UNIVERSITY OF
BUCHAREST
VIRTUTE ET SAPIENTIA

University of Bucharest

Faculty of Mathematics and Computer Science

Department of Computer Science

Pandora.cc

Multithreading and Cloud Computing in a box

Author:
Alexandru Velea

Supervisor:
Lect. dr. Păduraru Ciprian

BSc Thesis

February, 2020

Abstract

The purpose of this paper is to introduce the reader to a new C++ framework called Pandora.cc while showcasing its potential uses and the reasoning behind various design choices.

The framework is a blend of micro and nano services architecture. It provides fast and powerful parallelism capabilities for writing nanoservices using a standard RPC-server interface. The creation of bundles of nano services allows the creation of microservices that can be deployed in the cloud, allowing the code to scale without requiring code changes. Each Pandora.cc binary can be configured with ease to serve a subset of services along with technical hardware specifications regarding thread distribution, allowing lock-free code to emerge and to allocate more resources to latency-sensitive services.

The paper concludes with various benchmarks to present experimentally that the framework can be used successfully in both multithread and distributed systems. The presence of both code examples and UML diagrams showcase the slick and modern API design and functionality supported while the introduction of the various design principles helps users decide if Pandora.cc is a good fit for them.

Abstract

Scopul lucrării e de a familiariza cititorul cu noua librarie de C++, Pandora.cc, în acelaşi timp, arătând potentiale utilizări si explicând principiile de design care au stat la crearea acesteia.

Librăria combină nuanţe din architecturile software de micro si nano servicii. Oferă capabilităti rapide şi configurabile de papalelism pentru dezvoltarea applicaţiilor folosind nanoserviciilor împreună cu o interfaţă standard de tip RPC. Abilitatea de a crea grupuri de nanoservicii este folosită pentru a definii componente similare cu microserviciile, ca mai apoi acestea să fie rulate in cloud, permitând o scalabilitate orizontala fără a modifica codul existent. Fiecare executabil de tipul Pandora.cc poate fi configurat cu uşurinţă să satisfacă un subset din serviciile implementate, fiind posibilă o configurare pentru a preciza ce resurse hardware sunt folosite de servicii, astfel obtind un cod fără mutexi care poate gestiona mai bine resursele de intrare/ieşire.

Lucrarea se sfarşeşte cu diverse teste de performantă care atestă experimental că librăria poate fi folosită cu succes atât pentru aplicaţii multithreaded cât si ca o optiune de calcul distribuit. Prezenţa atât a digramelor UML cât si a exemplelor de cod atestă un API modern şi usor de folosit iar introducerea în lucrare a principilor de design din spatele librăriei permite utilizatorilor să decidă dacă poate fi considerată o optiune viabila pentru cazul personal.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Due to the rise of tech giants, microservices became more popular as a way to achieve horizontal scaling in large applications. As new projects are born, many developers look to this architectural style knowing they would be able to scale their application no matter the requirements while benefiting from the ease of development and testing which comes along.

In reality, trying to implement an idea using microservices from the start can be a changeling task since it enforces many rules and practices, which could make the process more robust. Such examples could be changing APIs, deploying in early-development, and involving additional layers of complexity such as containers and networking.

Pandora.cc tries to offer a fast and easy way to develop services in a new fast-changing decade. It tries to close the bridge between same-server parallel computing and cloud computing, being best suited for latency sensite application. The framework is build using lock-free data structures and CPU atomic data types to ensure that as little time as possible is wasted by waiting for resources on same-server case. The configurability which comes with the configuration system, called the bundling system, allows modifications of the network architecture without affecting the codebase, including which services are available on each instance. The source code of the framework can be found on GitHub at `github.com/alexvelea/pandora.cc`.

The paper makes three main contributions. First, we present three architectures to justify a new design, showcasing the advantages and disadvantages of such practices. Second, we present a series of tools that facilitate better use and coherency when developing software with Pandora.cc framework. Third, we showcase the development process as well as demonstrating experimentally that the approach is optimal and achieves all the design principles put in place.

Framework advantages

- **Easy API** All the development is done throw writing nanoservices, which are defined by a simple API consisting of a pair of Request-Response and the code to handle such a service. This process is as easy as designing a function that takes a single argument, the request, and returns a specialized value, the response.

- **High-speed messaging** Nanoservices calls on the same machine have reduced latency, peaking at 10^6 messages/second/core. This functionality enables programs to run fast and efficient in early development until multiple servers are required, that is, until the application cannot be vertically scaled anymore. Horizontal scaling should come very late in a product's life since 128 thread CPU's retail at around $2000 as of the date of the paper.

- **Thread selection** The ability to specify on which CPU-thread a nanoservice should run on, enabling lock-free code if all the services are designed to be run on a single thread, for example, a set-get pair of services. Extension of the nanoservices into the cloud, as micro-services, alongside a suite of features similar to the ones provided by Google's Protocol Buffers.

- **Hot Upgradeability** The rollout of new versions of specific nano-services can be done in production on targeted binaries while they are still running, meaning that even TCP connection could remain active while a rollout is taking place, reducing the down-time to 0.

1.2 Structure

The first part of the paper, chapters 2 to 4, presents several concepts regarding cloud computing and multithreaded workloads, providing insights on both the advantages and disadvantages of several software architectures and practices. It presents Pandora.cc framework, reasoning about its position on the presented options while synthesizing some problems that it tries to solve.

The first part starts with a short glossary section, chapter 2, for those unfamiliar with some terms, followed by a short theoretical introduction, into the framework, chapter 3, allowing readers to grasp the end-goal, usages, and motivation behind the framework. All this offers vital insights to look after while going throw the rest of the paper.

Chapter 4 goes into a in-depth comparison and preview of 3 software architectures: Microservices, Nanoservices and the Actor Model. While only a subset of the available technologies is showcased, it offers a good starting point for those unfamiliar with such

practices while presenting them in a way that it's easier to see how and why they were selected as inspiration for the framework. The paper presents then in a way that emphasizes good ideas that are going to be implemented while comparing the differences with each architecture. It concludes with a conclusion and tools section which ties all the design choices together, presenting potential drawbacks and putting, in contrast, these choices with the design principles taken into consideration while developing the framework. The tools can be seen as the most significant innovative contribution provided by the paper, further consolidating the design pieces.

The second part of the paper, (chaper 5) presents the implementation of critical components of the framework, showcasing performance metrics as well as code-snippets to showcase how the components are meant to be used together. Optimizations play a big part in developing a fast framework, covering them offering insights on the inner workings of the framework as well as raising awareness for potential pitfalls in further development on various projects.

Chapter 5.4, In depth example overview, presents an example of a possible service along with two diagrams to explain the inner workings of the framework. The first one is an UML diagram, showcasing the user code and the interactions with the framework, containing the whole development-stack. The second one, a sequence diagram, presents how a request interacts with various components while being processed by the service.

Chaper 5.5, Development process, offers more insights into the state of the project and the process of developing the framework itself. Although it doesn't affect using the framework, it may give readers a new perspective for future projects. It tries to tackle common problems and offer new ways to look at things, focusing on taking notes from agile development practices while not being drowned in the complexity of implementing them 'by the book'.

Chaper 5.6, Testing, offers essential insight since it showcases controversial practices. Such principles can be seen in the framework itself, such as doing the bare minimum to achieve great results without over-engineering mainstream solutions just because they are popular. It makes good use of various tools that may be overlooked or unheard of while arguing over the cost of maintenance and development of the selected method.

Chapter 2

Glossary

This chapter presents a short description of some popular terms which will be used throw out the paper, but it does not offer comparisons or advantages, but instead, it tries to paint a real picture that might escape the bounds of definitions found in the literature. The purpose is to familiarise readers with the terms and use cases so that it'll be easier to figure out what we're trying to achieve and what the limitations are.

2.1 Arhitectures

A software arhitecture defines the general structure of a software system. It embodies how the data is managed between, the relationships and requirements of the components. Chosing the right arhitecture can affect development in multiple ways, including reliability of the product, speed, development pace, scalability and upgradeability.

2.1.1 Microservices

Microservices, also known as microservice arhitecture is an arhitectural style that structures the core elements of the software into a collection of independent and scoped services. The term was introduced in 2013, being described as a granular approach to SOA(service oriented arhitecture).

While SOA breakes down functionality based on business logic, for example a simple `OrderService`, a microservice based design breaks down that functionality even further into `CustomerManagement`, `OrderManagement` and `OrderProcessing`.

Microservices are defined as grained services with precise business-capability, with a standardized interface and a lightweight protocol [Hassan et al. 2017]. Such microservices usually run in their own process, containerised, making them easily to deploy and upgrade [Alshuqayran et al. 2016]. The comunication between microservices is done using the network through a RESTfull API or RPC(Remote Procedure Call), making them language independent. This enables the use of multiple microservices in parallel behind a load

balancer to achieve horizontal scaling, i.e, scaling up with the number of servers not the number of CPU cores.

2.1.2 Nanoservices

Nanoservices are commonly viewed as granular microservices. They share a big part of the design principles of microservices, the only difference being that nanoservices offer just one endpoint. If in a `OrderService` SOA application contains multiple microservices, such as `OrderManagement`, this can be split even further into smaller modular pieces, for example `PlaceOrder` and `CancelOrder` so that they can form a microservice when bundled together.

Nanoservice advocates try to provide a usecase-based definition, trying to focus on where a nanoservice should be used and what are the key differences in doing so when compared to a microservice. This enables a more robust definition which is not tied to the size or the number of APIs exposed. More precisely, a nanoservice is defined as a service which is deployable and most importantly reusable. One such example could be a service which provides files from various sources. This would be helpfull since all the other services share one unified API to retrieve files and they are agnostic on the location of these files. Some of them could be stored in the database while others could be in the cloud, in a storage server.

2.2 Parallel computing

Parallel computing refers to the act of running multiple computations simultaneously. Even though this was possible from the introduction of the first multi-core CPU, the trend has reached a golden age with the inability to scale up the frequency of cores, manufacturers opting to increase the number of cores/CPU instead of increasing the frequency of the actual cores. Not to be confused with distributed computing where one task is split among multiple computers inside the same network.

2.2.1 Concurrency

Concurrent computing, usually being mistaken for parallel computing, is a form of computing in which multiple tasks can have an overlapping execution time, but no 2 tasks will be running at the same time. This paradigm allows more accessible design since there is no need for locking shared resources between tasks while allowing multiple flows to execute seamlessly in parallel if the work to be done is lightweight. Usually, when talking about concurrency, tasks do not switch context mid-run, meaning that one task should finalize or yield (force pause) before the schedules choose another one. This can be

seen in high-level languages that offer concurrency but not parallelisms, such as Python and JavaScript.

2.2.2 Threads

Threads, not to be confused with CPU threads, are theoretical code execution flows which belong to a process. A process can have multiple threads, each of them being can run in parallel. The OS(operating system) chooses when and which threads should run in one moment of time using the scheduler. When a thread switch is made, i.e., the OS chooses to run another thread, a context switch takes place, which adds an overhead for the operation. One main advantage of threads compared to having multiple processes running in parallel is memory sharing. While threads do not share the stack, they do share heap-allocated variables, making the transfer of information very convenient. Because of this, new atomic locking mechanisms need to be added, so synchronization of threads will be possible. These atomic operations can be seen as operations that happen one at a time.

2.2.3 Promises and futures

Promises and futures represent abstract ways to do parallel programming. These features can be implemented in the majority of languages, even if they do not support parallel programming per se. This is happening because they can be used as a means to retrieve information from the network, which can arrive at any time. A future can be viewed as a place holder for a variable that will be populated by another thread or action in the future. In C++, promises are a means of creating futures. The promise takes the action of setting the value of the promise for the generated child-future while other code-flows can wait for that to happen.

2.2.4 Callbacks

Callbacks represent another way to deal with asynchronous calls. Popularised by JavaScript, callbacks will execute some code when the result of an asynchronous call is determined. For simplicity, this concept is very similar to futures in the sense that it can be achieved by waiting on a future and executing the callback afterward while running the code between the initial function call and the future wait as another task on a different thread. Callbacks are generally used when the result is not influencing the state of the flow before the code, but either producing side-effects such as printing or changing UI or spawning additional calls on its own. An essential factor to consider is that while a future is being waited for, the thread doing so is waiting, wasting valuable CPU time. Based on this observation, callbacks tend to minimize wasted time.

2.2.5 The actor model

Being created in 1973 by Carl Hewitt, the actor model is a mathematical model that treats actors as the entities that carry out concurrent programming. An actor has a local scope state, the sharing of information being done throw sending messages. When an actor receives a message, it can carry out local operations, send other messages, create new actors, and modify its own state before sending out the reply. In this way, there is no need for lock-based synchronization, as in the case of thread-programming.

Nowadays, it's used mostly in proofs and in some functional programming languages since the coding mindset is more common than in imperative programming languages. Erlang was one of the first widely-used programming languages which adopted the actor model, introducing it's "let it crash" philosophy. This can happen because actors have limited scope, and in case of a fail, only that actor is affected by it, meaning it can be recreated with ease. With this being said, it's not uncommon to hear about applications that use the actor model, which has been running without any problem for more than 5 years. Erricson is reporting a 99.9999999% reliability over the course of 20 years for it's ATM switch, being down for only

Chapter 3

Framework Overview

This chapter tries to present a brief overview of what the library does and showcase some use cases as well as the motivation behind it. The contents of this chapter are optional, but it'll help to put things in a better perspective as we dive into various comparisons and implementations.

3.1 Motivation

State of the art

With the developments in the CPU world, high-level languages became more and more popular since the majority of the users could offer the required computing speed to overcome the overhead of such languages. In recent years this trend gained traction in the service world as well. Developers started opting for server-side code in Python or Node.js, which would require more servers or deliver worse latency in very sensitive cases. The majority of these arguments are usually countered by the fact that the cost of developing in higher-level languages is much higher than the cost of the extra hardware required.

History

Having some experience with parallel programming, the first big project served an essential roll in discovering the pitfalls and benefits of different practices along with various implications which can't be easily quantified, as development speed, code maintainability cost, and complexity for users which are not familiar with the codebase into the point where they are able to contribute.

The need for parallelism raised as a necessity to make the code faster, opting for vertical scaling in order to maximize the throughput. Due to the design of the application, some components couldn't be parallelized, yielding a codebase where the flow of a request was partially parallelized to increase the response time. Since most parallel programming

libraries added a substantial overhead of communication, they couldn't be used in order to decrease the time of a specific request. Due to this context, we opted for a low-level option, a thread-pool. This seemed like the best way to reduce overhead and write little bloatware. But in time, the complexity of the code increased, and the tasks being run in the thread-pools became chaotic really quick. This made the code really hard to follow and to synchronize. The first alternative to the problem was to use something similar to the microservices architecture, and with some digging, the nanoservices resurfaced as a viable solution.

Solution

Pandora.cc started out with the simple idea to offer a fast way for nanoservices to communicate, thus C++ being the language of choice. The target is to take function calls and run them on a different thread while being able to retrieve the result in a practical way while having some control over which hardware components it runs so it'll reduce the number of synchronization mechanisms required. The first part could've been easily achieved using std::future and std:: async, both being available in the standard library of C++. The problem was that they were too slow, thus increasing the desire for such a library. In time, the idea to make function calls to other threads evolved in making these calls to another computer altogether, all of these details being taken care of the framework, while someone looking at the code couldn't determine if the service call would be run on the same server or not.

3.2 What it does

Pandora.cc enables the development of applications using nanoservices as its primitive computational unit while highly rewarding having a good application architectural structures. These nanoservices benefit from a fast message mailbox while communicating on the same server, i.e., when no network call is involved. This allows them to become fast, maintainable, and easy to develop a way to express code.

To an outside user, Pandora applications look similar to applications that follow a 'less regulated' actor model due to its odd development practices. While this may be the case, the code is enforced to be more structured and to address data-sharing accordingly. After the development of the code, an unusual step arises, in the development of how the nanoservices will be run inside a Pandora application.

Developing a good architecture regarding how nanoservices should run on a service is a critical step when using Pandora. This is done using the bundle system. The bundle system allows for bundling of different nanoservices which are used together to form a microservice. Such an example can be a FileBundle, which consists of FetchFile, Process-

File, and SaveFile. While these nanoservices are generally used together, they will benefit from being able to run in parallel, and, if run on the same server, the communication overhead becomes negligible. The memory transfer is equivalent to copying an object inside the server's RAM while the framework reduces the latency for each call, being able to achieve 10^6 sequential calls between a pair of services (a $\rightarrow$ b $\rightarrow$ a $\rightarrow$ b ...).

Nanoservice thread specification is also achieved throw the bundling system. While specifying for each nanoservice how many replicas(running instances) should an application run, it's also possible to specify on which CPU threads this should happen. In this way, it's possible to share memory between nanoservices through a global state which does not require any synchronization mechanisms if done right. For example, 2 nanoservices SetValue and GetValue can be configured to be able to run on the same thread for lock-free code. Another use case is represented by nanoservices, which are hardware bound, like FetchFile and SaveFile. If both of these nanoservices ran in parallel, it would not speed up the process since the bottleneck is represented by the SDD/HDD, not the CPU. Service prioritization is another aspect to keep in mind since it can allow some latency-sensitive services always to have a free thread waiting for them so the task can start at once.

Distributed programming becomes available just by writing some config files. The nanoservice calls are unaware if they will be processed on this server or somewhere in the cloud. When deploying another Pandora binary, it'll automatically broadcast its list of available nanoservices endpoints to all other binaries in the network, via a service discovery process. Target use cases High-performance applications. Due to it's fast and customizable parallelism model, Pandora.cc can be successfully used as a viable option for developing single-server apps that require high thruput and can benefit from using multiple CPU-cores. Such examples can include stock-trading apps or cryptocurrency algorithm implementations that use proof of stake.

First framework for small applications. Due to the API of Pandora.cc, it's easy to migrate if needed to another parallel or distributed framework since the majority of the code could be salvaged with ease. This is backed up by the fact that the code is written with minimum glue-code, and the user is not tied in some technologies and can easily switch to a more classic microservices framework, like using Google's RPC framework along with protobuf.

Future scalability. While most projects start without a defined idea in mind and can find themselves in a tight spot due to limitations to enter horizontal scaling, Pandora.cc enables this by design while trying to protect developers from harmful practices. This is a crucial decision to make before choosing the framework since the majority of applications would not require vast resources, but this can become a reality in case of a surge in the number of users or just organic growth. Usually, this can have bad implications, which can be found especially in applications that follow the Monolithic Architecture pattern,

where developers find themselves refactoring a lot of time at once, trying to make the codebase scalable.

Fast scalability. While all microservice-based systems offer unlimited horizontal scalability by design, Pandora.cc tries to offer fast on-demand scalability as well. Some applications might require a boost in a specific service from time to time, due to special events such as sales or holidays. Such an example would be Black Friday when online shops would require to add more services that would serve products that a user might want to buy based on past purchases or currently opened pages. This can be done using Pandora.cc without codebase changes or changing configs for current-running services.

3.3 Target use cases

- **High-performance applications** Due to it's fast and customisable parallelism model, Pandora.cc can be succesfully used as a viable option for developing single-server apps which require high thruput and can benefit from using multiple CPU-cores. Such examples can include stock-trading apps or crypto currency algorithm implementations which use proof of stake.

- **First framework for small applications** Due to the API of Pandora.cc, it's easy to migrate if needed to another parallel or distributed framework since the majority of the code could be salvaged with ease. This is backed up by the fact that the code is written with minimum glue-code and the user is not tied in some technologies and can easely switch to more clasic microservices framework, like using Google's RPC framework along with protobuffers.

- **Future scalability** While most projects start without a defined idea in mind and can find themselves in a tight spot due to limitations to enter horizontal scaling, Pandora.cc enables this by design while trying to protect developers from bad practices. This is an important decision to make before choosing the framework since the majority of applications whould not require huge resources but this can become reality in case of a surge in the number of users or just organic growth. Usually this can have bad implications which can be found especially in applications which follow the Monolithic Architecture pattern, where developers find themselves refactoring a lot of time at once trying to make the codebase scalable.

- **Fast scalability** While all microservice-based systems offer unlimited horizontal scalability by design, Pandora.cc tries to offer fast on-demand scalability as well. Some applications might require a boost in a specific service from time to time, due to special events such as sales or holidays. Such an example would be Black Friday, when online shops would require to add more services which would serve products

that a user might want to buy based on past purchases or current opened pages. This can be done using pandora.cc without codebase changes or changing configs for current-running services.

Chapter 4

Architectural and Design Decisions

In this chapter, we look over the design decisions that influenced the shape of the library along with a theoretical comparison of various models or architecture, pointing out advantages and possible use cases for each of them. The purpose of such a section is to justify the way the framework is intended to be used as well as pointing out the differences in relation to other popular ways of achieving a similar result.

4.1 Design Principles

Understanding the design principals on which a framework was build is a key part of quantifying if the desired result was achieved whilst guiding the development process in the right direction. While many of them seem rudimentary or obvious, many others were left out, such as cross-platform compatibility, design for failing, or crash stability. Some of the design decisions are presented below, along with some reasoning behind them below, in order of importance.

1. **Speed** Speed is one of the most important factors we look after. In order for the framework to be speed-reliable, it should do a decent number of "ping-pong" (a→b→a→b...) operations per second, i.e., having 2 services that talk to each other as much as they can to share some meaningful information or to simulate an actor. As a decent benchmark for this, for the majority of the cases, 10^6 operations per second seems a decent enough number to strike for.

2. **Ease of use and reducing glue code** Being able to write code without having to copy-paster or follow a lot of generic code is a must. The most important thing is to be able to get something done with at little code as possible while maintaining a generic enough API. Glue code and heavy APIs represent an important source of bugs. Forcing developers to write meaningless code decreases the overall attention to details, letting subtle bugs to be created in the weirdest scenarios.

3. **Fast scaling** While scaling is an important factor and advantage in microservice-based applications, the speed scaling speed becomes an important factor to look after. The rise of AWS (Amazon Web Services) allowed new tools to be popularised in the DevOps(software development - Dev and information-technology operations - Ops) community such as elastic scaling. This allows for automatic deployment in case of a sudden increase in requests. Pandora.cc tries to take this a step forward, enabling a fast repurposing and reorganization of the structure of the project without any required downtime.

4. **Easy Deployment and Upgradeability** Being able to perform easy deploy operations is an important aspect of developing, especially if deployment is involved in the testing process. Sometimes seen as a new deployment, being able to upgrade current services that are already deployed is possible with Pandora.cc. In order to achieve this, some specifications needed to be set beforehand. We chose to opt to be able to change individual implementations of nanoservices while the Pandora binary is still running, the only downside being the need to upgrade all implementations, in case there are more running instances of such service, along with a loss in state for the specific services. Still, it's possible to implement cold storage caching, which will mitigate such cases.

5. **System design optimizations** Synchronization is seen as one of the biggest pitfalls in parallel programming. We try to allow developers to reduce the number of synchronization mechanisms required by giving more access to the underlying hardware. By individually specifying each nanoservice on which CPU-core it should run, 2 specific nanoservices could be forced to not run in parallel by forcing them only to be scheduled on a specific CPU. Another aspect to this is the fact that if the configuration is done right, a specific nanoserviced could be the only one being scheduled on a CPU-core, thus allowing for dedicated threads for latency-sensitive services, which should be available as soon as possible, without waiting for something to finish before they could start.

6. **Reducing bugs** Programming languages are not equal. Some of them are associated with more bugs, such as C, C++ or Python while others are not, such as Haskell, Scala or Clojure[1].

   While C++ is naturally associated with more bugs, developing in a specific way using a well-thought library can greatly reduce the number of bugs due to reducing some of the usual bug causes. The goal is to reduce such practices, such as race-conditions and the usage of pointers, explicitly the tracking of ownership. We try to achieve this by enforcing the use of nanoservices, which force the writing of code in such a way that the data needs to be shared less around scopes, making it

easier to keep track of and by offering another toolset to go around synchronization mechanisms.

Position relative to the state of the art

Instead of trying to implement an architecture based on the textbook definition, Pandora.cc tries to blend in ideas from various technologies in order to succeed in a particular area of interest. The main reason for this lies in the current state of the art: there are well-established frameworks that implement the microservice architecture (Google C++ RPC or POCO) for C++ and various other alternatives for different languages.

In order to make the framework worthwhile, some really important advantages needed to be added for specialized use-cases so that the advantages would overcome the cost of adding or switching to a new paradigm. Taking this into consideration, we'll evaluate 3 main technologies, namely microservices, nanoservices, and the actor model, with the intent to create a new blend that will fulfill the design principles.

4.2 Microservices

Even thou the concepts used by microservices were used before by SOA, a better definition of the paradigm in 2013 led to a rise in popularity in the following years[2]. Since the technology is relatively new and can be implemented differently, we chose to refer to [3] as a baseline on how the industry adopted and used this paradigm as well as finding out where the advantages and challenges are situated in contrast to the literature.

Advantages

- **Independent deployment** allows teams to upgrade the system as they roll out new features and bug fixes [4], the main beneficiary from this being the user, which gets the new and latest as fast as possible. This enables teams to be more independent from each other, thus reducing bureaucracy inside a company and allowing for faster development overall.

- **Scalability** is often mentioned as one of the main advantages, especially over monolithic architectures [5]. Due to the distributed design, microservices allow the scaling of one specific service instead of scaling the whole system. Maintainability is achieved due to the design of the communication pipeline, each microservice implementing a stable API while having RPC (Remote Procedure Calls) over the internet as the means of sharing or requesting information.

- **Language independence** is the staple of distributed programming. The use of REST API or cross-language serialized data structures such as Google's Protocol Buffers made microservices language agnostic for the most part, enabling a wide range of developers to be able to contribute to a shared codebase.

- **Easy deployability** is not necessary a microservice advantage per se, but more a result of the DevOps practices which are used when working the technology. The rise of containers such as Docker and the wide coverage for such technologies by server providers allow the deployment of microservices into Docker containers with ease, which can afterward be deployed in the cloud. This is possible due to the general aspect of docker containers and encapsulation of resources in the container itself.

Disadvantages and challenges

- **Expensive RPC** due to the overhead of the network involved in each operation can quickly add to a large overhead, especially in complex queries that require sequenced RPC to yield the final result[3].

- **Service faults** tend to be harder to debug or to track the origin of the problem due to the distributed aspect of microservices[3]. Maybe one of the biggest disadvantages compared to the monolithic architecture, miss behavior tends to be harder to identify, especially if the error is introduced in the runtime of another service than the one that crashed. This becomes even harder if that service is written in another language.

- **Testing** a stack of services can be challenging, especially due to random events that could occur in a normal deployment, such as network errors, DB bottlenecks, or random crashes[3].

- **The complexity of distributed systems** is no easy task, access to data or data synchronization being one of the major problems[3].

Other important characteristics

In the majority of cases and implementations, microservices are accompanied by some characteristics such as being autonomous components that run on one process[6]. This is specifically important since it enables them to self-heal in case of a system failure due to limited scoping, which so the failure doesn't affect other systems and redundancy to replace the failed service.

Applications

Monzo is a digital bank(neobank) startup based in London, UK, which has a user growth of 40'000/week[7]. It was founded in 2015, and it has a transparent policy regarding its development practices. This is why we chose to showcase some of their findings, as they reflect state of the art in terms of development practices.

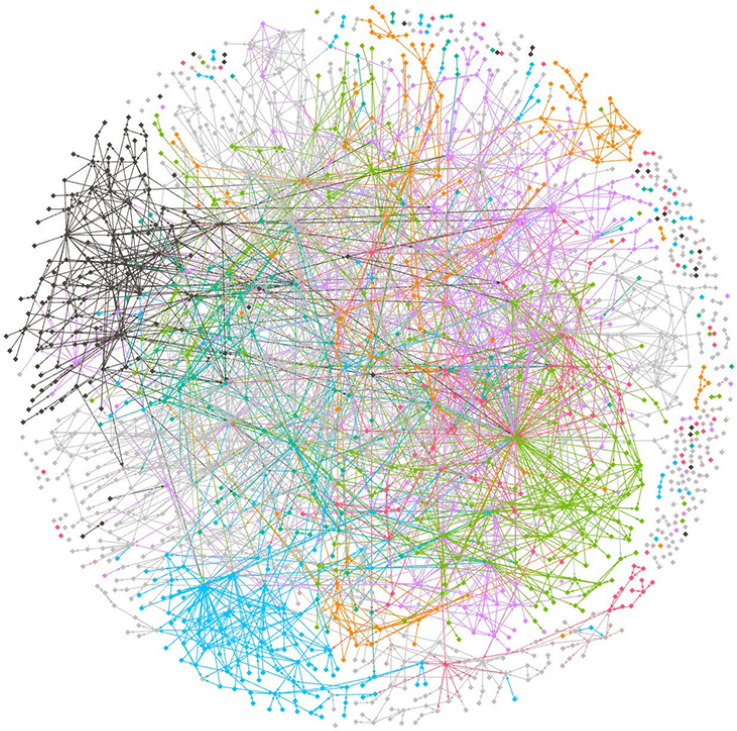

Figure 4.1: Monzo's microservices map

Figure 4.1[7], showcases the architecture of the digital bank, each node in the graph representing a service and one edge showcasing an enforced network rule between them. The nodes are colored based on the team that manages them.

The purpose of the figure is not to start a debate on whatever or not the company uses the architecture right, but to showcase that it's possible to achieve tremendous scaling compared to other architectures due to the widespread solutions to facilitate this, such as Kubernetes.

More details regarding their findings can be found in other blog posts as well, such as 'building a modern backend' which dates back in 2016, one year after the company's launch. Without going into detail, Monzo's findings are similar to the ones presented above[3].

Breakdown

The majority of the advantages and disadvantages of microservices have their root in remote calls. This enables services to be independent of each other while being harder to test and design. To allow a better deployment and failure handling, each microservice is seen as a separate entity, being independent of other components.

Takes

While microservices offer a great way to design full-scale applications, they lack in speed and usability for small applications or applications that did not meet the required necessity for such infrastructure yet. We argue that it's a big decision to be made, one that pays off in the big run, but it will surely not favor such practices to be adopted from the beginning. On the other hand, a later transition to this pattern requires a massive change in code-paradigm, which could take much time to be integrated.

The containerization of microservices is a widely used and implemented feature, but we feel it shouldn't be enforced. By allowing multiple microservices to be run in the same process, we can cut off the RPC induced latency completely, thus making the architecture a viable option for both vertical and horizontal scaling. By doing so, some advantages become harder to achieve, like easy deployability, independent deployment, and crash robustness. We try to offer a good workaround for the first two using Pandora.cc specific tools like the bundling system and the hot-swap service. These tools are explained in more detail in chapter 4.7.

4.3 Nanoservices

With the rise in popularity of microservices in 2014, people started talking about taking the paradigm a step forward. Microservices were seen as a smaller approach to SOA, while nanoservices were defined as a smaller microservice, ranging from 10 to 100 lines of code. It took 4 years until developers figured out a way to make this technology reliable. The rise of nanoservices was made possible due to the maturity of the microservices ecosystem and the parallel rise serverless, a new technology that tries to tackle the same problems. These changes didn't solve the disadvantages of nanoservices, but instead, it offered a different point of view to consider when considering to build a nanoservice.

Advantages

- **Deployability** is taken a step forward compared to microservices, nanoservices also containing all the required infrastructure necessary to be deployed. The definition of such service also handles all resources required during its lifetime. To integrate

the service in the ecosystem, a config file is used to specify different parameters of endpoints for events.

- **Reusability** is a significant advantage due to the limited scope and ease of integration. This is highly enhanced by the fact that nanoservices can be coded in any language, and they can be bundled into a Docker container. Due to this nature, a "Microservice Package Manager" could be more significant than any language-specific package manager, such as pip for python and npm for javascript.

- **Usability** is often showcased as an essential advantage. Nanoservices should do something or figure solve a problem, meaning that they are not to be used inside a complex chain and should be mostly situated at the end of the execution chain.

Disadvantages and challenges

- **"Nanoservice pattern is an anti-pattern"** is often used to describe nanoservices in relationship to microservices. Being more of a pitfall, building entire services around nanoservices could escalate quickly into unreliable code and an increase of complexity in deployment.

- **Communication overhead** is accentuated compared to microservices since nanoservices produce more remote calls due to the limited scope. This makes SLAs (service-level agreement) challenging to reason about since many interrupts can happen in the life-cycle of a request inside a nanoservice-based architecture.

Breakdown

Nanoservices occupy the role of containerized, scalable, and language-agnostic endpoints to different services, such as databases, storage units, or cache managers. Tooling is a common use-case for nanoservices, such as cost managers for cloud computing, alert systems, or data aggregators.

Takes

Nanoservices can be used in the majority of cases to take care of communicating with an endpoint or adding plugins to an application. We argue that by moving the nanoservices closer to the microservice, we can reduce the network overhead, raising the potential of such nanoservices. This enables nanoservices to be used as a computational part for the request, being a viable option to speed up specific independent tasks, taking a similar role to actors or tasks.

4.4 Actor model

Introduced by Hewitt et al, the actor model offers a scalable way to implement concurrent software using a 'share-nothing' paradigm[8]. We chose to explore this alternative due to its powerful claims and reduced use case (1), being implemented in Erlang, and various frameworks like Akka, which can be used in both Java and Scala. A new C++ implementation is offered by CAF (C++ Actor Framework) along with with paper[9].

We try to justify the case for (1) reasoning that the performance advantages aligns well with the intended design goals. We base our performance and advantages on the CAF paper while looking at the current state of the art to find systems that use the actor model, focusing on the impact of such cases.

Advantages

- **No synchronization required.** Actors do not share stare and achieve sharing data by passing messages. The lack of synchronization mechanisms enables less error-prone code due to race conditions.

- **Small memory footprint.** Actors are lightweight units of computation, each of them taking less then 0.5KBs of memory, including the associated mailbox. This enables the creation of 2^{20} actors in less than 10 seconds, having a memory footprint less than 500MB[9].

- **Fast communication for same-machine use cases.** A proper implementation of the mailbox can achieve performant results such as 20×10^6 messages being sent and processed by one actor in a $20 : 1$ scenario, in under 10 seconds.[9]

- **Fault tolerance.** Each actor can crash independently without affecting the whole system, such cases being handled by a supervisor, another actor who decides how to handle the event. [8]

Disadvantages and challenges

- **Immutability** is not necessarily a disadvantage, but it's most common in functional programming. This inherently makes developers adhere to a new programming paradigm, thus increasing the complexity of designing and writing code.

- **Dining philosophers problem** is often presented as one of the pitfalls of the actor model. The no synchronization paradigm creates a different range of problems that require different solutions. Similar to the immutability challenge, this imposes a different mindset than the general development process of parallel programming.

Breakdown

The actor model proposes a fast and reliable paradigm that best suits functional programming since it'll blend in with the fact that such languages do not use mutable variables and focus more on keeping a consistent state throughout the run of the service. While CAF offers a good API and competitive performance, we argue that the development process is to different from being considered as a viable option for an application.

Takes

The majority of the challenges that come with the actor model are related to the development process, and we argue that there are good partial-use cases to justify the case for the actor model. While applications that use only the actor-model are too demanding, using this practice for maintaining state is often the everyday practical use. For example, an online shop might choose to have one actor for each item sold in its store.

While the actor model does not fulfill the entire design principles, it offers a useful benchmark when taking into consideration speed and, in some ways, scaling.

4.5 Conclusions

All the architectures described in the section provide valuable advantages while having their disadvantages or posing difficult challenges. We acknowledge that other design patterns may solve some of the problems or provide suitable alternatives to the proposed ones, but we consider that this selection provides a right mix of options while keeping the selection base clean enough.

Considering all these, we designed Pandora.cc to offer a similar performance to the Actor Model while allowing a development style similar to the nanoservice architecture, all while having the deployment and scalability capabilities of the microservice architecture.

We argue that the Actor Model is an excellent paradigm if implemented in a functional language, but when used in an imperative language like C++, it lacks powerful tools that enable the immutability aspect of actors. This is solved by changing the actors with nanoservices.

Nanoservices can be used in a great variety of use cases due to their general aspect and easy reusability. While the communication and deployment overhead is considered as being too much when used in an application, we try to repurpose them by filling the role of an actor. This reduces the communication overhead dramatically while maintaining all the functional aspects of nanoservices.

Microservices appeared first as a loosely defined term to refer to a more granular Software Oriented Architecture. In time, it became more mature due to the support from

big tech companies, which benefited the most from the approach, allowing them to scale in a way to match the unprecedented growth. This enabled the microservice architecture to be implemented, deployed, and tested with ease due to the number of popular DevOps tools, such as Kubernetes, Docker and Jenkins. We argue that this architectural style was a success due to the abundance of tools, and such we try to allow Pandora.cc developers to take full advantage of them, by creating 'microservices' as bundles of nanoservices, like a plug-in system. This approach is made using the bundling system, a powerful tool that will be showcased in chapter 4.6.

Tradeoffs

- Since 3 different coding paradigms inspire Pandora.cc, the resulting features are not a union of the original ones. Some of them are diminished while some disadvantages are less severe.

- Since Pandora.cc is a C++ framework, microservice's language independence does not apply when developing with Pandora, compared with Google's RPC, which supports out of the box cross-language endpoints for different services. Services written in different languages can be supported, but a pandora-specific mockup service interface needs to be created.

- Nanoservices reusability is diminished since they need to be Pandora nanoservices. We still think it's easy to port C++ nanoservices wrote using other frameworks to be Pandora-compatible, but this won't come out of the box. The actor Model's fault tolerance is wholly negated due to the introduction of nanoservices instead of actors. Nanoservices alone are fault-tolerant, but when running inside the same process, a big fail can affect other nanoservices as well.

4.6 Tools

To make Pandora a competitive alternative, we created a series of tools to consolidate development practices. They are not inspired directly from any paradigm but instead developed to tackle some problems.

Bundling system. Since Pandora.cc organizes a project as a collection of nanoservices, the bundling system allows the creation of bundles, providing a level of abstraction while making the deployment and management process more straightforward. For example, one bundle could be called FileMicroservice, which would contain the FetchFile, WriteFile and DeleteFile nanoservices. The FileMicroservice can be used in the future as a single entity without worrying about future changes in the structure of the microservice, such as the inclusion of new nanoservices or changes in API.

Thread selection can be a powerful tool that if used correctly, but it can be tough to design applications this way. It allows both enforcing of restrictions to avoid race conditions as well as allocating resources for a service so it can start serving as fast as possible. To make the development more accessible, we chose to not enforce this, as it can be hard to catch potential bugs, but it allows for more control over the runtime of the system.

Nanoservice hot update. Since nanoservices are embedded in the executable, the naive way would be static link them into the binary. Instead, Pandora builds each nanoservice as a dynamic library that can load at runtime, allowing the libraries to be unloaded and reloaded, thus allowing updating the DLLs while the binary is running.

Chapter 5

Implementing the framework

In this chapter, we present the inner workings of the framework, along with various implemented optimizations. The focus centers on the multithreading aspect of the framework since the cloud aspect of it are very similar to current RCP-based frameworks, like Google RPC.

5.1 Mailbox

To facilitate a large throughput of messages from a service to another, the mailbox is an essential part of the framework, a case similar to the CPP actor framework[9]. We chose to make threads more independent, giving them a similar role as processes with a single core that use an event-queue to consume requests. Each thread is associated with a Multi Producer Single Consumer Queue (MPSCQ) from which it will retrieve the tasks which need to execute. For this purpose, we chose Dmitry Vyukov's non-intrusive lock-free unbound MPSC queue [10] design with Matt Stump's implementation, publicly available on GitHub[11]. The decision to give each thread its MPSCQ has it's own advantages and disadvantages compared to one global Multi Producer Multi Consumer Queue (MPMCQ).

Advantages

- Each thread can start consuming its requests without checking if it's able to execute run, due to restrictions imposed by the thread specification tool.
- It allows for better performance when sending over messages.

Disadvantages

- Each request receives it's designated MPSC at the time of its creation, which can create congestion on specific threads while others wait for new requests.

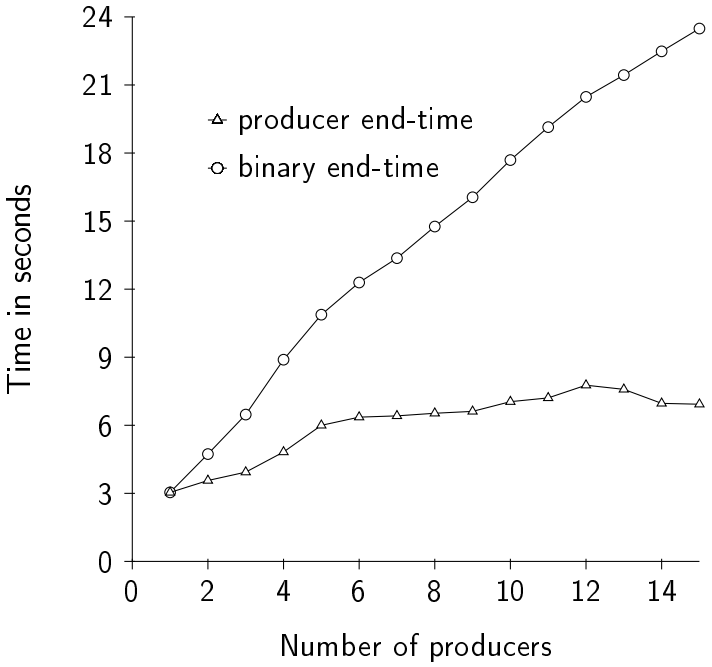

Figure 5.1: MPSCQ Benchmark

Figure 5.1 shows maximum time time to create 10^7 requests by all producers (producer end-time) and the time required by the consumer to finish consuming all the produced requests (producer end-time). The benchmark is done using a specialized instantiation of the MPSCQ to showcase its performance without adding any other components to it. It can be seen that it takes 1 producer 3.04 seconds to create all the requests, 4 producers 4.8 seconds and 10 producers 7 seconds. With further increase of the number of producers, the time stabilises at around 7 seconds. It can be seen that the consumer follows a stable trend, consuming 3×10^6 queries/second while the producers are working and 7×10^6 while there are no new requests inserted.

5.2 Futures and Promises

Because the framework uses nanoservices as agents to carry out computations, we chose to use futures and promises for the first run of the implementation as a means to communicate and retrieve data. Callbacks seem like a popular choice, especially in programming languages that offer concurrency but not parallelism, such as javascript, but we felt that futures offer a more straightforward practice when designing an application.

Design problems with Callbacks

Callbacks seem like a good pattern since it'll allow a suitable 'yield' mechanism, in the sense that the thread can process other requests while the futures are not deferred, but

it raises some hard design choices. In case a service makes multiple service calls, each of them with an associated callback, it's not clear where those callbacks should execute and which service is responsible for handling those callbacks.

While the most straightforward answer would be that they should be handled by the thread that spawned them and with the associated service-entity for that thread, some race conditions may appear. If the callback is associated with the service-entity for that specific thread, how does this affect the internal state of that service, namely, a service entry should only carry out one' request' operation before it can start processing another so that the internal state would remain uncorrupted. This difficulty can be solved using actors for carrying out the computation, since they can use callbacks without any of the disadvantages mentioned above, but they lack that state immutability aspect.

To fix this problem as well, we can choose a heavy architectural approach, showcased in the theoretical code below. For example, we'll like to build a microservice which will recommend ads for users while caching some information.

```
void GetAd(Request, Response) {
  // do some extra work
  auto cache_hit = MicroserviceLocalState.GlobalCache.Got(Request);
  if (cache_hit) { Reponse = cache_hit.Get(); }
  else { GetAdActor(Request, Response); }
}
void GetAdActor(Request, Response) {
  this->received_ads = vector<AdResponse>(10);
  for (int i = 0; i < 10; i += 1) // request 10 ads, keep the best one
    SuggestAd(Request).Callback([=](Response){ // keep track of responses
      this->received_ads[i] = Response; });
  WakeAfter(100ms).Callback([=]() {
    // set the actor response to be the 'best' ad received
    Response = BestResponseOf(this->received_ads);
    // update the cache
    UpdateGlobalCache({request, response}, {});
  });
}
void UpdateGlobalCache(Request, Response) {
  MicroserviceLocalState.GlobalCache.Set(Request.request, Request.best_ad);
}
```

Figure 5.2: Callback-based request handler prototype

In figure 5.2 we created 2 nanoservices, `GetAd` and `UpdateGlobalCache`, and 1 actor, `GetAdActor`. Since GetInformation forwards the response from another call, we can safely exit the `GetAd` nanoservice so it can get to process other requests. `GetAdActor`

can have multiple instances of the same state object since it's an actor, and it's reusable and lightweight by design, enabling the existence of multiple such actors at the same time to handle multiple wait-callbacks in parallel. This is enforced by the lifespan of the `std::vector received_ads`. Since the vector is stack-allocated, it'll destroy when the function exits, and if we embed it in the state of the state object, it'll clash with future calls in the case of nanoservices.

While it is possible to implement callbacks nicely with actors or as stateless functions that produce side-effects in nanoservices, we argue that the added complexity does not make this option compelling at the moment, even thou it gives more power to developers.

Choosing the right Future

Before committing to futures as a means of data sharing, we needed to make sure that this is a viable option, performance-wise. To test these, we created a benchmark environment using the already-implemented mailbox to facilitate the communication between 2 threads, the first one sending a request which will be enqueued in the MPSCQ and retrieved using a promise. This approach yielded 0.7×10^6 (a→b a→b a→b) calls/second, drastically affecting the base performance of the MPSCQ, making the standard implementation of futures an infeasible option.

To compensate this, we chose to create a specialized version of the future/promise pair, focusing on raw performance suited for the current use case, allowing the integration of the future/promise inside a pair of request and response, minimizing cache misses. This approach yields 1.8×10^6 calls/second, being 2.5 faster than a stantard approach using `std::future`

For the future implementation, we used a heap-allocated `BaseFuture` as the central part of the future-promise pair, the actual futures, and promises being similar to a `BaseFutureView` with different read or write access, for promises and futures accordingly.

```cpp
struct BaseFuture {
  std::atomic<bool> is_set;
  std::atomic<int>  counter;
  T                 value;

  void decrement_counter() {
      if (counter.fetch_sub(1) == 1) {
          delete this;
      }
  }
};
```

Figure 5.3: C++ Future implementation

In figure 5.3, we showcase our implementation of the `BaseFuture`. The implementation is similar to a shared pointer with a build-in reference counter and a bool flag to simulate a `std::optional` field. The code would be further developed to reduce heap-allocations and overhead.

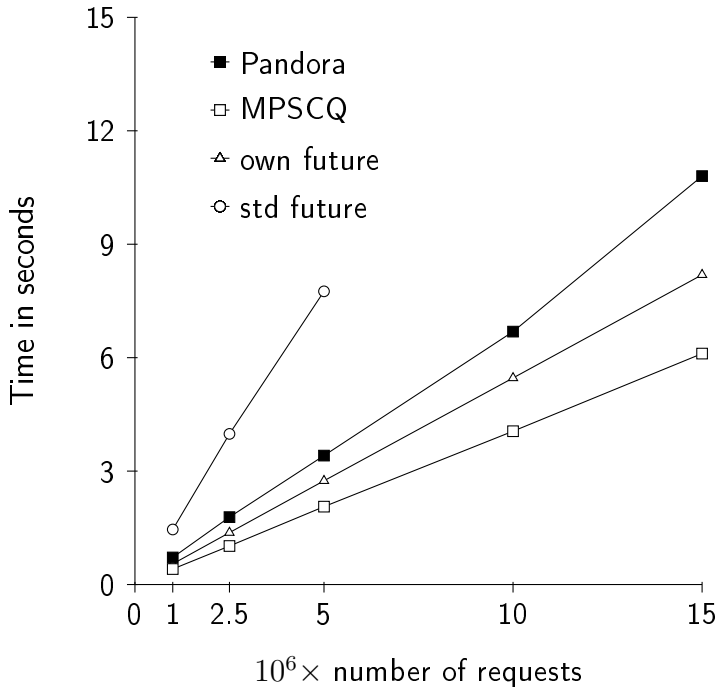

Figure 5.4: Future-Promise Benchmark

Figure 5.4 showcases time improvements over the std::promise as well as a throttled version of the first MPSCQ benchmark and a Pandora.cc running example. The benchmark involves a general 2 service application, one of them continually polling the other for information. In the MPSCQ throttled benchmark, the producer can send another request

only when the consumer has already processed the request to replicate an exchange of information. The Pandora.cc example is based on 2 simple nanoservices, Multiplication and Addition. The first one wants to compute $X \times 1$ by making X Addition calls to add 1 to the final result, where X is the number of requests on the X-axis.

5.3 Designing Nanoservices

Even thou we started with a set goal, to reduce glue code, we argue that it's better not to enforce strict standards from the beginning. With these in mind, we present a current implementation of a nanoservice, a rather explicit one. Our end-goal and view for this part is going to be explained in chapter 6.1, Future work.

```
struct Addition {
  struct Request {
    int a, b;
  };
  struct Response {
    int aplusb;
  };

  // state accesible by all Addition workers
  struct SharedState {
    atomic<int> cumulative_num_calls;
  };

  int num_calls = 0; // object specific state
  Addition(int thread_number);
  // public API which is used by other nanoservices
  static Future<Response> async(const Request&);
  static void blocking(const Request&, Response*);

 private:
  void handle_request(const Request&, Response*);
  static SharedState shared;
};
RegisterService(Addition);
```

Figure 5.5: Nanoserivce demo code

In figure 5.5, we present a working `Addition` nanoservice. Its purpose is to take 2 numbers and return the sum of those numbers. The code defines the Addition class with the subclasses `Request` and `Response` accordingly, one constructor, 2 static public API, and one internal handler. Besides all external API's there is one `SharedState` class

along with a `num_calls` field, enabling the nanoservice to keep a thread-local state. The `SharedState` static object allows the individual thread-specific instances to share information if needed. Since this object is shared, the requirement for synchronization mechanisms arrises to implement such cases correctly. The last piece of code is the `RegisterService` macro, which enables Pandora.cc to be aware of the presence of the service so it can run the specified services throw the use of the bundling system.

Even thou we do not enforce these names in the current iteration of the framework, we'll like to stick by them to check if the current API is sufficient, allowing future changes to be made more quickly if necessary. We opted for this way of writing services since it's somehow similar to other RPC frameworks, users being required to write only the `handle_request` method and the `Request` and `Response` messages, thus eliminating much necessary glue-code.

Internally, when performing an async call, the framework look-ups all the possible threads which can run the desired service, and it'll select one, adding the request in that's thread MPSCQ. In the case of blocking calls, the framework checks if the service is configured to execute on this specific thread and calls it as a normal function if that's the case. In case of a failure, it'll throw an exception to signal that. Due to this, it becomes especially tricky to troubleshoot cases where only a subset of threads accept running blocking calls for a specif service. The option to execute such blocking calls allows users to achieve maximum efficiency without forcing duplicated code.

5.4 In depth example overview

In this section, we want to explain the interactions between various components further analyzing a particular example. The example focuses on a service that recommends internet ads for various sites. Without losing generality, we can assume that we have access to a black-box function, which takes as argument some browser cookies and returns an ad. It's also important to note that we have access to an estimation function, which estimates the likelihood that a user is going to interact with that particular add.

A trivial implementation of this would respond with only one ad, which can yield suboptimal performance. To gain better user interactions, one can generate X ads in parallel and return the best ad based on the estimation function.

In some cases, the ads can take a while to generate. Instead of waiting for all X ads, the leading service can be configured to wait at most $100ms$ to ensure that the user receives an ad in a decent time, thus not violating the service's SLAs.

A simple Pandora implementation contains a `GetAd` nanoservice, which forwards the request to X `ProduceAd` nanoservices, waits at most $100ms$ and returns the best ad. The `ProduceAd` nanoservice calls the embedded black-box and returns the best ad that

is suggested.

Testing the actual performance of the black-box regarding user interactions can be hard to achieve since its hard to deploy different types of black boxes inside a simple service. Using Pandora, each binary could be specified to use a different type of implementation for the `ProduceAd` service, which encapsulates the specified black box.

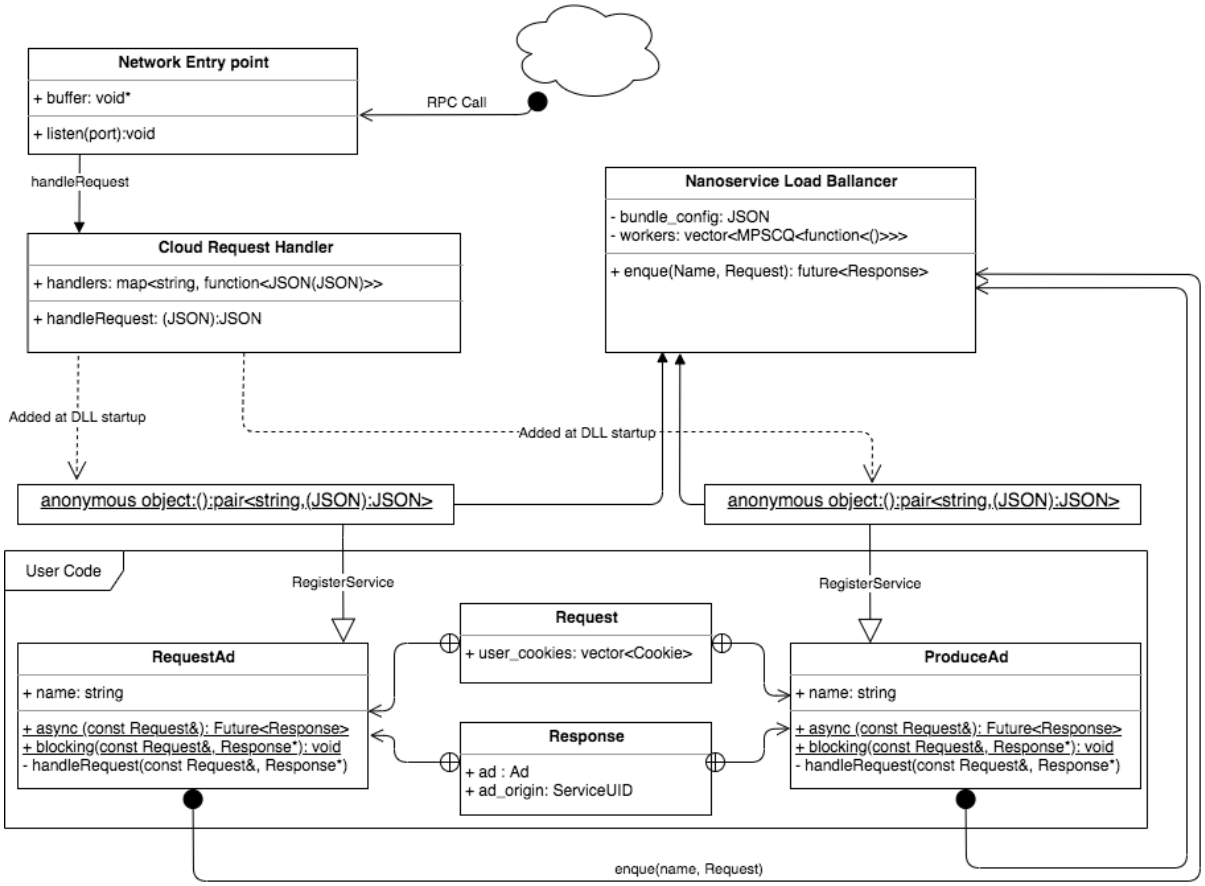

Figure 5.6: Ad service complete UML diagram

Figure 5.6 described the UML diagram of the entire Pandora binary. It's important to note the usage of the `RegisterService` macro. It's run at the load of the nanoservice and it embeds a request handler for that specific service in the `CloudRequestHandler`, so it'll be able to handle RPC requests without having to hard-code all handler functions in a big class. Due to this aspect, it's possible to just dinamicly load services directly in the binary, thus reducing the need to recompile the whole project. The relasionship between the nanoservices and the `LoadBallencer` is utilised in case of local RPC calls, which don't go over the network.

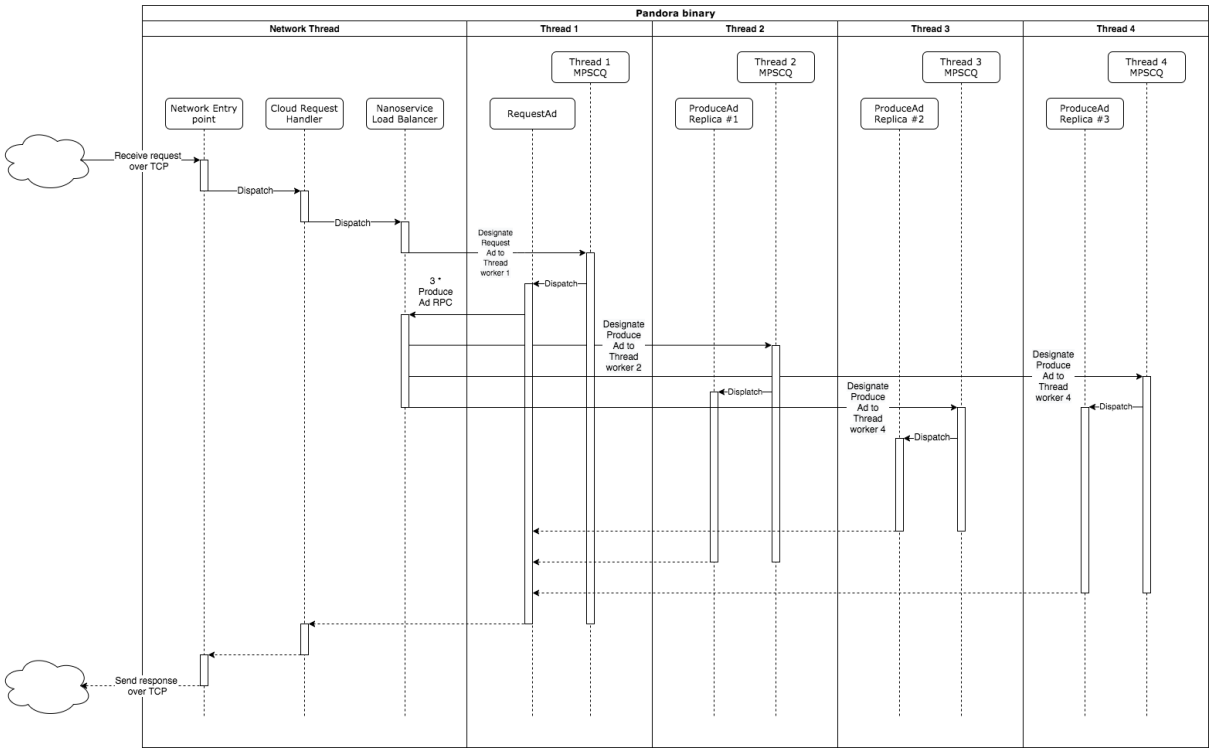

Figure 5.7: Life of a request sequence diagram

Figure 5.7 showcases a sequence diagram while specifying threads, to furter emphasise the importance of correctly configuring the system. It's important to note that with the current implementation the `RequestAd` nanoservice must wait for all 3 `ProduceAd` calls to end or to pass 100ms in order to serve another request.

5.5 Development process

As described in chapter 3.1, Motivation, the idea for the framework was a long-planned project. In the beginning, there was the research, figuring out which technologies are worth taking into consideration and what other developers are using. To facilitate the search for knowledge, GitHub C++ repositories offered much insight into current practices and trends. The research, shown in chapter 4, paved a long road to success, but it also paved a planned direction and some important end-results as well as some milestones and similar frameworks to keep track of the possibilities.

Before we started the actual development, the hardest part needed to be dealt with: finding a suitable name. With some inspiration, we settled on Pandora.cc, the first part referencing the classic Greek myth, showcasing Pandora, the first human created on the instructions of Zeus, being gifted by all the gods.

While all the research provided a compelling direction, we tried to stick to a strict and organized development plan from the beginning, using a streamlined, feature-based

approach, which is present in most agile paradigms. The use of the software provided by trello.com and the feature-based approach encouraged an organic growth of the project, offering only essential features while keeping the development process on the road to the end-goal. This pattern is also visible in the structure of this chapter; the mailbox is the first item in the development process, followed by an efficient way to facilitate communication. User-API and the bundle system were refactored numerous times due to code becoming mature. In the end, we introduce a series of other optimizations to facilitate a higher throughput.

Early introduction of examples into the codebase was one of the most critical assets in maintaining a coherent codebase throwout the development period. These examples constitute experiments in the early stages of development, single code endpoints that showcase a use case for a specific library or a header file. With a more refined user-API, the introduction of a hello-world service promoted a broader overview of the library.

5.6 Testing

The early introduction of tests was one of the main targets to look after while developing the framework. While tests may take a while to configure and write, they serve multiple purposes throwout the development process. Ensuring stability is probably one of the first advantages that developers advocate when choosing to test the code in the form of maintaining the correctness of the code.

While testing comes in a variety of forms and tools, we chose to adhere to a nonconformist way to achieve testing, due to the design and requirements of the project. The fact that one developer did all the development allowed much easier tooling and automatization, cutting out the necessity of stable testing and running environment in the form of a dedicated cloud server. Given this, the execution of all tests before any git-push command is performed using a git-hook. The testing system consists of a python script which executes the experiments and the examples and performs various checks to determine if they satisfy. While this system can be described as simplistic, but we argue that it's a right balance between secure development, maintainability, and initial overhead. In more standard terms, the testing is done using continuous integration, being facilitated by integration tests and performance benchmarks.

Integration tests are primarily used to test big interconnected services or applications. In this particular case, they can be successfully implemented to cover the whole majority of possible code-changes due to the vast logical dependencies of the codebase. This practice takes advantage of the presence of the experiments and examples, covering the entire codebase. To conclude, if the run is accepted or not, the binaries need to produce an expected output that tests their correctness. All the binaries execute 3 times, the first

one being the release version, compiled with `-std=c++17 -O3` while the second one uses `-O1 -fsanitize=address` to check for undefined behavior. The third test involves executing the binaries with `valgrind`, yielding a correct execution if there aren't any errors.

Performance benchmarks are not standard, but this fact changes as the application becomes more focused on performance. Subtle changes can affect the overall performance of the application in unsuspected ways, like a change in the task distribution algorithm. Another advantage is that many performance benchmarks offer a history of the performance, making subtle decrements in performance less of a concern to keep track of.

In conclusion, this testing practice offers many advantages to single-developers, being easy to configure and develop while allowing for automated tests so bad-changes do not enter the version control system. The addition of the endpoints (experiments and examples) allows easy testing while developing and a convenient way to achieve continuous integration.

Chapter 6

Conclusion

Even thou state of the art offers a consolidated option when talking about large-scale applications, throw the use of famous microservice frameworks, we showed that it's possible to introduce a new approach that might be useful in some cases.

While developers try to stay away from C++ and gravitate towards new languages such as Python, JS and Go, Pandora.cc offers outstanding performance combined with an intuitive and modern API. Being a blend of multiple approaches, we created something which can be described as "the best of both world" for some users. The reduced upfront cost of using the framework and the potential gains, even for multithreaded activities, might make Pandora a competitive option for this use-case alone.

The paper itself does a thorough job of educating the reader on various practices as well as showcasing the internals of the framework along with the great results that were achieved with this early build of the project. Being able to perform 1.5×10^6 operations per second, the overhead of the framework can be considered negligible even in latency-sensitive workload, without enforcing various restrictions to accommodate them.

Design principles are often considered as implicit when presenting a new product. The presence of such practices offers better insight into the product itself and its role in the ecosystem. The process of putting down the motives driving it and some clear end-goals it's a critical and often neglected step when creating a new product. In the end, both the creators and the end-users take advantage of this being publicly available; creators have a stable goal set in mind, being able to gather feedback along the way from external sources while being able to know that the product is going in the right direction, even if the road itself is not that straight forward. End-users can have an easier time choosing or understanding the product since they can understand the underlying principles on which it was built, reasoning if those coincide with their own.

Development process and Testing are often absent from such papers, but we hope to spark new ideas as well as offer a different approach to standard practices by including them. All of these things go hand-in-hand with the overall idea of framework, to analyze

the current state of the art and break it down into smaller pieces, so it'll be convenient to create something new, to better fit the individual requirements, without having the pay for extra features.

6.1 Future Work

It's no secret that the framework is not complete, or that it could use some more refinement. We believe that the current state offers a good starting point as well as a great proof of concept to showcase the potential gain in speed if nothing else. Still, we'll like to share our thoughts on the direction of the project, along with several concrete milestones to quantify the required work and areas of interest.

- **Binary interchange format for data-structures.** While coding with structs in C++ and using an automated JSON encoder/decoder can be seen as an easy option, the advantage and maturity of such binary protocols can't be overlooked. Maybe the most common way to implement such practices, Protocol Buffers by Google, is trendy, a lot of good and faster alternatives started to become mainstream, such as Cap'n Proto.

- **Enforced topologies.** While being able to take advantage of all the microservices at once is a good idea for a small application, as the requirements grow, it's common that all the resource that will be used by a microservice should be white-listed manually. This becomes a necessity if A-B testing or any form of experimental services are introduced.

- **Code generation.** A controversial topic nonetheless, code generation comes with both advantages and disadvantages. While it's not of most importance, it's good to keep in mind that such practices will come in handy when developing services.

- **Optimisations.** While the framework is fast, there is always room to improve. Careful resource management and allocations can yield some incredible performance improvements. Implementation changes and platform-specific optimisations such as `SSE` or `AVX` can further optimise the runtime.

Bibliography

[1] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality, 2019. arXiv:1901.10220.

[2] Sander Klock, Jan Martijn E. M. van der Werf, Jan Pieter Guelen, and Slinger Jansen. Workload-based clustering of coherent feature sets in microservice architectures. *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 11–20, 2017.

[3] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. 09 2018. Conference: VI Workshop on Software Visualization, Evolution and Maintenance, At São Carlos - Brazil.

[4] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[5] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications*, pages 44–51. IEEE, 11 2016.

[6] Sara Hassan, Nour Ali, and Rami Bahsoon. Microservice ambients: An architectural meta-modelling approach for microservice granularity. 04 2017. IEEE ICSA, pages 1–10, doi: 10.1109/ICSA.2017.32.

[7] Jack Kleeman. We built network isolation for 1,500 services to make monzo more secure, 11 2019. Available at https://monzo.com/blog/we-built-network-isolation-for-1-500-services.

[8] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[9] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming,*

and Applications (SPLASH '13), Workshop AGERE!, pages 87–96, New York, NY, USA, Oct. 2013. ACM.

[10] Dmitry Vyukov. Non-intrusive mpsc node-based queue. Available at http://www.1024cores.net/home/lock-free-algorithms/queues/non-intrusive-mpsc-node-based-queue.

[11] Matt Stump. A public domain lock free queues implemented in c++11. https://github.com/mstump/queues/blob/master/include/mpsc-queue.hpp.