function, but this is rarely the case for other endpoints. Consider a `/search` endpoint, which accepts its arguments as a query string and returns another string as the result; Our `/search` endpoint explicitly shows how endpoints differ

[9] Strictly speaking, "must" is too strong of a qualifier here: some languages, such as Python [20], permit the passage of arguments in arbitrary orders by explicitly specifying the name of the parameter to which we are passing a value. However, this doesn't detract from the point we're trying to make.

from the typical concept of functions: calling `/search?what=socks&quantity=10` should obviously work as well as `/search?quantity=10&what=socks`, but a naïve conversion from endpoints to types would give us $search\colon (str, int) \to str$ and $search\colon (int, str) \to str$, respectively — two incompatible types. Conversely, if `/search` accepted its arguments as part of its path (e.g., `/search/socks/10`), then there'd only be one unambiguous way to naïvely type the endpoint ($search\colon (str, int) \to str$).

In summary: applying a notion of "type" to an endpoint is not senseless, but it cannot be done naively. To some extent, types are always intertwined with the way they are represented, and functions as usually defined implicitly assume that it makes sense to speak of the order in which parameters/arguments are expected/passed. On the other hand, microservices' endpoints' types are indissociable from the mechanism through which they expect to receive their data — which may or may not have a meaningful notion of order.

3.2.2 Possible Approaches

There is one obvious way to solve this mismatch between the concepts of types and endpoints: sidestep the problem entirely by choosing a single mechanism through which data will be exchanged. As a matter of fact, we already showed this at the end of the previous section: by embedding values in the endpoints' paths we automatically gained a notion of order, thereby averting the problem. In effect, this approach is akin to saying "programmers write functions and function calls, and the way in which values are sent from callers to callees is merely an implementation detail that doesn't concern them".

There is nothing intrinsically wrong with this approach, and Seco et al.'s [21] work, one of the most influential on the development of our own, does something similar to this. In their work, Seco et al. propose a small programming language with a set of constructs that allow the programmer to create distributed systems built of multiple intercommunicating services, similarly to our goal with Regent. The difference between their work and ours is that in Seco et al.'s, all services are implemented in the same language — which allows their runtime to have richer information about the state of the system —, whereas Regent actively tries to minimise its assumptions about the implementation details of the microservices it manages. As a result, our notions of "function" and "endpoint" refer to the same thing in Seco et al.'s system: callers and callees define functions as would be usual in any other programming language, and the runtime takes care of deciding how to transfer the data between them.

As another example of this approach, consider the mechanism of Remote Procedure Calls (RPCs). Generically speaking, RPCs require that each client have a copy of the server's interface (typically known as a "stub"), expressed as a class/group of functions/etc in the client's own language, and a runtime to support the RPC mechanism. To interact with the server, the programmer simply calls the stub's functions as they'd call any other function defined using normal mechanisms. The RPC runtime intercepts these calls and transparently transmits them to the server, abstracting away details such as what format and protocol the data is transmitted in.

Initially, we started by designing Regent similarly to these examples: specifications would simply contain endpoints composed of a `Type` and a path (`At`) to identify them. But eventually, this raised the question of what a type actually *is* for Regent. As one of our system's design goals is to be agnostic to the implementation details of the user's microservices, we have no control over which programming language, communication protocols, or file formats each microservice uses. In this context, the notion of type in microservice A might be completely different from microservice B's notion of type!

After reviewing and iterating upon our initial design choices, we reached the conclusion that there was a mismatch between what microservices in our setting actually do and what we'd like them to do. Ideally, we'd like to be able to simply look at the types in two specifications and be able to decide whether they're compatible or not; In practice, we must also look at how the values are actually exchanged between microservices. Hence why our endpoints' signatures must also contain a `Body` and `Reply`, and why paths (`At`) may contain a query string at the end[10]. In Regent's case, an endpoint's type is *indissociable* from the way its data is transmitted.

3.2.3 Regent's Approach

Given the preceeding discussion, we are now ready to define Regent's compatibility relation, and to do so we will start with an example. Listings 3.5 and 3.6 present specifications for two HTTP services: a server — which offers an endpoint and does nothing else —, and a client — which uses the server's endpoint and does nothing else.

Immediately, it is easy to see that the `Type`s of each endpoint differ, with one being a function that transforms two integers into a single integer and the other being a function that transforms a record into an integer. It is also easy to see that everything else "matches": both descriptions of the endpoint say that it accepts a `POST` request, both say that its path is `/f`, both have the same query parameters `x` and `y` (although the arguments are not exactly the same), both say that its `Body` is empty, and both say that the `Reply` is an integer (although one refers to the reply as `z` and the other as `rank`).

For Regent, these two specifications are perfectly compatible, despite apparently being typed differently. In fact, Regent treats the `Type` as less of a type in the usual programming language sense and more like a mapping between keys and base types (i.e., `str` and `int`). To verify that these two specifications are compatible, Regent simply verifies that the parameters required by the server in its `Body` (none) and query string (`x` and `y`) are supplied by the client, with the same base type on both the client and server, and that the `Reply` expected by the client (a plain integer value) is sent by the server, also with the correct type.

[10] In section 3.2.1, we also mentioned how HTTP endpoints may receive data in the headers or as part of their paths. As we felt that HTTP request bodies, HTTP reply bodies, and query strings were enough to

```
Defines:
  Endpoints:
    - At: '/f?x={x}&y={y}'
      Type: '(x: int, y: int) -> z: int'
      Method: POST
      Body: ~
      Reply: '{{z}}'

Depends: ~
```

Listing 3.5: A Regent specification for a simple server.

```
Defines: ~

Depends:
  - Name: SocialNetworkService
    Endpoints:
      - At: '/f?x={user.friends}&y={user.likes}'
        Type: '(user: {friends: int, likes: int}) -> rank: int'
        Method: POST
        Body: ~
        Reply: '{{rank}}'
```

Listing 3.6: A Regent specification for a client that uses the server of listing 3.5.

Naturally, it is then fair to ask why we permit multiple ways to write Types which appear to differ if it makes no difference in Regent's compatibility check. The reason for this is usability: we could have restricted Type to be a mapping from keys to base types, but allowing Type to be structured as a function with records and base types makes it easier for readers of the specification to interpret what an endpoint is doing. Listing 3.6 demonstrates this, by writing the Type of the endpoint in listing 3.5 in a more semantically meaningful way. Similarly, usability is why we allow the keys used in Type to differ from the actual parameter names used in the query string, Body, and `Reply` (for example, in listing 3.5 the query parameter is x while the key in Type is `user.friends`).

Finally, let us consider an alternative client's specification, presented in listing 3.7. This specification clearly has much more "going on" when compared to the two we've just discussed. Even so, Regent would consider it to be compatible with the server specification in listing 3.5. Despite its apparent complexity, the specification in listing 3.7 guarantees that the microservice which it describes will supply exactly what the server of listing 3.5 needs and require no more than it provides: a query string containing two `int`egers x and y is sent to path `/f` and a plain `int`eger value is awaited as a reply. Everything else that the client promises to send is inconsequential, as Regent can transparently intervene (via the proxy mechanism discussed in section 3.3) and make the necessary

illustrate our approach, these mechanisms aren't currently supported in the proxy.

```
Defines: ~

Depends:
  - Name: SocialNetworkService
    Endpoints:
      - At: '/f?x={user.friends}&y={user.likes}&at={location}'
        Type: '(user: {friends: int, likes: int},
                points: {basic: int, premium: int},
                metadata: str,
                location: str) -> rank: int'
        Method: GET
        Body: '{currency:
                 {bsc: {{points.basic}},
                  prem: {{points.premium}}},
                metadata: {{metadata}}}'
        Reply: '{{rank}}'
```

Listing 3.7: A Regent specification for another client which also uses the microservice described in listing 3.5.

modifications for the values sent by the client to match those expected by the server. In this case: drop the `Body` entirely, change the HTTP `Method` from `GET` to `POST`, and drop the `at` parameter from the query string.

Herein lies the purpose of the compatibility relation. Suppose that, initially, two microservices were deployed — one using the specification of listing 3.7, and a server requiring everything that's sent by the client of listing 3.7. Further, suppose that after some time the developer concludes that some of the data required by the `/f` endpoint is actually not needed and that it could be rewritten into the simpler specification presented in listing 3.5.

Normally, deploying the rewritten server would require (1) checking that all of its clients (in this case, just one) are compatible with the new version and (2) updating all those that are not. Regent automatically deals with (1) via the compatibility relation, which rejects all attempts to deploy services whose interfaces would not be respected by their clients, and with (2) via the adaptation mechanism (discussed in section 3.3), which can automatically convert from what clients send to what servers expect by using the type information of the compatibility relation.

Finally, now that we've explained the design of the compatibility verification, we present it fully, step-by-step as the Regent Controller (see figure 3.1 of section 3.1.1) implements it:

- For each interface specification, Regent retrieves the endpoints in its `Depends` section and matches each one to the corresponding endpoints on other specifications' `Defines` sections. Notice how this is plural, since a dependency may be mapped to more than one definition (e.g., a client may potentially use any member of a set of identical servers).

- Then, for each endpoint `Depends`/`Defines` information pair, Regent checks two things: first, that every query parameter/key in `Body` in the `Defines` information is also present in the `Depends` information — that is, that the client supplies the server with at least what it requires (and possibly more); second, it verifies that every parameter/key has the same base type on both sides — i.e., that both are either an `int`eger or a `str`ing. If this step is successful, then Regent skips the next step (as it is unneeded).

- If there are query parameters or `Body` keys that the endpoint requires in `Defines` but the endpoint in `Depends` does not send, Regent tries two additional ways of providing the value itself: default values and mixed-mappings. To simplify the explanation, we base it on a concrete example that uses both of these mechanisms — listings 3.8 (a server) and 3.9 (a client).

  Given that the server requires two query parameters but the client sends none, obviously these specifications are not directly compatible. However, as the server defines a default value of `123` for query parameter `y`, there's no problem if the client doesn't send its own value. Additionally, the client does send a value in the body whose key (`x`) has the same name and type as the missing query parameter in the server. Thus, the value could just be mapped from the client's `Body` to the server's query parameter if needed, and these 2 listrings end up being deemed compatible.

- After performing these verifications for the `Body` and query parameters, Regent does almost the exact same thing for the `Reply` fields — the difference being that the roles are reversed, with it being the keys in the endpoint in `Defines` that has to supply all that the endpoint in `Depends` requires in its `Reply`.

- In our prototype's case, we assume that communication is performed over HTTP, and so the endpoint information also contains an HTTP `Method` entry. In practice, this entry is just a string and does not factor into Regent's compatibility checks — we can simply rewrite it to match whatever the server's definition requires.

- Lastly, Regent analyses the deployment information to retrieve the internet locations of the machines onto which the microservices are being deployed, along with the ports which will be open on the microservices. As multiple processes cannot be bound to the same port at the same location, Regent will reject any deployments which would cause an overlap between open ports.

To verify a deployment as compatible with the system, Regent performs the steps above twice: once between pairs of microservices in the deployment itself (to ensure that the deployment is self-consistent), and once between pairs of microservices where one is being deployed and another is already running (and whose information is therefore

```
Defines:
  Endpoints:
    - At: '/f?x={x}&y={y}'
      Type: '(x: int, y: int [123]) -> _'
      Method: POST
      Body: ~
      Reply: ~

Depends: ~
```

Listing 3.8: A Regent specification for a server demonstrating how default values and mixed-mapping work.

```
Defines: ~

Depends:
  - Name: SpecialValuesExample
    Endpoints:
      - At: '/f'
        Type: '(x: int) -> _'
        Method: POST
        Body: '{x: {{x}}}'
        Reply: ~
```

Listing 3.9: A Regent specification for a client demonstrating how default values and mixed-mapping work.

already saved in the Regent Controller's database). These two verifications correspond, respectively, to what we referred to as "intracompatibility" and "intercompatibility" in section 3.1.2.

3.3 Proxies & Adaptation

Logically speaking, the adaptation mechanism is just an application of the compatibility relation described in the previous section. That is: whereas the compatibility relation only verifies if things are compatible, the adaptation mechanism uses the same principles to actually intercept the communications between microservices and change the values being sent to the ones that are expected to be received. As a concrete example of this distinction: the compatibility relation is what evaluates listings 3.8 and 3.9 as compatible, whereas the adaptation mechanism is what actually inserts the 123 default value into y and moves the value of x from the Body to the query parameter.

Actually implementing the adaptation mechanism, however, requires multiple steps. As we already mentioned in previous chapters, the adaptation should be as transparent as possible. To achieve this, Regent injects a proxy into the Docker images created by the programmers and automatically configures it so that all outgoing TCP packets will

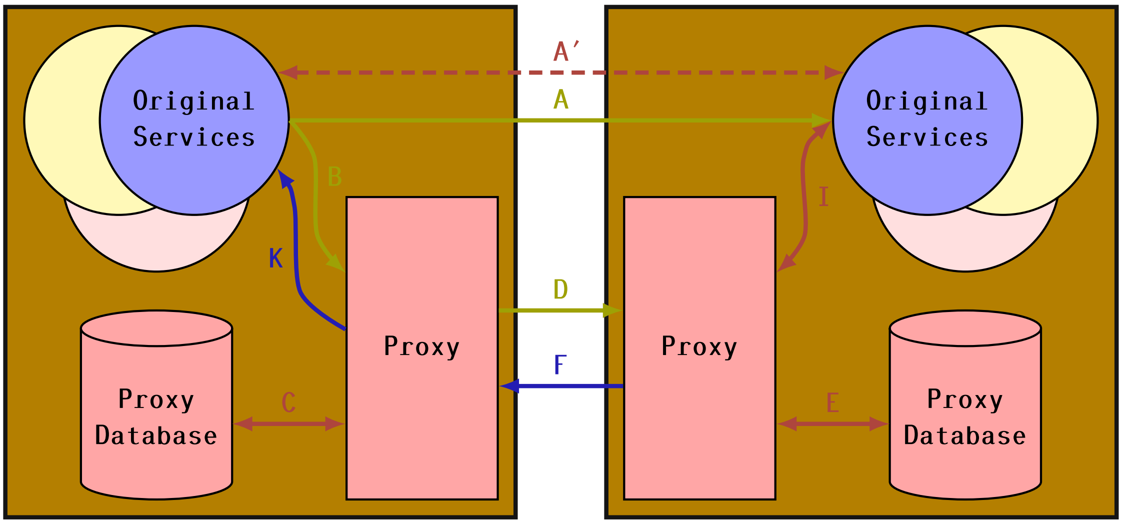

Figure 3.2: Depiction of the complete communication protocol between two microservices after they have been modified by Regent.

be redirected to the proxy. At run time, this proxy evaluates the outgoing requests to determine if they should be left unchanged or, if not to be left intact, how they should be modified.

Figure 3.2 depicts how two microservices communicate after having been modified by Regent. In it, orange boxes represent the boundaries of the microservices, with the internal components representing processes running in the microservices. Arrows represent (attempts) to communicate and, as we did for figure 3.1, we distinguish requests (in green) from their replies (in blue) when there are steps inbetween them that we must discuss, and make no distinction otherwise (in desaturated red).

In section 3.1.2 we discussed how proxies are built from a Dockerfile template (listing 3.4) which, among other things, "copies over (...) some initial data" that the proxy requires. This initial data is composed of the specification (i.e., the endpoints in `Defines` and `Depends`) for the microservice in which the proxy runs plus some deployment information for each of the microservices on which it depends — concretely, their hostname, their URL or IP, their open ports, and an unique ID for their specification. Once launched, the proxy seeds its database with this data, which it then uses when intercepting messages to decide how to handle them.

We now describe the process in figure 3.2 where the microservice on the left is acting as a client issuing an HTTP request to microservice on the right; We assume that the proxy's initialisation has already finished and that this is the first request being sent between these two microservices. The exchange proceeds as follows:

- At some point, one of the original processes in the image (from before it was modified by Regent) decides to send an HTTP request to another microservice controlled by Regent. From this process's perspective, what follows next is the exchange depicted by arrow `A′`: it is sending a request, and it'll receive a reply from the server it's contacting (naturally, we're assuming there are no timeouts, loss of network connectivity, etc). In practice, the reply to the proccess's request is not instantaneous; We label the HTTP request sent by the process as `A`.

- As described in section 3.1.2, Regent has altered the Docker image so that the (`launcher.sh`) script is run when the image is started as a container. This script configures iptables so that all outgoing communications are redirected through the proxy, and so the HTTP request `A` is transformed into HTTP request `B`. This request is delivered to the proxy process unchanged — crucially, the HTTP Host of the original recipient is preserved.

- Upon receiving the request, the proxy checks for a special header that proxies add onto HTTP requests to identify their origin (i.e., it checks if this request originated from another proxy). Since in this case the header is absent, it proceeds to check its database (`C`) for the existence of a specification associated with the HTTP Host of the request. Since the database was seeded with the deployment information of the microservice's dependencies, checking for the presence of the HTTP Host is equivalent to checking whether the request was intended for a microservice under Regent's control.

- If the HTTP Host were not present, then the proxy would forward the request without changing it and await a reply (not pictured), which it would then to the original process (`K`). Otherwise, if there is a specification associated with that host, the proxy will search it for the endpoint whose path matches the one to which the request is being sent. As this is the first request being sent to this host, the endpoint's information will simply be an entry in the `Depends` section of the sender's specification (e.g., for the client of listing 3.7, the first request sent to `/f` at the server of listing 3.5 will have `Type: '(user_info: {friends: int, likes: int}, ...) -> int'`, `Method: GET`, etc instead of `Type: '(x: int, y: int) -> int'`, `Method: POST`, etc).

- Alongside this, the proxy also searches to find the endpoint information that the requesting microservice (i.e., itself) *expected* the serving microservice to have. In this case, this endpoint information is *the same* `Depends` entry as in the last point. There is a subtlety here which may not be entirely obvious, but which is unavoidable since we want it to be possible for callers and callees to evolve separately[11]: the information that a specification gives for an endpoint on which it depends may be out of date. To deal with this, the proxy's database must actually maintain two copies of the information per endpoint: the information that describes what the client will send out, and the latest information that actually describes what the server will accept. Since the only specification used to seed the proxy is its own

[11] As per the "principle of caller-callee independence" from section 1.2.3.

microservice's, initially both of these endpoint descriptions are the same — but as we will see in a few steps, these descriptions eventually diverge.

- Once the proxy knows how the client is sending the information and how the server intends to receive it, it performs the necessary transformations (dropping fields, moving things back and forth from the query parameters and the Body, adding default values) to match the client's request to the server's expectation. Since these microservices are deployed, they must have passed Regent's compatibility checks, and so we have the guarantee that it is possible to perform one of these transformations[12].

- Finally, once the request's information has been rewritten the proxy changes the request's HTTP Host so that it is directed instead to the remote microservice's proxy (D). The request is also modified to include a header which signals that this request is originating from another Regent proxy (i.e., the header we mentioned at the start of this process), and a header which contains the unique identifier of the specification against which the proxy rewrote the request.

- The receiving microservice's proxy examines the request, just as the sending proxy did in the third step above. In this case, it does find the header which identifies the request as originating at another proxy, and so it checks the identifier against which the request was written. If this identifier matches its specification's identifier, then the request is valid and can be delivered to the originally intended recipient; Otherwise, the request is invalid and must be rewritten.

- For the sake of describing the full process, we'll assume that the request was invalid, meaning that the sending proxy's view of the receiving microservice's interface is outdated. To fix this, the receiving proxy retrieves its own specification from its database (E) and sends it, along with an error code, as a reply F to the requesting proxy.

- The rest of the process is very simple from now onwards. The requesting proxy simply updates its database (C) with the specification sent by the remote proxy and performs the same steps as before, transforming the data from the format in which it was sent out by the original processes into the format in which the server actually expects to receive it. After this, it sends the request (D) to the remote proxy again.

- The remote process undergoes the same sequence of steps as last time. It's possible for the remote microservice to have been updated again, making the new request outdated again, in which case this cycle of rejections and rewritings goes back and forth until both the requester and the recipient can agree on what is the latest

[12] This is where the invariant we speak of in section 3.1.2 comes into play: as long as the invariant is upheld, doing this is always possible.

version of the specification. Assuming that the request has finally been accepted by the recipient proxy, the last thing to do is deliver the modified request to the original destination process (as depicted in I).

- Once the remote proxy receives the original destination's result, it passes it on to the sending proxy (D). The proxy may, for one last time, perform some modifications on the result (e.g., to drop some fields sent by the server but which the client doesn't require), which is finally delivered to the original sending process in reply K.

In short: from their point of view, processes communicate without any regard for Regent's existence; Meanwhile, the proxies send requests back and forth until the sender's information matches what the recipient requires it to be. In this process, the sender updates its information on what is the latest state of the world every time it talks to a receiver whose interface has changed.

In this way, we afford clients and servers a great deal of flexibility on changing their interfaces by allowing them to "just change" without requiring them to also change the interfaces of all the services on which they depend or which depend upon them. And while our current implementation is bound by technical constraints (i.e., it requires that the HTTP protocol be used, that messages in the HTTP body be sent as JSON data, and so on), in principle it could be extended to support any combination of technologies.

Chapter 4

Evaluation

In this chapter, we present our evaluation of Regent. We do so by walking through the deployments of two different example systems and discussing the ways in which Regent facilitates their robust evolution.

We note that the evaluation presented in this chapter is qualitative rather than quantitative. We believe that this choice is justified for two reasons: first, collecting metrics such as the execution overhead introduced by our components would be a waste of effort at this stage — since Regent is still a prototype, it's much more important to verify that it's working correctly rather than efficiently; second, because Regent is a tool where performance is mostly incidental to its purpose — no amount of (reasonably achievable) quantitive metrics can justify a statement like "Regent makes it safer to perform deployments".

4.1 System #1: Number Generator Service

In this section, our aims are to establish what a full set of configuration files might look like and to give an example of how we perform our qualitative evaluations. Thus, the system described in this section is exceedingly simple: it involves only two deployments (i.e., a single update) and contains only three small microservices — two of which act as producers of random numbers and one which consumes these numbers. The stated goal of the system is to build an application capable of generating numbers at one of the producers and having them printed to the standard output at the consumer.

4.1.1 Deployment #1

Listings 4.1 and 4.2 describe the interfaces of a producer microservice and its consumer, respectively. Listing 4.3 presents the deployment file that builds the microservices corresponding to these interfaces.

```
Defines:
  Endpoints:
    - At: '/random'
      Type: '() -> r: int'
      Method: GET
      Body: ~
      Reply: '{{r}}'

Depends: ~
```

Listing 4.1: Interface of the 1st random number producer.

```
Defines: ~

Depends:
  - Name: RandomNumberServer
    Endpoints:
      - At: '/random'
        Type: '() -> r: int'
        Method: GET
        Body: ~
        Reply: '{{r}}'
```

Listing 4.2: Interface of the (only) number consumer.

Since the specification of the consumer interface mirrors the producer's exactly, there's not much to discuss in this version of the system, and in fact Regent provides essentially no added value to the developer — the same pair of microservices running without Regent's support would work with the same ease.

4.1.2 Deployment #2

In the second deployment, the goal is to create a new number-generating microservice and to update the consumer so that it uses both producers.

We present the new microservice's interface specification in listing 4.4 and the consumer's updated specification in listing 4.5, but omit the deployment file since it ends up being virtually equal to the one in listing 4.3. Concretely, the only significant changes are replacing the older producer's entry (which is unneeded because nothing about it is changed) with the newer producer's entry (which is similar and only differs in obvious details like the hostname, which ports are open, etc), and adding a new entry to the consumer's list of dependencies (under the `Expects` key) mapping "`BoundedRandomNumberServer`"(used in the consumer's specification, cf. listing 4.5) to the hostname of the new producer.

The difference between the new producer and the old one is that the new one accepts two parameters, `l` and `u`, which set the `l`ower and `u`pper bounds for the values it generates.

```
- Specification-File-Path: "client/client.yaml"

  Image-Location: Registry
  Registry-URL: ~
  Registry-Port: ~
  Registry-Repository: user123/number-service-client:test

  Hostname: RandomNumberCruncher
  Expects:
    - Name: RandomNumberServer
      Specified-By:
        - a.nice.NumberServiceServer

  Machine:
    URL: 172.17.0.1
    Port: 5353

- Specification-File-Path: "server/server.yaml"

  Image-Location: Registry
  Registry-URL: ~
  Registry-Port: ~
  Registry-Repository: user123/number-service-server:test

  Hostname: a.nice.NumberServiceServer
  Service-Port: 8080:80
  Proxy-Port: 19820:19820
  Expects: ~

  Machine:
    URL: 172.17.0.1
    Port: 5353
```

Listing 4.3: Deployment file for the 1st deployment.

```
Defines:
  Endpoints:
    - At: '/random?l={l}&u={u}'
      Type: '(l: int [0], u: int [100]) -> r: int'
      Method: GET
      Body: ~
      Reply: '{{r}}'

Depends: ~
```

Listing 4.4: Interface of the 2nd random number producer.

```
Defines: ~

Depends:
  - Name: RandomNumberServer, BoundedNumberServer
    Endpoints:
      - At: '/random'
        Type: '() -> r: int'
        Method: GET
        Body: ~
        Reply: '{{r}}'
```

Listing 4.5: Updated interface of the (only) number consumer.

However, to account for the fact that the consumer may choose not to supply these values, the interface description provides default values for the bounds. Upon detecting a request from the consumer without bounds, Regent transparently inserts the default values before delivering it to the producer.

While this system's developer could have simply edited the consumer to treat each producer differently (sending the bounds to one and not the other) or updated the older producer to also accept a range, that would not necessarily be feasible if this weren't a toy example. This serves to show how Regent uniquely enables a progressive approach to modifying the system: all the developer had to do was alter the client to also direct its requests to the new microservice, and Regent took care of the rest. This way, it became possible to progressively update the system piece-by-piece instead of being forced to deal with all of the interdependent microservices at once.

4.2 System #2: Game Service

In this section, we present a system whose purpose is to support a video game. The game works by having players write their own custom agents to play against each other in games like chess (also known as a "bot tournament"). We present the system as being built iteratively, in response to changes in its requirements, over the course of 4 deployments.

Although each of the microservices in our example is small and we've attempted to simplify the interfaces as much as possible, altogether they amount to more than 10 interfaces throughout the 4 deployments. Thus, we omit all deployment files for the sake of clarity and brevity, and simply indicate which components communicate with each other when relevant.

4.2.1 Deployment #1

The first deployment contains only microservices capable of hosting a game of chess (listing 4.6) and client microservices to host the players' custom agents (listing 4.7). Each game microservice supports 3 endpoints, all of which the client microservices use: getting the state of the board (`/chess/board`), attempting to move a piece to another place on the board (`/chess/action`), and authenticating an agent with a secret string (`/auth`).

```
Defines:
  - At: '/chess/board'
    Type: '() -> b: str'
    Method: GET
    Body: ~
    Reply: '{"board": {{b}}}'

  - At: '/chess/action?p={p}'
    Type: '(p: str, x: int, y: int) -> _'
    Method: POST
    Body: '{"x": {{x}},
            "y": {{y}}}'
    Reply: ~

  - At: '/auth'
    Type: '(id: str) -> succ: int'
    Method: POST
    Body: '{"id": {{id}}}'
    Reply: '{{succ}}'

Depends: ~
```

Listing 4.6: 1st interface for the chess game microservices.

Similarly to the 1st deployment of System #1, not much of interest happens at this stage since the system is just being created. However, we can already point out an instance of Regent's features facilitating the communication of the microservices: notice that the client sends its request to move a piece solely in the body of the request as a JSON object, whereas the chess server expects to receive the piece's ID as a query string and the position as a JSON object.

The reason why these microservices are capable of interacting despite this mismatch is Regent. On detecting that there's something with the exact same name ("p") being communicated in two different manners, Regent transparently moves it from the place it was sent in to the place it is expected in.

As this is once again a small example, there would be simpler ways of solving the mismatch (e.g., simply rewriting one of the endpoints). However, our focus is again on non-toy examples: this functionality gives developers more flexibility when linking together microservices with pre-existing assumptions about how to send data (e.g., if there were already a checkers game that expected to receive the data in the way the player agents send it and there were already another type of player agent that sent the data as the chess game expects to receive it).

```
Defines: ~

Depends:
  - Name: GameServer
    Endpoints:
      - At: '/chess/board'
        Type: '() -> board: str'
        Method: GET
        Body: ~
        Reply: '{"board": {{board}}}'

      - At: '/chess/action'
        Type: '(piece: str, row: int, column: int) -> _'
        Method: POST
        Body: '{"p": {{piece}},
               "x": {{row}},
               "y": {{column}}}'
        Reply: ~

      - At: '/auth'
        Type: '(id: str) -> success: int'
        Method: POST
        Body: '{"id": {{id}}}'
        Reply: '{{success}}'
```

Listing 4.7: 1st interface for the player agent microservices.

4.2.2 Deployment #2

The second deployment adds two new types of microservice to the architecture: login microservices (listing 4.8) and random number generation (RNG) microservices (listing 4.10).

```
Defines:
  - At: '/auth'
    Type: '(id: str) -> succ: int'
    Method: POST
    Body: '{"id": {{id}}}'
    Reply: '{{succ}}'

Depends: ~
```

Listing 4.8: 1st interface for the login microservices.

In our hypothetical system, the developer realises that the chess game microservice should not be directly responsible for the authentication of the players' agents and begins the extraction of this functionality into its own login microservice. However, removing it from the chess game specification would prevent already-existing player agent microservices from interacting with newly deployed chess game microservices.

```
Defines:
  - At: '/chess/board'
    Type: '() -> b: str'
    Method: GET
    Body: ~
    Reply: '{"board": {{b}}}'

  - At: '/chess/action?p={p}'
    Type: '(p: str, x: int, y: int) -> _'
    Method: POST
    Body: '{"x": {{x}},
            "y": {{y}}}'
    Reply: ~

  - At: '/auth'
    Type: '(id: str) -> succ: int'
    Method: POST
    Body: '{"id": {{id}}}'
    Reply: '{{succ}}'

Depends:
  - Name: RNG
    Endpoints:
      - At: '/random'
        Type: '() -> random: int'
        Method: GET
        Body: ~
        Reply: {{random}}
```

Listing 4.9: 2nd interface for the chess game microservices.

```
Defines:
  - At: '/random'
    Type: '() -> rand: int'
    Method: GET
    Body: ~
    Reply: {{rand}}

Depends: ~
```

Listing 4.10: 1st interface for the random number generator microservices.

As Regent separates interface specifications from the deployment details, this is not problematic: the developer can simply require that all new player agents use the login microservice as per the new specification in listing 4.11, which does not affect the older player agent specification. Simmultaneously, the developer can temporarily keep the /auth endpoint in the chess game's interface, guaranteeing that existing player agents will still be able to connect to newly-deployed chess game microservers.

In this situation, Regent's usefulness is in helping coordinate this transitional process by only accepting one of two scenarios: either the chess game's interface maintains the authentication endpoint, or it removes the endpoint while never being allowed to be contacted by the older player agents. If it were not for Regent, it would be easy to forget to account for the existing player agent microservices and end up in a situation where one of them tries to play a game of chess on a newer (incompatible) game microservice.

```
Defines: ~

Depends:
  - Name: GameServer
    Endpoints:
      - At: '/chess/board'
        Type: '() -> board: str'
        Method: GET
        Body: ~
        Reply: '{"board": {{board}}}'

      - At: '/chess/action'
        Type: '(piece: str, row: int, column: int) -> _'
        Method: POST
        Body: '{"p": {{piece}},
                "x": {{row}},
                "y": {{column}}}'
        Reply: ~

  - Name: LoginServer
    Endpoints:
      - At: '/auth'
        Type: '(id: str) -> success: int'
        Method: POST
        Body: '{"id": {{id}}}'
        Reply: '{{success}}'
```

Listing 4.11: 2nd interface for the player agent microservices.

Meanwhile, at this point the RNG microservices are only used by the updated chess games in order to offer variants on traditional chess (e.g., with a random number of pawns removed); We expand on the RNG microservices in the remaining deployments.

4.2.3 Deployment #3

The third deployment expands the login (listing 4.13) and RNG (listing 4.12) specifications, while leaving the chess game one untouched and only minimally refactoring the player agents' (listing 4.14).

```
Defines:
  - At: '/random?l={{low}}&h={{high}}'
    Type: '(low: int [0], high: int [100]) -> rand: int'
    Method: GET
    Body: ~
    Reply: {{rand}}

Depends: ~
```

Listing 4.12: 2nd interface for the random number generator microservices.

Similarly to our example in System #1, the RNG microservices are refactored to accept a pair of bounds between which to generate the random numbers. And just as in System #1, the bounds are given default values so that Regent can cover for microservices who do not set them themselves. As we already discuss the benefits of specifying default values for endpoints in that system's section (4.1.2), we avoid repeating ourselves here.

```
Defines:
  - At: '/login'
    Type: '(username: str, password: str) -> success: int'
    Method: POST
    Body: '{"username": {{username}},
            "password": {{password}}}'
    Reply: '{{success}}'

Depends:
  - Name: SecureRNG
    Endpoints:
      - At: '/random?l={{min}}&h={{max}}'
        Type: '(min: int, max: int) -> random: int'
        Method: GET
        Body: ~
        Reply: {{random}}
```

Listing 4.13: 2nd interface for the login microservices.

We also note that the inverse situation is supported: a microservice may be prepared to send out bounds and still decide to point its request at an older RNG microservice that doesn't support them. In this case, Regent intervenes by simply discarding the bounds before delivering the request to the RNG microservice. One possible use case for this functionality would be to update the users of the RNG microservices before actually updating the RNG microservices themselves — which might seem pointless in our toy example, but which may be useful in a system where the team developing the "senders" is ready to deploy its work before the team developing the "receivers".

```
Defines: ~

Depends:
  - Name: GameServer
    Endpoints:
      - At: '/chess/board'
        Type: '() -> board: str'
        Method: GET
        Body: ~
        Reply: '{"board": {{board}}}'

      - At: '/chess/action'
        Type: '(piece: str, row: int, column: int) -> _'
        Method: POST
        Body: '{"p": {{piece}},
                "x": {{row}},
                "y": {{column}}}'
        Reply: ~

  - Name: LoginServer
    Endpoints:
      - At: '/login'
        Type: '(username: str, password: str) -> success: int'
        Method: POST
        Body: '{"username": {{username}},
                "password": {{password}}}'
        Reply: '{{success}}'
```

Listing 4.14: 3rd interface for the player agent microservices.

At this point, we also finalise the login rework by turning it into a more traditional username/password based approach and updating the player agent specifications to depend on the new login. In addition to this, the login specification also becomes dependent on the RNG microservices, in order to generate cryptographic-quality randomness for the login algorithm.

```
Defines:
  - At: '/random?l={{lo}}&h={{hi}}'
    Type: '(lo: int [0], hi: int [100], flags: int) -> rand: int'
    Method: POST
    Body: {{token}}
    Reply: {{rand}}

Depends: ~
```

Listing 4.15: 1st interface for the cryptographically safe RNG microservices.

```
Defines:
  - At: '/login'
    Type: '(username: str, password: str) -> success: int'
    Method: POST
    Body: '{"username": {{username}},
            "password": {{password}}}'
    Reply: '{{success}}'

Depends:
  - Name: SecureRNG
    Endpoints:
      - At: '/random?l={{min}}&h={{max}}'
        Type: '(min: int, max: int, flags: int) -> random: int'
        Method: GET
        Body: {{token}}
        Reply: {{random}}
```

Listing 4.16: 3st interface for the login microservices.

4.2.4 Deployment #4

Finally, we reach the fourth and last deployment of our example. It is only now, with the login functionality reworked and extracted into its own set of microservices, that the developer decides to fully remove the /auth endpoint from the chess game specification (listing 4.17) — making it so that microservices created or updated using this interface are no longer allowed to accept requests from player agent microservices that have lingered on unchanged since the 1st deployment.

To finish our example, we present one last requirement change faced by our imaginary developer: for security reasons, it is now necessary to allow a more fine-grained control over the cryptographic RNG used in the login microservices.

The obvious way to handle this requirement would be to simply add a "flags" field to the existing RNG specification and give it a default value to handle callers who don't provide flags. Suppose, however, that there is no sensible default value for these flags and they must always be explicitly selected by the caller.

Given this situation, the only viable option becomes to create a new kind of "cryptographically safe" RNG microservice (listing 4.15) that's incompatible with the normal RNG microservices and their callers, and change the login microservices (listing 4.16) to use them.

Although it may seem like a minor detail, the way we handle this new requirement serves to showcase the benefit of fully separating the interfaces from their deployment details: as the usages of a specification start to naturally diverge, Regent makes it seamless to simply copy it, slightly modify it, and use it to start creating new microservices.

```
Defines:
  - At: '/chess/board'
    Type: '() -> b: str'
    Method: GET
    Body: ~
    Reply: '{"board": {{b}}}'

  - At: '/chess/action?p={p}'
    Type: '(p: str, x: int, y: int) -> _'
    Method: POST
    Body: '{"x": {{x}},
           "y": {{y}}}'
    Reply: ~

Depends:
  - Name: RNG
    Endpoints:
      - At: '/random'
        Type: '() -> random: int'
        Method: GET
        Body: ~
        Reply: {{random}}
```

Listing 4.17: 3rd interface for the chess game microservices.

4.3 Conclusions

As we explained at the start of this chapter, our evaluation of Regent was purely qualitative. Although we did implement most of our ideas into a prototype, and did manage to run some examples on it, we can see no useful quantitive metrics to support Regent's claims. Because of this, we chose to justify the worth of our approach by walking through some examples and discussing the features that Regent enabled when compared with a Regent-less approach.

Namely, throughout sections 4.1 and 4.2, we demonstrated how Regent's ability to automatically adapt requests that don't match what the requestee expects to receive — be it through injecting default values where none are provided, dropping extraneous parameters, or moving data from one being sent in the query string to the body and vice-versa — is useful for diminishing and/or delaying the work needed to update parts of a microservice-based system.

Additionally, we argued for the usefulness of Regent as a safeguard when refactoring parts of a system: as Regent rejects deployments that would lead to irremediably incompatible states — such as a microservice trying to use an endpoint that no longer exists on another one —, it becomes harder to accidentally "break" the system.

Finally, we demonstrated several times how malleable interface specifications are: a specification can always be modified without affecting previously deployed microservices that used that specification, thanks to how Regent separates deployment details from interface information.

Chapter 5

Future Work

In its current form, our prototypical implementation of Regent has accumulated a lot of technical debt due to the many different avenues we explored during its development. As such, we believe there's a significant amount of improvements that could be done just reimplementing the prototype from the ground up, in a more disciplined manner and with the benefit of now knowing which ideas work and which ones don't.

Particularly, we now believe that choosing Python (and the associated ecosystem) to implement Regent was a mistake, and that simply porting over the prototype to a language with a stricter type system would result in an immediate and noticeable increase in our prototype's quality. We believe our issues with Python ultimately come down to Regent having to perform so many different kinds of comparisons and conversions to enable the adaptation mechanism, that it becomes nearly impossible to accurately track the type of anything at all, forcing us to spend time testing the same things repeatedly to check that they really are what we expect them to be.

The experience of implementing Regent has also shown us how quickly some of the tools on which we built Regent degenerate once you attempt to push them beyond the "happy path". As a concrete example, the tools for inspecting and dynamically modifying Docker images quickly became unsuitable for our purposes and required us to implement our own *ad hoc* way of preserving the image's start-up command when attempting to modify it. Thus, a direction in which to develop Regent would be either finding better ways to integrate it with these tools or finding more powerful tools on which to build it.

Another interesting avenue to explore would be finishing some of Regent's more difficult or tedious to implement features, such as guaranteeing the transactionality of Regent Controller's deployment of microservices onto the Regent Participants, making the Controller capable of handling multiple requests in parallel instead of having to serialise them, creating automatic interface specification generation tools for mainstream languages like C or Java, or supporting technologies beyond HTTP and JSON.

Lastly, we believe it would be very interesting to look into expanding Regent's compatibility/adaptation relation. Specifically, would a relation capable of expressing pre-conditions and post-conditions be useful in Regent? If yes, how would it affect the usability of writing/reading interface specifications? These are questions which we've already spent some time on, but for which we do not yet have any concrete answers.

Bibliography

[1] Ada Mancini. *How to use an entrypoint script to initialize container data at runtime*. Accessed 2020-07-12. URL: https://success.docker.com/article/use-a-script-to-initialize-stateful-container-data.

[2] M. Autili, A. Di Salle, F. Gallo, C. Pompilio, and M. Tivoli. "CHOReVOLUTION: Automating the Realization of Highly–Collaborative Distributed Applications." In: *Coordination Models and Languages*. Ed. by H. Riis Nielson and E. Tuosto. Cham: Springer International Publishing, 2019, pp. 92–108. ISBN: 978-3-030-22397-7.

[3] M. Autili, A. Di Salle, C. Pompilio, and M. Tivoli. "CHOReVOLUTION: Hands-On In-Service Training for Choreography-Based Systems." In: *Coordination Models and Languages*. Ed. by S. Bliudze and L. Bocchi. Cham: Springer International Publishing, 2020, pp. 3–19. ISBN: 978-3-030-50029-0.

[4] Benjamin Podszun (darklajid), et al. *Moby Github Repository, docker build should support privileged operations - Issue #1916*. Accessed 2020-04-13. URL: https://github.com/moby/moby/issues/1916.

[5] Cloudfare, Inc. *What Is a Serverless Microservice? | Serverless Microservices Explained*. Accessed 2020-04-20. URL: https://www.cloudflare.com/learning/serverless/glossary/serverless-microservice/.

[6] Docker Inc. *Docker Documentation*. Accessed 2020-04-13. URL: https://docs.docker.com/engine/reference/builder/#cmd.

[7] Docker Inc. *Docker Documentation*. Accessed 2020-04-13. URL: https://docs.docker.com/engine/reference/builder/#entrypoint.

[8] Docker Inc. *Empowering App Development for Developers | Docker*. Accessed 2020-04-20. URL: https://www.docker.com/.

[9] Google LLC. *Protocol Buffers | Google Developers*. Accessed 2020-04-20. URL: https://developers.google.com/protocol-buffers/.

[10] gRPC Authors. *gRPC – A high-performance, open source universal RPC framework*. Accessed 2020-07-07. URL: https://grpc.io/.

[11] Koen Claessen et al. *QuickCheck: Automatic testing of Haskell programs*. Accessed 2020-07-12. URL: https://hackage.haskell.org/package/QuickCheck.

[12] L. Lamport. "Paxos Made Simple." In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[13] Microsoft Corporation. *API Management | Microsoft Azure*. Accessed 2020-04-20. URL: https://azure.microsoft.com/en-gb/services/api-management/.

[14] Microsoft Corporation. *Transform and protect your API with Azure API Management | Microsoft Docs*. Accessed 2020-04-20. URL: https://docs.microsoft.com/en-gb/azure/api-management/transform-api.

[15] F. Montesi, C. Guidi, and G. Zavattaro. "Service-Oriented Programming with Jolie." In: *Web Services Foundations*. Ed. by A. Bouguettaya, Q. Z. Sheng, and F. Daniel. New York, NY: Springer New York, 2014, pp. 81–107. ISBN: 978-1-4614-7518-7. DOI: 10.1007/978-1-4614-7518-7_4. URL: https://doi.org/10.1007/978-1-4614-7518-7_4.

[16] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. "How Amazon Web Services Uses Formal Methods." In: *Communications of the ACM* 58 (Mar. 2015), pp. 66–73. DOI: 10.1145/2699417.

[17] S. Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357.

[18] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer. "Live Functional Programming with Typed Holes." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 14:1–14:32. ISSN: 2475-1421. DOI: 10.1145/3290327. URL: http://doi.acm.org/10.1145/3290327.

[19] B. C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[20] Python Software Foundation. *Glossary — Python 3.8.2 documentation*. Accessed 2020-04-19. URL: https://docs.python.org/3/glossary.html.

[21] J. C. Seco, P. Ferreira, H. Lourenço, C. Ferreira, and L. Ferrão. "Robust Contract Evolution in a TypeSafe MicroServices Architecture." English. *Unpublished*. 2019. DOI: 10.4230/LIPIcs.ECOOP.2019.23. URL: https://bit.ly/microservices_paper_techreport.

[22] Smartbear Software. *About Swagger Specification | Documentation | Swagger | Swagger*. Accessed 2020-04-20. URL: https://swagger.io/docs/specification/about/.

[23] Smartbear Software. *API Code & Client Generator | Swagger Codegen | Swagger*. Accessed 2020-04-20. URL: https://swagger.io/tools/swagger-codegen/.

[24] *Sonic Pi - The Live Coding Music Synth for Everyone*. Accessed 2020-04-20. URL: https://sonic-pi.net/.

[25] The Kubernetes Authors. *Production Grade Container Orchestration - Kubernetes*. Accessed 2020-04-20. URL: https://kubernetes.io/.

[26] J. Thones. "Microservices." In: *IEEE Software* 32.01 (2015), pp. 116–116. ISSN: 0740-7459. DOI: 10.1109/MS.2015.11.

[27] Tianon Gravi (tianon), et. al. *Docker Official Image for postgres*. Accessed 2020-07-12. URL: https://github.com/docker-library/postgres/blob/master/docker-entrypoint.sh.

[28] Tony Mauro. *Adopting Microservices at Neflix: Lessons for Architectural Design*. Accessed 2020-04-20. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.