An Architecture for Interspatial Communication

Anil Madhavapeddy, KC Sivaramakrishnan, Gemma Gordon
Computer Laboratory
University of Cambridge
Email: first.last@cl.cam.ac.uk

Thomas Gazagnaire
Tarides
Paris, France
Email: first@tarides.com

***Abstract*—Digital infrastructure in modern urban environments is currently very Internet-centric, and involves transmitting data to physically remote environments. The cost for this is data insecurity, high response latency and unpredictable reliability of services. In this paper, we lay out a software architecture that inverts the current model by building an operating system designed to securely connect physical spaces with extremely low latency, high bandwidth local-area computation capabilities and service discovery. We describe our early prototype OSMOSE, which is based on unikernels and a distributed ledger store.**

I. Introduction

In his classic essay on "Computing for the 21st Century" [1], Mark Weiser observed that:

> The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

Since then, there have been tremendous advances in mobile and sensing technologies, and there is an ongoing rapid deployment of "smart" digital infrastructure that augments the physical environment. Consider the following scenario for a next-generation building:

"A group of people are meeting in an coworking space to discuss a project they are working on together. They are the first to arrive at the building on a cold winter morning, and are directed to a meeting room automatically by audio directions individually projected to them. As they walk into the room the lights are already on and the heating has been preset to a comfortable level. They wave to a wall where a display appears, and the building recognises the gesture and projects a shared folder of the project. It begins to record and encrypt the conversation with a group secret keyed to the participants. One of the participants receives a tight-beam audio notification informing them that they are needed elsewhere briefly. When they leave the room the shared recording is immediately paused within milliseconds to allow other participants to casually chat among themselves. Subsequently, the encrypted recording is transcribed, and made available in the datastores of each participant under the terms of their shared disclosure agreement for their project."

This narrative illustrates the distance between Weiser's 1999 vision of ubiquitous computing and where we currently are almost two decades on. The above scenario features *no wearables or mobiles* among the human participants – instead of an array of individual smart phones, watches and laptops, the situated environment of the building is providing secure, shared, multi-tenant infrastructure for audio and visual communications among the human participants. This is analogous to the shift from individual computers to multi-tenant cloud infrastructure in the past decade [2], but applied to the sharing of physical places and devices.

Another aspect to consider in the scenario is the seamless, low-latency nature of the interactions that were described between humans and electronic devices. When a human takes an action the environment is able to react within milliseconds to take action immediately, rather than delaying for seconds and causing the awkward delays that we have become used to with modern Internet-connected mobile devices. By providing immediacy, the technology can complement social interactions rather than interrupting them [3].

Encouragingly, all of the hardware required to physically assemble the described building is available off the shelf. E-ink display wallpaper, parametric audio speakers for directional broadcast, gesture recognition and smart lighting and heating are all available or relatively easy to construct. The missing link is the foundational software that manages, coordinates and secures the distributed hardware – the operating system for digitally connecting physical spaces together.

In this position paper, we begin to bridge this gap between available hardware and missing software. We construct an architectural model for *interspacial networking* – an operating system for the dense, interconnected and shared urban spaces that most humans live in. It aims to shift us away from the wide-area, mobile-oriented devices that are permeating early deployments of smart infrastructure, and towards a sustainable digital model that is far more similar to a conventional utility such as electricity or gas. Our system aims to make it far simpler and more secure to introduce, manage and rely on digital devices into day-to-day life in urban environments.

There is also an exciting new generation of upcoming applications that demand vast amounts of bandwidth and low-latency responses. Augmented and virtual reality, environmental e-ink displays, parametric directional sound, and robotic appliances simply do not work well if the latencies of interactions with them rise above a certain point. This paper proposes an operating software architecture aimed at fully supporting these new applications on modern hardware platforms. We will next describe some of the challenges in more detail, then present the design of our OSMOSE prototype OS (§II), and finally discuss some of the implications (§III).

A. The Problem with Existing Mobile Software

Given that the hardware is all available, why is the above scenario difficult to implement? The answer lies in the traditional operating systems and network architecture that power the current generation of smart devices. Because of the rise of cloud computing, they are typically built around communication to centralised Internet services.

For example, consider what happens when we speak into an Apple Watch in order to retrieve some information. For this to work, the watch must be connected to a mobile phone, which in turn needs to establish a cellular or wifi connection to the Internet, where Apple voice recognition services will dispatch a search query to Microsoft Bing servers. If any one of the services in this chain breaks (for example, the common case of the phone signal being "trapped" by a wifi authentication page), then the user experience is broken. Even when it does work, it can take seconds to respond the voice query, and with very variable latency. Once the response does come through, handoff to other devices is also difficult unless they are owned by the user.

Consequent to this centralised architecture, we are hitting several problems with deployment of digital devices in situated environments.

1) Reliability: Services deployed in physical environments need to work all the time, and be locally debuggable when they do fail. How do we shift from a light switch that "almost works" after being pressed several times to seamless voice or gesture-driven services that are always tuned and available in a given environment? They also need to work independently of Internet connectivity, so that every building can be an island of digital services even when offline.

2) Efficiency: Modern IoT devices often embed entire copies of Linux in them, and include wide-area cellular connectivity even if they are only used locally. This is often a terrible energy tradeoff compared to traditional "low-tech" solutions that have local physical connectivity. We need to slim down the devices to require less software running on each device, and rely on the situated environment to provide more operational support.

3) Security: The amount of sensitive data being captured in these environments is tremendous, and much of it should not leave the confines of the physical space without explicit permission from all parties involved. This is extremely difficult to police given the amount of Internet-wide coordination used in existing devices, but can be fixed if the local environment provides a structured mechanism for handling such storage securely with respect to local environmental policies.

4) Latency: Interactive services require response times beneath the uncanny valley of human perception. For the scenario above to really feel seamless, we need a new "latency first" application architecture that makes data and computation capacity available physically near the human users, with scheduling tolerances for responses in the milliseconds rather than seconds.

Solving these problems is difficult to do piecemeal across individual parts of the software stack, since they are currently general-purpose and loosely coupled. A typical IoT device might run its own copy of the Linux kernel, with an embedded userspace, a VPN into a centralised management server operated by the vendor for updates, and a mobile application for the user to manage it. There is little synergy or dependence on shared infrastructure to assist with the process.

B. Replacing the Existing Software Architecture

We now describe what a cross-cutting software solution to operating embedded physical infrastructure might look like, if designed with a clean slate in mind. The first step to making this a practical prospect is to adopt the discipline of deploying *unikernels* [4] on the hardware, and replacing the traditional layered operating system stack.

Unikernels are specialised operating systems that are compiled together with application source code and configuration, resulting in a specialised binary that can boot in milliseconds and eliminates traditional runtime layers in favour of optimised build-time assembly. Unikernels were originally developed from the concept of library operating systems [5] and applied to cloud computing, and we now argue that the same approach is a perfect fit to the world of resource-constrained embedded systems.

In a library operating system, all of the software layers are linked together with a small boot layer. This includes traditionally kernel-based interfaces such as hardware device drivers, which can now run in the same privilege level as the application driving them. This model is ideal for embedded devices, since it can result in direct low-latency and energy-efficient access to the hardware. Crucially, the approach also allows the application to be tailored to the operational model of the embedded hardware. For example, the same application source code could be compiled to a tiny embedded processor without an MMU, or also be recompiled into a conventional Unix process. One downside of this specialisation approach is that traditional multi-user (e.g. UNIX processes) operation is more difficult in a single device once the unikernel has been deployed (we address this in §II-B)

The unikernel approach lets us close the gap between the requirements of application programming interfaces and the diverse requirements of the embedded hardware. Instead of forcing vendors to squeeze an ever-growing operating system stack for trivial tasks (such as driving a lightswitch), adopting unikernel-based interfaces means that the same codebase can be repurposed and tailored across the variety of hardware that we can expect to see in a typical situated environment in the next few decades.

C. Towards Interspatial Applications

We can now map application interfaces directly onto the diverse hardware found in physical environments, and so turn our attention to how the applications themselves might operate.

When dealing with physical devices, the latency of response to external events is paramount. Safety critical systems often require "hard realtime" operation, or more often attempt to provide soft guarantees about when they respond. As we move

further up the stack to the modern mobile application, there are no such guarantees. Pressing a button on an iOS or Android device goes through many layers of scheduling and network connectivity before a response is generated.

Our interspacial architecture will let us deploy complex "latency-first apps" into physical environments – ones that are designed to run locally with response budgets in the milliseconds, and with minimal external connectivity needs. This implies that there is simply no time for doing conventional operations such as network scanning or service discovery in serial – by the time the user has asked a question, the physical environment should already have the appropriate resources established.

While it is currently possible to spin up services on demand within milliseconds [6] it is difficult to conduct an end-to-end response for complex clustered services within a time budget of a few milliseconds. A device that is starting "cold" will need to establish a network connection, authenticate the user(s), negotiate security keys, perhaps perform version negotiation between devices, and usually interrupt the user at an inconvenient time for a security update.

The reason that every device has to currently do all this work is partly due to the end-to-end principle that guides the design of Internet protocols. Every node is a "dumb" router that knows how to forward protocol packets, but lacks higher-level application information. Existing system interfaces (such as the BSD sockets API, or DNS resolution) implement this network stack, which is in turn baked into existing application logic. Every single node needs to repetitively perform the same actions to establish connectivity, with the surrounding network environment being unable to assist due to the end-to-end principle.

To gain back our latency budget, we reach back in time to an older design based on circuit switching [7] – as seen in the venerable telephone network. In this model, each hop in the network establishes a connection to its peers and supplies a simpler interface to each node. Each node is aware of all connections emanating to its peers, which means that it can choose how to schedule or optimise traffic across links locally. For our interspacial applications, this is exactly what we want – a building should be able to precisely manage all the local resources (include bandwidth or storage) just as it does so with other utilities such as electricity. In return, individual applications do not need to manage their own networking, storage and authentication stacks – they are provided by the surrounding environment.

D. Scheduling Ambient Dataflow

Our interspatial applications require a lot of immediate interaction with the physical environment. This could involve reading sensor input (e.g. gesture recognition, audio inputs, pressure sensors) and actuating outputs to sensors (e.g. heating, lighting, speakers, displays). The interspatial operating system thus establishes network circuits to include a *physical network topology* that describes the containment relationships present in a physical environment – a chair is on a floor beside a wall within a room that is on the first floor of a building in Cambridge in the United Kingdom.

As users interact with their environment, network circuits are set up as they move *in parallel* with their actions. In our earlier scenario, when the user first enters the building they are authenticated with the building systems and given a pseudonym that is used to subsequently track them. As they walk towards the meeting room, their preferences are reconciled with other participants. Once they enter the room, they are allocated a network slot inside that room's wireless network, and their voiceprint details uploaded to local embedded processors. Every time a new participant enters, their collection of pseudonyms are aggregated to generate temporary encryption keys for any shared communication. All video and voice are encrypted in real-time using these keys and saved to secure storage enclaves within the walls, available for immediate processing by locally connected computation capacity. Once a participant leaves a space, their security keys expire and the data is immediately garbage collected, ready for re-use by other humans.

This design inverts the conventional model of establishing on-demand connections to remote services with a model that incrementally establishes nearby circuits for applications, with local data processing and resource allocation done as the humans move around the situated environment in real time. Since connections are established incrementally as humans move around, when an action actually has to be taken it can be done so with a single-hop packet, thus minimising round-trip times and error-prone connection establishment protocols that would normally add latency variance to an interaction.

II. The Interspatial Operating System

We now present a prototype interspatial operating system based on a unikernel and circuit switching software architecture, dubbed OSMOSE. Its scope is to drive all the hardware in a single physical space – such as a building – including the thousands of sensors and actuators that may be present. As background requirements, we assume that:

- we have a reliable physical topology available of the situated environment. Details are out of scope for this position paper – there are a number of indoor localisation technologies available [8].
- users and devices have an out-of-band method to authenticate to the operating system – this might be biometric or face recognition or a simple password, but this has to happen as they enter the physical space.
- there is ample hardware available locally for applications – for example an array of embedded devices in each room – and the hardware is all connected via room-local reliable wired networks, with wireless at the last hop only.

Since OSMOSE needs to run across a variety of hardware with differing resource constraints, the traditional model of booting a single image no longer holds. Instead of application binaries, the input to the system is a collection of declarative application fragments that represent processing logic and how it ties together. This is then combined with physical topology

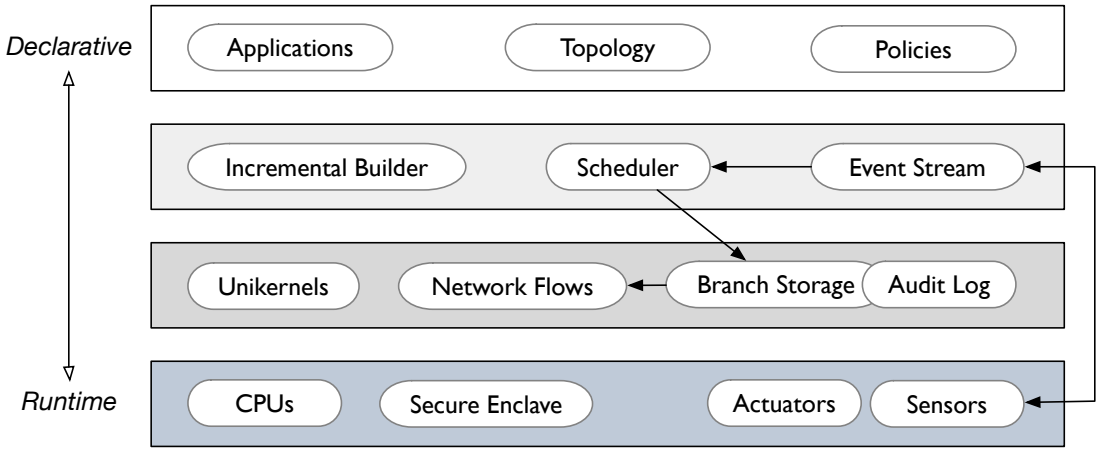

Fig. 1. Architecture of the interspacial operating system. The system is configured declaratively via a set of application source code, policies and network topology, and is then incrementally built and deployed onto hardware. All communications happens via the Merkle-tree based branch storage.

information and security policies to compile a set of unikernels that are booted on the building's hardware. This cycle of compilation happens continuously and incrementally – as applications are introduced or policies change, the unikernels are recompiled in real-time to all the devices.

Applications do not communicate freely between each other over the network with conventional TCP/IP – instead, the OS sets up explicit channels that provide a high-level streaming interface suited to the application needs (for example, a low-bandwidth control channel *vs* a high-bandwidth video processing ring buffer). The resulting set of communication circuits is tracked as a dynamic dataflow graph, with the building hardware representing nodes and the application logic as edges. All communications in the system are dispatched through a distributed storage system that immutably stores interactions for audit and debugging purposes.

Figure 1 illustrates the overall architecture of OSMOSE. It implements these components using the MirageOS[1] unikernel framework, written in the OCaml programming language.

A. Declarative Layer

Applications are partitioned up into declarative source code fragments that represent units of computation, and then scheduled together by via a structured event stream using the Capnp[2] RPC framework. This is becoming a popular architecture on container-native computing infrastructures via the Serverless movement [9], since it allows applications to run on distributed virtual machines that can be rescheduled depending on load and hardware availability. In an interspatial deployment, the application logic is compiled to run directly on the physical hardware as a unikernel [10], with appropriate network channels established depending on the available circuits (e.g. a local point-to-point wireless link).

The services are composed together via the domain-specific language of combinators provided by MirageOS. For example, a voice recognition application can request an audio stream, which can then be mapped onto a feature extraction module. The application does not know the exact source of the audio stream at this stage. In the case of our prototype, we use the MirageOS device driver model to define many of these sources and sinks. The resulting programming model is event-driven and reactive, and encourages the programmer to provide ongoing incremental updates so that user interfaces can be updated rapidly.

B. Builder and Artefact Layer

Once we have all the source code (in the form of application logic, topology and configuration code), we need to compile it into executable unikernels that can run on the available hardware. Unlike a conventional operating system that runs on a single hardware host, the code is compiled into a heterogenous set of bootable kernels that are specialised to the various devices available in the cluster. For example, a portion of the application logic may be compiled to run on a GPU or FPGA, and another piece to run on an embedded ARM CPU without an MMU. This is accomplished via an incremental build service that can rapidly link and specialise the source code into unikernels.

The build service tracks all of the source code that via a branching datastore that is similar to Git. Every single line of application, operating system and configuration logic is stored in the same place, and so the builder can easily track precisely what code is running on the array of hardware. We use the Irmin [11] datastore in our prototypes. Irmin is a unikernel implementation of the Git model for MirageOS, and provides the facility to handle complex distributed datastructures just like source code.

Irmin is structured as a series of OCaml libraries that expose increasingly complex storage functionality. Applications request the minimal functionality they need for their purposes, and the appropriate storage layer is crafted from the combination of requirements. For example, one application (a button) might just need a key-value store, whereas another

[1]See https://mirage.io
[2]See http://capnproto.org

(a projector) might want a read-only filesystem. Importantly, all of the storage is backed by the same Merkle-tree-based storage, granting us the ability to precisely track the behaviour of every hardware node.

C. Scheduler Layer

The scheduler is responsible for triggering the builder, deploying the resulting unikernels to hardware, and establishing network circuits between devices. It receives an event stream from the hardware (such as network topology changes, or node failures). This layer is the most radical departure from a conventional operating system model. Instead of a process-based system, a deployment can be viewed as a graph of distributed hardware nodes, through which event stream data from the environment flows.

Every single coordination operation between nodes is piped through the Irmin datastore, meaning that the complete state of the system can be tracked with strong provenance. We do assume that there is a large amount of persistent local storage available in the deployment, but this can run in a separate processing node from the low-power embedded hardware. The basic model for the Irmin-based communication has been validated over the past decade via a series of shared-memory implementations in the Xen hypervisor [12]. In order to keep latency down despite the use of Merkle-tree-based coordination, Irmin can expose a shared-memory endpoint to each node, and local operations between two nodes are asynchronously reconciled to a remote store via a set of short-lived branches.

When applications need access to the outside world, they do so via RPC calls to the scheduler. We use the Capnp RPC framework for this purpose, which effectively acts as the system call layer. Capnp features a very efficient low-level serialisation mechanism (important when communicating between embedded devices), but also a compiler for protocol schemas that integrates secure capabilities [13]. These capabilities can be passed around nodes as opaque references and used to authenticate access to different parts of the system. Since a capability can only be generated and communicated through the Irmin store, this means that tracing distributed function calls through a deployment are available "for free" by inspecting and reconstructing the storage.

D. Hardware Layer

The unikernels stored in the artefact layer are regularly booted on the various pieces of embedded hardware throughout the deployment. There is no need for dynamic network discovery since static topology information is baked into every booting kernel. If the network environment changes, then the scheduler will trigger a recompile via the branch store and perform a rolling upgrade. This is a similar discipline to a Continuous Integration / Deployment pipeline commonly adopted in the cloud, except applied to embedded firmware via unikernels. It is a model that is easily possible as a result of the pervasive use of the Irmin datastore.

At an individual device level, it is now common to find domain-specific hardware. The most notable is the presence of a secure enclave (e.g. SGX or Trustzone) that can be used to store private user data [14]. All of the data that is captured from sensors can only be committed to the local Irmin store, and so it can be directly encrypted via a secure enclave before ever being stored persistently. Although this may seem expensive from a resource usage perspective, the data is not stored for long periods of time, and nor does it need to be transmitted externally. It can, however, be buffered for long enough for interested local applications to process it directly (e.g. transcribe a voice recording to a smaller, but still encrypted, textual equivalent).

The design of OSMOSE that we have described in this section is still quite a low-level one, and is concerned with establishing the basic primitives required to boot and run unikernel code on an array of embedded hardware. However, the use of a schema-based RPC compiler for inter-component communication means that it is very easy to extend the system to accommodate new device drivers and data processing models. The MirageOS application framework that we are using has been under development since 2007, and has a growing set of library-based abstractions for networking, storage and coordination that work well for cloud-based deployments. We anticipate that as OSMOSE matures, there will be many equivalent abstractions (e.g. real time video processing) contributed for interspacial device drivers.

III. Discussion

Our OSMOSE prototype brings the traditional benefits of an operating system to the distributed array of hardware that comprises a physical building — resource scheduling, hardware isolation and a userlevel programming interface. The programming model is aimed at building real-time, interactive interfaces that can do complex data processing local to the user, without having to ship data remotely to the cloud.

Any operating system is only worth the applications that run on it. While our application interfaces are intentionally not backwards compatible with existing frameworks to give us room to experiment, their programming model is a familiar one to those who program cloud-based infrastructure. We are hoping that the combination of a successful microservice-like service model combined with automated deployments on embedded devices [15] will be compelling to developers. Our next task is to design user studies of particularly latency-sensitive interactive applications and evaluate them in OSMOSE versus a conventional cloud-driven deployment.

Cyberphysical security is vital to creating a trustworthy digital future for our smart environments. Accordingly, every layer of OSMOSE is engineered with this in mind. At the lower levels, the unikernel approach is designed to eliminate unnecessary code and promote the use of safer programming languages throughout the software stack. At the hardware level, unikernels can compile flexibly and efficiently enough to make use of the limited resources in CPU-based secure enclaves [14]. At the storage level, the use of capabilities

and distributed ledgers (aka "blockchain") to track provenance throughout the system means that there is accountability builtin from the ground up.

Pervasive data recording comes with some obvious drawbacks. The European General Data Protection Regulation that is being introduced from mid-2018 means that any interspacial deployment also needs to engineer *deletion* for all the tracked data. OSMOSE is designed to ensure that deletion is an expensive but reliable operation – since every bit of data is tracked, a distributed garbage collector can traverse everything and rewrite history on all nodes to remove any individual data that should be excised. The system also regularly runs garbage collection on data to compact or delete it once the immediate processing needs are met. Most obviously, OSMOSE is designed to move computation to the data instead of uploading the data to a remote cloud. This is the biggest improvement for security and privacy for individuals – they can benefit from the futuristic augmented reality interactions while knowing that their data is kept physically on the premises.

In order to meet Weiser's vision of truly ubiquitous computing, the applications running on OSMOSE also need to provide the same or better levels of data analysis as cloud-based recommender systems currently operate. There is complementary research ongoing to house data silos near the user [16], and combining this local data with broader global data from online social networks [17]. The data capture systems in OSMOSE can replace the role currently taken by mobile devices [18], and also provide a corresponding increase in accuracy due to not requiring the data to be transmitted remotely before being processed.

IV. Conclusion

We have presented an early design for a distributed operating system designed for deploying a new generation of low-latency, interactive applications in urban environments. Our design inverts the existing model of funnelling data to the cloud, and instead provides the infrastructure for rapidly processing data locally. In return, this will provide a foundation for sustainably and securely managing the trillions of embedded devices that form the emerging smart cities movement.

Our next steps are to implement and evaluate realistic next-generation interactive applications on this platform, and build a real physical prototype in a building environment. Source code is open-source under a BSD-style license and available from https://github.com/mirage.

Acknowledgements

The authors would like to thank Mark Wormald from Pembroke College for coining the term "interspacial". We also thank the MirageOS development team for their contributions that made OSMOSE possible, and in particular to Thomas Leonard for writing the Capnp MirageOS library.

References

[1] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999. [Online]. Available: http://doi.acm.org/10.1145/329124.329126

[2] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, Jul. 2008.

[3] J. Deber, R. Jota, C. Forlines, and D. Wigdor, "How much faster is fast enough?: User perception of latency & latency improvements in direct and indirect touch," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 1827–1836.

[4] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGPLAN Not.*, vol. 48, no. 4, pp. 461–472, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451167

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266.

[6] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[7] G. W. Luderer, H. Che, J. P. Haggerty, P. A. Kirslis, and W. T. Marshall, "A distributed unix system based on a virtual circuit switch," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 160–168.

[8] Z. B. Tariq, D. M. Cheema, M. Z. Kamran, and I. H. Naqvi, "Non-gps positioning systems: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 57:1–57:34, Aug. 2017.

[9] A. Kanso and A. Youssef, "Serverless: Beyond the cloud," in *Proceedings of the 2Nd International Workshop on Serverless Computing*, ser. WoSC '17. New York, NY, USA: ACM, 2017, pp. 6–10.

[10] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici, "Unikernels everywhere: The case for elastic cdns," *SIGPLAN Not.*, vol. 52, no. 7, pp. 15–29, Apr. 2017.

[11] B. Farinier, T. Gazagnaire, and A. Madhavapeddy, "Mergeable persistent data structures," in *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, D. Baelde and J. Alglave, Eds., Le Val d'Ajol, France, Jan. 2015.

[12] T. Gazagnaire and V. Hanquez, "Oxenstored: An efficient hierarchical and transactional database using functional programming with reference cell comparisons," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '09. New York, NY, USA: ACM, 2009, pp. 203–214.

[13] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Comput. Surv.*, vol. 8, no. 4, pp. 409–445, Dec. 1976.

[14] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "Scone: Secure linux containers with intel sgx," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 689–703.

[15] P. Persson and O. Angelsmark, "Kappa: Serverless iot deployment," in *Proceedings of the 2Nd International Workshop on Serverless Computing*, ser. WoSC '17. New York, NY, USA: ACM, 2017, pp. 16–21.

[16] A. Chaudhry, J. Crowcroft, H. Howard, A. Madhavapeddy, R. Mortier, H. Haddadi, and D. McAuley, "Personal data: Thinking inside the box," *Aarhus Series on Human Centered Computing*, vol. 1, no. 1, p. 4, 2015.

[17] S. S. Rodríguez, L. Wang, J. R. Zhao, R. Mortier, and H. Haddadi, "Personal model training under privacy constraints," *CoRR*, vol. abs/1703.00380, 2017.

[18] S. A. Ossia, A. S. Shamsabadi, A. Taheri, H. R. Rabiee, N. D. Lane, and H. Haddadi, "A hybrid deep learning architecture for privacy-preserving mobile analytics," *CoRR*, vol. abs/1703.02952, 2017.