UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CLENIMAR FILEMON SOUZA

RESOURCE-EFFICIENT MANAGEMENT OF STATEFUL CONFIDENTIAL APPLICATIONS

CAMPINA GRANDE

2020

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Resource-efficient management of stateful confidential applications

Clenimar Filemon Souza

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sistemas Distribuídos e Computação em Nuvem

Andrey Elísio Monteiro Brito
(Orientador)

Campina Grande, Paraíba, Brasil

S729r Souza, Clenimar Filemon.

Resource-efficient managemente of stateful confidential applications / Clenimar Filemon Souza. - Campina Grande, 2020.

61f. : il. Color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2020.

"Orientação: Prof. Dr. Andrey Elísio Monteiro Brito".

Referências.

1. Computação na Nuvem. 2. Computação Confidencial. 3. Provisionamento de Recursos Intel SGX. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU 004.056.53(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS CRB-15/93

RESOURCE-EFFICIENT MANAGEMENT OF STATEFUL CONFIDENTIAL APPLICATIONS

CLENIMAR FILEMON SOUZA

DISSERTAÇÃO APROVADA EM 27/10/2020

ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG
Orientador(a)

THIAGO EMMANUEL PEREIRA DA CUNHA SILVA, , UFCG
Examinador(a)

CHRISTOF FETZER, Dr., TU Dresden
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O alto nível de poder computacional oferecido por provedores de nuvem pública, combinado com a flexibilidade, eficiência e custo reduzido, fazem desta um ambiente atrativo para a implantação de serviços e aplicações. Contudo, num ambiente altamente dinâmico e sem controle sobre a localidade precisa de dados e código, ou quem pode acessá-los, surgem preocupações a respeito de confidencialidade e integridade. De fato, a maioria das preocupações relacionadas à adoção da nuvem pública é relacionada a segurança. Embora algumas aplicações e serviços possam tolerar estas preocupações em nome da redução de custo, da maior flexibilidade e simplicidade de gerenciamento, outras aplicações têm requisitos tão rigorosos de confidencialidade, integridade e isolamento que a maior parte dos recursos ofertados por provedores de nuvem pública é simplesmente inadequados.

Estas aplicações, a exemplo de *internet banking* e finanças, sistemas de saúde, de cidade inteligente (*smart grids*) ou serviços de documentos confidenciais, podem utilizar de tecnologias assistidas por hardware, como Ambientes de Execução Confiável, ou TEEs (do inglês, *Trusted Execution Environments*), para prover garantias de confidencialidade e integridade, mesmo em infraestruturas não-confiáveis. Os principais provedores de nuvem pública têm começado a oferecer instâncias com suporte a TEEs. Essas tecnologias, no entanto, comumente dependem de recursos de hardware especializados e escassos, o que se traduz em custos elevados e desempenho subótimo, especialmente para atender a uma larga escala. Este trabalho define um conjunto de requisitos de aplicação para confidencialidade, integridade e isolamento, e apresenta uma abordagem que visa à otimização do uso de recursos através do gerenciamento dinâmico de instâncias de aplicação em infraestruturas de nuvem confidencial. A abordagem é implantada em infraestruturas gerenciadas por Kubernetes, o gerenciador de contêineres padrão da indústria, e avaliada no contexto de uma aplicação de processamento de dados confidenciais providas por medidores de energia inteligentes (*smart meters*).

Abstract

The high level of computing power offered by cloud providers, combined with the flexibility, efficiency and reduced costs make the public cloud an attractive environment for deploying services and applications. However, this highly dynamical environment and the uncertainty about the actual location of the application and data, or who has access to it, raise questions about confidentiality and integrity of running applications and services in such environments. In fact, most of the concerns that prevent an even broader adoption of public cloud providers are related to security. Although some applications can overlook such concerns in the name of the cost reduction and almost infinite resources, some other applications have such a strict set of requirements with respect to confidentiality, integrity and data isolation that most public cloud offerings are simply not suitable.

These applications, such as internet banking and finance, healthcare systems, smart grid or confidential document services, can rely on hardware-assisted technology, such as Trusted Execution Environments (TEEs), to provide confidentiality and integrity guarantees, even in untrusted infrastructures. The major public cloud providers have also started to offer TEE-enabled instances. However, these technologies usually rely on scarce hardware resources, that often translate to higher costs and subpar performance, especially when deploying for large scales. This work defines a set of application requirements w.r.t. confidentiality, integrity and data isolation, and presents a resource-efficient approach to dynamically manage large sets of application instances in confidential cloud infrastructures. This approach is deployed to infrastructures managed by Kubernetes, the industry-standard container orchestrator, and evaluated in the context of an application that manages sensitive smart meter data.

Contents

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

The rise of public cloud providers is certainly one of the key factors that allowed the World Wide Web to transition to its current, almost ubiquitous state. The increasingly cheaper costs associated with the readily available computing power, more adaptable to the users needs, have long turned public cloud computing into the *de facto modus operandi* of the industry. In fact, a large amount of the internet is running on public cloud providers, such as Amazon Web Services, Microsoft Azure and Google Cloud Platform. Running on the cloud allows applications and services to be hosted at multiple different sites, providing not only high availability and fault tolerance, but also reducing the average response times of many people, contributing to an overall better user experience. In most cases, it is also more cost effective, as the costs of maintaining an on-premises computing infrastructure (*e.g.*, internet connection, electricity, cooling, operators, maintenance) tend to be far more expensive than paying a public cloud provider bill at the end of the month.

However, there are also many concerns, most of them directly related to security and/or privacy. Huge data centers and server farms help reduce costs and deliver availability and virtually unlimited computational power, but also make it nearly impossible to track the physical location of the applications and data being processed, or who has access to it. In fact, recent reports[1] indicate security as the most relevant challenge when it comes to adoption of the cloud (81%), followed by managing cloud spend (79%).

Although many applications and services can overlook such concerns and choose the practicality and lower costs of public cloud providers, there are classes of application and services whose security and privacy requirements are not compatible with the standard public cloud offering. Such require-

[1]Flexera 2020 State of the Cloud Report: https://info.flexera.com/SLO-CM-REPORT-State-of-the-Cloud-2020

ments are usually associated with private and/or sensitive data being processed. Data that must not be seen, leaked or modified by unauthorized parties. Another common issue is related to protection of intellectual property, where the client wants to protect the piece of software itself, be it the business logic, an algorithm, a machine learning model. Internet banking and other financial services, health services (patient records and exams), social security, confidential document management, smart grid applications are examples that do require integrity and confidentiality and, in some cases, strong context isolation during runtime.

Public cloud providers usually offer some sort of security or privacy-aware features and services. More recently, with the term confidential computing gaining some traction, some providers have started to offer support to Trusted Execution Environments (TEEs)[2]. TEEs provide hardware-assisted confidentiality and integrity guarantees to the application execution. It is usually possible to attest the piece of software running inside protected regions of the memory to ensure it was not modified. Neither the operating system or a systems administrator, even with physical access to the machine, are able to read the contents of these regions. Since TEEs do require specialized hardware to establish trust and to allow the secure execution, through private memory regions and additional feature set, most chip manufacturers have hardware technologies that implement the TEE standard. Intel, for example, offers Security Guard eXtensions, or SGX [1; 13]; AMD offers Secure Encrypted Virtualization, or SEV [10]; and ARM offers TrustZone [21].

However, it is important to note that TEE technologies, such as the three mentioned before, have their downsides. They have inherently different security guarantees, which are based on the actual implementation of the technology. But regardless of the security guarantees and the feature set, these technologies impose some overhead, notably in performance [20], since the code runs in a protected, often limited, memory space, which also requires special setup or even attestation with external services. Another issue is the development overhead, since these technologies often limit the scope of the application (e.g., limited system calls) or even require that the application be modified, or even rewritten. Intel SGX, for example, provides an SDK (Software Development Kit) that only supports C/C++ applications.

Another limitation is the hardware requirements, since most features are dependent on a specific feature set, often included in the chip. When the hardware is being offered by a cloud provider, the specialized hardware and the rather scarce amount of protected resources, as it is in most TEE technologies, translate directly into higher operational costs.

[2]https://searchcloudcomputing.techtarget.com/feature/How-public-cloud-vendors-tackle-confidential-computing

Therefore, it is a challenge to allow applications that leverage TEEs to ensure their strict set of confidentiality, integrity and isolation requirements to be deployed to third-party confidential cloud infrastructures. Depending on a specialized, scarce resources such as protected memory might be prohibitively expensive and slow, especially when running a large set of applications. Another challenge is to allow these complex architectures to benefit from well-established and ubiquitous infrastructure management tools.

1.1 Problem

The following set of requirements defines the class of applications that this work considers, and is shared among all use cases aforementioned, such as internet banking, financial applications, health care services, smart cities applications.

1. The application must be available at all times.
2. The application requires confidentiality and integrity guarantees at runtime, even when data is in use, as provided by a TEE technology.
3. The application runs multiple instances in separate processes, for better context isolation.
4. Each application instance serves only one client at a time, for the period that the client is active (e.g., within a session).
5. The client data is encrypted at rest.
6. All communications between the application and clients are end-to-end encrypted.
7. The application must be usable in an interactive form, with no noticeable interaction overhead.

This set of requirements gives a stateful quality to the application, since it holds a particular, unique state which belongs to the client being currently served. Moreover, the application is supposed to keep serving the same client until the end of the active period, which is henceforth denoted as a *Session*. Additionally, the isolation requirement imposes restrictions to the deployment process, since clients cannot be served by the same process.

Deploying a large set of stateful, isolated application instances within TEEs, where each application instance has a particular confidential state (*e.g.*, the data of the user being served) raises a number of challenges. How to distribute such instances across a large set of nodes? How to perform the correct routing of the messages if the communication uses end-to-end encryption? How to overcome the quality-of-service impact of having complex applications running inside TEEs, which are inherently slower than native, unprotected applications? How to monitor the state of this set of applications? How to optimize resource consumption in order to lower the operational cost of keeping this service running? How to enable a reasonable auto-scaling mechanism for this set of applications?

Each one of the challenges listed above is both non-trivial and relevant.

1.2 Objective and research questions

This work is focused on a subset of the challenges previously posed. We focus on resource-efficiency, because it translates into cheaper deployments, especially on a public cloud environment, where cost is usually one of the main concerns. We try to achieve it without modifying the original application. We also ensure that the proposed solution would not violate any of the application requirements described in Section 1.1. For instance, the communication between client and server must remain encrypted, and the client should not notice any difference in user experience at all.

More specifically, this work evaluates the *Standby Servers approach* (Chapter 4) for dynamically provisioning a large set of isolated, stateful instances of an application that runs inside of TEEs. The research questions are defined as follows.

Q1. Can this approach be implemented with off-the-shelf cloud orchestration tools (such as Kubernetes), minimizing the complexity of managing the infrastructure?

Q2. How to correctly and efficiently route messages that are end-to-end encrypted in a stateful setting?

Q3. What are the expected performance benefits when comparing such an approach with the trivial approach of having all instances running simultaneously?

1.3 Contributions

The main contributions of this work are:

- An architecture that allows stateful, isolated applications that run in TEEs to be deployed in confidential cloud infrastructures[3].

- An approach for routing end-to-end encrypted messages in a stateful application setting.

[3]This architecture is a joint work between UFCG and Scontain U.G., see Section 6.3.

- An evaluation of the proposed architecture with respect to its viability, performance, quality-of-service and resource consumption.
- Insights on how to minimize the latency overhead imposed by TEEs and scarce specialized resources (e.g., protected memory).

This work assumes the threat model of a TEE implementation based on Intel SGX, where the attacker has privileged access to the machine and can control the whole software stack running on it, including the operating system. The attacker leverages this privileges to extract sensitive data and cryptographic keys, or to modify the source code of applications. This way providers and operators do not need to be trusted. We only trust the application code, the SGX support software and the SGX hardware, which are assumed to be bug-free. We also assume that side-channel attacks are outside of the scope of SGX. It is up to the software developer to mitigate such risks (either by implementing risk-reducing mechanisms themselves or by leveraging existing tools, such as shielded execution environments).

1.4 Document structure

The rest of this document is organized as follows:

Chapter 2. Background. This chapter presents a set of concepts that are relevant to better understand the problem, the motivational use case and the proposed solution. It also overviews existing solutions to similar problems in the literature, and compares such solutions to the approach presented by this work.

Chapter 3. Use case: smart grids. This chapter introduces a use case to illustrate the problem. This example will be used throughout the work, and will serve as a base to illustrate the proposed solution and its evaluation.

Chapter 4. Dynamic provisioning of stateful applications. This chapter presents an approach for deploying stateful, isolated application instances that run inside of TEEs in a more efficient way. Two other alternative approaches are presented for comparison.

Chapter 5. Evaluation. This chapter presents the results of the analysis of the proposed architecture in terms of performance and resource consumption.

Chapter 6. Final remarks. This chapter presents final remarks and future work.

Chapter 2

Background

2.1 Trusted Execution Environments with Intel SGX

Trusted Execution Environments, or TEEs, provide confidentiality and integrity guarantees through special features, such as private regions of the memory, special CPU instructions or hardware-based attestation, which protect the application execution. There are several TEE technologies, and the most popular are Intel SGX, ARM TrustZone and AMD SEV.

Intel SGX [1; 13], or Secure Guard eXtensions, leverages a special set of CPU instructions to create private regions of the memory, called enclaves, and protect the execution of sensitive applications in untrusted environments. The access control to these regions is enforced by the processor. An attacker with administrative privileges or physical access to the machine would not be able to see the contents of an enclave, which guarantees the confidentiality of the computation being executed. In addition, it is possible to attest the contents of an enclave, ensuring the integrity of what is running (*i.e.*, it was not modified by anyone). Confidentiality and integrity are the key concepts of Intel SGX technology, which is available in most off-the-shelf Intel processors (*e.g.*, it is available in Intel Core family since its 5th generation) and some server processors as well (*e.g.*, Intel Xeon family).

The private region of the memory, which is called Enclave Page Cache, or EPC, and is part of the Processor Reserved Memory, has a limited size, usually 128 MB (a few new processors have 256 MB). Some internal metadata structures are also kept inside the EPC, leaving a smaller amount of protected memory available to actual application enclaves (*e.g.*, approximately 93 MB in a 128 MB EPC machine). All the enclave pages are stored in the EPC. If the enclave is bigger than the available EPC, old pages are evicted from the EPC to the main memory. This process usually imposes a significant overhead to the application performance, since the page contents have to be encrypted before leaving the enclave (and decrypted when the pages are loaded back from main memory).

Intel SGX applications always have an insecure (*i.e.*, not protected by the TEE) portion, which is responsible for executing system calls, since enclaves are isolated from the operating system. The insecure portion also starts enclaves, and all information exchanged between the application enclave(s) and its insecure portion is done through special calls for entering and leaving the enclave (*ecalls* and *ocalls*). SGX applications have to ensure that no sensitive information leaves the enclave in a plain text form.

Intel SGX also supports remote attestation, which allows applications to verify the identity of the enclaves and the platform on which they are running (*e.g.*, whether the machine has SGX and an up-to-date firmware, or if CPU features such as hyperthreading are enabled). The enclave identity, or MRENCLAVE, uniquely identifies an enclave through the hashing (SHA256) of its contents, including the associated code and security flags. The signer of the enclave is also uniquely identified by the signer identity, or MRSIGNER. The attestation process has two steps. First, a special enclave provided by Intel, called *Quoting Enclave*, generates a report on the enclave to be attested. This report is signed by the *Quoting Enclave*, generating a *quote*. The *quote* contains all the information needed to identify the encalve and the platform. The *quote* is then sent to an *Attestation Service*, which verifies whether the enclave can be trusted or not. The *Attestation Service* could be Intel itself, through its Intel Attestation Service (IAS), or a local *Attestation Service* implemented with Data Center Attestation Primitives (DCAP). Using DCAP allows for more flexible attestation scenarios, where one wants to avoid delegating trust decisions to third-parties or higher latencies due to contacting a remote service.

2.1.1 Intel SGX SDK

Intel provides a Software Development Kit (SDK) for developing SGX applications. The Intel SGX SDK allows the developer to create applications that leverage enclaves to run sensitive workloads. As mentioned before, the application is split in two portions: secure (*i.e.*, inside SGX enclaves) and insecure (*i.e.*, outside of enclaves, on user space; communicates with the operating system and other applications). The communication between both parts is defined in a special language, called EDL (Enclave Definition Language), which specifies the interface for both secure and insecure portions (*i.e.*, which calls are allowed from the enclave to the insecure part and vice-versa). The developer has to implement the code that actually implements these interfaces.

```
enclave {
    trusted{
        public void app_to_enclave([in, string] char *secretIn);
        public void enclave_to_app([out, size = len] char *secretOut,
            size_t len);
    };
    untrusted{
        void print_debug([in, string] char *dbg_message);
    };
};
```

Source Code 2.1: Sample EDL file for an application that sends a secret to the enclave and retrieves it.

Source Code 2.1 shows an enclave definition using EDL language. The actual code has to implement these interfaces, and each interaction between the two portions of the application has to be defined. This requirement imposes a high effort to adapt applications to benefit from Intel SGX security guarantees. Not only the developer has to rewrite their code and use, in most cases, a completely new toolset, but also the toolset itself limits the possibilities, as Intel SGX SDK only supports C and C++ languages currently.

2.1.2 SCONE framework

As discussed in the previous subsection, the Intel SGX SDK limitations reduce dramatically its applicability, especially when pre-existing, complex applications are considered. Frameworks and shielded execution environments have been created to address such limitations, by leveraging SGX whilst providing more flexibility and better support for pre-existing applications.

SCONE (Secure CONtainer Environment) [2] is a framework that allows entire applications to run inside of Intel SGX enclaves, thus providing the same integrity and confidentiality guarantees. SCONE has also additional features, such a more robust attestation mechanism, transparent file system encryption and automated secret generation. However, having the whole application fit inside the enclave (whose EPC is limited in most cases) might impose some performance overhead, as the data needs to be encrypted before going to the main memory, and decrypted when coming back to the EPC.

SCONE relies on a set of tools, which include a modified GCC compiler and system libraries, such as *glibc* or *musl-libc*, to compile existing applications to run inside of Intel SGX enclaves. It supports dynamically and statically linked applications, and a broader set of high-level programming languages (such as C, C++, Python, Go, Java, JavaScript, C#).

Once inside the enclave, the application talks to the SCONE runtime, which handles all system calls through an asynchronous interface. An external container interface with lock-free queues processes system call requests and responses. SCONE has two special types of threads: *ethreads* and *sthreads*. *ethreads* are responsible for executing the application threads inside of enclaves. *sthreads* are responsible for handling system calls on behalf of the application threads. The number of *ethreads* and *sthreads* is configurable, as well as the number of lock-free queues. Each application has an ideal amount of threads and queues, depending on the nature of the workload. These threads can also be pinned to specific CPU cores for improved performance (avoid context switching, for example).

The SCONE runtime also enables two important features of the SCONE framework: remote attestation and transparent encryption. The remote attestation mechanism of SCONE relies on two components: the Local Attestation Service (LAS) and the Configuration and Attestation Service (CAS). LAS runs locally on the machine and attests and enclave locally (through a *Quoting Enclave*). The generated quote is then sent to CAS, which is the single source of trust and can establish whether an enclave can be trusted or not based on previously registered security policies. Additionally, CAS can provide secrets and configurations in a secure fashion (*e.g.*, certificates, keys, files) to an attested enclave. The transparent encryption feature leverages the shielding mechanism of the SCONE runtime to encrypt and decrypt files and network traffic transparently (*i.e.*, the application does not need to be aware of encryption). CAS also allows the persistence of attestation information in a local encrypted file, called *vault file*. If this mode is enabled, the enclave only needs to talk to CAS once. Every subsequent attestation request would be performed by the local vault file, as long as the enclave *quote* is exactly the same. This is useful when the application owner wants to avoid the latency of an external request during every startup/attestation.

The expected state of a SCONE application is defined in security policies called *session files*. *Session files* are defined by the user and describe the enclave that can be trusted (and thus have secrets and files delivered to it after attestation). *Session files* include the expected enclave identity of the application (MRENCLAVE), as well as the expected state of the underlying platform (whether hyperthreading is allowed, or outdated firmwares, for example). Secrets and configuration files can also be defined in a *session file*.

```
name: sample-session
version: 0.3

```

```
4  services:
5    - name: hello-world
6      mrenclaves: [41f0117a3c62966b48...ff719b1f48abbf417e855fff0546a8e0d]
7      command: /usr/local/helloworld --name "Bob"
8      pwd: /
9      image_name: hello-world
10     environment:
11       SCONE_MODE: hw
12       A_SECRET: $$SCONE::secret1$$
13 images:
14   - name: hello-world
15     injection_files:
16       - path: /etc/secret.txt
17         content: $$SCONE::secret2$$
18 secrets:
19   - name: secret1
20     kind: ascii
21     size: 16
22   - name: secret2
23     kind: ascii
24     value: "this is a secret!!!"
25 security:
26   attestation:
27     tolerate: [debug-mode, hyperthreading, outdated-tcb]
28     ignore_advisories: "*"
```

Source Code 2.2: Sample SCONE *session file*.

Source Code 2.2 shows an example of a SCONE *session file* named *sampe-session* (line 1). This session defines one *service* (lines 4 to 12). A *service* defines an application execution. The application (*/usr/local/hello-world*) has its expected enclave identity defined in line 6: any modification to the original code would result in a different enclave identity, thus failing the attestation process. It is possible to specify arguments, such as `-name "Bob"` (line 7) or environment variables (lines 10 to 12). Arguments and environment variables defined in the *session file* are only delivered to the enclave after it is attested by CAS. Lines 18 to 24 define to *secrets*: *secret1* is a randomly-generated 16 bytes ASCII string, whilst *secret2* is the string "`this is a secret!!!`". They are made available to the *service* via an environment variable (*secret1* is injected as the environment variable *A_SECRET* on line 12) and a file (*secret2* is written to */etc/secret.txt* on lines 16 and 17). The file */etc/secret.txt* does not exist on the local file system: it is rendered by the SCONE runtime inside the enclave (once it is attested). Finally, lines 25 to 28 define security aspects of the underlying platform, such as which security vulnerabilities are tolerated (line 27), and which Intel Security Advisories (security updates that are recommended by Intel) can be ignored.

In order to be attested, a SCONE application has to define the following environment variables: *SCONE_CAS_ADDR* (CAS address), *SCONE_LAS_ADDR* (LAS address) and *SCONE_CONFIG_ID* (the session and service definition the application is requesting, in the format *SESSIONNAME/SERVICENAME*).

2.2 Kubernetes

The grow in popularity faced by microservice architecture and container-based workloads [4] in the last decade have created the need for specialized tooling to manage and monitor large sets of production containers. This class of applications, known as *container orchestrators* act on top of an infrastructure (be it composed by baremetal machines, virtual machines, or both), sometimes being able to let the operator act on a higher abstraction level, which avoid coupling between applications and infrastructure and, ultimately, increases overall portability.

Kubernetes [6; 8] is the de-facto container orchestrator of the industry. Released in 2014, the open-source project reused many concepts that had been applied at Google internally for years, helping manage millions of productions containers daily. The first release of Kubernetes also marked the foundation of the Cloud Native Computing Foundation, or CNCF. The mission of CNCF is to define standards and spread best practices that aim at a portable, scalable and fault-tolerant world of applications, completely aligned with the ever-growing resource offer from cloud providers around the world.

Kubernetes manages an infrastructure by grouping a set of nodes in a *cluster*. Kubernetes clusters have two types of nodes: *masters* and *workers*. Master nodes are supposed to run the core components of Kubernetes, responsible for maintaining the cluster, such as its controller manager, scheduler and API server. It is a good practice to use the master nodes exclusively for workloads related to cluster management. It is common to see clusters with a single master node. However, in order to achieve more strict high-availability and fault-tolerance requirements, one should have at least 3 master nodes. All cluster state is persisted to etcd[1], a distributed key-value storage.

[1] https://etcd.io/

One of the key aspects of Kubernetes is its application-centric approach. Its powerful declarative API offers objects, such as *Pods*, *Deployments*, *DaemonSets* and *Services*, that allow the operator to not care about infrastructure specifics, application scheduling or even resource allocation. This allows Kubernetes to manage fairly large infrastructures in an efficient way, almost completely transparent to the user. As of today, Kubernetes officially supports infrastructures with up to 5000 nodes. Another direct result of having a declarative, uniform API that takes care of infrastructure specifics is portability. An application that runs on a Kubernetes cluster will most likely run in any other Kubernetes cluster, regardless of the underlying infrastructure (for instance, on different cloud providers).

When a user deploys an application to a Kubernetes cluster, they specify the desired state of the application. A set of controllers then make sure that the state is the desired at all times, taking action whenever is necessary. This allows tasks such as replica management, failure recovery and application updates to be taken care automatically. Moreover, the Kubernetes API has been consistently evolved and extended over the years, resulting in a rich and powerful cloud-native ecosystem. Various tools and solutions for monitoring, logging, access control, storage and more have been created and integrate seamlessly to the Kubernetes API.

The main Kubernetes API objects and the behaviors associated with them are detailed below. They are *Pod*, *Deployment*, *Service*, *Ingress*, *DaemonSet* and *StatefulSet*. Kubernetes offers many more API objects, as well as provide an API for custom objects, called *CustomResourceDefinition*.

Pod. The smallest application unit. Represents one or more containers that share a common goal and can be seen as one. For instance, a database application, such as MariaDB or redis, bundled with an auxiliary container that acts a proxy or authorizer of said database. *Pods* have their own network namespace (IP address and ports), which is shared among its constituent containers. *Pods* are also considered to be ephemeral by the Kubernetes scheduler, which can terminate or move them for some reason (*e.g.*, to free resources in a node under some kind of resource pressure). Consequently, *Pods* should always be created with an associated replication controller, which is offered by other objects, such as *Deployments* and *StatefulSets*.

Deployment. Represents a *pod* associated with a replication controller, which ensures the desired state is achieved. It is possible to define the amount of replicas and failure recovery policies. For instance, if a node of the cluster fails and becomes unavailable, causing a *deployment* to have less replicas than specified, the replication controller will act (*i.e.*, spawn new replicas) until the desired state is established again.

Service. Allows *pods* to be discovered and accessed by other entities. Since *pods* are ephemeral, their IP addresses can vary constantly, which is not a reliable option for communication. A *service* is a layer 4 load balancer (according to the OSI model) for *pods*. Through *labels* and *selectors* one can bind certain *pods* to *services*. *Services* have an IP address that does not change and entry in the in-cluster domain name server (DNS), which can be accessed by other in-cluster entities (*e.g.*, other *pods*). To allow applications to be reached from outside the cluster, there are two distinct *service* types. *NodePort services* expose a random port within a limited range in all nodes of the cluster for the application associated with the *service*. This way, an outside client could talk to any node IP address on the specified port. *LoadBalancer services*, on the other hand, contact the underlying cloud provider to provision a managed load balancer for the application.

Ingress. *Ingress* are layer 7 load balancers (according to the OSI model). They expose *services* to outside of the cluster, through a public IP address. Being a layer 7 (*i.e.*, application layer) load balancer gives *ingresses* a broader set of features, such as TLS termination, virtual host routing or canary deployments. It is important to note that Kubernetes does not implement the *ingress* logic, and the *ingress* API must be implemented by an external controller, chosen by the user. The *ingress controller*[2] is responsible for deploying the actual load balancer *pods* and managing *ingress* objects. The most popular load balancers on the market, such as HAProxy, NGINX and Envoy, offer an *ingress controller* solution.

DaemonSet. Represents a Pod and a replication controller that ensures that there will be exactly one replica per cluster node. It is possible to filter out nodes through selectors and labels. Useful to deploy applications such as log collectors, storage, monitoring or attestation agents, and many more.

StatefulSet. Represents a Pod and a stateful replication controller, which allows each replica to have its identity persisted. Useful to deploy stateful applications, such as a Database Management System. It is important to note the Pods are still ephemeral, but now they have an identity that is unique and persistent, which can be combined with data persistence techniques, such as volume provisioning.

2.3 Related work

AMD SEV [10] is a TEE technology provided by AMD. Based on AMD Memory Encrytion technology, it has recently faced an increase in popularity since it was announced as the default TEE for Google's new confidential computing offerings. Applications do not require software change, and the protected application can use as much memory as it is available on the machine. However, SGX offers a more strict control over the protected memory (including memory integrity protection and access

[2]https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

control) and defines a clear boundary between the trusted and untrusted portions of an application. Therefore, for applications that deal with sensitive data that require confidentiality and integrity, SGX is most suitable, despite its performance overhead [14].

ARM TrustZone [21] focus more on embedded and mobile systems with a single purpose. In TrustZone architecture, the processor is partitioned in secure and insecure portions, offering no further isolation within these regions. Intel SGX supports multiple enclaves, which are isolated from each other, enabling multi-purpose, multi-tenant use cases.

For Intel SGX SDK, the need to rewrite the code to use the enclave interface and define secure and insecure functions is prohibitive in most cases. Also, enclave attestation becomes a non-trivial process with SDK. Graphene-SGX [19] and SGX-LKL [15] are shielded execution frameworks that allow unmodified applications to run inside Intel SGX enclaves. Unlike SCONE, however, they include the operating system library within the enclave, broadening the attack surface considerably.

Ataíde *et al.* [3] and Sampaio *et al.* [16] use a similar approach of having generic application instances in TEEs managed by an application wrapper in the context of batch processing in untrusted infrastructures. As both works target batch processing, they make design decisions that are not adequate for handling interactive applications, as is our case. Segarra *et al.* [17] evaluates stream processing of confidential batch jobs using a modified Spark inside of SGX enclaves. Although streaming favors latency, it is still a rather different approach as compared to interactive services.

Serverless architectures have become very popular in the last few years. In fact, the serverless paradigm has some similarities with this work. Notably the ability to dynamically provision resources as needed, which is particularly appealing in a pay-as-you-go model. Most public cloud providers have serverless offerings. However, such services have some limitations. In the Function-as-a-Service (FaaS) model, for example, workloads that are more complex (which requires them to be split into several functions) usually suffer from performance issues, since the functions cannot communicate with each other directly and efficiently, requiring some sort of intermediate service (usually a shared storage, which tends to be slower). Stateful workloads suffer from the same issue, since each function invocation is meant to be stateless and independent. Managing data or state across function invocations requires an external, intermediate service [9]. Furthermore, access to specialized hardware, which is essential to benefit from TEEs capabilities, is also not supported in the current serverless offerings in the public cloud. Brenner and Kapizta [5] propose a generic confidential Function-as-a-Service platform that shares some goals with this work, such as minimizing the performance overhead (specially cold-start times) and optimizing resource usage. However, in order to achieve it, their application instances, which are JavaScript functions, share the same secure interpreter, and thus the same Intel SGX enclave. Their isolation is provided by software. Our approach provides further isolation by providing each application instance its own enclave. We also support a broader range of applications, since we aim at deploying services, which are inherently more complex than standalone functions.

Teodoro *et al.* [18] propose a stateful session-based load balancing mechanism for clustered web servers. Their approach relies on a central component, called *session directory*, that assigns user sessions to back-end servers, similarly to our Routing Manager. However, their work assumes that the load balancing is performed by the application itself, which has the ability to migrate sessions to other servers based on the current server load. Our approach for message routing relies on battle-tested, off-the-shelf load balancing solutions, such as HAProxy and Kubernetes, which allows it to be seamlessly integrated with any application, not to mention the broader set of features that it inherently offers (*e.g.*, SSL termination, SNI support, etc).

Chapter 3

Use case: smart grids

This chapter presents a use case and a sample application that follows the set of requirements described in Section 1.1. This use case and its implementation are used to better motivate the solution proposed in Chapter 4, and also to guide its evaluation in Chapter 5.

Consider the Electric Power Distribution Operator (EPDO) of a large city or region. The EPDO is responsible for distributing electricity from transmission lines to end customers through a distribution grid. End customers include urban, rural and also industrial facilities. The grid is equipped with smart meters and other sensors, which send metrics about the overall state of the network constantly. This data is used to detect failures, frauds and anomalies, to analyze the quality of the transmission, and to guide decision making, aiming at a more optimized system. It comprises important information about the distribution network, which is in itself sensitive, as it exposes points of failure, but also contains detailed metrics and consumption profiles of all end customers.

A large volume of data is processed by the system constantly, as the smart meters and sensors push their metrics. The service provides an interface to visualize and query the data. All communications are end-to-end encrypted, and unauthorized access must not be allowed. Data at rest is also protected through encryption. Only the owner of the data is able to access, visualize or query it. It means that the EPDO is not able to visualize end customers data, despite being the application provider. It only has access to metrics of the distribution and overall network state (pushed by sensors and smart meters that belong to the EPDO).

The system must be available at all times, and also provide a reasonable response time to the end customer, whilst guaranteeing the isolation, confidentiality and integrity of their respective private records.

3.1 Architecture

The system is composed by a server application and two types of client applications. One client application (Type 1) runs on smart meters and sensors, and its only function is to push new metrics to the server. The other client application (Type 2) allows the user to perform queries and visualize their data. All user data is saved to an embedded relational database, and each individual user has its own database, which is stored in a single file in the server file system. The database file is encrypted with an individual encryption key, which is provided to the server after a connection is successful established through a `LOGIN` operation (Section 3.3.1). All supported operations are performed through HTTPS requests to the server, and all communications between client and server are encrypted with TLS (Transport Layer Security).

Figure 3.1 shows the service architecture, and illustrates the communication between clients (both smart meters and customers) and servers through an encrypted connection (TLS). Distinct client types support different sets of operations. The encrypted connection might go through a load balancer that does not terminate TLS.

3.2 Data model

The data model of the example application is relational and SQL-compatible. The tables are described below.

- **`metrics`** `(id INTEGER PRIMARY KEY AUTOINCREMENT, timestamp INTEGER, power REAL, impedance REAL, tension REAL)`. Stores collected metrics pushed by smart meters and other sensors.
- **`alarms`** `(id INTEGER PRIMARY KEY AUTOINCREMENT, metric TEXT, threshold REAL)`. Stores user-defined alarms, with a metric threshold. Clients can rely on these alarms to trigger actions.

3.3 Supported operations

The server application supports a set of operations that can be performed by the different client applications to transform and query the user data. All operations, except the `LOGIN` operation, are performed in the context of a *Session*. The `LOGIN` operation creates a *Session*, where cryptographic

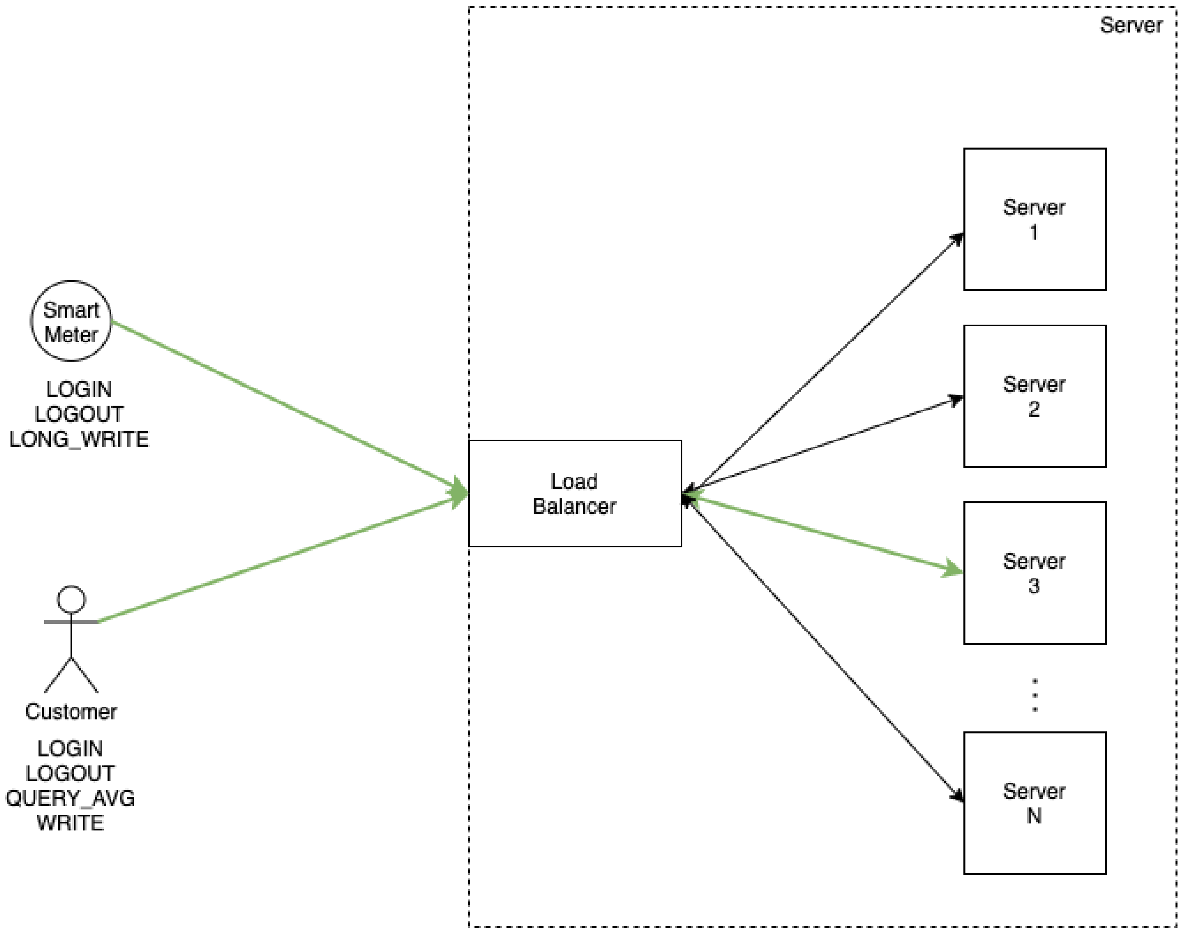

Figure 3.1: Use case service architecture.

data is exchanged and the secure communication is established. The complete list of operations supported by the server application are listed below.

3.3.1 `LOGIN`

The `LOGIN` creates a user *Session* in the server, and must be performed by the client before any other operation, in order to authenticate the server and exchange cryptographic information to decrypt the user database. All further operations must be executed within a *Session* context. A *Session* has also a limited duration, after which it expires, requiring the user to perform another `LOGIN` operation and create a new *Session*.

After a secure mutual TLS connection is established, which means that both server and client authenticated each other, a *Session* is created, with a unique *authorization token*. After the client authenticates the server (*e.g.*, with a TLS certificate that could only be provided after an initial SGX attestation), it shares its encryption key, which will be used by the server to decrypt the user records.

3.3.2 `LOGOUT`

A `LOGOUT` operation destroys the current *Session* and closes all open connections and resources, committing the user data to the database. It can be performed by the user or automatically triggered after a *Session* expires. After a `LOGOUT`, the server is idle, i.e., not actively serving anyone.

3.3.3 `WRITE`

`WRITE` updates the user database with new information via an SQL `INSERT` statement. It is intended for Type 2 client actions (*e.g.*, updating metadata, creating alarms and thresholds, or registering devices). In the context of the example application, it writes a few alarms to the `alarms` table. One `WRITE` operation might trigger multiple `INSERT` statements at a time, in which case the server application runs them inside of an SQL transaction, that is either commit, if all inserts are successful, or rolled back, if at least one of the inserts fail.

3.3.4 `LONG_WRITE`

`LONG_WRITE` also updates the user database with new information via an SQL `INSERT` statement. However, it is intended for Type 1 clients pushing new metrics collected from a smart meter or sensor. This actions are usually much larger, since the client accumulates a large number of measurements (*e.g.*, the 900 one-second measurements for the last 15 minutes or 3600 from the last hour). For this reason, they are isolated in a separate type of operation. A `LONG_WRITE` operation triggers dozens of `INSERT` SQL statements at a time, which are all executed by the server within an SQL transaction context.

3.3.5 `QUERY_AVG`

The `QUERY_AVG` operation allows the user to execute queries on their data through a `SELECT` SQL statement to calculate metric averages (*e.g.*, consumed power, impedance, tension) for a given period of time (including all time). The server processes such queries and return their results, which can then be used to plot visualization or feed analysis algorithms, for example. Only Type 2 clients are allowed to perform this operation.

3.4 Implementation

3.4.1 Server

The server application is implemented in C, which allows for an efficient memory management and overall small resource footprint. To manage the user database, the example application uses SQLite[1], a library that implements a small, fast and highly reliable SQL database engine. The database lies in a single file, which makes SQLite the perfect choice for embedded or self-contained applications. In fact, SQLite is built-in in nearly all mobile phones today, making it arguably the most used database in the world[2]. In the example application, SQLite allows for a better performance, if compared to a full-fledged DBMS (database management system). Its file format also makes it easier to encrypt the whole database at once.

We use SCONE to shield the application execution. C applications provide the smallest enclaves when running on SCONE. The enclave size of SCONE applications does not vary at runtime, and the entire enclave size (defined by environment variable *SCONE_HEAP*) is allocated at startup time, which incidentally increases cold-start times. Also, having the smallest possible enclave size helps with overall performance, even when the startup time overhead is not considered. Larger enclaves, due to the EPC memory constraint, result in more swapping from the protected memory to the main memory, which hurts performance severely. Go, for example, has an absolute minimum enclave size of 152 MB in SCONE. As the application grows and becomes more complex with added dependencies, this minimum requirement can reach up to a few gigabytes, which would make deploying many instances prohibitive. Not only the swapping from EPC to main memory would be expensive, but also the usage of main memory is not efficient, especially when considering deploying a few dozen servers per node. In most infrastructures, this would result in memory pressure. Table 3.1 shows the absolute minimum enclave size requirements in SCONE for C, Go and Python applications, alongside their average startup time. The test application is empty and returns immediately. Please note that interpreted Python code needs to be encrypted in order to provide the same integrity and confidentiality guarantees as C or Go applications.

The C server exposes an HTTP REST API, and uses TLS (Transport Layer Security) to encrypt the messages end-to-end. The server API has distinct endpoints for each operation. Since each server is supposed to serve only one client (due to the isolation requirements), the server certificate is issued to the client UID name (*i.e.*, the server certificate's *Common Name* field will be the client UID).

[1]https://www.sqlite.org/index.html

[2]https://www.sqlite.org/mostdeployed.html

Language	**Minimum enclave size**	**Avg. startup time**
C	8 MB	0.116 second
Go	152 MB	62 seconds
Python	8 MB	0.23 second

Table 3.1: Minimum enclave size requirements in SCONE.

The SCONE configuration of the server uses only 1 queue, 1 *ethread* and 1 *sthread*, both unpinned. The enclave size (*SCONE_HEAP*) is 64 MB.

3.4.2 Client

The client application is written in Go, and does not run inside of Intel SGX enclaves. It can simulate clients of Type 1 or Type 2, and supports different execution parameters that allow for different client workloads. The list of parameters supported by the client application is shown in Table 3.2.

The server address (parameters `-server` and `-port`) is, in fact, the load balancer address. The communications are end-to-end encrypted with TLS. The load balancer does not terminate the connection. It uses the SNI information (that the client defines via `-sni` parameter) to indicate the correct server. Once the communication to the server is established, the client sends the private key used to decrypt their data (`-keydir`).

Parameter	**Description**
`-server`	Server address
`-port`	Server port
`-sni`	Client UID, which is also the server name (as in the server certificate)
`-cert`	Path to client certificate to establish a TLS connection
`-key`	Path to client private key to establish a TLS connection
`-ca`	Path to client CA certificate to authenticate the server identity
`-keydir`	Path to the key used to decrypt user's records
`-mode`	Whether the client is simulating clients of Type 1 or Type 2
`-duration`	How long the session will be. Long sessions might require another `LOGIN`
`-frequency`	How often requests will be sent within a session
`-writeRatio`	The proportion of `WRITE` operations to be performed, on average
`-csv`	Print output in CSV format

Table 3.2: List of execution parameters supported by the client application.

Chapter 4

Dynamic provisioning of stateful applications in TEEs

Provisioning containerized applications with a strict set of privacy and isolation requirements, as posed in Section 1.1, is already a non-trivial problem. When such applications run on a third-party infrastructure, such as public cloud providers, it raises the complexity of any acceptable solution. In fact, when running applications in a public cloud infrastructure, the addition of TEEs might cover for the various privacy concerns, especially when assuming a threat model that does not trust the cloud itself. However, the addition of TEEs also brings some other concerns, especially related to performance (which, in a public cloud setting, incurs in more financial cost to the application owner).

This section presents an approach that aims at solving such concerns, by providing an architecture in which such strongly-isolated applications (*i.e.*, each application instance is a separate, independent process) can be deployed to serve a large number of users. Two alternatives approaches are also presented for comparison.

The three approaches are named *Trivial*, which is the simplest, *On-demand* and *Standby Servers*. The later will be used throughout the rest of this work, and compared against the trivial approach.

4.1 Alternative approaches

4.1.1 *Trivial* approach

If the applications have strong isolation requirements (*i.e.*, one application instance serves one user), as described in Section 1.1, the trivial approach would be to have all application instances running at all times, ready to serve the user whose state they hold. This is the simplest approach, and in fact, probably the most responsive too (from the user perspective), since no additional work is needed to provision the requested state.

However, the trivial approach is not resource efficient, which translates into a much higher operational cost. At any time, all the states are available and ready, but only a fraction of them is being actively requested. For instance, if a certain service has 100 users, but the average user activity is 25% (*i.e.*, 25 active users), the application would have 75% of the instances ready and running, but idle. This is clearly not efficient and, in fact, prohibitive when the user base start to grow over millions of users.

4.1.2 Dynamic provisioning: *On-demand* approach

A second approach, which resembles serverless architectures, is to spawn application instances only when there is an active client. At any time, the number of running application instances is equal to the number of active users, and this number is dynamic. This approach introduces two new components to the architecture:

- Routing Manager: some sort of proxy responsible for processing the incoming client requests and deciding whether a new application instance must be started or not.
- Application Wrapper: the component that would actually start the application instance when requested by the Routing Manager. This component might or might not be under the application owner's responsibility. In a serverless environment, for example, it would be provided by the cloud provider itself (*e.g.*, the serverless API). New application instances can be either new processes (which run in an independent enclave, therefore being isolated from other instances) or new containers.

This dynamic provisioning approach is more resource-efficient, since only the needed amount of resources is being used at any given time. However, it might come at the cost of performance and overall complexity of the solution. Now there are a few additional processes between the user request and the server response. For instance, when a user request a new state:

1. The user requests a new state via a `LOGIN` operation (*i.e.*, a new application must be spawned);
2. The Routing Manager decides that a new instance is needed;
3. The Routing Manager talks to the Application Wrapper to spawn a new instance;

4. The Application Wrapper spawns a new instance;
5. The requested state is provisioned, either by the Application Wrapper, in case of having some sort of dynamic data provisioning, or by the application instance itself, in case the state is pulled from an external entity (*e.g.*, a database or a content delivery network);
6. The application instance receives the user request;
7. The application instance responds to the user request.

Some of this processes might happen over the network (*e.g.*, 3 and 5), or depend on external entities (*e.g.*, 5). Also, processes 4 and 5 are particularly more expensive if strict privacy requirements are in place. Starting applications that run inside of TEEs is way more expensive than running native applications. Also, if the state being provisioned is encrypted, an additional overhead (also directly proportional to the size of the state) is imposed to the application instance, and overall response times. All of this extra steps (2–5) combined point to an expensive cold start and prohibitively high tail-latency times.

If we consider the scalability aspect, the approach should scale well as long as the Routing Manager is able to consider multiple nodes to which it can route requests or request new application instances to be deployed.

4.2 Dynamic provisioning: *Standby Servers* approach

The previous approaches have advantages and disadvantages. For instance, the Trivial approach, although the simplest of the three, does not provide a good solution from resource consumption and scalability points of view. The on-demand covers for the resource-efficiency bit, which is important when the infrastructure is run by a third-party. However, starting application instances on-demand also imposes a severe performance overhead, which may be prohibitive for the user experience and quality of service.

The approach presented below is very similar to the on-demand approach, having the same architecture and components. However, it tries to mitigate the overhead imposed by starting the application on demand within a TEE. It does that by keeping a number of application instances running, but without any state loaded. Such application instances are ready to serve any user. Whenever a new user arrives, the Routing Manager forwards their requests to one of the idle application instances. Such instance then assumes the right identity and responds to the user, who it will continue to serve until the end of the session.

In this approach, an idle application instance assumes the correct identity by retrieving the requested state from an external source after the performs a `LOGIN` operation. The external source can be, as already described in the previous approach, a database, another service or website, a local file system, or even a dynamically-provisioned file system or volume.

The amount of application instances that run is fixed and should be calibrated based on the average load of the service. If all the idle application instances are already serving someone, the amount of servers can be increased, leading the Application Wrapper to spawn more idle application instances to accommodate the extra load.

This architecture also supports multiple nodes, as long as the Routing Manager is able to coordinate between all the Application Wrappers. Thus, assuming a fair ratio of application instances deployed by this approach, the latency increase caused by instantiating a new application within a TEE is attenuated.

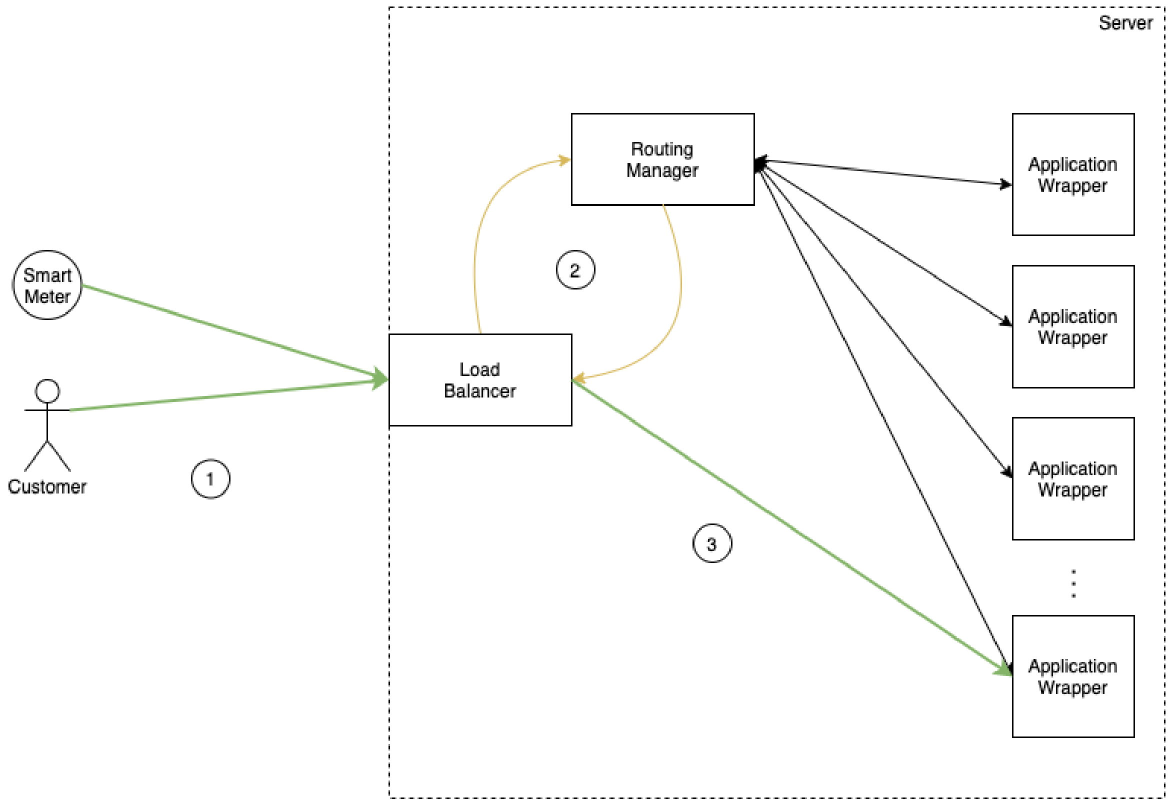

Figure 4.1: Standby Servers architecture.

Figure 4.1 shows the *Standby Servers* architecture. Requests sent by the clients (smart meters or customers) are encrypted (1). The Load Balancer queries the Routing Manager (2) for a routing decision without inspecting the request. The request is then forwarded to the assigned server, which runs inside one of the Application Wrappers (3).

4.3 Integration with example application

The example application (as described in Chapter 3) is integrated with the *Standby Servers approach* and the *Trivial* approach (to provide a baseline performance for comparison). Both approaches (including all needed components) are defined in Kubernetes manifests, allowing them to be quickly deployed in any Kubernetes cluster with Intel SGX support. The following subsection discuss implementation aspects of the components.

4.3.1 Application Wrapper

The Application Wrapper is responsible for managing the life cycle of the server applications, and for reporting their status to the Routing Manager. It is implemented in Python and does not run inside of Intel SGX enclaves. Each application server is managed by a sidecar thread created by the Wrapper. Each application server runs in a separate process in its own enclave inside the Wrapper's container. The Wrapper is application-agnostic, and the communication between application and Wrapper is performed by the sidecar thread, which listens to specific events logged by the application, taking appropriate action upon them. Thus, the sidecar thread is the interface between Wrapper and application, and must be adapted to listen to the right set of events. In this example, the sidecar thread listens to events logged to the application standard output (Table 4.1).

Server event	**Wrapper action**
`WAITING_LOGIN`	Report the instance status as *Ready* (*i.e.*, accepting new connections) to the Routing Manager.
`LOGIN <ClientUID>`	Report the instance status as *Active* (*i.e.*, currently serving someone) to the Routing Manager. Provision the encrypted user data for `<ClientUID>`.
`LOGOUT`	Report the instance status as *Stopped* to the Routing Manager. Deprovision the encrypted user data. Close all open connections and resources. Restart the server application.

Table 4.1: Server events and Application Wrapper actions.

The Application Wrapper has two operation modes: *standalone* and *dynamic*. If in *standalone* mode, the Application Wrapper has the needed parameters (such as the Routing Manager address, the number of servers to run, the server executable binary, whether to use dynamically provisioned volumes or not) in its environment, and starts all sidecar threads and applications right away. In *dynamic mode*, the Application Wrapper is remotely started or stopped by REST API calls, allowing it to be integrated to an external service, such as an auto-scaling controller. In *standalone mode*, the number of servers is fixed. In this work, we use the Application Wrapper in *standalone mode*, which means that the number of applications is fixed. Although using dynamic mode would provide a more flexible and production-like behavior (*e.g.*, number of servers varying dynamically), it would also bring more complexity to the architecture and introduce the need for another component (*e.g.*, an auto-scaling controller that manages the Wrappers via their API), thus being considered out of the scope of this work.

After starting, the Wrapper contacts the Routing Manager and advertises itself, sending its current status and how many applications it manages. Then, each sidecar thread also contacts the Routing Manager to advertise the status of the application it manages (*i.e.*, *Ready*, *Active* or *Stopped*). Each status change in the application is also reported by its sidecar thread to the Routing Manager. Besides that, the Wrapper also reports periodically the state of all applications. Figure 4.2 shows the Application Wrapper architecture.

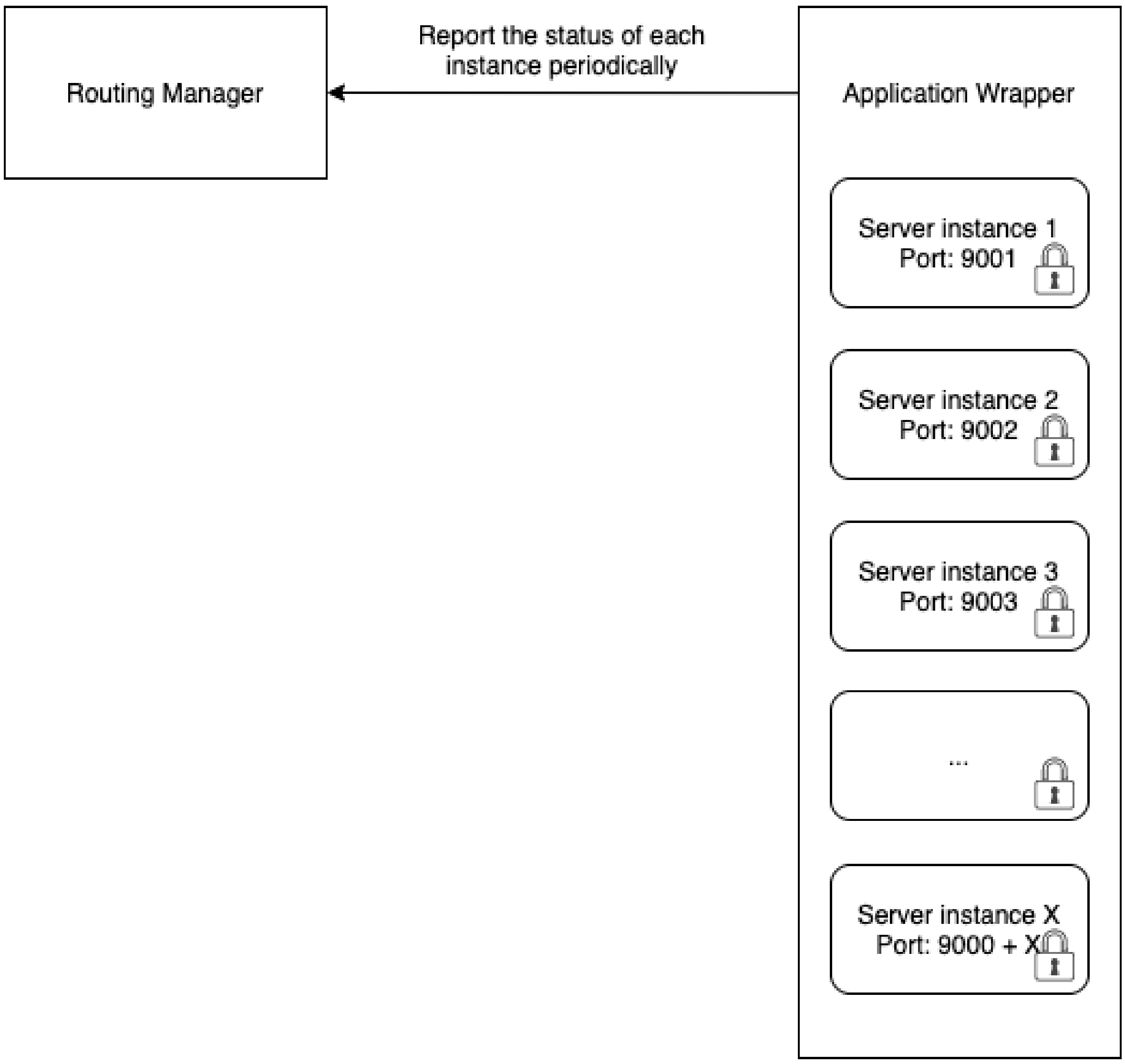
Routing Manager
Report the status of each instance periodically
Application Wrapper
Server instance 1 Port: 9001
Server instance 2 Port: 9002
Server instance 3 Port: 9003
...
Server instance X Port: 9000 + X

Figure 4.2: Application Wrapper architecture.

Wrappers are deployed to Kubernetes clusters as Deployment resources. The servers run inside of the Wrapper container, each one listening to a different port. Thus, although the Wrapper is seen as a single unit from the Kubernetes API perspective, it also contains all the applications that it manages running inside of the same container. Running each application as a process in the same container is more efficient, since spawning a Pod per application would take an additional round trip to the Kubernetes API, additional resources, and a much more noticeable overhead (from performance and management complexity perspectives). Each application is then exposed to the cluster through a Service, acquiring a local IP address and an entry in the in-cluster DNS server.

4.3.2 Routing Manager

The Routing Manager is responsible for the dynamic assignment of server instances to incoming clients. It holds the state of each application instance and updates it according to the reports sent to it by the Wrappers periodically. The Routing Manager exposes a Report API, used by the Wrapper and the sidecar threads, and a Query API, which is used by the load balancers in order to make a routing decision. Whenever a new client connects to the load balancer, the Routing Manager is queried and returns an unassigned application instance, to which the client request is forwarded. Further requests from the same client within the same *Session* can either have the load balancer query the Routing Manager again, or just bypass it via some sort of persistent session (*e.g.*, session cookies or the *Keep-Alive* HTTP header). Figure 4.3 illustrates the routing decision process of the Routing Manager.

The Routing Manager is deployed to Kubernetes as a *Deployment* resource, and is exposed by its own *Service* resource, which allows Wrappers, sidecar threads and load balancer to reach it via its local in-cluster DNS record. It is implemented in C and stores the state of the system (wrapper statuses and assignments) in an in-memory data structure.

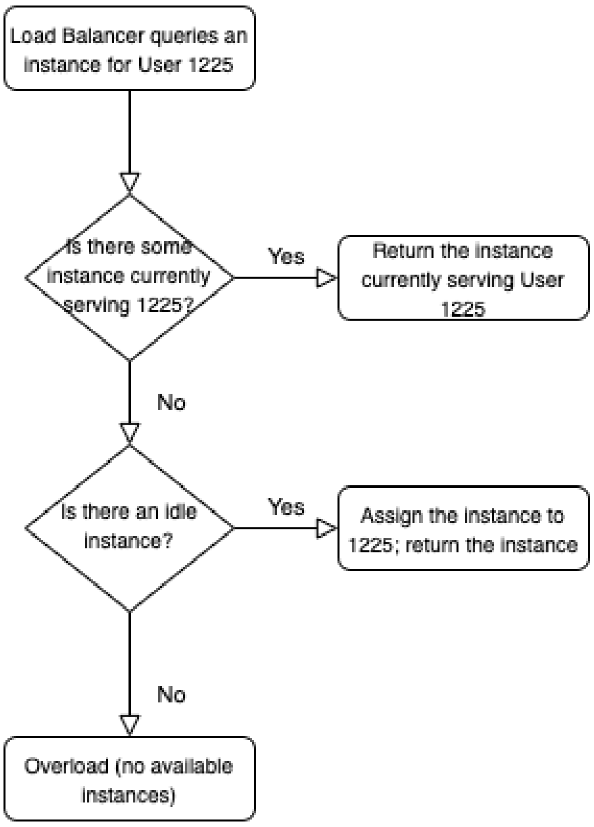
Load Balancer queries an instance for User 1225
Is there some instance currently serving 1225?
Yes
Return the instance currently serving User 1225
No
Is there an idle instance?
Yes
Assign the instance to 1225; return the instance
No
Overload (no available instances)

Figure 4.3: Flowchart of Routing Manager's decision making process.

4.3.3 Load balancer

The load balancer is the outermost component of the architecture and a key point of the end-to-end communication. It exposes a single endpoint through which the clients can contact the application instances transparently. It runs replicated, balancing and routing the incoming traffic based on queries made to the Routing Manager, which holds the state of the system. In this approach, the load balancer replicas are not supposed to run inside of Intel SGX enclaves. Therefore, they are not able to inspect the requests or any sensitive information: they should only route the incoming request to the appropriate server. The routing is possible through the SNI (Server Name Indication) TLS extension, which is set by the client. The SNI extension indicates the target server of a TLS request in the handshake process, allowing the load balancer to query the Routing Manager for the appropriate server. The content and the headers of the request are only accessible when the TLS is terminated, which only happens inside the application server (with the appropriate certificate and inside of Intel SGX enclaves).

The load balancer is deployed to Kubernetes as an Ingress resource (as discussed in Section 2.2), which exposes each Service for application servers. Since Kubernetes does not provide a default implementation for *Ingress* controller, it is required to pick one third-party controller among the many currently available. A preliminary investigation was conducted in order to define the most suitable Ingress controller solution.

Picking an Ingress controller

The official Kubernetes documentation page for Ingress controllers lists at least 15 different options for Ingress controllers, each one with its own target audience and feature set. The most popular load balancers and proxy solutions on the market have their own Ingress controller implementation. In order to find the most suitable Ingress controller implementation, the following requirements were defined.

1. Support the TLS SNI extension. Required for the routing in end-to-end encrypted channels, so the load balancer does not need to terminate the TLS connection.

2. Support some sort of mechanism that allow the load balancer to be integrated with the Routing Manager.

The following three implementations met these requirements, and were further compared.

- **Contour**. A Cloud Native Computing Foundation incubating project, Contour is an Ingress controller based on Envoy Proxy [11], an open-source layer 7 (according to the OSI model) proxy created by Lyft[1]. Envoy is a very popular, production-ready cloud-native proxy written in Go. It supports SNI routing natively. Even though Contour does not provide an easy way to integrate with external services (the feature that would allow this use case, called *External Authorization*, is currently under development), it is possible to leverage its dynamic route management to mimic that. In Contour, each route (*i.e.*, the binding to a server application) is a separate Kubernetes CRD (Custom Resource Definition), which allows for more flexibility. The Routing Manager, however, would need to be extended in order to talk to the Kubernetes API and create new routes dynamically.

- **NGINX Ingress Controller**. Based on NGINX proxy [2], one of the most famous proxy solutions on the internet, created and maintained by the community. It supports SNI-based routing natively, and its Lua extension support allows for a straightforward integration with the Routing Manager through sockets.

- **Voyager**. Based on HAProxy[3], which is another famous load balancing solution, Voyager is an Ingress controller created and maintained by Appscode[4]. It supports SNI routing natively and Lua extensions, similar to the NGINX alternative, which allow for socket use to contact the Routing Manager.

To compare the three Ingress controller solutions, a preliminary experiment was conducted. A simple web server application, written in C and built with SCONE, was deployed to a 10-node Kubernetes cluster. Each Ingress controller solution was deployed to manage 800 backend servers. A separate client machine ran one or two clients per backend server, with TLS and SNI, for a 10-minute benchmark consisting on HTTPS requests. The load balancers (*i.e.*, the actual proxy Pods) were replicated (10 replicas, one per node). The output variables are **latency**, in milliseconds, **throughput**, in requests per second, and total requests. Metrics for CPU and RAM usage of the load balancers were also collected.

[1]https://www.lyft.com/
[2]https://www.nginx.com/
[3]http://www.haproxy.org/
[4]https://voyagermesh.com/docs/10.0.0/concepts/overview/

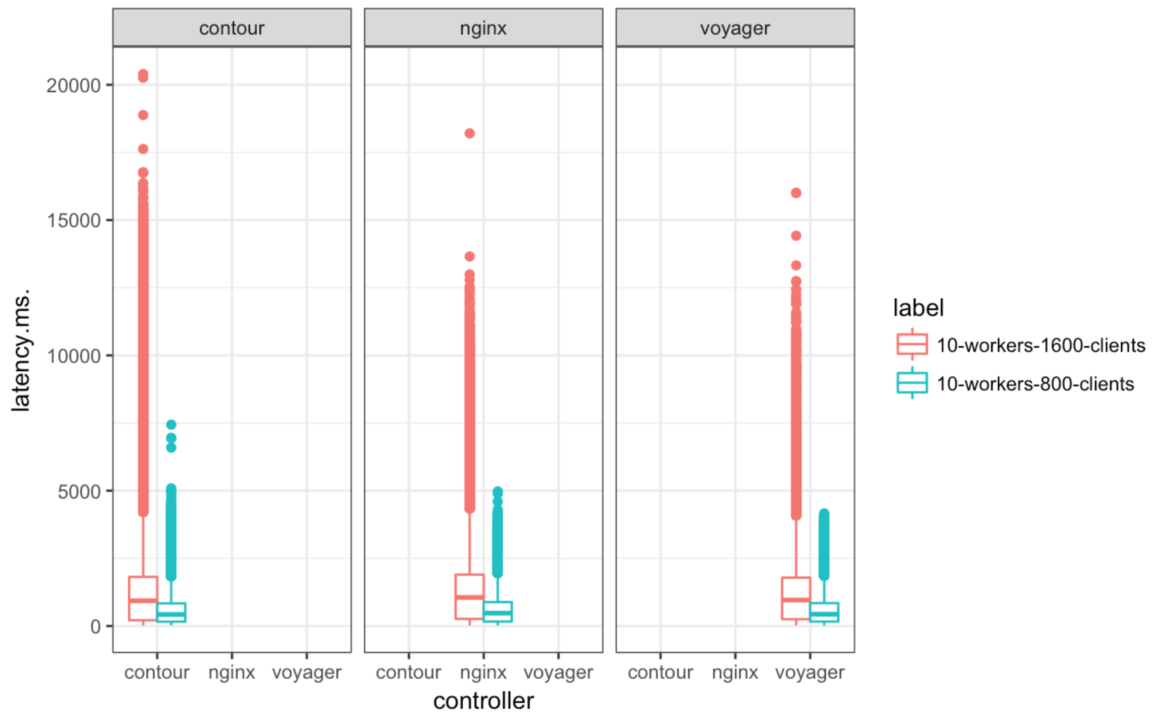

Figure 4.4: Ingress controllers: latency in a 10-minute benchmark.

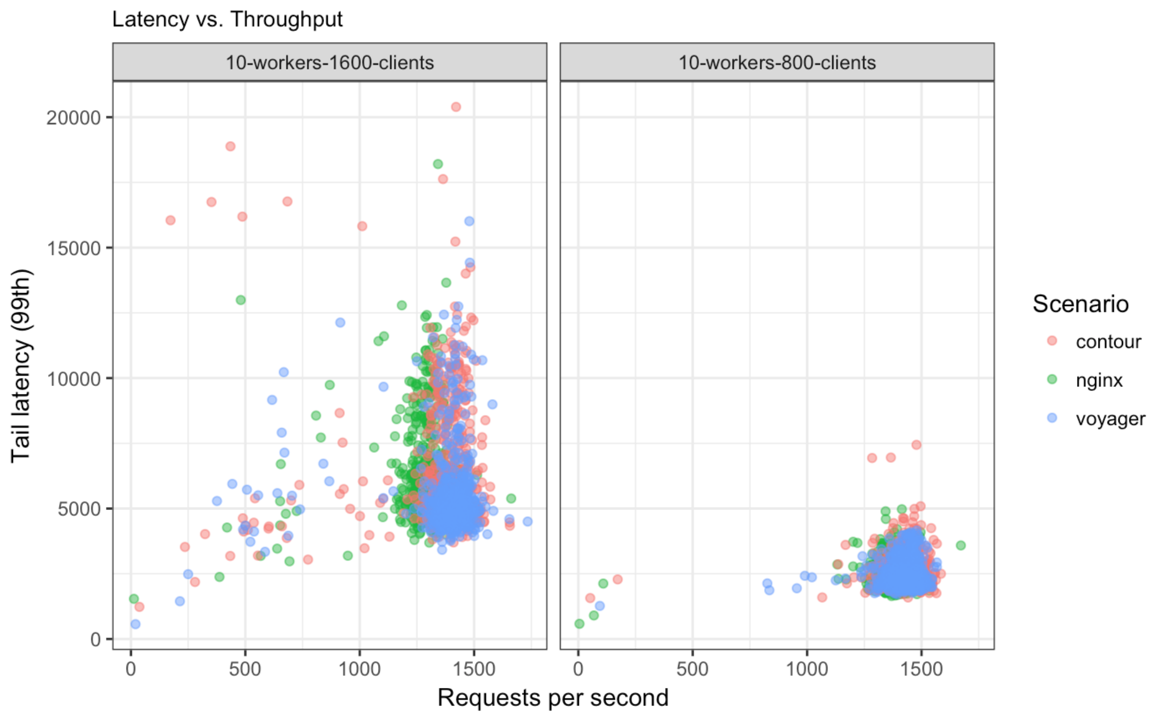

Figure 4.5: Ingress controllers: tail latency (p99) versus throughput in a 10-minute benchmark.

Voyager had a slightly better performance, presenting lower latencies and also smaller performance drops when doubling the benchmark clients (Figure 4.4). Figure 4.5 also puts Voyager in the lead when comparing tail latency and throughput. Finally, Table 4.2 confirms the slightly more stable behavior of Voyager, which also presented the smallest resource footprint of the three Ingress controllers.

Ingress controller	**Avg. total requests, 800 clients**	**Avg. total requests, 1600 clients**
Contour	859208	816512
NGINX	830716	771935
Voyager	854320	831612

Table 4.2: Ingress controllers: total requests processed in a 10-minute benchmark.

For this reason, Voyager was picked as the Ingress controller for this work. In addition, HAProxy, on which Voyager is based, is well-known for its reliability and efficiency, being used in an enormous number of production systems.

In Voyager, the integration with the Routing Manager is achieved by a simple Lua action that is triggered upon every incoming TCP connection. It extracts the SNI information and sends a request to the Routing Manager, which responds with the name of the appropriate backend. The socket operation performed by Lua is non-blocking, which attenuates the performance impact of an HTTP request to an external service.

4.3.4 Attestation

As discussed in Section 2.1.2, SCONE supports remote attestation. Remote attestation enforces the integrity of the code being executed. It does so through the Local Attestation Service (LAS), which attests enclaves locally, generating a *quote* that contains information about the enclave being attested and the underlying platform; and the Configuration and Attestation Service (CAS), responsible for attesting *quotes* based on the expected state of the enclaves, which are defined in *session files*. Once the enclave is attested, CAS also delivers secrets and configuration directly to it.

LAS is essentially a special enclave that attests other enclaves locally, by generating signed reports (the keys are unique to each physical machine) and the *attestation quote*. The communication with LAS is performed via TCP. When deploying LAS to Kubernetes, one must ensure the two following requirements are satisfied, in order for the local attestation to work: (1) A LAS replica runs on every node of the cluster that is capable of creating enclaves; and (2) the applications must contact the LAS replica running locally, on the same node. The first requirement is met by using a *DaemonSet* resource for LAS, making sure that exactly one replica runs on each node. To meet the second requirement, the natural choice would be a *Service*, providing the LAS replicas a name and an IP address. However, since *DaemonSet* replicas are considered to be the same, the *Service* load balancer only performs a round-robin between all available replicas. Therefore, a request sent by an application is not guaranteed to arrive at the replica running on the same node. To overcome this situation, the application defines the local container engine network interface as the LAS address (*e.g.*, `SCONE_LAS_ADDR=172.17.0.1` for Docker). An alternative is to define a special parameter to the LAS *Service*, `hostPort`, which allows it to listen on a port of the host network. The applications would then use the IP address of the node as *SCONE_LAS_ADDR* (it is possible to retrieve the node IP dynamically).

CAS is deployed to the Kubernetes cluster as a single-replica *StatefulSet* resource. An associated *PersistentVolume* resource binds a volume that allows that the CAS database to be persisted. In the event of a crash or restart of the CAS replica, the new replica would be able to retrieve the database state. We persist the attestation information in a local *vault file*, in order to avoid round trips to CAS every time an application instance is attested, thus reducing the startup times.

4.3.5 Data encryption and provisioning

The user data is provisioned through an auxiliary script, that leverages SCONE CLI to encrypt all the user data transparently (*i.e.*, the server application does not have to deal with cryptography explicitly). The input to the script is the plain data, which is an SQLite database. The script outputs the encrypted data directory and the respective keys. The keys are distributed to clients, while the encrypted data is distributed to all of the cluster nodes. Servers access the encrypted user data via a local mount of the host file system. Ideally, the data would be provisioned by a dynamic volume provisioner, such as Ceph RBD (as shown in Figure 4.6. In order to simulate the delay imposed by a dynamic volume provisioner, the servers impose a time penalty of 100 milliseconds when opening the user data during a `LOGIN` operation.

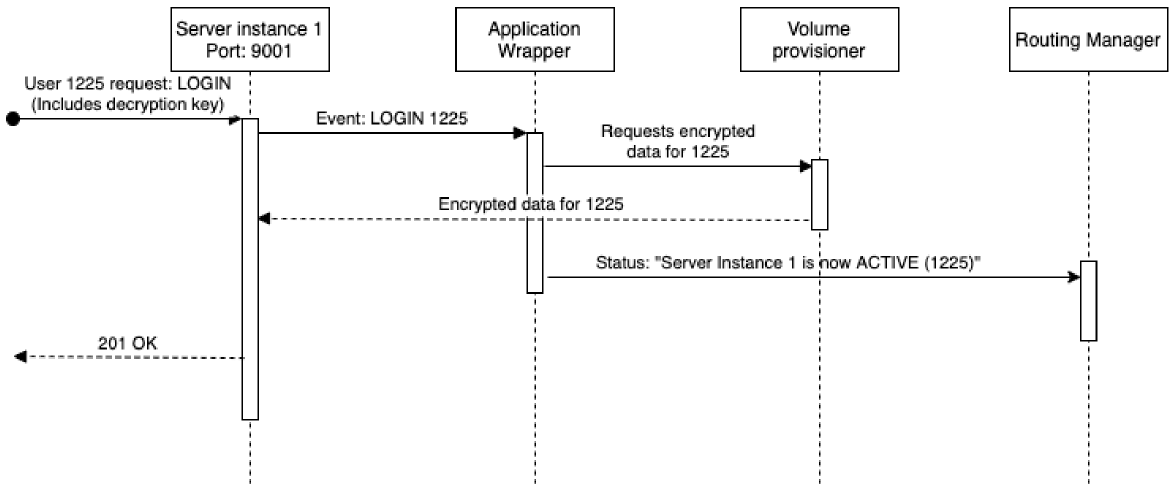

Figure 4.6: Sequence diagram for dynamic data provisioning after a `LOGIN` operation.

Chapter 5

Evaluation

This section evaluates the proposed architecture (Chapter 4) with respect to its performance, resource consumption and viability to deploy using popular, off-the-shelf infrastructure management tools, such as Kubernetes.

5.1 Enclave startup times

The alternative *On-demand* approach, as described in Section 4.1.2, relies on the dynamic instantiation of server instances, which are created on a per-need basis, as the clients arrive. This approach, which resembles serverless architectures, is more resource-efficient than the *Trivial* approach. The on-demand instantiation of server instances, however, imposes a high latency penalty to the end user. This issue, known as cold start, is also a frequent concern in serverless architectures. In our example application, the latency overhead is enhanced due to the addition of TEEs.

We run an experiment to evaluate the latency overhead that using the *On-demand* approach would impose to the end user.

5.1.1 Methodology

We evaluate the startup times of our example server application. The following scenarios are explored.

- **Enclave**. Measure how long the application server takes to start from a container that is already running.
- **Container+Enclave**. Measure how long the application server takes to start from a new container spawned by Docker.

We execute 30 sequential starts per run. Sequential starts are 1 second apart. The EPC memory is emptied before each run, and there are no other SGX applications running on the same VM. The application server has an enclave size of 64 MB, and remote attestation is disabled.

Experiment environment

The experiment is executed on a virtual machine (VM) in a private cloud environment managed by OpenStack. The VM has 4 vCPUs, 8 GB of RAM, 100 GB of storage and 30 MB of EPC memory. This cloud environment does over commit CPU (*i.e.*, the sum of virtualized allocatable resources exceeds the real amount of resources in the physical host), but does not over commit RAM nor EPC.

Experiment tools

To measure the startup times, we use the built-in Linux command `time`. `time` outputs three different times: *real*, *user* and *sys*. We only consider the *real* time, as it is the wall-clock time (*i.e.*, real elapsed time). The server application is started with the same SCONE parameters as for production use (*i.e.*, same enclave size, same amount of queues and threads). A special flag is set to make the application exit right after starting. Our clock stops when the application exits.

5.1.2 Discussion

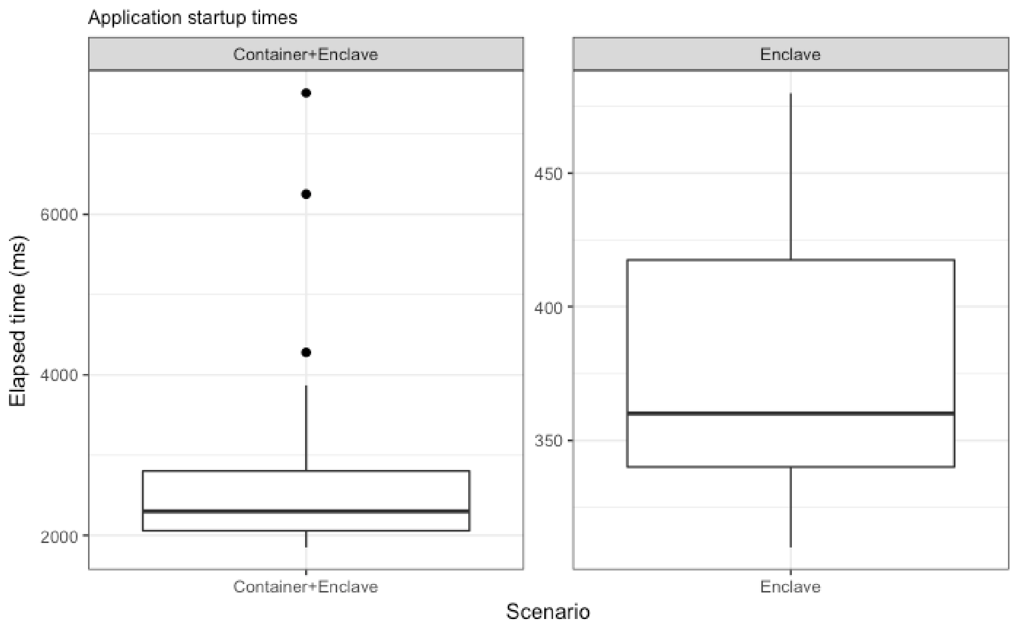

Figure 5.1: Application startup times.

Figure 5.1 shows the distribution of the startup times collected in the experiments. When starting an application server enclave from an existing container, it took on average 380 milliseconds (ranging from 310 to 480 milliseconds) for the enclave to start. When starting the enclave in a new container, however, the average startup time was 2778 milliseconds (ranging from 1854 to 7509 milliseconds). In both scenarios, the cost of starting a new server instance is noticeable for the user. Additionally, this experiments do not consider, for example, the time that the application takes to be able to serve requests (since we exit right after starting), or the cost imposed by Kubernetes (*e.g.*, the communication to the API server, the creation and scheduling of new resources). Since this application must not exceed a certain latency threshold, the *On-demand* approach does not meet all requirements.

5.2 *Standby Servers* approach

The proposed architecture was evaluated with respect to its efficiency and performance managing server instances of the example application described in Chapter 3. The proposed approach is compared to its *Trivial* counterpart, where all server instances are running at all times.

5.2.1 Methodology

The performance with the *Trivial* approach defines the baseline for our test infrastructure. The proposed approach is evaluated in different scenarios. Each scenario is composed by variables of a fractional factorial design experiment, and correspond to a distinct user activity pattern. The variables are listed below.

- **Active User Ratio (AUR)**. The maximum amount of users that the application is expected to have active at any given time. Since the *Standby Servers* approach allows for the dynamic serving of users through a fixed set of servers, it is necessary to define the target user ratio. The total amount of users is defined by the baseline experiment, which uses the *Trivial* (or all servers running at all times) approach. The levels for this variable are: 50%, 25% and 5%.
- **Write Requests Ratio (WR)**. Defines the write/total request ratio. For instance, if `WR` is 50%, half of the client requests, on average, will be `WRITE` operations. The levels for this variable are: 50%, 10% and 0% (*i.e.*, read-only).
- **Request Frequency (RF)**. Defines how often the client will send requests within a session. The levels for this variable are: 10 seconds and 60 seconds.

The duration of the user session is fixed at 600 seconds, or 10 minutes. All sessions are started with a `LOGIN` operation, and are closed by a `LOGOUT` operation. The levels for each variable were chosen based on the average usage of our sample application. That is why we choose a 10-second request interval as being our "most frequent" period, instead of 1 second or less. Such intervals would not be realistic for this specific application, but that might not be the case for other applications. The same logic applies to the values chosen for Active User Ratio, Write Request Ratio, and the *Session* duration of 10 minutes.

The output variables of this experiment are define below, and allow to evaluate the server performance from the user perspective (latency), and the resource consumption from the server perspective.

- **Latency** (client). Time spent by the client in each request, in milliseconds. Clients start the tracking of time right before sending the request, and stop immediately after a server response arrives. Therefore, request preparation and response inspection are not included in this time.
- **CPU utilization** (server). CPU time ratio for the server instances, per node, as reported by the Docker engine through cAdvisor.
- **Total RAM memory usage** (server). Total memory allocated to the server instances, as reported by the Docker engine through cAdvisor. This metric is the sum of the memory allocated for each node, expressed in GB.
- **EPC pages** (server). The amount of free, allocated and evicted EPC pages. Each EPC page has 4 KB.

Experiment environment

The servers (along with routing managers, load balancers and application wrappers) are deployed to an 8-node Kubernetes cluster. Each cluster node is a virtual machine (VM) in a private cloud environment managed by OpenStack. The cluster has 1 master node, which is used exclusively to run the cluster control plane and Kubernetes-related workloads (such as the API server). The master node VM has 2 vCPUS, 4 GB of RAM, 60 GB of storage and no EPC memory (*i.e.*, it is not able to run Intel SGX workloads). The rest of the cluster is composed by 7 similar worker nodes. Each worker node has 4 vCPUs, 8 GB of RAM, 100 GB of storage and 30 MB of EPC memory (*i.e.*, able to run Intel SGX workloads). The 7 worker nodes are distributed among 3 similar physical host machines, which means that co-located VMs probably affect one another at some degree. This cloud environment does over commit CPU (*i.e.*, the sum of virtualized allocatable resources exceeds the real amount of resources in the physical host), but does not over commit RAM nor EPC. Each physical machine has a 3.9 GHz quad-core Intel Xeon E3-1280 v6, 32 GB of RAM and 128 MB of EPC (although only 93 MB are available for applications, approximately). The clients run on a single physical machine, with a 3.1 GHz dual-core Intel Core i7 5557U, 16 GB of RAM, 512 GB of storage and no EPC memory. Each client is spawned on its own Docker container. The client machine is not in the same physical network as the server VMs, as it is expected in this use case, where clients access services hosted in a remote cloud.

Experiment tools

To collect the latency of the servers in different scenarios, the client application (as described in Chapter 3) is modified to track and log the time spent in every request, until the server response arrives. Request preparation and response inspection are not included in this time. The client applications use the standard Go time library (`time`) to track time. The approach used to deploy the server instances is completely transparent to the client application, which always talk to the same endpoint (*i.e.*, the load balancer endpoint). An auxiliary *bash* script spawns parallel clients within a given scenario and collects their results and logs in a CSV file.

To evaluate the resource consumption of the cluster, we use TEEMon [12], a continuous performance monitoring framework for TEEs. TEEMon supports Kubernetes clusters and provide a series of dashboards where it is possible to monitor not only TEE state (*e.g.*, active enclaves, active EPC pages, EPC page fault occurrences) but also the infrastructure (CPU, RAM, network) or specific Docker containers. Metrics are stored in Prometheus, a popular monitoring system and time-series database, and visualized in Grafana, a popular observability platform. Our metrics of interest, on the server side, come from two exporters deployed by TEEMon: cAdvisor (v0.36.0), which exposes container metrics (*e.g.*, CPU, memory and network); and SGX-exporter, which provides EPC-related metrics (*e.g.*, free pages, allocated pages, evicted pages). Metrics are scraped from exporters every 15 seconds.

5.2.2 Baseline performance

The baseline performance is defined by the *Trivial* approach, since we want to define the maximum amount of servers that can run comfortably in the test infrastructure, whilst still providing a reasonable tail latency for all operations. The preliminary experiment aims at defining a reasonable amount of servers for our experiment environment, which will then be considered as the total amount of users.

Table 5.1 shows the baseline experiment levels, which configure the most intensive usage scenario: a write-intensive (WR is 50%) set of frequent operations (every 10 seconds).

The *Trivial* approach is deployed to Kubernetes as a set of *Deployments*, one per application instance. It means that each application instance runs in its own container, which is exposed by a *Service*. Each *Service* is then added to the *Ingress*, and the load balancing is performed by a standard HAProxy load balancer (controlled by Voyager), as we do not need any extension to talk to external services. To avoid imbalance between application instances and nodes, we add `topologySpreadConstraints` to the *Deployment* manifests. This special field defines spread constraints to the Kubernetes scheduler. In our experiment, the scheduler spreads the application instances equally among the nodes.

We measure baseline performance only for clients of Type 2, which are more affected by latency issues. Clients of Type 1, such as smart meters and sensors, not only tolerate higher latencies, but also are assumed to have retry mechanisms to push any pending metrics to the server.

Servers per wrapper	**Total servers**	**Write ratio**	**Request frequency**	**Session**
15	105	50%	10 seconds	600 seconds
20	140	50%	10 seconds	600 seconds
25	175	50%	10 seconds	600 seconds

Table 5.1: Levels for baseline experiment with *Trivial* approach.

The output variable is *Latency*, or the total time spent by each client request to be completed, in milliseconds. We are specially interested in the tail latency, here defined as the 95th and the 99th percentiles of *Latency*. The n^{th} percentile of the *Latency* distribution is the maximum latency for the fastest $n\%$ of all requests. For instance, if the 99th percentile latency of a given operation is 500 milliseconds, it means that 99% of the requests were processed under 500 milliseconds. The median, which is the 50th percentile, is also shown.

Servers	**`LOGIN` p50 latency**	**`LOGIN` p95 latency**	**`LOGIN` p99 latency**
105	676.812 ms	1012.090 ms	1131.732 ms
140	736.101 ms	1141.018 ms	2105.009 ms
175	755.821 ms	1824.261 ms	2719.989 ms

Table 5.2: Latency percentiles for `LOGIN` operations.

Servers	**`LOGOUT` p50 latency**	**`LOGOUT` p95 latency**	**`LOGOUT` p99 latency**
105	365.972 ms	740.487 ms	1081.462 ms
140	443.303 ms	905.167 ms	1427.415 ms
175	591.772 ms	3079.142 ms	5307.941 ms

Table 5.3: Latency percentiles for `LOGOUT` operations.

Servers	**`QUERY_AVG` p50 latency**	**`QUERY_AVG` p95 latency**	**`QUERY_AVG` p99 latency**
105	173.064 ms	489.750 ms	1211.930 ms
140	185.184 ms	731.192 ms	1299.095 ms
175	186.661 ms	819.088 ms	1299.523 ms

Table 5.4: Latency percentiles for `QUERY_AVG` operations.

Servers	**`WRITE` p50 latency**	**`WRITE` p95 latency**	**`WRITE` p99 latency**
105	344.935 ms	706.655 ms	1386.174 ms
140	381.992 ms	975.072 ms	1565.018 ms
175	422.520 ms	1163.695 ms	1824.909 ms

Table 5.5: Latency percentiles for `WRITE` operations.

Figure 5.2 shows the performance degradation caused by the increase in the number of active application instances. More active applications result in more clients being imposed prohibitive latency times. Figure 5.3 shows the *Latency* distribution, which lets us analyze the tail latency for each setting.

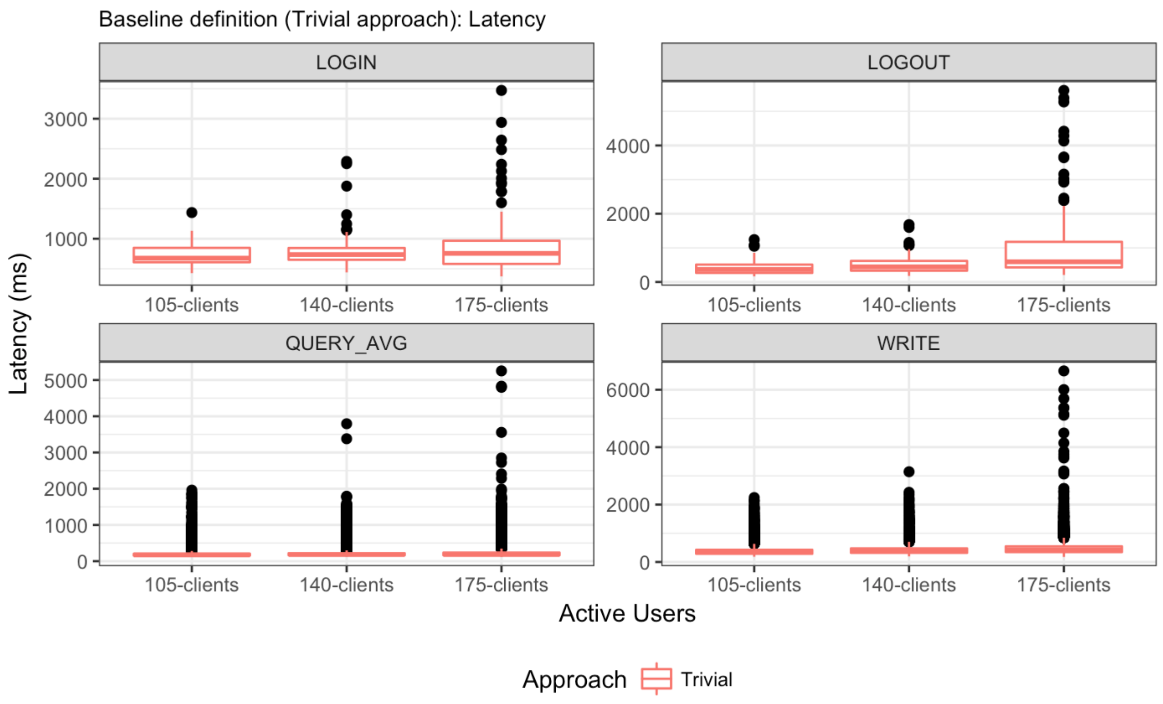

Figure 5.2: Baseline: per-operation latency (Trivial approach).

As seen in Figure 5.3, the performance of 15 servers per wrapper presents the best latency curve for all operations, being the most reasonable when it comes to the test infrastructure. Therefore, we consider 105 servers to be the total amount of servers (and users) in this experiment.

Resource-wise, we have a similar landscape. Since SCONE enclaves are statically allocated at startup time, we expect memory usage to be constant for a given amount of running servers. The application instances inside enclaves are not allowed to allocate more memory. The memory usage is shown in Table 5.6. CPU utilization graphs (Figure 5.4) show a clear increase in CPU time with more running servers, which is also expected.

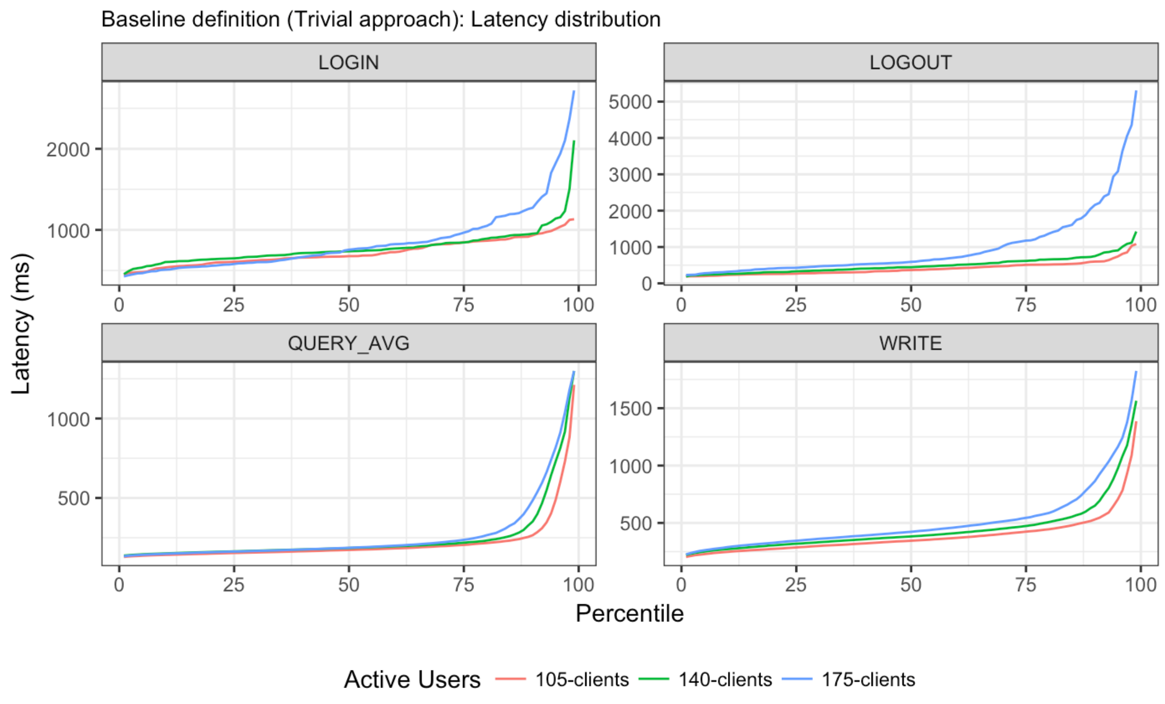

Figure 5.3: Baseline: per-operation latency distribution (Trivial approach).

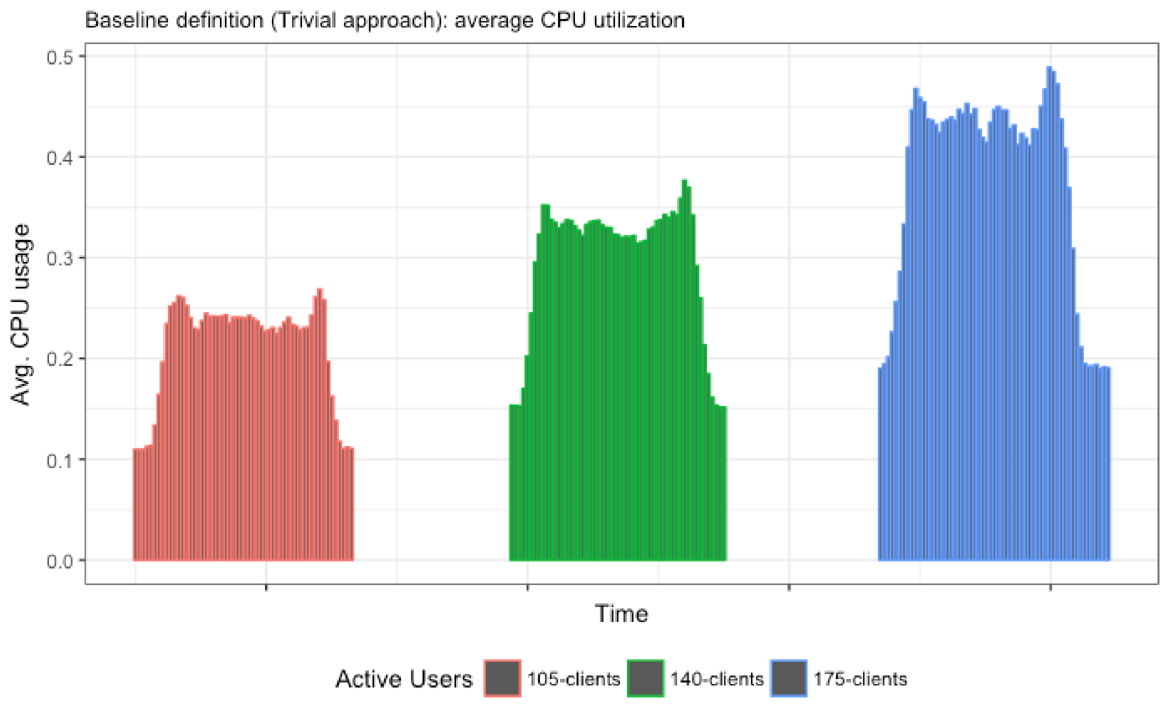

Figure 5.4: Baseline: average CPU utilization (Trivial approach).

Servers	Allocated memory
105	13.28 GB
140	17.78 GB
175	23.02 GB

Table 5.6: Baseline: total memory allocated to application instances in the cluster.

5.2.3 Discussion

Routing Manager resource impact

We evaluate the impact of the Routing Manager with respect to it resource consumption. The Routing Manager is deployed as a single-instance *Deployment* in Kubernetes. Since only one instance is needed to serve the entire cluster, the resource usage of the Routing Manager is remarkably low. Its implementation, in C, and the fact that it does not run inside of SGX enclaves also contribute to the low resource footprint. The average CPU usage is under 1%, whilst the average memory usage do not go beyond 4 MB. Figure 5.5 shows a typical resource usage profile for the Routing Manager during a benchmark.

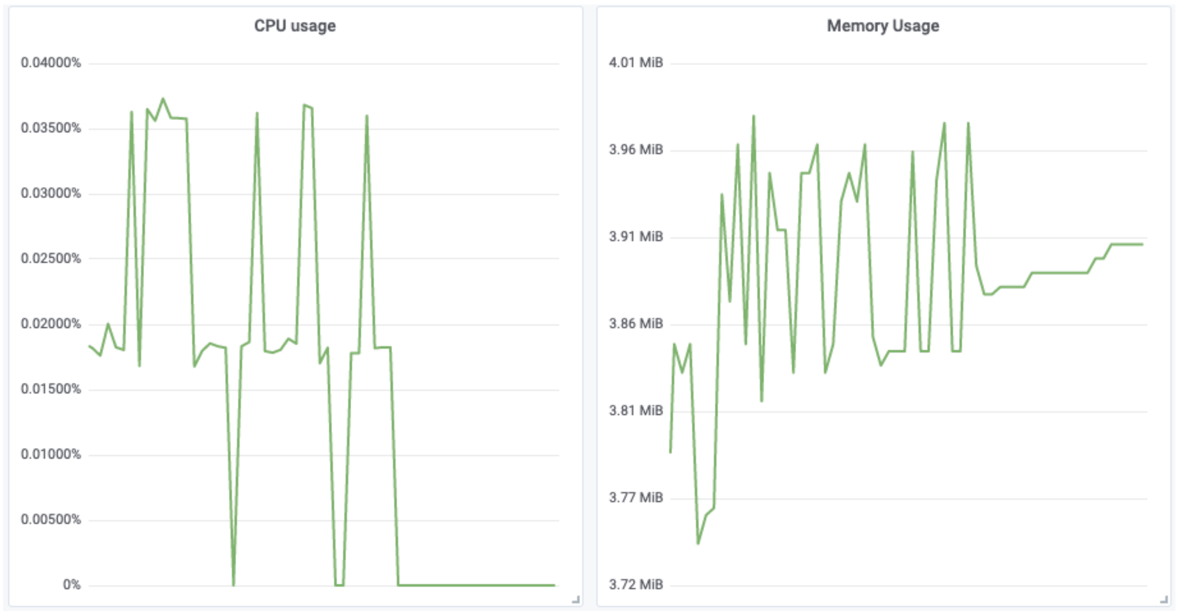

Figure 5.5: Typical Routing Manager resource footprint during benchmark.

Per-operation performance, Type 1 clients

To evaluate and compare the performance of both approaches for Type 1 clients (*i.e.*, smart meters and sensors), we execute a LONG_WRITE operation to simulate a smart meter pushing metrics to the server. When acting as a Type 1 client, the benchmark performs a LOGIN, a LONG_WRITE and a LOGOUT. The LONG_WRITE operation simulates metrics for different periods being pushed: 1h, 3h and 12h. In the three scenarios, simulated metrics are collected every 15 seconds. We use an Active User Ratio of 100% (*i.e.*, all 105 users). Note, however, that the *Session* is much shorter, which reduces the overlap between clients and the overall load at the servers. Figure 5.6 compares the *Latency* for different accumulated metric periods. The performance between the two approaches is comparable.

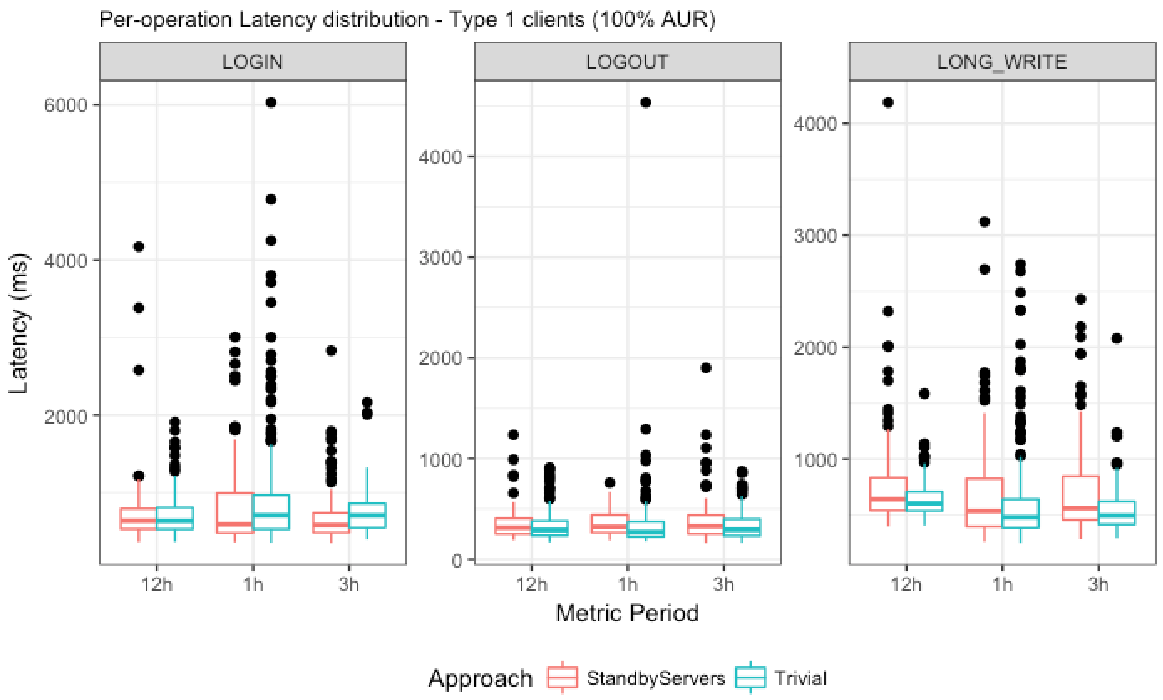

Figure 5.6: Latency distribution for Type 1 clients - 100% AUR.

Per-operation performance, Type 2 clients

We start by comparing the per-operation *Latency* for the different values of Active User Ratio (AUR). AUR defines the amount of active user states during the benchmark. Although we consider AUR values of 50%, 25% and 5%, the actual amount of servers and clients is rounded up to avoid imbalance issues. The servers are equally distributed among different nodes for each experiment.

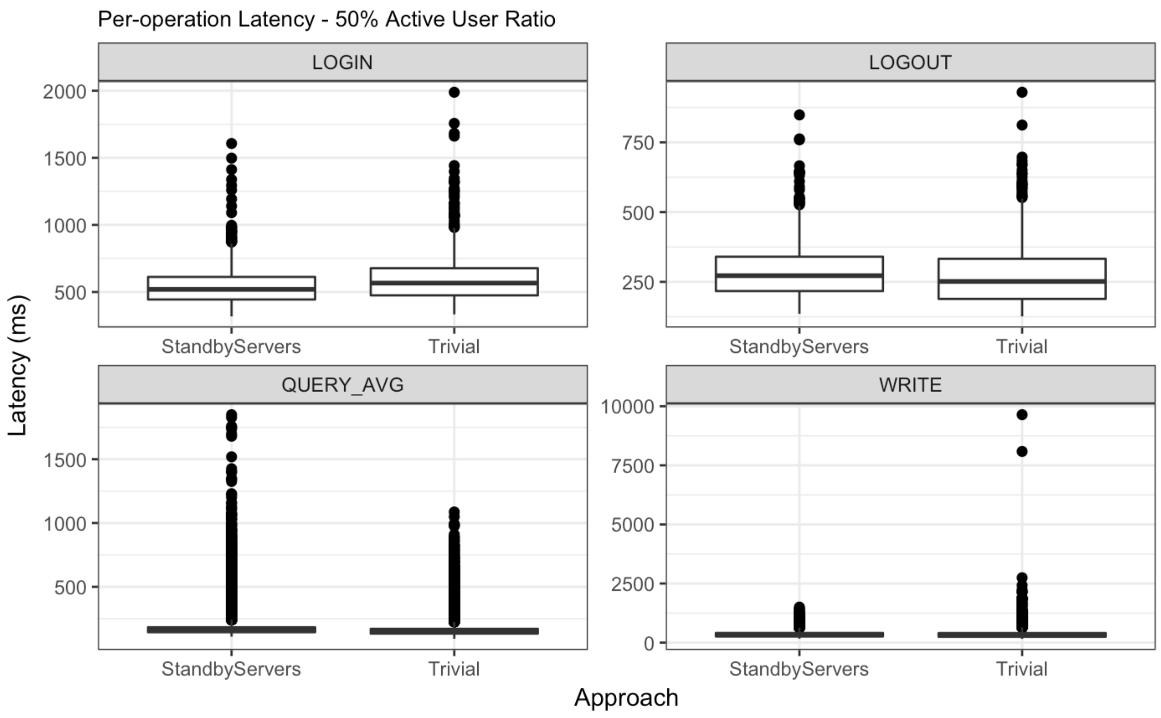

Figure 5.7: Latency distribution for 56 clients, or 50% Active User Ratio.

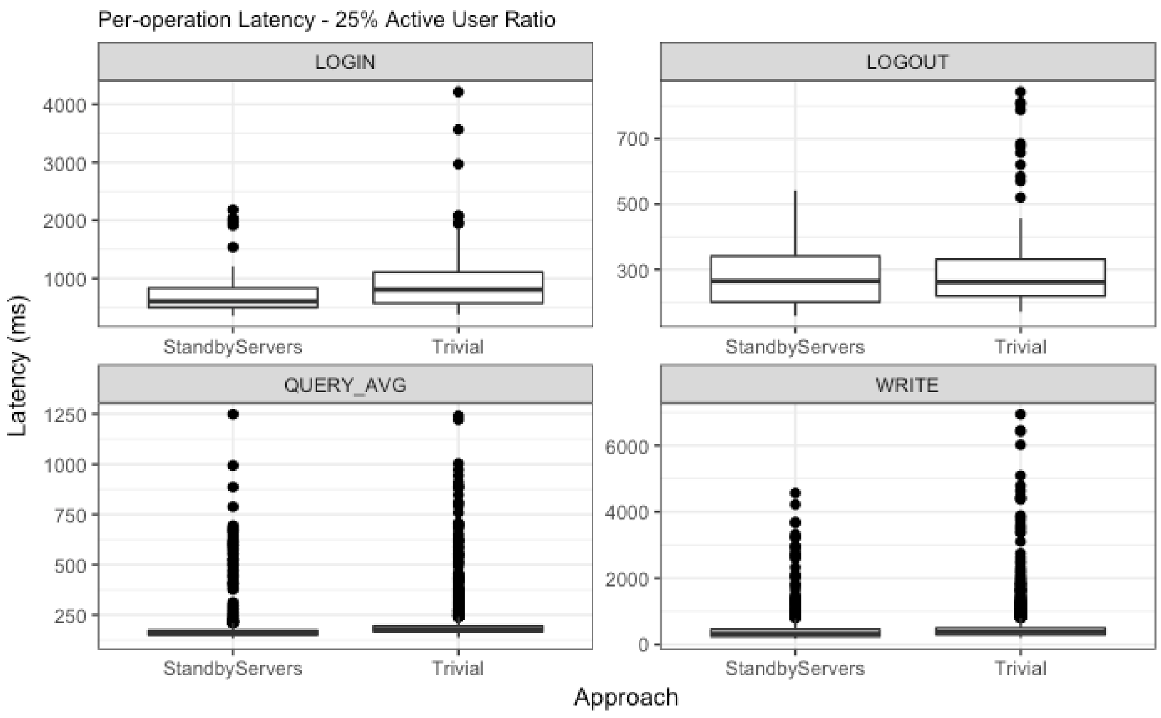

Figure 5.8: Latency distribution for 28 clients, or 25% Active User Ratio.

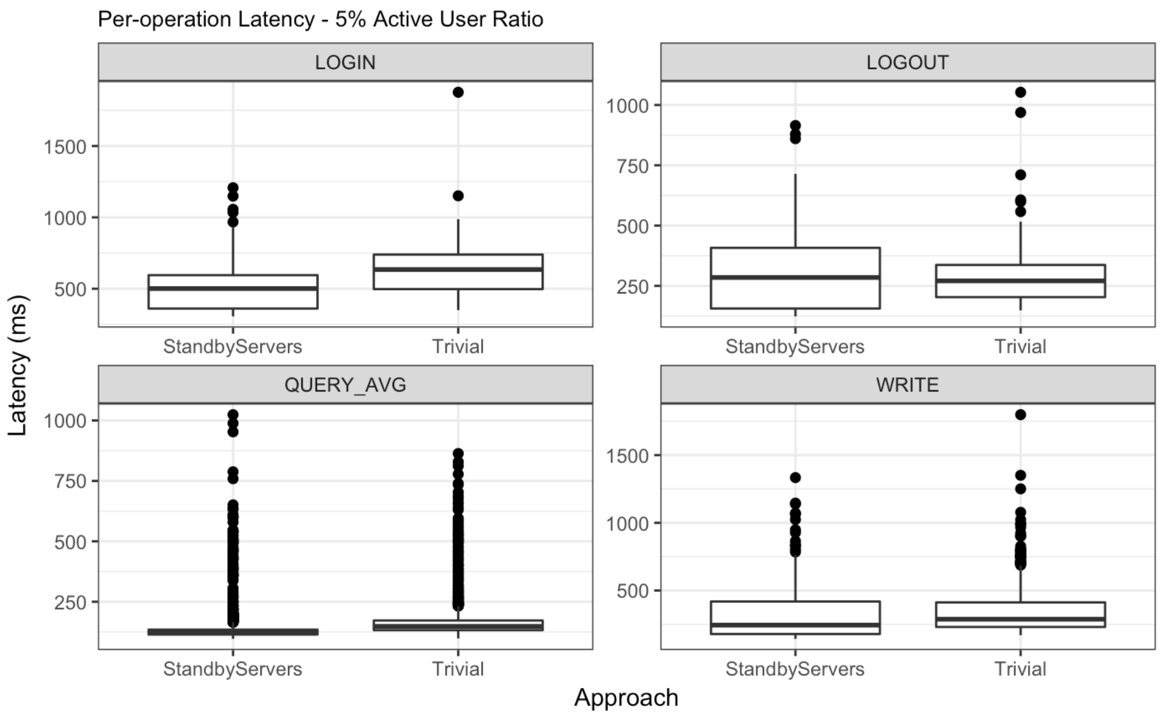

Figure 5.9: Latency distribution for 7 clients, or 5% Active User Ratio.

As per Figures 5.7, 5.8 and 5.9, the per-operation *Latency* faced by clients of Type 2 is comparable for both *Standby Servers* and *Trivial* approaches. The boxplots show many outliers in all observed AUR configurations. It is important to note that the clients are not located in the same physical network as the servers. Although this arrangement is more realistic, it is also more prone to interference in the observed *Latency* (external network instabilities, for example).

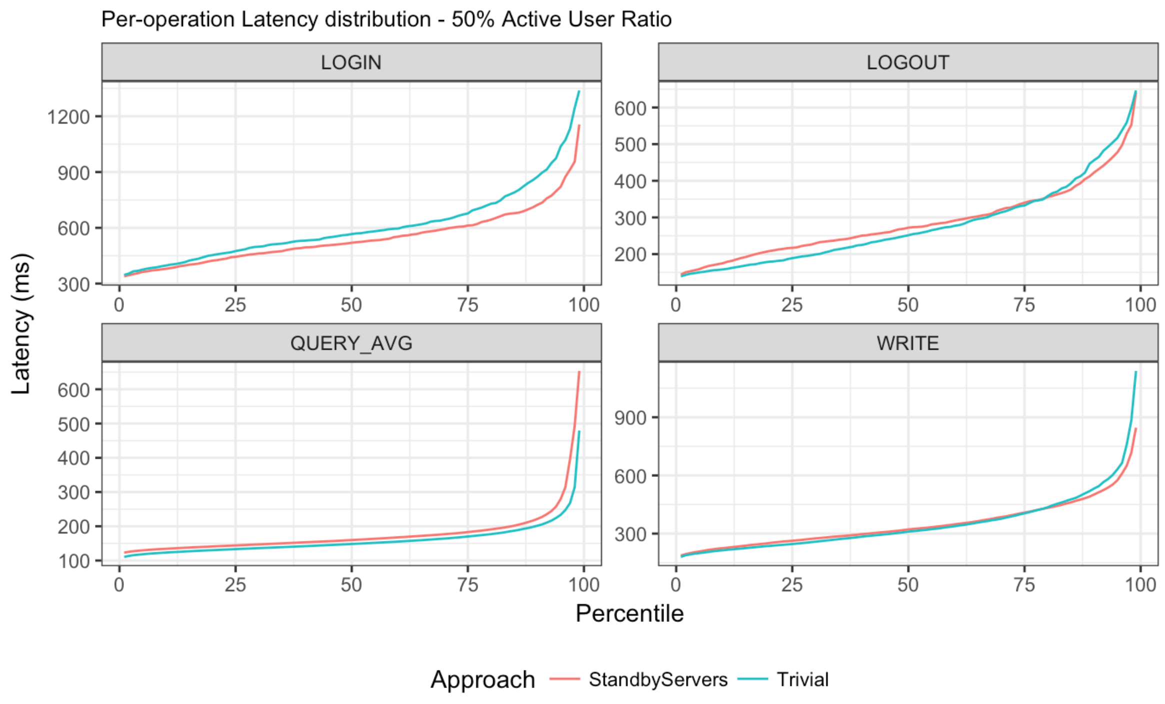

Figure 5.10: Latency curves for 56 clients, or 50% Active User Ratio.

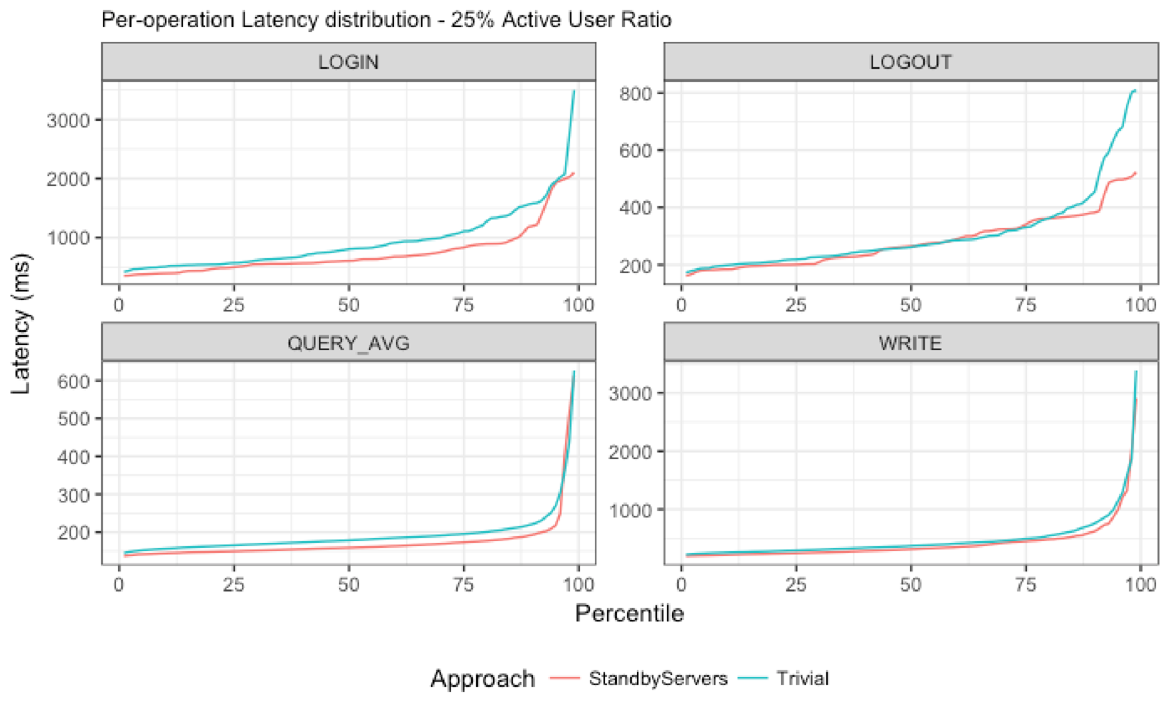

Figure 5.11: Latency curves for 28 clients, or 25% Active User Ratio.

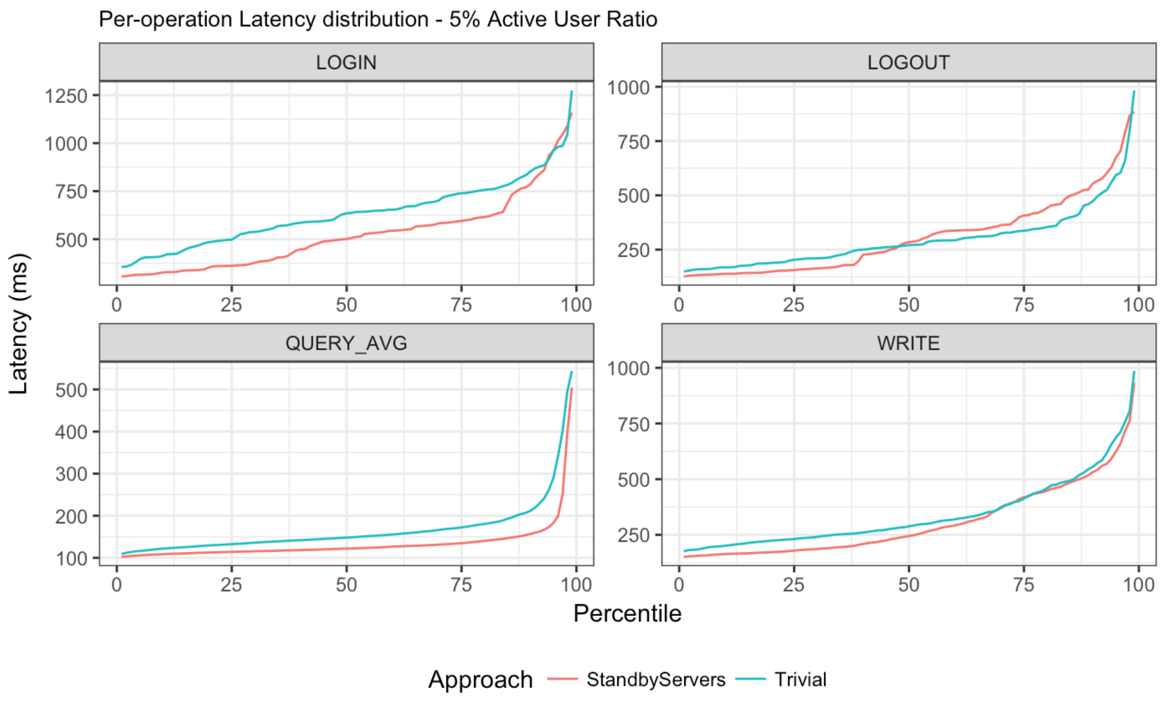

Figure 5.12: Latency curves for 7 clients, or 5% Active User Ratio.

The *Latency* curves allow in Figure 5.10, 5.11 and 5.12 allow to evaluate the per-operation tail *Latency* for each AUR configuration. Overall, the *Standby Servers* approach behaves comparably to the *Trivial* approach, and most operations show a slightly better curve, even when considering the additional overhead of querying the Routing Manager.

When the *Latency* measurements are split by experiment level (each one corresponding to different usage patterns), the *Standby Servers* approach is also comparable to the *Trivial* counterpart (Figure 5.13). Each color represents a different combination of *Frequency* (F) and *Write Requests Ratio* (WR).

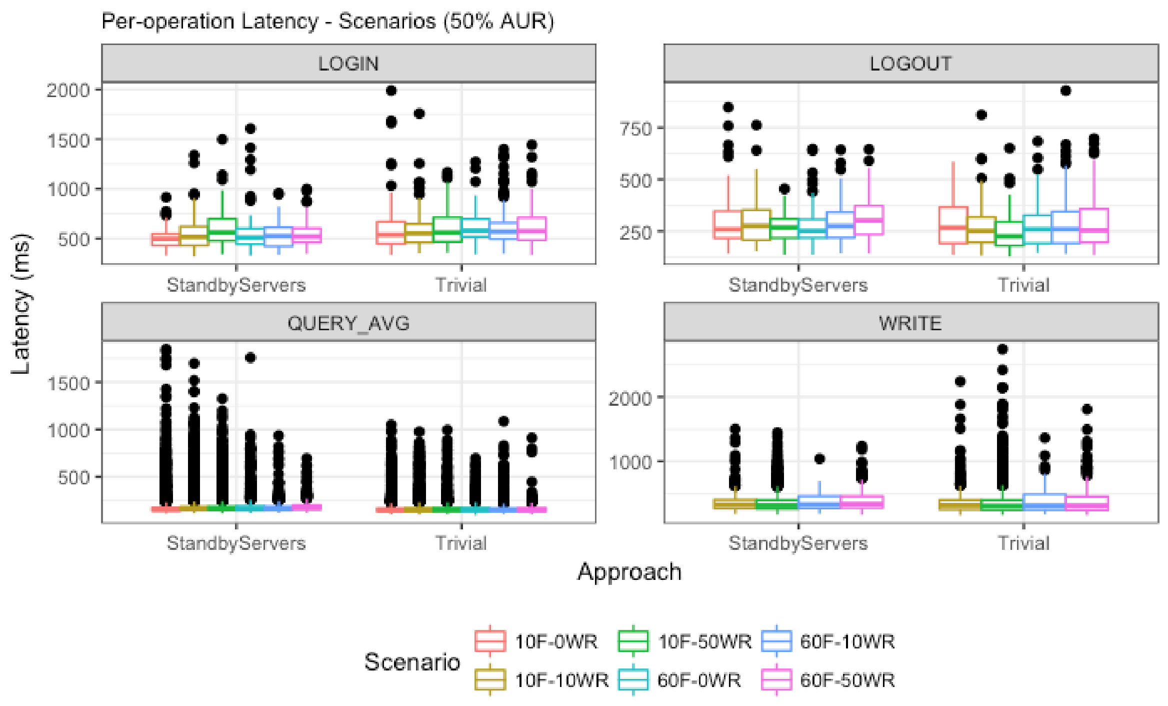

Figure 5.13: Latency for different usage scenarios (50% AUR).

Since the enclave sizes are fixed, having them managed dynamically to serve only active users is more resource-efficient. Figure 5.14 compares the total memory utilization for the server instances in the cluster, as reported by the Docker engine through cAdvisor. Total memory of the cluster is 56 GB (8 GB x 7 nodes). The value shown in the vertical axis, *Allocated memory (GB)*, is the sum of the memory allocated by server instances in each node. The observed reduction in memory utilization provided by the *Standby Servers* approach ranges from 44.80% (for an Active User Ratio of approximately 50%) to 91.71% (for an Active User Ratio of approximately 5%).

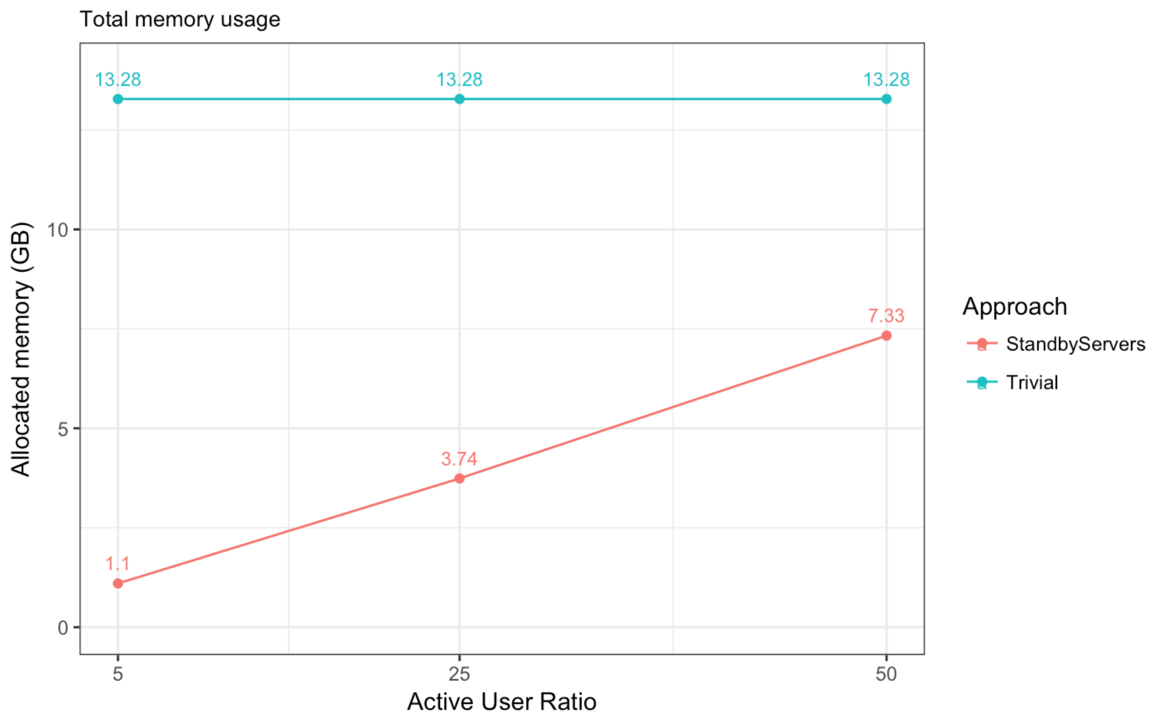

Figure 5.14: Total memory allocated to application instances in the cluster.

Having less active server instances also contributes to less EPC memory usage. Figures 5.15 and 5.16 compare the amount of free, allocated and evicted pages during a typical deployment (25% AUR). The *Standby Servers* approach uses more than 50% less pages than its *Trivial* counterpart.

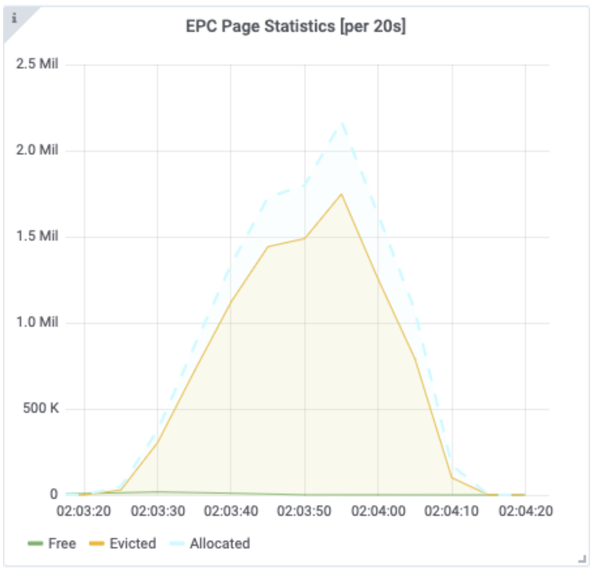

Figure 5.15: EPC pages for Trivial approach.

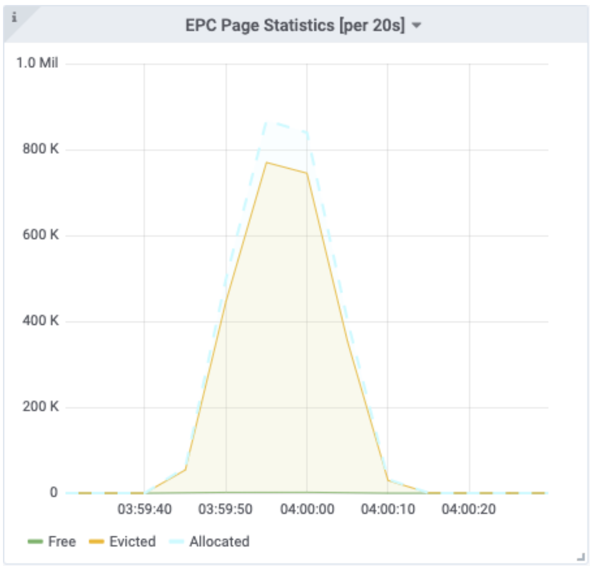

Figure 5.16: EPC pages for Standby Servers approach (50% AUR).

Based on the observed latency distribution and average resource consumption profile, the *Standby Server* approach presents itself as viable alternative to deploy applications that fit into the architecture, and whose requirements are satisfied by it. Its performance is similar or better than the alternative *Trivial* approach in almost all tested scenarios, whilst using less computing resources (especially memory and EPC) and mitigating the cold-start issue (since requests are forwarded only to servers that are already running). The *Standby Servers* approach also supports much larger user bases, since it only focus on the amount of active users at any moment. An application with 10000 users could be served by the *Standby Servers* approach in our test infrastructure, for example, for an Active User Ratio of 1% (100 active users), whilst the *Trivial* approach would require a much larger infrastructure to achieve the same.

Moreover, all experiments were run on Kubernetes, the industry-standard container orchestration tool, which allowed a much simpler and more straightforward infrastructure management. Other popular tools, such as Prometheus and Grafana were also taken advantage of in the task of monitoring resource consumption and even TEE metrics, thanks to the TEEMon framework. The uniform API provided by Kubernetes also ensures that the architecture can be deployed to any other Kubernetes cluster with Intel SGX support.

5.3 Threats to validity

The work has the following threats to validity.

- The experiments are prone to interference due to the fact that clients are not in the same physical network as the servers.
- The experiments do not consider extra SCONE tuning parameters, such as number of queues, ethreads and sthreads, which can yield significant performance improvements.
- The experiments consider only one application. It is not straightforward to estimate the performance for applications with different access patterns.

Chapter 6

Final remarks

6.1 Conclusion

This work presented an approach to deploy applications with a strict set of confidentiality, integrity and isolation requirements. These requirements include end-to-end communication encryption and data-at-rest encryption. The *Standby Servers* approach allows the dynamic provisioning of such application instances, that run inside of Intel SGX enclaves, aiming at attenuating some of the overheads imposed by running inside of a TEE. The approach leverages a stateful routing mechanism powered by off-the-shelf load balancer solutions and a Routing Manager to dynamically assign incoming users to server instances that are ready and running, but do not hold any particular state. With the help of an Application Wrapper, which manages the application instances, the (encrypted) state is dynamically loaded into the applications' file system. After the client and server establish a secure communication channel and the data is provisioned, the client provides the decryption key, and the server starts a *Session*, serving all the subsequent requests from that same user within the *Session* duration.

The architecture is generic enough so that different applications can be integrated to the Application Wrapper with little implementation effort.

This work also evaluated the approach applicability in the context of industry-standard cloud infrastructure tools, such as Kubernetes, which provides simplicity, portability and a rich and well-established ecosystem of tools and services. The approach presented itself as viable to be implemented, with comparable end-user latency distributions and a much lower resource footprint, especially for memory and Enclave Page Cache, when compared to the alternative, static approach of having all the application instances running at all times. This dynamic provisioning of stateful, isolated instances also allows large user bases to be served by rather small infrastructures, since we only need to take into account the active states at any time (*i.e.*, active users).

Furthermore, the following publications were produced, directly or indirectly, within the context of this research work, some of them in international collaboration.

- *"Implementing Quality of Service and Confidentiality for Batch Processing Applications"* [3], on the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion;
- Demo *"Asperathos: Running QoS-Aware Sensitive Batch Applications with Intel SGX"* [16], on the 37th Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos;
- *"TEEMon: A continuous performance monitoring framework for TEEs"* [12], on the 21st ACM/IFIP International Middleware Conference (accepted, to be published);
- Tutorial *"Processamento confidencial de dados de sensores na nuvem"* [7], on the 20th Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais.

Finally, this work also led to the presentation of the technical talk *"Secure Data Analysis with OpenStack and Asperathos"* at an international computer industry conference, the 2019 OpenInfrastructure Summit[1].

6.2 Future work

This work evaluated a complex architecture. Naturally, there are many interesting paths that deserve being further investigated and evolved. A few of them are listed below.

- **Explore auto-scaling.** One of the limitations of the *Standby Servers* approach is that the number of application instances is fixed per wrapper. However, the Wrapper offers a REST API that allow an external entity to start and stop application instances. This way, the Wrappers could be managed by an auto-scaling controller, for example, that could add or remove Wrappers (*e.g.*, in the face of an infrastructure or topology change). Furthermore, the Wrappers themselves can be extended to listen to events (*e.g.*, the application could log response times for each request) and take scaling decisions based on them, in order to meet previously defined performance constraints.

[1]https://www.openstack.org/summit/denver-2019/summit-schedule/events/23579/secure-data-analysis-with-openstack-and-asperathos

- **Support generic applications seamlessly.** Although the architecture is generic enough to accommodate different applications, some implementation effort is still needed to make the sidecar threads listen to the right events. The Application Wrapper can be extended to allow these events (and respective actions) to be described in a declarative way.
- **Compare to serverless.** Compare the advantages and disadvantages of this approach with that of serverless architectures. Dynamic loading code to enable more efficient resource usage is also a goal of the serverless paradigm. Generalizing the applications supported would also enable a thorough comparison with this alternative paradigm.
- **Profiling and optimization.** Profile each component of the architecture in order to find places for optimization. For instance, understanding the impact of the load balancing solution to overall latencies, and finding ways to improve it, such as using sockets more efficiently, via a connection pool, or even bundling load balancers and Routing Manager to attenuate round-trip times during the routing decision.

6.3 Acknowledgments

This work was partially funded by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) through the Master's program at the Departamento de Sistemas e Computação at Universidade Federal de Campina Grande. The current use case and architecture has been developed within the project UFCG-EMBRAPII ScalableSGX in a research collaboration between Universidade Federal de Campina Grande (UFCG), Scontain U.G., and Technische Universität Dresden (TU Dresden). The Routing Manager implementation is a contribution by Sergei Arnautov (Scontain U.G.). Fábio Silva (UFCG) contributed to purging mechanisms in the Intel SGX driver. Gabriel Vinha (UFCG) contributed mainly to the implementation of monitoring approaches for SGX enclaves. Lucas Cavalcante (UFCG) contributed mainly to the dynamic provision of client data using Ceph RBD. Although some of these aspects were not included in the scope of this work, their participation in the project provided continuous feedback.

Bibliography

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. Citeseer, 2013.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[3] Igor Ataide, Gabriel Vinha, Clenimar Souza, and Andrey Brito. Implementing quality of service and confidentiality for batch processing applications. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 258–265. IEEE, 2018.

[4] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[5] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[6] Eric A Brewer. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 167–167, 2015.

[7] Andrey Brito, Clenimar Souza, Fábio Silva, Lucas Cavalcante, and Matteus Silva. Processamento confidencial de dados de sensores na nuvem. *Minicursos do XX SBSEG*, 2020.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[9] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[10] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[11] Matt Klein. Lyft's envoy: Experiences operating a large service mesh. 2017.

[12] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. TEEMon: A continuous performance monitoring framework for TEEs. In *Proceedings of the 21th International Middleware Conference*, page to be published, 2020.

[13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[14] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel sgx and amd memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[15] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[16] Lília Rodrigues Sampaio, Clenimar Souza, Gabriel Silva Vinha, and Andrey Brito. Asperathos: Running qos-aware sensitive batch applications with intel sgx. In *Anais Estendidos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 89–96. SBC, 2019.

[17] Carlos Segarra, Ricard Delgado-Gonzalo, Mathieu Lemay, Pierre-Louis Aublin, Peter Pietzuch, and Valerio Schiavoni. Using trusted execution environments for secure stream processing of medical data. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 91–107. Springer, 2019.

[18] George Teodoro, Tulio Tavares, Bruno Coutinho, Wagner Meira, and Dorgival Guedes. Load balancing on stateful clustered web servers. In *Proceedings. 15th Symposium on Computer Architecture and High Performance Computing*, pages 207–215, 2003.

[19] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 645–658, 2017.

[20] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213, 2018.

[21] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, 2008.