IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier 10.1109/ACCESS.2017.DOI

The Continuum of Computing

Giovanni Jiayi Hu
[1]Department of Mathematics, University of Padua, Italy I-35121

ABSTRACT Work in Progress.

I. INTRODUCTION

The Internet has evolved significantly since its inception. It has grown into a ubiquitous platform for everyday services from just a simple communication layer for information sharing between researchers. Many changes and infrastructure trends drive this transformation.

First, during the past decade, the sharing of server and networking capabilities - known as the cloud computing paradigm - has become a reality, giving users and companies access to virtually unlimited amounts of storage and computing power. Nowadays, there is an immense amount of seamlessly connected mobile devices, servers, and network components that offer their virtualised capabilities to their users.

Second, as mobile computing evolved, people started bringing their devices with them and accessing the Internet anywhere and anytime. Nowadays, we are amid the so-called Internet of Things (IoT), where devices (things) are connected to the Internet and each other. These "things" comprise a multitude of heterogeneous devices ranging from consumer devices, such as mobile phones and wearables, to industrial sensors and actuators [23], and smart transportation, grids [21], and cities [22].

Thus, it is natural that as consumers, we want our Internet-capable devices (e.g., mobile phones, thermostats, electric vehicle) to adapt seamlessly to our changing lives and expectations, regardless of location [12].

Besides, our capacity for collecting data is expanding dramatically. Nevertheless, our ability to manage, manipulate, and analyse this data to transform it into information and act upon it has not kept the pace [11]. Data sources and their volumes of generation are growing exponentially and have outpaced the Internet ability to transport this data in a reliable and timely manner to the cloud. IoT applications might require a short response time, involve private data, and produce a large quantity of data which could be a heavy load for networks. Cloud computing and Internet infrastructure are not capable enough to support these applications and their exploding multitude.

In the vision of this thesis, the future holds us an information-rich pervasive computational continuum that seamlessly combines this data and computing power to model, manage, control and make use of virtually any realisable sub-system of interest [11]. Use cases exist in diverse application areas, from managing extreme events (e.g. environmental monitoring) to optimising everyday processes (e.g. manufacturing) and improving life quality (e.g. healthcare and smart cities).

A natural consequence of this vision is that computational capabilities are gradually being introduced in edge networks during recent years. New approaches that combine distributed services close to the data sources (i.e. edge nodes) with resources in the cloud and along the data path can constitute a computing continuum and effectively process this data. Depending on the use case and service level requirements (SLR), user applications may require processing and storage locally, in the cloud, or somewhere between (fog).

This trend paves the way for novel and ubiquitous services in a wide range of application domains. To cite Haller et al. [1], this thesis believes that we are close to "a world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes. Services are available to interact with these 'smart objects' over the Internet, query their state and any information associated with them, taking into account security and privacy issues."

This thesis proposes a vision that follows naturally from the previous premises - the **continuum of computing** - a ubiquitous system where distributed resources and services on the whole computing continuum are dynamically aggregated on-demand to support different ranging services.

In this thesis's vision, there will be pervasive service platforms anywhere the user is and a multitude of services available over the Internet. These services' granularity will be very different, ranging from high-level business services to low-level sensor services provided by the Internet of Things.

It is fundamental that services are treated as first-class citizens, allowing services to be dynamically placed by decoupling them from their location. The continuum must also facilitate the elastic provisioning of virtualised end-to-

end service delivery infrastructures. Virtual capabilities are leased, dynamically configured and scaled as a function of user demand and service requirements. The continuum must behave as a fluid, which continuously adapts its shapes to fit the surroundings, as first described in [10].

However, this vision goes beyond the traditional elasticity of clouds. In addition to computational, network and storage resources, the managed capabilities include end-to-end network configuration and high-level service and device functionalities as well.

The notion of "the continuum of computing" is not entirely novel and has been anticipated before. The authors of [10] envision a Fluid Internet, which "seamlessly provisions virtualised infrastructure capabilities, adapting the delivery substrate to the dynamic requirements of services and users, much like a fluid adapting to fit its surroundings". [11] also presents the notion of computing in the continuum, that is, "realising a fluid ecosystem where distributed resources and services are programmatically aggregated on-demand to support emerging data-driven application workflows". Similarly, authors of [12] present the continuum of computing and seek to develop approaches that include the entire computing continuum as a collective whole.

Both cloud computing and IoT topics gained popularity in the last decade, and the number of papers dealing separately with cloud and IoT has been showing an increasing trend since 2008. More recent works like [17] focus on integrating the cloud and IoT but keeping the two as distinct spaces where the latter sends data and offloads computation to the former. The few publications mentioned in the previous paragraph are the only ones to foresee a continuum of computing encompassing the whole Internet as a substrate for ubiquitous services. Unfortunately however, to the best of the author's knowledge, very few effort has been carried out on the development of platforms to make this vision possible and optimized.

The remaining parts of this paper are organised as follows. Section II discusses the present trends which lead to the continuum and presents the challenges of this vision. Section III shows the problem space addressed by this thesis and the technology baseline. Section IV contains the evaluation of the architecture presented in the preceding section and, lastly, the thesis concludes in Section V with the final remarks.

II. THE PROBLEM STATEMENT

In the past decade and now more than ever, cloud computing has provided users with the potential to perform computing tasks utilising resource physically distant. A popular definition of cloud computing [2] is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

The successful cloud architecture can be split into four layers: datacentre (hardware), infrastructure, platform, and application [18]. Each of them can be seen as a service for the layer above and as a consumer for the layer below. The resulting model is divided into mainly three types of service: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

Conventional single provider infrastructures hosting cloud services offer undoubtedly many benefits but not without challenges. A large data centre's energy consumption is high to keep it operational, and like any other centralised computing model, the resulting issues would be adverse in case of a failure.

Another issue is that the required data must be transferred and stored to separate places from the source. Data centres are often geographically distant from the application users, notably when the data is generated at edge locations. However, the exchange of sensitive or personal data is considered critical and private for applications.

Besides, the evolution of the Internet of Things is having a significant impact on cloud computing generally, and it is stretching the limitations of the latter. The number of connected devices increases exponentially with estimations of dozens of billions of "things" going live in coming years [24].

As a consequence of connecting these sensors to the Internet, large volumes of data are being generated at unprecedented volumes, variety and velocity. This data is currently transferred and stored in the cloud in a centralised manner. Data transfer, especially in these volumes, is costly and retards computational performance.

Thus, it becomes evident that traditional data processing methods where all data is collected and processed in a central instance will not suffice. A more decentralised solution is required where data processing could take place before transfer and storage. The key is to reduce the number of messages and the amount of data transmitted throughout the continuum so that data should be evaluated locally or only where it makes sense.

With these premises, in this thesis's vision, we shall apply the cloud computing definition as mentioned above to the Internet of Things and the network along the data path (fog and edge). The rationale is two-fold. First, it offers on-demand virtual capabilities regarding storage, memory and processing units that augment IoT devices and components with limited computation capabilities due to form factors. Note, however, that fog and edge nodes cannot scale "infinitely" as the cloud data centres.

Second, scaling fog and edge infrastructure is costly and time-consuming, even more as in cloud infrastructure, due to additional factors such as heterogeneity, network unreliability and difficulty to predict capacity needs in advance. Underprovisioning means potentially losing business, while overprovisioning means wasting money on unused infrastructure. Moreover, user demand at the edge can be very often spiky, meaning that companies would need to provision for anomalous peaks like gatherings and events, investing in significant infrastructure that sits underutilised most of

the time. The overprovisioning also has an environmental cost when underutilised infrastructure consumes significant amounts of power.

The cloud computing model applied to the fog and edge as services is attractive since it allows businesses to decrease the required capital expenditure (CAPEX) and frees them from the need to invest in the infrastructure. Businesses can rent resources according to their needs and only paying for the usage [17]. Moreover, it reduces operating costs (OPEX), as service providers do not have to provision capacities according to maximum peak load. Resources are released when service demand is low.

On the other hand, the cloud can benefit from the fog and edge by extending its scope to deal with real-world things in a more distributed and dynamic manner and delivering new services in a large number of real-life scenarios. Such an extension will impact future application development, where this new flow of information gathering, processing, and transmission will generate new challenges.

A. DECENTRALIZATION

As stated before, highly distributed networks are the most effective architecture for the continuum, particularly as services become more complex and more bandwidth-hungry. Although often referred to as a single entity, the Internet is actually composed of thousands of different networks. The net result is that content generated at the edge must travel over multiple networks to reach its centrally-hosted data centre.

Unfortunately, inter-network data communication is neither an efficient nor reliable operation and can be adversely affected by a number of factors.

Capacity at peering points where networks exchange traffic typically lags demand mainly due to the economic structure of the Internet [8]. The economic incentive flows in at the first mile (cloud data centres) and at the last mile (IoT), with however very little interest to invest in the middle network composed of peering points. These points become thus the cause for packet loss and increase latency.

Across the Internet, outages are happening all the time, caused by a wide variety of reasons such as cable cuts, misconfigured routers, DDoS attacks, power outages, or even earthquakes and other natural disasters. While failures vary in scope, large-scale occurrences are not uncommon [9].

When dealing with a pervasive system such as the continuum, it is therefore no longer sufficient to simply have enough server and network bandwidth resources. One must consider the throughput of the entire path from IoT devices to data centres to end-users. The bottleneck is not likely to be at just the far ends of the path. It could be at a peering point, as mentioned, or due to the network latency between server and device.

Autonomous vehicles are a great example of these issues. One Gigabyte of data will be generated by the car every second, and it requires real-time processing for the vehicle to make correct decisions [13]. Network bandwidth and reliability can be quickly challenged for their capability of supporting a large number of vehicles in one area. Moreover, if all the data were to be sent to the cloud for processing, the response time would be too long. In such cases, the data needs to be processed at the edge for shorter response time, more efficient processing and smaller network pressure.

Secondly, in many other cases, most of the end nodes in IoT (e.g. sensors and actuators) are energy-constrained things, so offloading some computing tasks to the edge could be more energy efficient. Even end mobile phones benefit significantly from reduced energy consumption, especially for tasks like Mobile Augmented Reality [25].

For these reasons, alternate models such as fog computing and edge computing have emerged in recent years. A fog and edge-based platform, with servers anywhere the end device, can achieve the scale needed. Each location supports higher orders of throughput, low response times, and higher energy efficiency.

However, the nearest physical layer may not always be a good option. If the task requires long computation, as in big-data analysis, it will be better to offload to the more distant but capable node or even the cloud. A request's latency is the sum of two components: the computing latency and the transmission latency. A high computing latency can outweigh the transmission efficiency. Therefore, edge computing's wish is to determine the ideal tradeoff between computing latency and transmission latency, leveraging the whole continuum.

Similar attention must be dedicated to the energy issue. The battery is the most precious resource for things at the edge of the network, but the wireless communication module is usually very energy-hungry [13]. For a given workload, it is not always the case that the most energy-efficient solution is to offload the whole workload to the edge rather than compute locally. The key is the tradeoff between computation energy consumption and transmission energy consumption. Disburdening to another node is preferable only if the latter overhead is more negligible than computing locally.

It shall be clear by now that the vision of a computing continuum is not that one form of computing (e.g. edge computing) will supplant another (e.g. cloud computing), but that we must try harnessing the entire computing space. Some trending computing models relevant to the decentralisation above are not presented.

1) Fog Computing

Fog computing [20] is a decentralised computing infrastructure that is used mainly as a complement to cloud computing. It leverages the compute resources at the edge network and brings the computational processing closer to the data source by offloading workload to edge nodes from cloud data centres. The network nodes near the edge providing these resources are called fog nodes and they are usually a single hop away from the end-users.

Overall, any device with computing, storage and network connectivity can constitute a fog node, such as switches and routers, industrial controllers, embedded servers and video

surveillance cameras [16]. In the recent years, the spread of fog computing has been facilitated by the availability of suitable hardware in the form of small, affordable, low-power computers (e.g. Raspberry Pi [52]), along with improvements in virtualisation technologies that enable slicing resources between different users to provide isolated environments (see Section §II-B3).

The Cloudlet [7] architecture is one of the first approaches that offer an example of fog computing with virtual machine (VM) techniques. Its computing power is much less than a conventional cloud infrastructure as cloudlets are composed of less powerful processors and are significantly smaller in size.

2) Mobile Edge Computing

Edge computing [13] is any computing and network resource that takes place only on the edge of the network. The rationale of edge computing is that computing should happen in the proximity of data sources, eliminating the costly data transfer to a remote data centre. The result significantly improves user Quality of Service (QoS) as, similar to fog computing, and there are considerable network latency reduction and bandwidth consumption by the end-users.

When the end-users are mobile devices, edge computing is referred to as mobile edge computing (MEC) as well. In contrast to the broader term edge computing, MEC focuses on co-locating computing and storage resources at base cellular networks stations. Being co-located at base stations, MEC servers' computing and storage resources are available close to mobile users like smartphones and smart vehicles and can support mobility between cellular cells. MEC is seen as a promising approach to increase the quality of experience in cellular networks and a natural direction for the evolution to 5G networks [26].

3) Serverless Computing

Serverless computing [27] involves building, running and providing applications and services without considering the server-side. "Serverless" does not mean that there is no server usage, but rather the main focus is on the application itself rather than what happens on the physical infrastructure.

Serverless computing is synonymous with Function-as-a-Service (FaaS), and event-based programming as the execution of an application will be executed only when necessary and not all the time, thus meaning that an event can trigger the execution of a function. In particular, FaaS triggers a server only when a function is requested, then executes the expected operations and terminates.

The major advantages of this model for cloud consumers are increased scalability and applications' infrastructure independency and lower costs. Similarly, cloud vendors also have strong incentives for services to be built on serverless architectures as opposed to following a fixed-price model for long-running VMs. VM deployments tend to be very static as users deploy them for long periods of time. In contrast, the FaaS computing model tends to offer more variation in memory and CPU utilisation. As a result, the serverless computation gives a better opportunity to optimise resource usage and to bill users by function invocation and network utilisation.

Recently, the main cloud vendors (e.g. Amazon and Microsoft) are putting effort into making it easier to integrate edge functionality into their clouds mainly by providing serverless computing and tight integration with their other cloud services. As example, AWS Lambda@Edge [53] allows using serverless functions at the AWS edge location in response to CDN event to apply moderate computations. On the other hand, AWS Greengrass [54] and Azure IoT Edge [55] are commercial solutions specifically targeting serverless for IoT.

Recent works in literature have shown the advantages of the FaaS model for edge computing too. The authors of [15] present an augmented reality use case for mobile edge computing, in which computation and data-intensive tasks are offloaded from the devices to serverless functions at the edge, outperforming the cloud alternative up to 80% in terms of throughput and latency.

B. CHALLENGES

Having discussed the trend towards a decentralised interconnected continuum, this section presents the challenges imposed by such a system. As an example, the heterogeneity of the connected devices is immense. It can be seen from different perspectives, such as computing performance, storage and network requirements, communication protocols, energy consumption, to name a few. Such diversity poses questions about how to deal with the virtualisation of the underlying resources and the nodes' interoperability.

In order to achieve the continuum, this thesis identifies the following challenges as the most significant. Being the edge characterized by a very high heterogeneity as just mentioned, it lacks different important properties such as service orientation, interoperability, orchestration, reliability, efficiency, availability, and security. Therefore, it is natural that many challenges in the realisation of the continuum overlap with open research problems in the edge computing. At the same time, other issues are originated, or at least exacerbated, from the holistic integration of cloud, fog, and edge computing (such e.g. providers' interoperability).

1) *Service orientation*: as stated in the introduction of this thesis, services must be treated as first-class citizens, allowing them to be dynamically placed and decoupling them from their location. Consumers should only be concerned with what they want to do and accomplish and providers with how that could be done and provided to the user;
2) *Orchestration*: advanced orchestration systems are required to support the resource management of heterogeneous devices and be adapted to many applications running on the edge connected devices;
3) *Virtualisation*: the system has to provision resources and provide access to heterogeneous IoT resources

and hardware such as GPUs. Besides, today's common practice in the cloud assumes enumerated resources and predictable computing capabilities. However, in the continuum, the capabilities and numbers of components change dramatically over time and require virtualisation solutions to provide a reliable service. Lastly, edge computing must support multi-tenancy and the virtualisation techniques are likely to be asked to deliver execution environments even more hardened than in cloud centres.;

4) *Dynamic configuration*: TODO
5) *Interoperability*: applications on a continuum should be able to amalgamate services and infrastructure from different locations seamlessly. Every node on the Internet should additionally expose a uniform interface to its resources and services;
6) *Portability and Programmability*: developers should be provided with the languages and tools to program applications and services to run in such a dynamic environment without having to reinvent themselves;
7) *Mobility*: efficient migration of each application and service has to be supported from platform to platform and to follow the users' movements in the network (roaming);
8) *Reliability*: services in the continuum could fail due to various reasons, notably at the edge. The establishment of reliable communication between nodes must be supported;
9) *Security and Privacy*: personal and sensitive user data are subjected to high risk while many users access Internet services from anywhere. Security and privacy matter even more when personal data may have to be stored closer to the users/devices to facilitate computing and processing on the edge or fog layer;
10) *Context awareness*: services need to be aware of different aspects of the environment they are acting in, whereas today's services are rarely context-sensitive;
11) *Energy efficiency*: obtaining energy efficiency in both data processing and transmission is an important open issue;

1) Service orientation

This thesis visions service-oriented paradigms as necessary for organising and utilising distributed capabilities that may be under the control of different ownership domains. A Service-Oriented Architecture (SOA) provides "a uniform mean to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations" [6].

According to this model, a basic SOA's major components and their possible interactions are constituted by a service provider that publishes his service interface via a service registry where a service requester/consumer can find it and subsequently bind to the service provider [1].

The SOA reference model's central concept is services that provide access to capabilities by well-defined interfaces to be exercised following a service contract with constraints and policies. This enables the loose coupling of services and minimisation of mutual dependencies.

Services are provided by the service providers and are to be used by others, the service consumers. Services may be composed based on other existing services, thereby adhering to the principle of reuse. They are responsible only for the logic and information they encapsulate, uniformly described and publicly retrievable via service discovery mechanisms. This design is in net contrast to today's practice. We continue to build ad-hoc programs confined in relatively small single nodes, defining individual device computing behaviours [12].

However, to enable the seamless and agile interoperation of services in the continuum, several challenges still exist concerning their organisation and implementation.

First of all, the lack of neutral, trustworthy and widely accepted service intermediaries on the Internet of today still prevents the establishment of a Continuum as described above. Service intermediaries are necessary to facilitate the efficient retrieval of services that match a given user need and provide service performance by monitoring service quality and availability. Unfortunately, so far, single cloud providers such as Amazon web Services, Google Cloud Platform or Microsoft Azure have shown no interest in inter-cooperation.

Secondly, the lack of means for composing various services towards higher-order services meeting actual user demands imposes additional challenges to the emergence of the Continuum of Computing. The lack of interoperation prevents services from being set up quickly and easily. Cloud providers and their respective systems adhere to different conventions with respect to interfaces and communication protocols. For instance, Google Cloud Platform services use Protocol Buffers [28], a Google technology for serialising structured data. For this reason, a significant effort will be required to uniformly provide the service semantics and interfaces in a comprehensible way for both humans and machines. As example, the system would need to map high-level descriptions to vendor specific implementations (e.g. "key–value store" would map to different AWS or Google Cloud products).

On this latter matter, services on the Internet of today can be considered as mute and dull. In the future, services are expected to become more open and reactive to their respective environments, particularly in two respects [1]. Existing service interfaces (e.g., REST-based) are not yet designed to be interpreted and used by machines but rather by humans. This issue frequently prevents the systems from easily discovering and interacting with them. Methods are needed to "put a face" on services and improve user-service interaction and machine-to-machine (M2M) communication. Fortunately, recent REST community efforts are trying to compensate for the lack of M2M communication [29].

a: Service-oriented architectures

The idea of service-orientation can be partially traced back to the object-oriented programming (OOP) literature [3].

However, the evolution of objects into services, and the relative comparisons, has to be treated carefully since the first focus on encapsulation and information is hidden in a shared-memory scenario. In contrast, the second is built on the idea of independent deployment and asynchronous communication via well-defined interfaces. It is, therefore, a paradigm shift, where both the paradigms share the common idea of componentisation.

The last decade has seen a further shift towards the concept of service first and the natural evolution to microservices afterwards. Service-oriented architectures (SOA) have been introduced to harness distributed systems' complexity and integrate different software applications. In SOA, a service offers functionalities to other components, accessible via message passing. Services decouple their interfaces (how other services access their functionalities) from their implementation.

This design is in striking contrast to monolithic architectures, where the application is composed of a single program, typically providing a user interface and data access through a database. However, monolithic architecture poses real challenges and difficulties when growing exponentially because it needs to scale up.

Service-Oriented Architectures (SOA) are built around services designed to work in an orchestrated manner to modularise the system. It is more challenging to divide the application into multiple services, but it enables greater flexibility, extensibility and reusability of existing services for multiple use cases. This model's benefits are application modularity and service reusability [4], which are all very important to a successful computing continuum. However, a major disadvantage is a complexity in orchestrating and monitoring all the services, especially when the project is complex and the components are huge.

The first generation of SOA architectures defined daunting and nebulous requirements for services (e.g., discoverability and service contracts) [3], which hindered the adoption of the model. Microservices are the second iteration, introduced as a solution for the gaps in the SOA approach.

The microservices approach divides applications into more granular components by distributing them into small self-contained services. Each service implements and attends to separate business functions and capabilities to maintain independence from other services. They are mainly deployed in an automated manner, through a container (more on that in §II-B3) and communicating through REST APIs [5], thus making the impact of programming language insignificant.

In terms of the continuum, the *as-a-service* model allows consumers to be only concerned with what they want to do and accomplish, and providers with how that could be done and provided to the user. A successful interface establishment between those two actors can lead to minimal direct consumer interaction with the provider's infrastructure, allowing complete control to the provider and no cost of ownership.

Furthermore, this results in a system where various service implementations should already exist, maybe provided in the same fashion as service marketplace (e.g. Shopify App Store [30]), and the consumer himself does not have to be an expert and develop them. Even the infrastructure's physical resources should not be consumer-aware, and there may be several diverse implementations to meet specific service demands. These implementations can differ in hardware type and could be characterised by different price and performance attributes.

2) Orchestration

The complexity of the continuum environment and services makes service orchestration a central task to coordinate and schedule the operation of a myriad of distributed service components. Besides, it will also be a crucial feature for many IT organisations and DevOps adopters to speed the delivery of services, simplify optimisation, and reduce costs [8]. However, the orchestration of virtualised environments is challenging due to the scale, heterogeneity, and diversity of resource types and the uncertainties of the underlying environments. The uncertainties arise from a number of factors, including resource capacity demand (e.g. bandwidth and memory), failures (e.g. failure of a network link), user access pattern (e.g. the number of users and location) and lifecycle activities of applications.

Another difficulty caused by the heterogeneities of the underlying resources is providing different pricing models, locations, and resource types and sizes. Taking into account the heterogeneity would enable for instance to dynamically provision resources of different pricing models to the cluster to satisfy application-specific needs with minimum cost.

Generally speaking, orchestrating the services in the continuum is a huge challenge. It encompasses different technologies from different fields, including, among others, wireless cellular networks, distributed systems, virtualisation, and platform management. It imposes new models of interaction among different heterogeneous clouds, which require mobility handover and migration of services at both local and global scale.

3) Virtualisation

The rapid pace of innovation in datacentres and the software platforms within them has transformed how companies build, deploy, and manage online applications and services. Until the last decade, basically every application ran on its own physical machine.

The high costs of buying and maintaining large numbers of machines, and the fact that each was often under-utilized, led to a great leap forward virtualisation.

Virtual Machine Monitors (VMM) [45] were born at the end of the 1960s as a software-abstraction layer that partitions a hardware platform into one or more virtual machines (VM) with the aim to run existing software unmodified. Nowadays however, a VMM is more a solution for security and reliability. Functions like migration and security, which are difficult to achieve in modern operating systems, seem much better suited to implementation at the VMM layer.

KVM (Kernel-based Virtual Machine) [46] is an example of VMM built into the Linux kernel that allows a host machine to run multiple, isolated virtual machines.

On the contrary, Linux Containers (LXC) [47] are an OS-level virtualisation method for running multiple isolated applications sharing an underlying Linux kernel. A container namely consists of one or more processes, with reduced privileges, having restricted visibility into kernel objects and of host resources.

With containers, applications share an OS and as a result these deployments will be significantly smaller in size than VM deployments, making it possible to store nowadays a few thousand containers on a physical cloud host (versus a strictly limited number of VMs). Adopting containers leads to a platform where the end-user can deploy services and applications on edge/fog computing platforms on heterogeneous devices with minimal efforts. Several works in literature (e.g. [56] and [57]) have proved the feasibility of container virtualisation applied to cheap low-powered devices as well, namely the already mentioned Raspberry Pies.

Containers provide resource isolation, self-contained packaging, anywhere deployment, and easiness of orchestration. All these features make containers a very compelling technology for the continuum.

To be precise however, some of these features are merit of the underlying image technology rather than the container virtualisation per se. Images package applications with individual runtime stacks, making the resultant software independent from the host operating system. This self-containment makes it possible to run several instances of an application, on different platforms. Many leading cloud providers like Amazon, Google, Microsoft, IBM and others use this technology for enabling PaaS and FaaS.

Existing serverless platforms such as Google Cloud Functions [58], Azure Functions [59] and AWS Lambda [60] isolate functions in ephemeral, stateless containers. The use of containers as an isolation mechanisms causes however latencies up to hundreds of ms or even 5 seconds [62], which are inacceptable for latency-sensitive services operating ad the edge. To achieve better efficiency, these platforms cache and reuse containers for multiple function calls within a given time window; e.g. 5 minutes, whereas some users send artificial requests to keep the containers alive. In the edge environment, the long-lived and/or over-provisioned containers can quickly exhaust the limited node resources and become impractical for serving a large number of IoT devices. Therefore, supporting a high number of serverless functions while providing a low response time (say 10ms [63]) is one of the main performance challenges for resource-constrained edge computing nodes.

In addition to the latency issue, containers have a not negligible resource footprint and isolation issues. Containers offer relatively weak isolation, to the point where containers are ran in virtual machines to achieve proper isolation. Unfortunately, virtual machines' resource requirements are too demanding for the edge or fog nodes. A more lightweight yet strong isolation solution is a hot research question.

Accordingly, recently lightweight isolation platforms have been introduced as a bridge between containers and full system virtualisation. Both Firecracker and gVisor [42] rely on host kernel functionality. Firecracker provides a narrower interface to the kernel by starting minimalistic guest VMs and providing full virtualisation, whereas gVisor has a wider interface. They both have low overhead in terms of memory footprint though. As an instance, Amazon Firecracker has a memory footprint of 5MB and startup latency of about 125ms [61].

Lastly, it is worth mentioning that Unikernels [49] are another novel virtualisation techniques. Unikernels comprise a minimal operating system and a single application, making them a natural alternative to containers. Unikernel-based serverless experiments yield at least a factor of 6 better latency and throughput [50]. Unikernel implementations, however, lack the maturity required for production platforms, e.g. missing the needed tooling and a way for non-expert users to deploy custom images. This makes porting containerized applications very challenging. Besides, container users can rely on a large ecosystem of tools and support to run unmodified existing applications.

As concluding note, undeniably, the basic units of execution have continuously shrank in the past years: from VMs to containers, unikernels, and serverless functions. Such smaller execution units are dictated by the need for increased flexibility and control, along with the strong economic incentives to increase resource utilisation. As a consequence, applications are increasingly becoming comprised of distributed components and the number of lines of code associated with each unit of execution, and the duration of each execution unit have all greatly decreased.

4) Dynamic configuration

Edge systems and IoT nodes must promptly recognise the environmental context changes under they are subject, which are often critical to many applications like video analysis [16]. This failure is because the service controller running on the edge can validate the data's usefulness (e.g. video frame) only after receiving the devices. Meanwhile, the IoT device will continue to perform poorly (or even uselessly) until the controller makes adjustments.

Dynamic configuration on constrained devices would allow dynamically adapting to environmental context changes based on the application requirements by application-specific computation on the node itself. The experimental results in [16] show that adapting multiple IoT cameras to serve applications and dynamically reconfiguring the cameras based on application desiderata can significantly enhance object detection accuracy, computational cost, and response time.

At the same time, opening the embedded systems to arbitrary code execution exposes the system to malicious intents, meaning that even a single buggy line of code can lead to system compromise, threatening the controlled physical assets, and potentially human safety. Current microcontroller

software stacks are complicated to use in trustworthy embedded systems. These stacks generally lack strong facilities for isolation.

Hardware support is often used to isolate applications from each other and the OS from applications. Dual-mode protection enables the kernel code to be inaccessible from applications, at the cost of requiring mode transitions. In addition, Memory Protection Units (MPUs) enable the partitioning of memory into subsets, each accessible by separate applications. On the other hand, MPUs require tight OS, hardware, and build-system support and thus limit the adoption of hardware memory isolation has been limited in microcontrollers.

A further challenge of dynamic configuration of IoT devices is allowing isolation execution with an acceptable compromises in efficiency loss and energy consumption. A common solution is adopting languages (e.g. Lua) that provide type safety and ensure that all memory accesses adhere to the proper type of the data being accessed. Thus, memory accesses are constrained only to memory provided by the language runtime, usually via an interpreter, and are prevented from corrupting the device. As alternative, Software Fault Isolation (SFI [64]) techniques provide instead memory safety by constraining loads and stores to a sandbox. When this data sandboxing is paired with control flow integrity (CFI), which ensures that execution can only follow paths intentionally generated by the compiler, the application's execution and memory accesses are constrained to the sandbox. Unfortunately, both the overheads of interpreting in the language runtime and performing the necessary SFI checks in SFI are a concern, given the limited resources of microcontrollers.

5) Interoperability

There are and will be many different technologies for connecting and integrating all the things into the continuum. In embedded systems, standards are in place, or standardisation efforts are underway. For example, ZigBee, IPv6 over Low-Power Wireless Area Networks (6LoWPAN), MQTT and CoAP [31] are gaining popularity in the wireless sensor networking area, and OPC [32] is well accepted in factory automation. However, the technologies are too different from expecting any standard to be able to cover them all.

For all these reasons, it is clear that we will have to deal with heterogeneity when building the continuum's edge infrastructures. Standards are surely helpful but will be hardly achieved. What is asked for here is interoperability. The key is to separate the functionality from its technical implementation. Service-oriented architectures are ideal for this since they encapsulate functionality in services with a common interface, abstracting from the underlying hardware and protocols. TODO

Having infrastructures that allow connecting and integrating a diverse set of technologies is not just a "necessary evil" but rather a strength since it offers two key benefits. First, it allows for applying different solutions to different applications. Depending on the application requirements, the best-fitting technology can be used.

Secondly, an infrastructure where diverse technologies can easily be integrated into will be future-proof. Especially in edge computing and IoT, the technical developments are not complete, and we can expect that new technologies, protocols and standards will arise. An infrastructure built with technology diversity in mind will allow interoperability with existing and already deployed devices and networks.

With that said, cloud platforms heterogeneity is also a non-negligible concern. Cloud services typically come with proprietary interfaces, causing resource integration to be adequately customised based on specific providers. This issue can be exacerbated when users when services in the continuum can depend on multiple providers to provide the necessary resources an application may require or improve application performance, and resilience [17].

Both cloud and IoT services and applications have typically been conceived as isolated vertical solutions. All system components are tightly coupled to the specific application context or the cloud provider. The IoT should instead ease service delivery without any vendor lock-in by applying the cloud service delivery models.

This challenge involves several aspects, where solutions need to be investigated in terms of unifying cloud providers [33], interoperable programming interfaces, and means for coping with data diversity.

TODO: data-diversity

6) Portability and Programmability

In cloud computing, users program their code and deploy them on the cloud. The cloud provider is in charge to decide where the computing is conducted in a cloud. Users have only partial knowledge of how the application runs. This easiness is one of the benefits of cloud computing: the infrastructure is transparent to the user. Usually, the program is written in the programming language the developer is most familiar with between the supported ones (e.g. JavaScript, Java, Python, .NET) and compiled for a specific target platform since the program only runs in the cloud.

As mentioned before, cloud developers tipically use containers that are spawned from images that include everything that is needed to run them. This includes code, libraries, settings, and system tools. More importantly, these images can be constructed from filesystem layers and hence are lightweight and use considerably less space than VMs. They offer benefits in terms of ease of deployment, testing, and composition to developers. Besides, they can be used to capture a development environment that is known to work for each device revision, and to share this environment among a team of developers. Containers are being widely used by organizations to deploy their increasingly diverse workloads to the cloud datacentres.

However, in edge computing and thus in the continuum, the nodes have diverse platforms. Mobile devices may have Android and iOS operative systems, and edge nodes are

likely to have different Linux distributions or versions of the same distribution. Embedded devices tipically lack any operative system feature, such as a file system or even the notion of process. Even the diversity of CPU architectures (x86, ARM32, ARM64, RISC-V) give programmers a hard time compiling for the different platforms. For all these reasons, the nodes' runtime differs from each other, and the programmers face considerable difficulties in writing a service that may be deployed in the continuum paradigm.

Docker images attempt to overcome this issue by supporting multiple architectures. A single image may contain variants for different architectures or even for different operating systems, such as Windows. Most of the official images on Docker Hub [65] provide a variety of architectures. For example, official images usually support amd64, arm32v5, arm32v6, arm32v7, arm64v8, i386, ppc64le, and s390x. As an instance, when running an image on an x86_64 / amd64 machine, the x86_64 variant will be pulled and run. Despite this feature, a significant pain point is the fact that developers are still required to configure and build their applications multiple times, for each platform. Moreover, the absense of an OS on embedded devices or the limitation on resource capacity block any hope of running containers on such devices, mining the idea of containers as a unified solution for portability in the continuum.

To address the programmability in the context of edge computing, [13] proposes the concept of computing stream defined as a serial of functions applied to the data along the data propagation path. Likewise, [12] proposes to program the continuum so that new algorithms and deep learning models can be pushed to appropriate locations (i.e., edge, fog, cloud, or HPC computing resources) using a simple FaaS abstraction.

Serverless architecture naturally solves two critical problems for portability and programmability [14]. First, the serverless programming model dramatically reduces users or developers' burden in developing, deploying, and managing applications. There is no need to understand the complex underlying procedures and distributed system to run the applications. Second, the functions are flexible to run on either edge or cloud, which achieves the desired portability.

However, in this thesis's view, the serverless computing model can satisfy only a limited subset of services, notably those with event-driven and request-reply nature. However, a common type of applications is long-running services that require high availability and must handle latency-sensitive requests. Examples include user-facing service or control loops. Another type of applications are batch jobs, which have a limited lifetime and are more tolerable toward performance fluctuations. Examples include scientific computations or MapReduce jobs [66]. Batch jobs are more suitable to be run in the cloud, but there are literature examples that show the advantages of big-data analysis on the edge [67].

7) Mobility

In the continuum, services can be reallocated to follow the end users' movements, and the data and state along should be reallocated as well. Therefore, the collaboration issues (e.g., synchronisation, the aforementioned data/state migration, etc.) have to be addressed across multiple infrastructure layers.

When mobility is required, provisioning data and services also needs to be performed with high reactivity and reliability. For instance, in the context of smart mobility, vehicles are often on the move, and vehicular networking and communication are often intermittent or unreliable [19].

Being co-located at base stations, MEC servers are available anywhere the user moves and can support mobility between cellular cells.

TODO

8) Reliability

When applications are deployed in resource-constrained environments, many challenges related to device failure or unreachability exist. Things at the edge of the network could fail due to various reasons, and they could additionally report failure to report data under unreliable conditions such as low battery level.

Various new communication protocols for IoT data collection have been proposed in the last years (e.g. CoAP, MQTT, AMPQ) [31]. These protocols serve well for the support of low energy and highly dynamic network condition. However, their connection reliability is not as good as BlueTooth or WiFi. If both sensing data and communication are not reliable, the system must leverage multiple reference data sources and historical data records to provide a reliable service.

9) Security and Privacy

The integration of edge computing, fog computing and cloud computing will raise some new and unforeseen security issues. Unique and unstudied scenarios, such as the interplay of heterogeneous edge nodes, and the migration of services across global and local scales, create the potential for original channels of malicious behaviour [35].

Like the health record data, end-user data collected at the edge of the network should be stored at the edge, and the user should be able to control if service providers should use the data. However, as edge computing stores and processes data at the edge, the privacy-sensitive information associated with end-users could be exploited and be more vulnerable than cloud servers.

Moreover, for applications like smart grids or sensor networks, an adversary could report false data, modify the other user data, tamper with their smart meter, or spoofing IP addresses, further disrupting sensor management's effectiveness in IoT systems.

10) Context awareness

Today's web services are rarely context-sensitive. However, to support context-sensitive services at the edge, they need to be aware of different aspects of the environment they are acting in. Context services can provide such information to the application services that implement local control loops and trigger specific actions based on context events. Such context services can be composed of lower-level services which deliver individual sensor readings.

On this matter, the MEC architecture provides the advantage of data locality, which lets a given MEC server store the context data only regarding the devices within the region covered by its base station. Such an advantage is two-fold: providing location-aware services is more accessible, and less data must be persisted on each MEC server. As an example, MEC servers in [25] can retrieve the features of the points-of-interest in an augmented reality scene, match them against a local database, and return the corresponding data (information about monuments, buildings and other points of interest) to the client application.

11) Energy efficiency

Obtaining energy efficiency in both data processing and transmission is an essential open issue. For handling such issue, several directions have been proposed: more efficient data transmission and compression technologies, data caching mechanisms for reusing collected data in time-tolerant applications, middlewares to improve availability and to compress data in case of continuous and long-duration monitoring of data.

III. SYSTEM DESIGN

Having illustrated the challenges of this thesis' vision, this chapter presents issues tackled by the author's work and the technology baseline.

A. ADDRESSED CHALLENGES

This thesis attempts to address the following challenges within the list mentioned in the previous section:

1) *Service orientation*: the web is proposed as substrate for the pervasive communication between services in the continuum and organized following the Representational State Transfer (REST) architectural style;
2) *Orchestration*: Kubernetes III-D is proposed as orchestrator for resource orchestration, workload orchestration and service orchestration of the continuum. On this matter, Kubernetes is extended to support the orchestration of IoT devices with an attempt to overcome the heterogeneity and unreliability of the underlying environment. Furthermore, the author presents a platform to allow independent vendors and developers to provide services running on popular service orchestrators, notably Kubernetes;
3) *Virtualisation*: WebAssembly III-E is used as platform-agnostic virtualisation technology to provide lightweight sandboxing capabilities on every node of the continuum based on software fault isolation and control flow integrity. Importantly, WebAssembly is a portable bytecode that is a compilation target of potentially any programming language, both the ones that use garbage collection (e.g. Go, JavaScript, and Python) and those which don't (e.g. C, C++, and Rust);
4) *Dynamic configuration*: the combination of a small WebAssembly interpreter and the low-overhead binary format of the bytecode result in intriguing opportunities for dynamic configuration of IoT devices;
5) *Interoperability*: in the author's view, the web is the substrate for the communication between services in the continuum and Representational State Transfer the architectural style to follow to support high-level interoperability between services;
6) *Portability and Programmability*: the thesis suggests the usage of the Open Container Initiative (OCI) Distribution Specification to extend Docker images to support the distribution of WebAssembly applications which are not container filesystem images. This solution significantly improves the integration with development existing tools and lowers the entry barrier for developing applications in the continuum;

Because of the large problem space, mobility, reliability, security, privacy, context awareness and energy efficiency are not addressed in this thesis and are left as open research questions. Besides, by any mean, it is also not in the intentions of this thesis being complete in the technical solutions proposed here. The author views them as a suggested technological ground for future work and as proof of concept of the continuum's viability.

B. WEB SERVICES

Tim Berners-Lee's original vision for the web focused on documents and their inter-relationships, but it was soon clear that the future would have laid in a web of applications. The web has since become the world's most successful vendor-independent application platform.

First and foremost, the web is a loosely coupled application-layer architecture and applications today depend on the web architecture, using HTTP to access information and perform updates. The most dominant architectural style is Representational State Transfer (REST) [5] that makes information available as resources identified by URIs. Applications communicate by exchanging representations of these resources using a transfer protocol, HTTP precisely. HTTP is the most popular application protocol on the Internet and the pillar of the web architecture, although new communication protocol (e.g. CoAP in Section III-F) are emerging and HTTP itself is undergoing revisions (e.g. HTTP/3 or QUIC [69]).

The most prominent characteristics of REST are:

- Resource orientation: resources are an information abstraction that allows servers to make any information or service available, identified via Uniform Resource Identifiers (URIs). The REST architecture allows the

server to own the original state of a resource, and client negotiate and access a representation of it. Such representation negotation is suitable for interoperability, caching, proxying, and redirecting requests and responses, enabling seamless interoperation of services in the continuum through proxies. Under the REST architecture, web resources often advertise links to other resources creating a distributed web and resulting in a highly scalable and flexible architecture.

- Uniform interface: clients access these server-controlled resources in a request–response fashion using a small uniform set of methods with different semantics (GET, PUT, POST, and DELETE).
- Communication protocol independency: for constrained environments, REST needs to be based on a protocol different than HTTP. HTTP is a powerful and well-tried protocol, but it's relatively expensive both in implementation code space and network resource usage. Part of the problem is that HTTP has undergone more than a decade of organic growth, leading to considerable implementation baggage that overwhelms small devices [37].
- High-level interoperability: the REST architectural style enables interoperability between RESTful protocols through proxies or, more generally, intermediaries that behave like a server to a client and play a client toward another server. REST intermediaries play well with assumption that not every devices must offer RESTful interfaces directly. In a number of cases it may not be possible to change the underlying communication protocol, but intermediaries allow exposing the resources through a RESTful API nevertheless. The interactions behind that RESTful interface are invisible and may include highly specialized protocols for the specific implementation scenario (e.g. OPC UA or MQTT) [38]
- Machine-to-machine communication: communication protocols must be specifically designed for efficient machine-to-machine communications without introducing overhead in network load, delay, and data processing. In the machine-to-machine (M2M) environments that will be typical of IoT applications, devices must be able to discover each other and their resources. Resource discovery is common on the web. One form of web discovery occurs when a user access a server's default resource (such as *index.html*), which often includes links to other web resources available on that or related servers. Machines can also perform web discovery if standardized interfaces and resource descriptions are available. New approaches from the IETF include the well-known resource path */.well-known/scheme* (RFC 5785) and the HTTP link header (RFC 5988). In IoT, we're dealing with autonomous devices and embedded systems; thus, the importance of uniform, interoperable resource discovery is much greater than on the current web.
- Stateless: REST requires requests from clients to be self-contained, in the sense that all information to serve the request must be part of the request. Statelessness helps applicatios to scale, as persisting state can be troublesome at the edge which have limited capabilities, and supports easier mobility in the continuum space.

From the previous points, it's clear that the design goals of RESTful web systems and their advantages for a decentralized and massive-scale service system align well with the field of pervasive computing. For these reasons, REST an ideal candidate to build an universal API (Application Programming Interface) for services and devices in the continuum, letting developers of web-based applications continue using their existing skills.

C. OPEN SERVICE BROKER

Application development teams require services and managed services enable developers to concentrate on their application code rather than operating these services, which can be very costly. Besides, self-service on-demand marketplace services can increase developer velocity and minimize time to deliver value to market. In the author's view, the continuum will offer a marketplace with vendors adding services based on application developer demand. This idea is not novel and the most notable example is Cloud Foundry's Marketplace [73], now dismissed unfortunately. A very akin one, still active and thriving, is the Shopify App Store [30] where developers can extend their e-commerce with third-party services. In the continuum's marketplace, providers control the access to the services and payment plans, but allow developers to bring their own services to the catalog. Over the years a rich ecosystem of services will be developed and accessible via simple well-documented APIs.

Contrary to this vision, cloud standards have failed to gain traction and therefore the need to find mechanisms for bridging the heterogeneity gap between platforms, and enabling data integration are more relevant than ever. Most orchestration technologies working across administrative domains use a broker to orchestrate resources at different levels within a provider (e.g. the cloud and the edge network) and across providers [33]. As the number of cloud vendors is limited, it is possible to build adapter and brokering layers that tried to homogenise access to different clouds.

On this matter, several stakeholders (namely the Kubernetes community, OpenShift, IBM and Google) have shown interest and contributed to working on a standard Open Service Broker (OSB) API [74]. Components that implement the OSB REST endpoints are referred to as service brokers and can be hosted anywhere the application platform can reach them. Service brokers offer a catalog of services, payment plans and user-facing metadata.

The real value of the broker API is, however, in abstracting service-specific lifecycle operations from the platform. The service brokers translates RESTful requests from the platform to service-specific ones for operations such as create, update, delete and generate credentials to access the provisioned services from application istances. Service brokers

can offer as many services and plans as desidered and multiple service brokers can be registered with the marketplace so that the final catalog of services is the aggregate of all services. The platform is thus able to provide a uniform user experience for application developers consuming these services.

D. KUBERNETES

Kubernetes [68] is a open-source framework designed to manage containerized workloads on clusters and originated from Google's experience with cloud services. The basic building block in Kubernetes is a Pod. A Pod encapsulates one or more tightly coupled containers that are colocated and share the same set of resources. Pods also encapsulate storage resources, a network IP, and a set of options that govern how the Pod's container(s) should run. A Pod is designed to run a single instance of an application enabling horizontal scaling by replicating multiple Pods across the nodes (workload scheduling). The amount of CPU, memory, and ephemeral storage a container needs can be specified when creating a Pod. This information can then be used to make decisions on Pod placement (resource scheduling). These compute resources can be specified both as a requested amount or a as cap on the amount the container is allowed to consume.

From a technical perspective, Kubernetes allows for various types of container runtimes to be used, with Docker natively supported by the platform. Thanks to the standardisation of the container runtime interface (CRI) API, Kubernetes supports other container technologies such as containerd [70] or Firecracker and gVisor mentioned before. Moreover, etcd is an open-source highly available distributed key value store and a fundamental part in supporting Kubernetes in the management of the cluster nodes and jobs. Specifically, etcd is used to store all of the cluster's data and acts as the single source of truth for all of the framework's components.

Overall, Kubernetes is a highly mature system; it stemmed from 10 years of experience at Google and it is the leading container-based cluster management system with an extensive community-driven support and development base. It provides users with a wide range of options for managing their Pods and the way in which they are scheduled, even allowing for pluggable customized schedulers to be easily integrated into the system.

Besides the extensible scheduler, which makes Kubernetes already an intriguing choice as the go-to orchestrator for the continuum, it also supports label-based constraints for the Pods' deployment. Developers can define their own labels to specify identifying attributes of objects that are meaningful and relevant to them, but that do not reflect the characteristics or semantics of the system directly. An example, it would allow to specify that a IoT device component must be reachable from the host or a persistent storage available, e.g. a Solid State Drive. But more importantly, labels can be used as well to force the scheduler to colocate services that communicate a lot into the same availability zone, improving drastically the latency and paving the way for context-aware services. To realize the heterogeneity goals however, it is required to continuously filter unqualified resources and new resource affinity models must be proposed to rank the resources when provisioning for different applications.

Finally, Kubernetes is also designed with multitenancy in mind, compared to Docker Swarm [71], which is another popular open-source orchestrator often cited as main competitor for edge orchestration due to its simplicity. Support for multitenancy is a must and well aligned with their goal of executing heterogeneous services on a set of shared resources.

1) Akri

Akri [75] is an open-source project which extends the Kubernetes Device Plugins API [76] to allow visibility to edge devices from applications running within the cluster. The Device Plugin API is designed to expose hardware resources attached to a node (called Kubelet in Kubernetes), such as GPUs or Solid State Drives. Akri leveraged this extension interface to implement discovery of IoT devices, with support for the diversity of communication protocols and the ephemeral availabilities.

Akri's architecture can be divided into three main component: the agent, the controller and the configuration. A configuration defines a communication protocol (e.g. OPC UA [32]) and related information, such as the protocol discovery parameters or the image of the agent to deploy.

The Akri agent instead is responsible for discovering devices following a protocol standard and support for new protocol can be easily added to the system. The agent will then track the state of the device and keep the Akri controller updated with the status. At the time of writing, the project has built-in support for ONVIF, udev and OCP UA [32] discovery handlers, with an incoming proposal for CoAP [37] coming from this thesis' author. By using Akri, the Kubernetes cluster is able to carry out dynamic discovery in order to take advantage of new resources as they become available, as well as move away from decommissioned/failed resources. Discovering edge devices is usually accomplished by scanning all connected communication interfaces and enlisting all locally available resources.

Lastly, the Akri controller advertises discovered leaf devices to the Kubernetes resources manager, making them visible to any application that demands edge devices to work. The controller helps application communicating with the device as well, by deploying a broker as intermediary.

The broker can by any application. As an example, the cluster could deploy a broker that exposes a gRPC [77] interface and translate the requests to the underlying edge communication protocol. In this thesis' view, the broker should be a web server that abstracts the actual communication between devices and applications behind a RESTful API in light of the previous reasons to choose the REST architecture as go-to for uniform interface in the continuum. Akri should automatically find all the devices in the environment and

make them available as web resources. As an instance, the agent discovers the devices on a regular basis by scanning for CoAP (Section §III-F) resources.

Besides, the broker could offer local aggregates of device-level services. For example, it can offer a service that returns the combined temperature measurements of all the things connected to it at any given time.

A RESTful broker can also help to scale with the number of concurrent HTTP requests by implementing highly performant cache mechanisms. The edge resource periodically sends its sensor readings to the broker, where the values are cached locally. Each application request is then served directly from this cache without accessing the actual device, improving in the average roundtrip time.

The broker architecture has also the advantage of fully decoupling the leaf node from the cluster workload, as it only needs to send an update packet with a frequency short enough to ensure the validity of data. On the other hand, the staleness of the retrieved data will depend on the frequency of updates sent by the device, while forwarding the HTTP request (properly translated) has the advantage of returning always the most recent sensor reading when the request was processed. Obviously, this will not hold true for non-cacheable requests (e.g. HTTP POST) that must be sent to devices, such as turning on LEDs or changing application state.

As many distributed monitoring applications are usually read-only during their operation (e.g. sensors collect data), this architecture exhibits a greater scalability level. The ultimate goal is to enables new types of applications where physical sensors can be shared with thousands of users with little, if any, impact on the latency and data staleness. A representative of such case is tracking public transportation with sub-second accuracy.

E. WEBASSEMBLY

WebAssembly (Wasm) [48], first announced in 2015 and released as a Minimum Viable Product in 2017, is a nascent technology that provides a strong memory isolation (through sandboxing) at near-native performance with a much smaller memory footprint. WebAssembly is language designed to addresses the problem of safe, fast, portable low-level code on the web. The project is developed by the World Wide web Consortium (W3C) with support from all major web browser vendors (Mozilla, Google, Microsoft, and Apple), thus increasing the likelihood that it will avoid serving the purposes of a single entity (e.g. Java and .NET). Developers who wish to leverage WebAssembly may write their code in a high-level language such as C++ or Rust and compile to a portable binary which runs on a stack-based virtual machine. Even though WebAssembly technically is a binary code format, it can be presented as a language with syntax and structure. This was an intentional design choice which makes it easier to explain and understand, without compromising compactness or ease of decoding.

A Wasm binary takes the form of one or more modules. It contains definitions for functions, globals, tables, and memories. The computation is based on a stack machine: code for a function consists of a sequence of instructions that manipulate values on an implicit operand stack, popping argument values and pushing result values. WebAssembly represents however control flow differently from most stack machines. It does not offer simple jumps but instead provides structured control flow (SCF) constructs more akin to a programming language. This ensures by construction that control flow cannot contain arbitrary branches. The SCF allows WebAssembly code to be validated and compiled in a single pass. SCF disassembled to a WebAssembly text format (.wat) is easier to read as well, an often overlooked but important human factor on the web.

The memory of a WebAssembly program is a large array of bytes referred to as a linear memory or simply memory. All memory access is dynamically checked against the memory size and out of bounds access results in a trap. Linear memory is disjoint from code space, the execution stack, and the engine's data structures. Each Wasm memory access addresses linear memory at an offset from the base, n, of the linear memory. Thus, there is some amount of address virtualisation as an address b in the sandbox is located at $b + n$ in physical memory. The Wasm runtime is responsible for translating linear memory accesses, and bounds-checking them to prevent accesses outside the sandbox.

Therefore, compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behaviour, especially when the original language is memory-safe like Rust (Section §III-G). At worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory. This memory and state encapsulation is applied at module level rather than at application level, meaning that memory and functions of a module cannot leak information unless explicitly exported/returned. This granularity in sandboxing is extremely important as security incidents have been increasingly exploiting vulnerabilities in the dependency chain, as reuse of third-party software is pervasive in modern languages like JavaScript, Rust or Go. As an example, JavaScript (and consequently TypeScript [78]) strongly relies on its package manager NPM and the latter has been subject to several security incidents in the last years [79].

This means that even untrusted modules can be safely executed in the same address space as other code. Additionally, it allows a WebAssembly engine to be embedded into any other language runtime without violating memory safety, as well as enabling programs with many independent instances with their own memory to exist in the same process. These sandboxing features make WebAssembly a compelling technology upon which to implement a virtualisation stack for the continuum. On the web, the substrate of the continuum, code is fetched from untrusted sources.

Another important safety feature of Wasm is the type system. Code must be validated before it can be executed safely.

Validation rules for WebAssembly are defined succinctly as a type system. This type system is, by design, simple. It is designed to be efficiently checkable in a single linear pass, to allow parallelisable binary decoding and compilation. Moreover, function pointers cannot be dereferenced directly. A call to a function pointer is translated into an activation of a function in a runtime table of valid entry points and types. The type of the function is checked dynamically against the expected type in the said entry. The dynamic signature check protects integrity of the execution environment. In case of a type mismatch or an out of bounds table access, a trap occurs.

With that being said, the design goals of WebAssembly strongly advocate for its suitability for this thesis:

- *Safe to execute*: protection for arbitrary code has traditionally been achieved by providing a managed language runtime that enforces memory safety, preventing programs from compromising user data or system state. However, managed language runtimes have traditionally not offered much for portable low-level code, such as C/C++ applications, that do not need garbage collection;
- *Fast to execute*: low-level code like that emitted by a C/C++ compiler is typically optimized ahead-of-time. On the contrary, managed runtimes and sandboxing techniques have typically imposed a steep performance overhead on low-level code. WebAssembly is, however, very competitive with native code and benchmarks using Wasm runtimes on modern browsers have shown slowdown within 10% compared to native and almost aways within 2x of native [48];
- *Language, hardware, and platform independency*: as mentioned before, the web spans not only many device classes, but different machine architectures, operating systems, and browsers. Code targeting the web must therefore be independent from any underlying hardware or platform to allow applications to run across all software and hardware types with the same behaviour;
- *Deterministic and easy to reason about*: WebAssembly has been designed with a formal semantics from the start, for both execution and validation, including a proof of soundness. Besides, the Wasm binary can be compiled into a .wat file to simplify learning and debugging;
- *Simple interoperability*: WebAssembly is similar to a virtual ISA in that it does not define how programs are loaded into the execution engine or how they perform I/O. The embedder (i.e. the host runtime) defines how modules are loaded, how imports and exports between modules are resolved, provides foreign functions to accomplish I/O, and specifies how traps are handled. It is possible, by design, to link multiple modules that have been created by different authors, from different original languages. However, as a low-level language, WebAssembly does not provide any built-in object model. It is up to producers to map their data types to numbers or the memory. This design is supposed to provide maximum flexibility to producers, and unlike previous VMs like Java or .NET, does not privilege any specific programming or object model while penalizing others. The downside of this design is that interoperability with object references is cumbersome when it involves exchanging references between the Wasm application and the host code or between modules originated from different language. A step in easing this issue is the recent introduction of Reference Types [80]. Though WebAssembly has a programming language shape, it is an abstraction over hardware, not over a programming language. A WebAssembly module may be compiled once and moved freely between different hardware architectures with no reconfiguration;
- *Compact*: the binary code is designed to be compact, especially compared to bytecode in text format. For this reason, it is recommended distributing the binary code on the Internet code and using the text format only for learning and debugging. Code transmitted over the network should be as compact as possible to reduce load times, save potentially expensive bandwidth, reduce memory usage on constrained devices attached to the network and improve overall responsiveness;
- *Easy to validate and compile*: validation proceeds by checking on-the-fly while the incoming bytecodes are arriving, with no intermediate representation (IR) being constructed. Benchmarks run on mainstream browsers in [48] prove that validation can be fast enough be performed at full network speed of 1Gib/s;
- *Streamable and parallelisable*: a Wasm runtime can minimize latency by starting streaming compilation as soon as function bodies arrive over the network. It can also parallelize compilation of consecutive function bodies. For instance, each function body is preceded by its size so that a decoder can skip ahead and parallelize even its decoding.

Despite the name WebAssembly, there has been a significant effort in the last years in adopting Wasm for a native execution, as it is a portable target for compilation of various high-level languages. Wasm standard does not necessarily make web-specific assumptions and there has been substantial work to standardize the WebAssembly System Interface (WASI) to run Wasm outside Web.

At the time of writing, there are a number of Wasm runtimes for programs written in different languages to embed Wasm applications. At same time there are different compilers which can compile languages to Wasm, e.g. the Wasm back-end for LLVM [81] works for C, C++, and Rust. Commercial solutions have also been slowly but steadily gaining popularity. Cloudflare's Service Workers [82] began to offer support for creating and hosting serverless functions in WebAssembly. In March 2019, the edge-computing platform Fastly has announced and open-sourced Lucet [83] that provides a compiler and runtime to enable native execution of Wasm applications and can instantiate a Wasm module within

$50\mu s$, with just a few kilobytes of memory overhead. Parity is also actively experimenting with writing smart contracts using Wasm [84].

Because of its design features, WebAssembly is currently under experimentation as a new method for running portable applications without the use of containers. This method leverages its binary format a portable vehicle to inherent memory and execution safety guarantees via its language features and a runtime with strong sandboxing capabilities. Thanks to the Wasm security model to enable execution of multiple untrusted modules in the same process, thereby providing significantly lighter-weight isolation compared to VMs and containers for multi-tenant serverless execution [39].

This idea is still at its infancy but there has been some interest in the recent years, as shown by the works done in [43] and [44]. All these works focus primarly on serverless computing via WebAssembly, an expression of the gain in popularity of such computing model. As this thesis have stated before however, if serverless were the only execution model, it would severley limit the variety of applications in the continuum.

1) Krustlet

The current WebAssembly System Interface (WASI) standard only contains a few methods for working with sockets, namely *sock_recv*, *sock_send*, and *sock_shutdown*, that are not enough for complete networking support. Adding support for connecting to sockets is fundamental to allow Wasm modules to connect to web servers, databases, or any service.

2) Wasm for embedded microcontrollers

An emerging use case for Wasm is microcontroller and interpreters are gaining popularity on resource-limited devices. wasm3 [85] is a popular open-source interpreter capable of running on any system with at least 64KiB of storage and 10KiB of memory. The authors of eWasm [51] have also explored various mechanisms for memory bounds checking and evaluated the trade-offs between the processing efficiency and memory consumption. Just-In-Time (JIT) compilers for Wasm exist (e.g. Wasmtime [86]) and receive more attention for the community, but their size and complexity make them unsuitable for microcontrollers.

Although, Wasm interpreters can often approximately 11x slower than native C [87], they useful for dynamically updating system code and debugging, but may not be applicable for code on devices extremely sensitive to performance and energy efficiency.

In the author's view, Wasm interpreters on microcontrollers remains an intriguing technology nevertheless, worth the attention of the research community. They offer a persuasive alternative to other language runtimes, e.g. Lua interpreters which are commonly used on embedded devices to support dynamic configuration [88]. The Wasm standard has a number of features that make it appealing for embedded devices [51].

- Broad support: there is a large ecosystem of vendors, tools, and languages providing Wasm support. With the ability to leverage Wasm, the embedded system community would benefit from a broader ecosystem;
- Portability: Wasm is a platform-independent Intermediate Representation (IR) that can be generated from different source languages, and can run on many CPU architectures. Solving how to effectively run Wasm on microcontrollers would open the possibility to include the embedded world to the continuum as intelligent computational space, rathern than only as mere data collector and dummy actuator;
- No mandatory garbage collection. Many broadly used language runtimes such as javascript, lua, or python cannot provide predictable execution and require excessive memory for a microcontroller;
- Lightweight runtime: Wasm mandates only a small number of runtime features around maintaining memory sandboxing. These light requirements help in an embedded adaptation.

F. COAP

CoAP [37] (RFC 7252) is a web communication protocol for use with constrained nodes and constrained (e.g. low-power, lossy) networks. The protocol is designed for machine-to-machine (M2M) applications and provides a client-server architecture between edge nodes, supports built-in discovery of services and resources. The protocol is built upon key concepts of the web such as URIs and RESTful interaction [36]. CoAP easily interfaces with HTTP for integration with web services while meeting specialized IoT requirements such as multicast support, very low overhead and simplicity for constrained environments.

A central element of CoAP's reduced complexity is that, instead of TCP, it uses UDP and defines a very simple message layer for retransmitting lost packets. The rationale for UDP is that TCP carries significant overhead and can have suboptimal performance for links with high packet loss, both of which are common across the Internet [8] and especially on remote locations where sensors might be deployed. Compared to HTTP as well, the additionally the multiple round trips required for the latter requests can quickly add up, affecting device performance and energy efficiency.

Within UDP packets, CoAP uses a four-byte binary header, followed by a sequence of options, each up to two bytes. The protocol's specification also defines the familiar four request methods: GET, PUT, POST, and DELETE. Similarly, response codes are patterned after the HTTP response codes.

The URI format allows exposing device data as resources and the use of standard and specialized service endpoints. For instance, CoAP servers are encouraged to provide resource descriptions available via the well-known URI */.well-known/core* to achieve resource discovery. Clients then access this description with a GET request on that URI, usually via a IPv4 or IPv6 broadcast message. The description format is based on the CoRE Link format (RFC 6690), which

is simple and easy to parse. Ease of parsing allows more efficient M2M discovery and inter-communication between nodes themselves.

The CoAP protocol allows to support different resource representations, in line with the representation negotation advocated by the REST architecture. The default format is textual for its convenience when reading and easiness to parse. The binary format is efficient to communicate, but requires external tools to make it readable by users. XML is understandable and very well structured, but the size of its messages is big and it is much worse to parse compared to binary formats. JSON is understandable, well structured and compact, but may still put unnecessary burden on the limited device. With all constrained devices, the flash or RAM memory consumed is one of the biggest problem, notably on devices with network connectivity where the buffer space needs to be allocated.

Another advantage of CoAP is that, by using standard web technologies, the human interactions follow a familiar and intuitive pattern already used by many developers and thus the learning curve is smoother. This feature cannot be understimated as allowing developers to use a familiar and seamless programming experience is essential to achieve the success of the continuum.

Interoperability with the rest of the continuum can be achieved by following the REST architecture's proxy pattern. We can generally build intermediaries that speak CoAP on one side and HTTP on the other without encoding specific application knowledge. This allows deploying new applications without having to upgrade the intermediaries involved. On the other hand, an intermediary can perform the translation between CoAP and HTTP without posing further requirements either on the client or server. Because equivalent methods, response codes, and options are present in both HTTP and CoAP protocols, the mapping between the two is straightforward. As result, an intermediary (e.g. Akri brokers) can discover CoAP resources and make them available at regular HTTP URIs, enabling web services to access CoAP servers transparently.

However, the HTTP client-initiated interaction model may be unsuited for event-based and streaming systems, where data is sent asynchronously to the clients as soon as it is produced. To overcome this issue, CoAP uses an asynchronous approach to support pushing information from servers to clients: observation. In a GET request, a client can indicate its interest in further updates from a resource by specifying the "Observe" option (RFC 7641). If the server accepts this option, the client becomes an observer of this resource and receives an asynchronous notification message each time it changes. This kind of communication, combined with an intermediary broker, allows streaming data updates via WebSocket (RFC 6455) and overcoming the client-pull interaction model of HTTP. The broker can also help achieving a more reliable communication by transparently changing the underlying sensor in case of unavailability or by just avoiding closing the connection in case of temporary loss of connection to the device.

G. RUST

As the continuum integrates computing throughout the network, the attack surface greatly increase. Security has to be a top priority during software development, yet most of the infrastructure is programmed using the C, C++ programming languages. These two programming languages are chosen due to the low overhead on memory and the high processing performance. Embedded systems has favour the two languages because of the need for low level control over the hardware. However, C and C++ not particularly known for producing secure software as evidenced by the large number of vulnerabilities reported against software written in them. Notably, the main cause of security vulnerabilities are memory safety bugs like data races and buffer overflows. Even higher-level languages suffer from these kinds of safety issues. The Go language, upon which most of the modern distributed systems are built-in (namely Kubernetes), is not exempt from exploits based on data races [89].

On the other hand, Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions", meaning that safety checks are done at compile time and runtime checks (e.g. out-of-bounds access) have the minimum overhead and come with a predictable cost.

Not all operations can be verified to be memory safe at compile time. One example is indexing a slice, a partial view into an array. In this case, the indexing operation contains a runtime check to check if the index is within bounds; at runtime slices carry information about their length for this purpose. If the index is out of bounds then the result is a panic. A panicking condition can result in either aborting the whole program or just unwinding the stack of the thread that ran into the panic. The unwinding process walks up the stack freeing all live resources before terminating the thread.

Therefore, the runtime overhead is not null but there is strong emphasis in the community on building safe abstractions with almost zero runtime cost. Safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g. casts are checked and unions are tagged), reads from uninitialized memory, mutations of non-mut data, and data races, among other misbehaviours.

The borrow checker, the most innovative feature of the language compiler, runs as part of the compilation process and catches bugs like use after free, pointer invalidation and data races. Security cannot be achieved without memory safety so it is fundamental that memory safety is a property of the language and not a concern of the developer only. That is why Rust is almost a mandatory choice as the implementation language for the infrastructure of the continuum. Thanks to its vast community of open-source libraries and productivity

tools, it can be a sensible alternative even for high-level services.

The main four aspects of the Rust memory safety are:

- Ownership: the compiler tracks the lifetime of resources, like memory allocations, through variable assignments and function calls. A resource assigned to a variable is said to be owned by the variable. Passing a variable to a function causes the ownership of the resource to be transferred to the callee; this operation in known as a move in Rust terminology. After moving the resource out of a variable the variable can no longer be used to access the resource. When a variable goes out of scope the resource it owns is freed, i.e. the destructor is called;
- References: when a move is not desired, borrowing can be used. Borrowing a value creates a reference to it. References in Rust are pointers that are guaranteed to be valid at compile time. References are never null and always point to a live and valid memory location. To cope with the problem of unsynchronized mutation the Rust programming languages provides two kinds of references: immutable references and mutable references. At any time, the language allows many immutable references to the same memory location *or* a single mutable reference to exist;
- Lifetime: the compiler uses lifetimes to track the liveness of variables. The borrow checkers checks that borrows do not outlive (have a span longer than) the lifetime of the data they refer to;
- Concurrency: Rust provides two marker traits in the core library that are used to build concurrency primitives: Send and Sync. Send types can be transferred across thread boundaries, whereas Sync types are safe to share references between threads. Values are required to implement these traits to be accesses concurrently. Concurrent access is not limited neither to two or more cores accessing the value in parallel. Single core runtimes can likewise access a value concurrently. Examples of the latter scenario include POSIX signal handlers on a single-core system, time-sliced threads running on a single-core system, and interrupt handlers on a embedded system.

Rust is one of the few alternatives to C and C++ in the embedded software development space that can equally compete in terms of performance, low-level control and cross-platform support. Its use case is not limited to system development though. Although initially advertised as a system language, Rust is nowadays dedscribed as a language to build reliable and performant software. Rust software ranges from embedded systems and operative systems to web servers and user interfaces (e.g. via WebAssembly on the browser).

Its toolchain is backed by the LLVM modular compiler infrastructure, which it relies on for translating high-level code from languages such as C++ to its own intermediate representation (IR). In addition to code translation, LLVM also provides several tools which aid in tasks such as optimization and dead code elimination. Thanks to the integration of the LLVM, the Rust compiler rustc can transform this IR to generate WebAssembly binary code as well. The join of Rust and WebAssembly constitutes a powerful combination. Developers can write source code in Rust to achieve high productivity and yet efficient memory-safe binaries. WebAssembly can bring universal portability of such binaries without compiling or distributing multiple versions.

1) RTIC

Rust is gaining popularity as language for embedded systems as well. The language provides a rich standard library, which relies however on dynamic memory allocation provided by the host operating system, not available on embedded devices. Fortunately, the standard library is built on a core library, that neither requires dynamic allocations nor any host operating system and can thus be built for and executed on bare metal targets. Many Rust libraries that may be needed on embedded come with support for "no_std", i.e. the library can be compiled without the standard library.

The Rust compiler rustc supports a wide range of platforms and architectures as well. Passing a compilation target to rustc makes it compile the source code to machine code optimized for the target platform.

A further step towards the maturity of Rust as language for embedded systems is the framework Real-Time Interrupt-driven Concurrency (RTIC), previously known as Real-Time For the Masses (RTFM). RTIC is a framework developed by the Embedded Systems group at the Luleå University of Technology (LTU) for leveraging the interrupt controller to schedule tasks of different priority, while the risk of deadlock is mitigated thanks to all resource-accesses being handled by the Stack Resource Policy (SRP) [90].

Stack Resource Policy is a method to handle resource access in real-time systems with a single shared stack. RTIC uses SRP to handle access to resources and it guarantees a race and deadlock-free preemptive execution, along with bounded priority inversion. Additionally, the model does not use excessive memory for the stack since the tasks all share common stack space.

RTIC implements the SRP using hardware interrupts. An interrupt signal is a hardware-generated event that usually occurs asynchronously to the execution of the program. Examples of signals are: an electrical pin changed its logical state from 0 to 1 or vice versa, a counter (timer) reached a certain value, or data became available on some communication interface. In response to these signals the processor executes an interrupt handler, a special subroutine. Interrupts have also an associated priority level. In the case of the ARM Cortex-M architecture all interrupts have higher priority than the thread mode, reserved for the common procedures, so handlers will preempt code running in thread mode.

Preemption works as follows: the processor suspends the execution of the current subroutine, saves the state of that subroutine (the processor registers) onto the stack and then

jumps to the interrupt handler. The interrupt handler runs to completion and returns. Upon returning from the interrupt handler the state of the suspended routine is restored by popping the registers off the stack and the preempted routine is resumed.

In the Cortex-M architecture each interrupt signal is serviced by a different interrupt handler and up to 240 different interrupts are supported [91], although the exact number depends on the implementation and most vendors only use a few dozens of interrupts. All these interrupts are managed by the Nested Vector Interrupt Controller (NVIC) peripheral. The NVIC is a standardized interrupt controller found on all Cortex-M devices. The NVIC provides an interface to control and configure interrupts in the form of memory mapped registers.

The way the NVIC handles interrupts [92] exactly matches how the RTIC task model prioritizes tasks. This fact is used to provide a highly efficient implementation: the RTIC runtime lets the NVIC, the hardware, do all the task scheduling so no book-keeping needs to be done in software. The priority of the task is thus bound to the interrupt service routine (ISR) matching that priority, associating an ISR to each task. This alternative to time sliced threads provides several advantages like reduced overhead, efficient memory utilization, predictable performance (constant time overhead), deadlock freedom and suitability for static analysis.

H. ARCHITECTURE

Having presented the technology background, the architecture of this thesis is now presented. To the best of the author's knowledge, this thesis is the first of kind which tries to combine the vision to the development of a Proof of Concept (POC). Notably, the goal of the POC is to prove that the technologies mentioned above can be merged together to support the realisation of the continuum's infrastructure layer.

The infrastructure layer is composed of data and computational resources. The data can be generated by various streaming IoT devices (e.g. cameras, smart watches, and smart infrastructure). The computational resources can be heterogeneous and distributed through the infrastructure, from the cloud to the edges.

In order to support a truly ubiquitous system where services are everywhere and anywhere the end-user is, by even following the movements across zones, a federation layer is also needed. The federation layer is responsible for orchestrating the geographically distributed resources composing the infrastructure. However, this thesis will not explore this layer and it is left as open research question of future works on the continuum.

With the goal being said, a reference architecture is shown in Fig. 1. From a high-level perspective, users submit applications to the orchestrator. The orchestrator provisions the necessary resources and services according to the requirements indicated by the application, then assigns the latter to compute nodes in the cluster, where they are executed. The compute cluster is an abstraction of interconnected nodes that can be on different underlying infrastructures such as clouds or fog nodes.

Users submit their applications in the form of a manifest. The manifest describes resources requirements in terms of the amount of CPU and memory they will require for example. Likewise, the manifest contains the service requirements as well. Services may range from common dependencies like database and storage to sensor data, which the cluster will take care to provide based on the stated requirements. Service requirements allows developers to describe contraints on the services, e.g. latency needs and service plans. Unfortunately, as of time of writing, this thesis has not explored standard manifest formats to describe such service requirements. The POC leverages the Kubernetes YAML format, which specifies information like the container image, placement constraints, and resource requests.

1) Orchestrator control plane

The Kubernetes control plane is the core of the orchestration system. It has a resource monitor module responsible for keeping track of real-time resource consumption metrics for each node in the compute cluster. This information is usually accessed by the scheduler to make better optimization decisions. The scheduler is responsible for determining whether there are enough resources and services available in the cluster to execute the submitted application. In case resources are insufficient, applications can be rejected or put on wait until the resources are freed. Another possible solution would be to increasing the number of cluster nodes to place the incoming application. After determing if requirements can be satisfied, the scheduler maps application components, or more specifically Kubernetes Pods, onto the cluster resources. This is done by considering several factors, e.g. the availabilities, the utilization of the nodes, affinities, priorities, or constraints.

In cases where the addition of new cluster nodes is possible, an automatic process is necessary. The provisioner is responsible for dynamically adding nodes (physical or virtual) to the cluster when the existing resources are insufficient to meet the applications' demands. It will also decide when nodes are no longer required in the cluster and will shut the nodes down to prevent incurring in additional costs.

2) Compute nodes

Each machine in the cluster that is available for deploying Pods is a compute node, called Kubelet in Kubernetes terminology. Each of these nodes has an implementation of the Kubelet APIs with various responsibilities. First, it collects local information such as resource consumption metrics that can be periodically reported to the control plane. Second, it starts and stops Pods and manages local resources, via a container runtime or a Wasm runtime. Finally, it monitors the Pods deployed on the node, sending periodic status to the control plane.

The reason for the presence of two types of compute nodes lies in the lack of maturity of the Wasm ecosystem at the

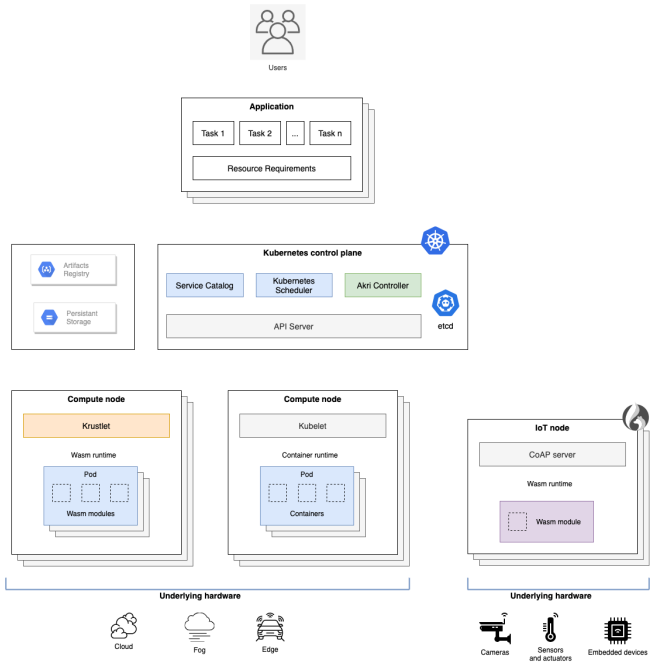

FIGURE 1. Reference architecture

time of writing. As mentioned previously, Krustlet is still far from reaching feature parity compared to a standard Kubelet implementation. As an example, support for the Kubernetes networking is not implemented yet, nor machine metrics are collected and sent to the control plane. Besides, as mentioned before, the current WebAssembly System Interface (WASI) standard doesn't have a complete networking support yet. Adding support for connecting to sockets is fundamental to allow Wasm modules to connect to web servers, databases, or any service. Because of the lack of network support, both as API in the Wasm module and as network configuration in the Kubernetes cluster, the POC uses a hybrid approach where conventional kubelet implementations and container runtimes are used as fallback.

3) IoT nodes

IoT nodes are embedded devices that act as sensors and actuators, provided as service to the cluster. The IoT nodes are heterogeneous in runtime implementation and communication protocols, therefore applications in the cluster interface with the them via Akri brokers as described before. However, in the POC of this thesis, embedded devices use the RTIC runtime implemented in Rust and support dynamic configuration by running arbitrary Wasm modules in a lightweight Wasm runtime.

The wasm runtime is an interpreter capable of executing any regular Wasm binary, assuming the file size and hardware requirements can be satisfied by the limited device.

4) Underlying infrastructure

One of the main benefits of containers is their flexibility in being deployed in a multitude of platforms. Because of these, the cluster machines can be either VMs on public or private cloud infrastructures, physical machines on a cluster, or even mobile or edge devices among others.

IV. EVALUATION

First, we evaluate our Sledge compiler aWsm and runtime on x86_64 and AArch64 architectures using two systems:

- MacBook Pro 15" 2018 with 6-Core Intel Core i7 2,2GHz and 16GB of RAM;
- Raspberry Pi 4 Model 3B+ with Quad-core Cortex-A53 (ARMv8) 64-bit SoC at 1.4GHz and 4 GB physical memory.

Sensor networks represent a crucial component in IoT environments [35]. For example, they can cooperate with RFID systems to better track things, get information about the position, movement, temperature, etc. Sensor networks are typically composed of a potentially high number of sensing nodes, communicating in a wireless multi-hop fashion.

Wireless sensor networks (WSNs) can provide various valuable data and are being utilised in several areas like healthcare, government and environmental services (natural disaster relief), defence (military target tracking and surveillance), hazardous environment exploration, seismic sensing, etc.

Recent technological advances have made efficient, low-cost, and low-power miniaturised devices available for large-scale remote sensing applications. In this context, the timely processing of massive and streaming sensor data, subject to

energy and network constraints and uncertainties, has been identified as the main challenge.

A. KUBERNETES

- OSB is not meant for machines (M2M) - The Device Plugin APIs is meant for hardware attached to a node, not for advertising edge nodes

B. WEB ASSEMBLY

The startup latency for initiating a function instance in a FaaS platform can vary significantly: from a few milliseconds to several seconds. A function execution might be a “warm-start", reusing a VM or container from the previous event of the same function, or it might be a “cold-start", where a new VM/container has to be launched.

Application developers employ work-arounds to avoid startup delays, including artificially activating functions (to avoid container shutdown) or over-provisioning resources required to run their functions. On a container-based serverless computing platform in an edge computing environment, containers that are long-lived or overprovisioned can quickly degrade performance due to limited available resources. Setup and control of these features requires multiple syscalls and IPC between container parent and child processes before a serverless function can begin executing.

Each Docker container consists of one or more separate processes, whereas each Wasm instance is contained within the same V8 process. This provides a Wasm-based solution the advantage of warmer caches and reduced context switch penalties, creating better potential for speedup when concurrently executing multiple serverless functions. Although in general a container’s overhead may be acceptable for long-running applications, this overhead quickly proves an impediment to meeting the low-latency demands of serving emerging IoT applications.

Wasm’s primary advantage over container-based solutions is the absence of a large cold start penalty. There are two reasons for this advantage. First, container runtimes such as Docker incur a large amount of overhead in ensuring their support for containers is as broad as possible.

The Sledge runtime decouples the heavy-weight function linking and loading process from function instantiation process.

Many of the existing Wasm runtimes can exhibit significant overhead in properly sandboxing code.

We must provide a way for our solution to limit an application’s execution time and maximum memory usage. In container-based platforms, this resource control is achieved via the Linux control groups feature. This feature allows one or more processes to be organized into groups which have their resources monitored and limited by the kernel. Maximum Memory Usage: current runtimes rely on WebAssembly’s JavaScript API to perform this memory creation operation. The WebAssembly.Memory() API function allows us to set initial and maximum sizes for the linear memory to be created. Execution Time: nope.

The use of WebAssembly in a serverless platform should not require application developers an undue amount of work in porting their existing code to the new runtime. Currently, support for many popular programming languages (e.g., C#, Go, Python, Java) is in active development across numerous open-source projects.

Creating several new programs and adapting existing popular libraries for use in WebAssembly, the amount of manual work involved was minimal.

There is an underlying issue with implementing network servers: there is no multi-threading support in WebAssembly - which means that a server running in WebAssembly is either going to be single-threaded, or its implementation would have to be significantly more complex (see Node’s event loop). At the same time, there is a WebAssembly threads proposal that would define operations for handling atomic memory access across threads, but it is only at stage 2 of the Wasm standardization process.

1) Wasm for microcontrollers

We use STM32F767ZIT6, a Cortex-M7 based microcontroller for evaluation of wasmi. It runs at 168MHz,128kB SRAM, and 5121KiB flash.

Linear memory is dynamically sized. It is extended in increments of a 64KiB page size.

Running SFI on resource constrained microcontrollers presents a number of challenges.

- Memory consumption. Wasm’s 64 KiB pages are too large for microcontrollers that often have between 16-256 KiB SRAM. [51] implementation departs from the standard by allowing byte-granularity, 1 KiB page-granularity
- Performance. Though the bounds checking on linear memory and the indirect checking of function pointer invocations are necessary for proper isolation, they add overhead over non-sandboxed code.
- Assumed ISA features. Wasm specifically is a 32-bit virtual architecture and supports floating point, but not all Cortex-Ms do. In such cases, floating-point emulation is broadly deployed.

C. RUST

Code generation is done via Rust macros 41 detailed in the advanced features part of the Rust book. Macros are what is known as metaprogramming, which essentially is writing code which manipulates code.

The downside to implementing a macro instead of a function is that macro definitions are more complex than function definitions because you’re writing Rust code that writes Rust code. Due to this indirection, macro definitions are generally more difficult to read, understand, and maintain than function definitions.

The tooling for debugging rust macros are not easy to use, even though tools like cargo-expand 42 significantly improves the situation. It produces the expanded output of

the macro, meaning that the code passed to the compiler can be inspected in order to verify correct operation.

D. RTIC

RTFM implements best effort scheduling based on static task priorities. For static analysis under Rate Monotonic scheduling, minimum inter-arrival is assumed to be larger or equal to the deadline of the task.

Static analysis and in particular Worst Case Execution Time (WCET) and stack usage analyses are required components of the certification process of safety critical software. Due to their age and popularity plenty of C and C++ tooling exists for static analysis: from sanitizers that find data races and memory bugs in software to static stack usage analysis tools that find the worst case stack usage of an application. Only a handful of these tools, namely sanitizers, can be used on Rust programs.

To indicate that these pointers need special attention from the programmer the language has the concept of unsafe operations. Unsafe operations cannot be proven to be memory safe by the compiler; the programmer must manually verify that they are indeed memory safe. This makes them easy to spot and facilitates the process of auditing Rust code for memory bugs.

When writing embedded Rust code only a subset of the standard library is available: the core library.

One can adopt a custom allocator, e.g. heapless which is fallible by construction (allowing the user to directly face OOM upfront, thus allowing system availability to be maintained, even if the allocator runs out of resources.

The problem that can arise with this layout is that the stack can grow too large and collide into the heap. This situation is called a stack overflow, or stack overrun.

cargo-call-stack is a Cargo subcommand that performs whole program stack usage analysis of a Rust application and produces a DOT file that contains the call graph of the program where each node in the call graph is annotated with its individual stack usage and cumulative (worst-case) stack usage.

if things go wrong and the documentation does not specify how things are done, many developers turn directly to the code for understanding the implementation details. As discussed previously understanding and then debugging Rust macros is one major roadblock many get dissuaded by due to the general complexity.

V. CONCLUSION

TODO

.

Appendixes, if needed, appear before the acknowledgment.

ACKNOWLEDGMENT

TODO

REFERENCES

[1] Haller S., Karnouskos S., Schroth C. (2009) The Internet of Things in an Enterprise Context. In: Domingue J., Fensel D., Traverso P. (eds) Future Internet – FIS 2008. FIS 2008. Lecture Notes in Computer Science, vol 5468. Springer, Berlin, Heidelberg.

[2] Mell, P., and Grance, T. 2011. The NIST definition of cloud computing, Recommendations of the National Istitute of Standards and Technology, NIST Special Publication 800-145.

[3] Dragoni N. et al. (2017) Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham.

[4] T Lynn, JG Mooney, B Lee, PT Endo. 2020. The cloud-to-thing continuum: opportunities and challenges in cloud, fog and edge computing.

[5] Roy Fielding. 2000. Representational State Transfer (REST). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Accessed on: Apr. 15, 2021.

[6] MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., Metz, R. and Hamilton, B.A., 2006. Reference model for service oriented architecture 1.0. OASIS standard, 12(S 18).

[7] Satyanarayanan, M., Bahl, P., Caceres, R. and Davies, N., 2009. The case for vm-based cloudlets in mobile computing. IEEE pervasive Computing, 8(4), pp.14-23.

[8] Nygren, E., Sitaraman, R.K. and Sun, J., 2010. The akamai network: a platform for high-performance internet applications. ACM SIGOPS Operating Systems Review, 44(3), pp.2-19.

[9] Prolonged AWS outage takes down a big chunk of the internet. 2020. [Online]. Available: https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet. Accessed on: Apr. 15, 2021.

[10] Latre, S., Famaey, J., De Turck, F. and Demeester, P., 2014. The fluid internet: service-centric management of a virtualised future internet. IEEE Communications Magazine, 52(1), pp.140-148.

[11] AbdelBaky, M., Zou, M., Zamani, A.R., Renart, E., Diaz-Montes, J. and Parashar, M., 2017, June. Computing in the continuum: Combining pervasive devices and services to support data-driven applications. In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS) (pp. 1815-1824). IEEE.

[12] Beckman, P., Dongarra, J., Ferrier, N., Fox, G., Moore, T., Reed, D. and Beck, M., 2020. Harnessing the Computing Continuum for Programming Our World. Fog Computing: Theory and Practice, pp.215-230.

[13] Shi, W., Cao, J., Zhang, Q., Li, Y. and Xu, L., 2016. Edge computing: Vision and challenges. IEEE internet of things journal, 3(5), pp.637-646.

[14] Yi, S., Hao, Z., Zhang, Q., Zhang, Q., Shi, W. and Li, Q., 2017, October. Lavea: Latency-aware video analytics on edge computing platform. In Proceedings of the Second ACM/IEEE Symposium on Edge Computing (pp. 1-13).

[15] Baresi, L., Mendonça, D.F. and Garriga, M., 2017, September. Empowering low-latency applications through a serverless edge computing architecture. In European Conference on Service-Oriented and Cloud Computing (pp. 196-210). Springer, Cham.

[16] Jang, S.Y., Lee, Y., Shin, B. and Lee, D., 2018, October. Application-aware IoT camera virtualisation for video analytics edge computing. In 2018 IEEE/ACM Symposium on Edge Computing (SEC) (pp. 132-144). IEEE.

[17] Botta, A., De Donato, W., Persico, V. and Pescapé, A., 2016. Integration of cloud computing and internet of things: a survey. Future generation computer systems, 56, pp.684-700.

[18] Zhang, Q., Cheng, L. & Boutaba, R. Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl 1, 7–18 (2010).

[19] He, W., Yan, G. and Da Xu, L., 2014. Developing vehicular data cloud services in the IoT environment. IEEE transactions on industrial informatics, 10(2), pp.1587-1595.

[20] Cisco fog computing solutions: Unleash the power of the Internet of Things. [Online]. Available: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-solutions.pdf. Accessed on Apr. 15, 2021.

[21] Mugarza, I., Amurrio, A., Azketa, E. and Jacob, E., 2019. Dynamic software updates to enhance security and privacy in high availability energy management applications in smart cities. IEEE Access, 7, pp.42269-42279.

[22] Mitton, N., Papavassiliou, S., Puliafito, A. and Trivedi, K.S., 2012. Combining Cloud and sensors in a smart city environment.

[23] Chen, B., Wan, J., Celesti, A., Li, D., Abbas, H. and Zhang, Q., 2018. Edge computing in IoT-based manufacturing. IEEE Communications Magazine, 56(9), pp.103-109.

[24] Gartner. Leading the IoT. 2017. [Online]. Available: https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf. Accessed on Apr. 15, 2021.

[25] Baresi, L., Mendonça, D.F. and Garriga, M., 2017, September. Empowering low-latency applications through a serverless edge computing architecture. In European Conference on Service-Oriented and Cloud Computing (pp. 196-210). Springer, Cham.

[26] Yousaf, F.Z., Bredel, M., Schaller, S. and Schneider, F., 2017. NFV and SDN—Key technology enablers for 5G networks. IEEE Journal on Selected Areas in Communications, 35(11), pp.2468-2478.

[27] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. and Gonzalez, J.E., 2019. Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

[28] Protocol Buffers. [Online]. Available: https://developers.google.com/protocol-buffers. Accessed on Apr. 15, 2021.

[29] OpenAPI Specification, Version 3.0.3. [Online]. Available: https://swagger.io/specification/. Accessed on Apr. 15, 2021.

[30] Shopify App Store. [Online]. Available: https://apps.shopify.com/. Accessed on Apr. 15, 2021.

[31] Naik, N., 2017, October. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In 2017 IEEE international systems engineering symposium (ISSE) (pp. 1-7). IEEE.

[32] Grüner, S., Pfrommer, J. and Palm, F., 2016. RESTful industrial communication with OPC UA. IEEE Transactions on Industrial Informatics, 12(5), pp.1832-1841.

[33] Grozev, N. and Buyya, R., 2014. Inter-Cloud architectures and application brokering: taxonomy and survey. Software: Practice and Experience, 44(3), pp.369-390.

[34] Bernstein, D., 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 1(3), pp.81-84.

[35] Yu, W., Liang, F., He, X., Hatcher, W.G., Lu, C., Lin, J. and Yang, X., 2017. A survey on the edge computing for the Internet of Things. IEEE access, 6, pp.6900-6919.

[36] Pereira, P.P., Eliasson, J., Kyusakov, R., Delsing, J., Raayatinezhad, A. and Johansson, M., 2013, March. Enabling cloud connectivity for mobile internet of things applications. In 2013 IEEE seventh international symposium on service-oriented system engineering (pp. 518-526). IEEE.

[37] Bormann, C., Castellani, A.P. and Shelby, Z., 2012. Coap: An application protocol for billions of tiny internet nodes. IEEE Internet Computing, 16(2), pp.62-67.

[38] Guinard, D., Trifa, V. and Wilde, E., 2010, November. A resource oriented architecture for the web of things. In 2010 Internet of Things (IOT) (pp. 1-8). IEEE.

[39] Gadepalli, P.K., McBride, S., Peach, G., Cherkasova, L. and Parmer, G., 2020, December. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. In Proceedings of the 21st International Middleware Conference (pp. 265-279).

[40] Elbamby, M.S., Perfecto, C., Liu, C.F., Park, J., Samarakoon, S., Chen, X. and Bennis, M., 2019. Wireless edge computing with latency and reliability guarantees. Proceedings of the IEEE, 107(8), pp.1717-1737.

[41] Security Overview of AWS Lambda. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.pdf. Accessed on Apr. 15, 2021.

[42] Caraza-Harter, T. and Swift, M.M., 2020, March. Blending containers and virtual machines: a study of firecracker and gVisor. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (pp. 101-113).

[43] Hall, A. and Ramachandran, U., 2019, April. An execution model for serverless functions at the edge. In Proceedings of the International Conference on Internet of Things Design and Implementation (pp. 225-236).

[44] Shillaker, S. and Pietzuch, P., 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIXATC 20) (pp. 419-433).

[45] Rosenblum, M. and Garfinkel, T., 2005. Virtual machine monitors: Current technology and future trends. Computer, 38(5), pp.39-47.

[46] Kernel Virtual Machine. [Online]. Available: https://www.linux-kvm.org/page/Main_Page. Accessed on Apr. 15, 2021.

[47] Bernstein, D., 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 1(3), pp.81-84.

[48] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.F., 2017, June. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 185-200).

[49] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S. and Crowcroft, J., 2013. Unikernels: Library operating systems for the cloud. ACM SIGARCH Computer Architecture News, 41(1), pp.461-472.

[50] Koller, R. and Williams, D., 2017, May. Will serverless end the dominance of linux in the cloud?. In Proceedings of the 16th Workshop on Hot Topics in Operating Systems (pp. 169-173).

[51] Peach, G., Pan, R., Wu, Z., Parmer, G., Haster, C. and Cherkasova, L., 2020. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(11), pp.3492-3505.

[52] Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/products/. Accessed on Apr. 15, 2021.

[53] AWS Lambda@Edge. [Online]. Available: https://aws.amazon.com/it/lambda/edge/. Accessed on Apr. 15, 2021.

[54] AWS IoT Greengrass. [Online]. Available: https://aws.amazon.com/greengrass/. Accessed on Apr. 15, 2021.

[55] Azure IoT Edge. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/. Accessed on Apr. 15, 2021.

[56] Pahl, C., Helmer, S., Miori, L., Sanin, J. and Lee, B., 2016, August. A container-based edge cloud paas architecture based on raspberry pi clusters. In 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW) (pp. 117-124). IEEE.

[57] Bellavista, P. and Zanni, A., 2017, January. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In Proceedings of the 18th international conference on distributed computing and networking (pp. 1-10).

[58] Google Cloud Functions. [Online]. Available: https://cloud.google.com/functions. Accessed on Apr. 15, 2021.

[59] Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/. Accessed on Apr. 15, 2021.

[60] AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/. Accessed on Apr. 15, 2021.

[61] Firecracker. [Online]. Available: https://firecracker-microvm.github.io/. Accessed on Apr. 15, 2021.

[62] Mohanty, S.K., Premsankar, G. and Di Francesco, M., 2018, December. An Evaluation of Open Source Serverless Computing Frameworks. In CloudCom (pp. 115-120).

[63] Elbamby, M.S., Perfecto, C., Liu, C.F., Park, J., Samarakoon, S., Chen, X. and Bennis, M., 2019. Wireless edge computing with latency and reliability guarantees. Proceedings of the IEEE, 107(8), pp.1717-1737.

[64] Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L., 1993, December. Efficient software-based fault isolation. In Proceedings of the fourteenth ACM symposium on Operating systems principles (pp. 203-216).

[65] Docker Hub. [Online]. Available: https://hub.docker.com/. Accessed on Apr. 15, 2021.

[66] Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K. and Sears, R., 2010, April. MapReduce online. In Nsdi (Vol. 10, No. 4, p. 20).

[67] He, J., Wei, J., Chen, K., Tang, Z., Zhou, Y. and Zhang, Y., 2017. Multitier fog computing with large-scale iot data analytics for smart cities. IEEE Internet of Things Journal, 5(2), pp.677-686.

[68] Kubernetes. [Online]. Available: https://kubernetes.io/. Accessed on Apr. 15, 2021.

[69] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J. and Bailey, J., 2017, August. The quic transport protocol: Design and internet-scale deployment. In Proceedings of the conference of the ACM special interest group on data communication (pp. 183-196).

[70] containerd. [Online]. Available: https://containerd.io/. Accessed on Apr. 15, 2021.

[71] Docker Swarm overview. [Online]. Available: https://docs.docker.com/engine/swarm/. Accessed on Apr. 15, 2021.

[72] Hoque, S., De Brito, M.S., Willner, A., Keil, O. and Magedanz, T., 2017, July. Towards container orchestration in fog computing infrastructures. In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC) (Vol. 2, pp. 294-299). IEEE.

[73] Cloud Foundry Launches Online Marketplace for Expanding Ecosystem. [Online]. Available: https://www.cloudfoundry.org/blog/cloud-foundry-launches-online-marketplace-expanding-ecosystem/. Accessed on Apr. 15, 2021.

[74] Open Service Broker. [Online]. Available: https://www.openservicebrokerapi.org/. Accessed on Apr. 15, 2021.
[75] Akri. [Online]. Available: https://github.com/deislabs/akri. Accessed on Apr. 15, 2021.
[76] Kubernetes Device Plugins. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/. Accessed on Apr. 15, 2021.
[77] gRPC. A high performance, open source universal RPC framework. [Online]. Available: https://grpc.io/. Accessed on Apr. 15, 2021.
[78] TypeScript. [Online]. Available: https://www.typescriptlang.org/. Accessed on Apr. 15, 2021.
[79] Microsoft spots malicious npm package stealing data from UNIX systems. [Online]. Available: https://www.zdnet.com/article/microsoft-spots-malicious-npm-package-stealing-data-from-unix-systems/. Accessed on Apr. 15, 2021.
[80] Reference Types Overview. [Online]. Available: https://github.com/WebAssembly/reference-types/blob/master/proposals/reference-types/Overview.md. Accessed on Apr. 15, 2021.
[81] LLVM. [Online]. Available: https://llvm.org/. Accessed on Apr. 15, 2021.
[82] WebAssembly on Cloudflare Workers. [Online]. Available: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. Accessed on Apr. 15, 2021.
[83] Announcing Lucet: Fastly's native WebAssembly compiler and runtime. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime. Accessed on Apr. 15, 2021.
[84] Write Wasm smart contracts with ink! 2.0. [Online]. Available: https://www.parity.io/write-wasm-smart-contracts-with-ink-2-0/. Accessed on Apr. 15, 2021.
[85] wasm3. [Online]. Available: https://github.com/wasm3/wasm3. Accessed on Apr. 15, 2021.
[86] wasmtime. [Online]. Available: https://github.com/bytecodealliance/wasmtime. Accessed on Apr. 15, 2021.
[87] wasm3 performance. [Online]. Available: https://github.com/wasm3/wasm3/blob/main/docs/Performance.md. Accessed on Apr. 15, 2021.
[88] Brzoza-Woch, R., Konieczny, M., Nawrocki, P., Szydlo, T. and Zielinski, K., 2016, May. Embedded systems in the application of fog computing—Levee monitoring use case. In 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES) (pp. 1-6). IEEE.
[89] Golang data races to break memory safety. [Online]. Available: https://blog.stalkr.net/2015/04/golang-data-races-to-break-memory-safety.html. Accessed on Apr. 15, 2021.
[90] Baker, T.P., 1990, December. A stack-based resource allocation policy for realtime processes. In [1990] Proceedings 11th Real-Time Systems Symposium (pp. 191-200). IEEE.
[91] Joseph, Y., 2014. The definitive guide to ARM Cortex-M3 and Cortex-M4 processors.
[92] Interrupt inputs and pending behaviors, Y., 2014. The definitive guide to ARM Cortex-M3 and Cortex-M4 processors. pp 246.