Isomorphic Internet of Things Architectures With Web Technologies

Mikkonen, Tommi

2021-07

Mikkonen , T , Pautasso , C & Taivalsaari , A 2021 , ' Isomorphic Internet of Things Architectures With Web Technologies ' , Computer : a publication of the IEEE Computer Society , vol. 54 , no. 7 , pp. 69-78 . https://doi.org/10.1109/MC.2021.3074258

http://hdl.handle.net/10138/336610
https://doi.org/10.1109/MC.2021.3074258

unspecified
acceptedVersion

Isomorphic IoT Architectures with Web Technologies

Tommi Mikkonen[1], **Cesare Pautasso**[2], **Antero Taivalsaari**[3]
[1]**University of Helsinki, Helsinki, Finland**
tommi.mikkonen@helsinki.fi
[2]**USI, Lugano, Switzerland**
cesare.pautasso@usi.ch
[3]**Nokia Bell Labs & Tampere University, Tampere, Finland**
antero.taivalsaari@nokia-bell-labs.com & antero.taivalsaari@tuni.fi

Abstract

A typical Internet of Things (IoT) system consists of a large number of different subsystems and devices, including sensors and actuators, gateways that connect them to the Internet, cloud services, end-user applications and analytics. Today, these subsystems are typically implemented with a broad variety of programming technologies and tools, making it difficult to migrate functionality from one subsystem to another. In this paper, we predict the rise of *isomorphic* IoT system architectures in which all the subsystems will be developed with a consistent set of technologies, thus allowing different parts of the system to run the same code.

Keywords: Isomorphic Software, Software Architecture, Internet of Things, IoT, Software Deployment, Deployment in the Large, Programmable World

Introduction

Recent years have witnessed an avalanche of digitalization technologies. Processing capabilities have grown dramatically, cloud computing has become commodity, data science has blossomed due to increasing amounts of data, and Artificial Intelligence and Machine Learning (AI/ML) have emerged as everyday technologies even in devices with limited capabilities such as mobile phones. These changes are leading us to a *Programmable World* [17] where everyday things around us will become connected and programmable.

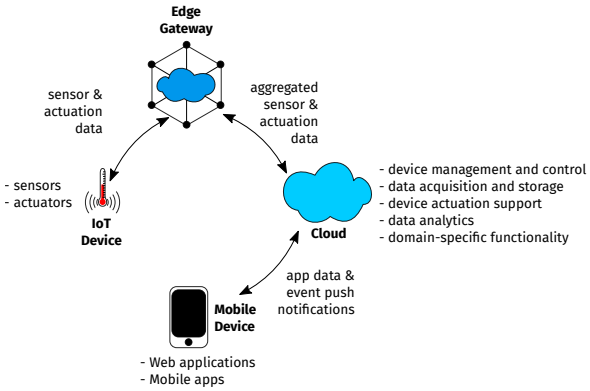

Figure 1. Typical IoT System End-to-End Architecture.

A hallmark of the trend towards the Programmable World is Internet of Things (IoT) development. A typical IoT architecture comprises a number of components, including (i) sensors and actuators that are at the edge of the network; (ii) gateways that connect them to the Internet; (iii) cloud services that offer access to large amounts of storage; (iv) end-user applications that enable access to data, sensors, and actuators; and (v) scalable analytics facilities that are connected to the cloud [14]. Typical IoT system end-to-end architecture is illustrated in Fig. 1. Today, a wide

variety of implementation technologies are used for developing different parts of the end-to-end IoT system (see Fig. 2). This results in diverging development and deployment practices as well as higher integration costs.

In this paper, we argue that a unifying software layer is needed to manage the complexity of IoT development and to liberate the developers from highly fragmented IoT architectures of today. The work presented in this paper is a continuation of a series of vision papers that describe liquid, multi-device software architectures [15] and the Programmable World concept [13]. In the present paper, we push the envelope towards isomorphic IoT systems, following the same line of thought and motivation.

On Isomorphic Software

Isomorphic means "with the same shape". The word isomorphism is derived from the Ancient Greek: $\iota\sigma o\zeta$, or *isos* = "equal", and $\mu o\rho\phi\eta$, or *morphe* = "form" or "shape". Isomorphism is a popular, well-established concept in mathematics. However, in the context of software development the concept has emerged relatively recently. For instance, in the context of web applications, isomorphism refers to the ability to run the same code both on the backend (cloud) and in the frontend (web browser). More broadly, isomorphic software architectures feature software components that do not have to be modified ("change their shape") when running across the different hardware or software components of the system; some examples of isomorphism in the context of software systems are listed in Table 1.

Technology	Description of Isomorphic Features
Java (1995)	The "write once, run everywhere" slogan popularized by the Java platform [1] captures the essence of static isomorphism in software. In Java, the concept meant that it is possible to run the same software on different computer architectures and operating systems using a virtual machine.
Squeak Smalltalk (1996)	Virtual machines for the Squeak Smalltalk system are available for many operating systems and hardware platforms, making it possible to run bit-identical images across all [4].
Unity (2005)	Unity 3D development platform was born within the gaming domain, but it has recently branched out to cinematics, automotive and architecture domains. Applications written for Unity can run across 25 different platforms, including gaming consoles, but also mobile devices, virtual reality headsets and smart TVs.
Lively Kernel (2007)	The Lively Kernel is a web framework, where applications are composed with JavaScript, and the code can be run on either in the client or on the server side [5].
Isomorphic web apps (2013)	The term 'isomorphic web app' was introduced in the context of web applications in mid-2010s, referring to the ability to allocate a part of a web application functionality either on the server or on the client [12]. While the term was new, the same idea has been used in the context of the Web previously, e.g., in the Lively Kernel mentioned above.
Universal Windows Platform (2015)	Within the Microsoft ecosystem, this platform enables developers to write and run the same software on computers and tablets running Windows 10, Xbox One gaming machines and HoloLens devices.
Liquid web apps (2015)	Liquid web applications [11] allow migration of their user interface components on the fly, allowing the users to flexibly use applications on different devices and screens. The main focus in this work is on user experience: how to seamlessly move, clone and adapt user interface components and entire user experiences from one device to another.

Table 1. Examples of Isomorphic Software.

In principle, writing software for isomorphic architectures is fundamentally simpler, since the same code can run everywhere. Because the underlying technologies are handled uniformly, developers do not have to master various different development technologies, and thus complexity is tamed considerably.

Several different levels of isomorphism can be identified. At the first level, isomorphism refers to the consistent use of the same development technologies across the different computational elements in the entire system. In contrast with such *static*, development-level isomorphism, in *dynamic* isomorphism a common runtime engine or virtualization solution is used, so that the same code can run in different computational elements without recompilation. In an even more advanced system, dynamic migration of code from one computational element to another is enabled.

In case of IoT applications, the same (i.e., isomorphic) software can ideally be deployed throughout the end-to-end system to run on edge devices, gateways, mobile clients and cloud services. However, as we are going to discuss, current IoT systems are a far cry from this ambition. Today, IoT application developers must be aware about the deployment context for their code, and they must be familiar with many different

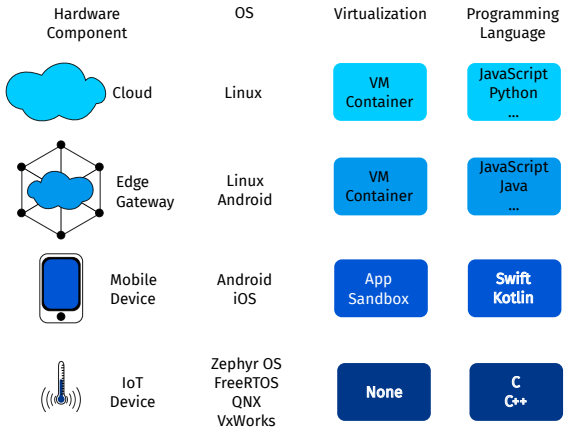

Figure 2. Illustrated Example of Current Platform Diversity in the Context of IoT Systems.

programming languages and virtual runtime environments and communication protocols (Fig. 2). This platform diversity can make it impossible, for example, to redeploy components from the edge to the cloud without a complete rewrite.

While isomorphic architectures will make it easier, faster and potentially cheaper to develop IoT applications and systems, we predict that they will also enable new kinds of dynamic applications which take advantage of the possibility to dynamically redeploy and migrate application components from the edge to the cloud (and vice versa).

Challenges in IoT Development – Diversity of Programming Models

Today, the vast majority of software developers have been trained to do either mobile development or web development [18]. Many of these developers tend to assume that their skills would be directly applicable to IoT development. However, IoT systems have many characteristics that do not apply to mobile or web applications. IoT developers must consider several factors that are unfamiliar to most application developers. Such factors include

(i) multidevice programming,
(ii) heterogeneity and diversity of devices,
(iii) intermittent, potentially unreliable connectivity,
(iv) the distributed, always-on and nature of the overall system, and
(v) the general need to write software in a highly fault-tolerant and defensive manner.

Moreover, a typical IoT application is *continuous and reactive*. On the basis of observed sensor readings, computations get triggered (and retriggered) and eventually result in various actionable events. The systems are essentially *asynchronous, parallel, and distributed*. These qualities alone make IoT applications very different from traditional PC, mobile or web applications, in which software is typically written for a single client that may communicate with a single backend server.

In general, IoT devices are bringing back the need for *embedded software development* skills and education. Software development for IoT devices is very similar to ”classic” embedded systems development, as they both require small-memory and energy-aware software development skills. This is a relevant note especially from an education viewpoint, since in the past 10-15 years a lot of universities – at least in Northern Europe – have scaled back their courses on embedded systems and control theory, focusing on presumably more modern and desirable areas such as Web and mobile software development instead. Recent Developer Economics survey reports strongly confirm the focus on higher-level programming skills [18].

While IoT *device* development is bringing back the need for embedded software, in the other end of the spectrum of IoT end-to-end systems, *cloud* development relies heavily on multiple layers of virtualization. In a modern microservice-based software architecture, the built-in assumption is that all the microservices must be turned into Docker containers. Furthermore, those Docker containers are then assumed to be run in a Kubernetes cluster. This has to be done in spite of the fact that the underlying components are commonly written in Python or JavaScript/Node.js (thus requiring a virtual machine language runtime), and they typically run in a virtual machine rented from third-party cloud service providers such as Amazon or Microsoft.

Dockerization and the use of Kubernetes effectively means that modern software systems commonly use a minimum of *four virtualization layers* even for the simplest of cloud components. The additional virtualization layers often offer little additional value, add overhead to the devel-

opment process, slow down application execution, and make debugging of the system more difficult. Nevertheless, the use of virtualized software environments is rapidly spreading also to IoT edge systems, including gateway development.

Virtualization layers add complexity to just about every step of the development process. Dealing with this complexity necessitates a lot of boilerplate software that presumably helps, but often distracts the developers from focusing on the essentials of the applications. Despite extensive use of virtualization, IoT systems still suffer from a rigid and fragmented architecture in which tasks cannot be reallocated easily from one computational element to another.

Industrial Example

Let us present a brief industrial IoT system example to illustrate the current diversity. In this system, a company has developed an industrial measurement and tracking solution that consists of a large number of devices that have been custom-built for different measurement and tracking tasks. Examples include devices for tracking air quality (temperature, humidity, air pressure and indoor air pollution), tracking movement (based on both inertial measurement units and indoor localization technologies), as well as devices for measuring ambient noise and luminosity (including infrared light level). Devices are connected to the network either via short-area radio solution such as Wi-Fi or low-power wide-area network (LPWAN) solution such as Narrowband-IoT (NB-IoT) or LTE-M. Data uploading and actuation is performed using the MQTT protocol (https://mqtt.org/).

In this case study system, sampling rates and data upload rates are relatively moderate. On average, data uploading is performed only every few minutes. This simplifies the implementation of the cloud backend quite considerably, since there is no requirement for continuous data streaming or extremely low latencies. Fig. 3 provides an overview of our case study system.

The measurement devices are built on top of available off-the-shelf hardware. For Wi-Fi based devices, popular ESP32 (https://www.espressif.com/en/products/socs/esp32) platform was used. For NB-IoT/LTE-M based devices, Nordic Semiconductor's nRF91 (https://www.nordicsemi.com/Products/Low-power-cellular-IoT/nRF9160) was chosen. The development language for both of these device platforms is C, but APIs, libraries and tools vary considerably since a different real-time operating system (RTOS) is used in each device platform.

Since data uploading is performed over Wi-Fi or cellular, this use case does not require any custom-built gateway devices running dedicated protocol translation stacks. However, for device configuration purposes, an Android mobile app – written in Java utilizing the Android libraries – was developed as well.

A lot of focus in the development effort was placed on developing the cloud backend. The logical components of the backend are depicted in Fig. 3. The majority of backend components were implemented using open source components. For security perimeter/reverse proxy implementation, NGINX (https://nginx.org/) was chosen. For data acquisition (collection of sensing data from devices), both Apache Kafka (https://kafka.apache.org/) and RabbitMQ (https://www.rabbitmq.com/) are used. For logging and system monitoring, Grafana (https://grafana.com/), Graphite (https://graphiteapp.org/) and Icinga (https://www.icinga.com/) were picked. Data analytics capabilities were originally implemented using Apache Storm (http://storm.apache.org/), but these capabilities were later replaced with Apache Spark (https://spark.apache.org/). Domain-specific microservices were all implemented in Node.js.

The entire backend is Dockerized; for instance, each of the microservices runs in its own Docker container. Ansible (https://www.ansible.com/) and OpenStack (https://www.openstack.org/) were utilized in the original deployment, but Later on the entire system was migrated to run in a Kubernetes cluster (https://kubernetes.io/).

In addition to the devices and the cloud backend, some additional web and mobile applications were developed for data visualization purposes as well as for system administration and monitoring. In web application development, an earlier version of Angular.js (https://angularjs.org/) was used, whereas mobile apps were written in Java/Android Studio.

As can be determined from the discussion above, the development of the entire end-to-end system required a very broad palette of technolo-

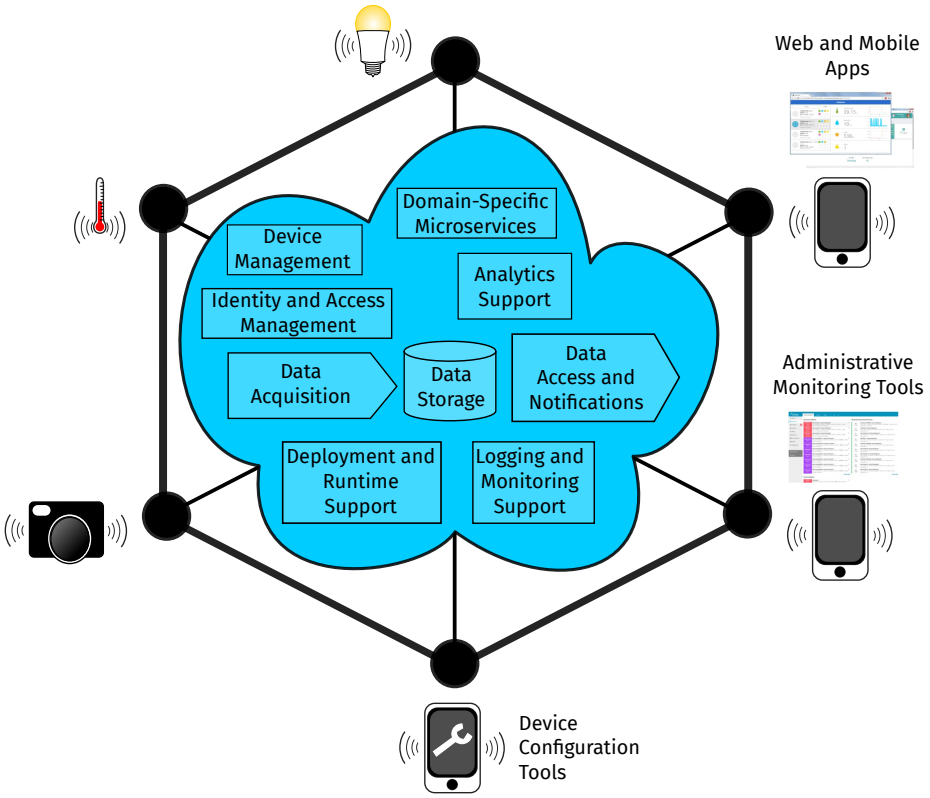

Figure 3. Overview of Our Case Study System, Including Its Key Subsystems and Related Applications.

gies ranging from embedded, mobile and web application technologies to a spectrum of popular cloud backend implementation components. Given the breadth of the technologies, it would be almost impossible for an individual developer or a small startup company to master all the necessary technologies to develop the entire system. Furthermore, because of the selected technologies, each of the components is rather tightly coupled with a specific computational element in the end-to-end system.

Additional Catalysts for Change

Intelligence in the Edge. In ”classic” IoT systems such as our case study system above, the majority of computation and analytics are performed in the cloud in a centralized fashion. However, in recent years there has been a noticeable trend in IoT system development to move intelligence closer to the edge. Historically, the computing capacity, memory and storage of edge devices were limited. Due to increasing computational capabilities of edge devices and requirements for lower latencies, though, intelligence in a modern end-to-end computing system is gradually moving towards the edge, first to gateways and then to devices. This includes both generic software functions, and – more importantly – time critical AI/ML features for processing data available in the edge with minimal latency. The requirement to run advanced AI/ML and analytics algorithms in the edge increases the demand for consistent programming technologies across the end-to-end system.

Increasingly dynamic nature of IoT Systems. In IoT systems that consist of a massive number of devices overall, device topologies can be expected to be highly dynamic and ephemeral. This dynamism calls for technologies that can cope with dynamically changing “swarms” of devices and their dynamically evolving responsibilities. The increasingly dynamic nature is not only related to software features, but also to AI/ML capabilities where reinforcement learning can introduce unexpected situations – something that worked yesterday might not work today, and vice versa. Furthermore, since such features are wrapped in software components, it is often

expected that they can be relocated to the best-suited context for execution.

To simplify development, deployment and long-term use, we expect that future IoT systems will need to support very flexible allocation of responsibilities, so that the roles of devices can evolve over time. This calls for a platform in which different computational entities can run the same code.

Predicting the Rise of Isomorphic IoT Systems

With the ever-increasing complexity, dynamism and sheer amount of software, we are at a dangerous trajectory at the moment. Rigid system architectures, broad spectrum of technologies, abundant use of cargo-cult reuse (picking certain implementation technologies and methods simply because others have done so), inconsiderate use of virtualization, and highly virtualized deployment and package management approaches are leading us to IoT systems that become increasingly difficult to manage. It is time for a change.

Going forward, we need technologies that liberate us from rigid task allocation and support the use of consistent implementation technologies. Dynamic component deployment should be supported, but in a fashion that emerges from application needs – especially in relation to performance and reliability – and not due to constraints imposed by dominant development platforms and tools. Moreover, various features (especially AI/ML capabilities) may require flexible migration of code and models from the cloud to the edge (and vice versa), depending on data availability and required response times.

Our prediction is that these demands will eventually lead us to *isomorphic IoT system architectures*, in the spirit of isomorphic web applications [12]. By isomorphic web applications, it is commonly referred to the ability to use the same development technologies and code between the frontend and backend. Just as what we are currently witnessing in the IoT area, isomorphic Web applications emerged from an initially fragmented technological landscape in which the development technologies for the web browser and the web server were entirely different. Many web developers will surely still remember the era when backend functionality

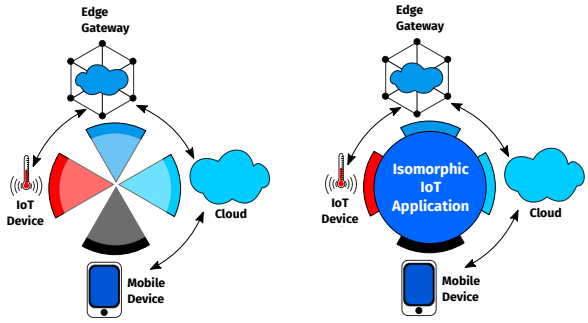

Figure 4. Classic vs Isomorphic IoT Architecture.

was written in PHP or Perl, resulting in a deep divide between programming languages used on the client and on the server side. Once the use of JavaScript spread to the backend [16], it became gradually possible to run the same code on both sides, as long as code would rely on compatible library dependencies and comply with different sandboxing restrictions.

In an isomorphic IoT system, devices, gateways and cloud backend features and frontend applications will be written using the same technologies, and will ideally be able to run the same software components, allowing flexible migration of code between components in the overall system. Instead of having to learn many incompatible software development platforms, in an isomorphic architecture one base technology will suffice and will be able to cover all aspects of end-to-end development; the same tools can then be used to compose the software across all the computing units (Fig. 4).

The two key technical elements that are needed for implementing such systems are (i) *uniform API* for accessing features of different subsystems, and (ii) a *common runtime* that is fast but small enough for embedded devices yet powerful enough to implement lightweight containers in order to deploy applications everywhere. In addition, an orchestrator function (such as those described in [7], [8]) is needed that will guide the deployment and potential migration of the different subsystems.

More broadly, the ”holy grail” in the IoT area would be a common Programmable World API that would cover device discovery, data acquisition, data access, device actuation, device management, code updates, debugging, and other

relevant topics in a universal fashion – thus working universally across devices from different domains, manufacturers, and the necessary security mechanisms. While it is debatable whether there will ever be a single API to cover IoT devices from entirely different domains, it is safe to bet that in five to ten years IoT devices and their APIs will have converged significantly. It is also very likely that the necessary infrastructure will grow around the already existing IP networking and Web infrastructure.

Isomorphic IoT with Web Technologies

In seeking for concrete technology candidates for implementing isomorphism, we have turned to Web standards, since they have played a unifying role in many other contexts. For uniform APIs in the isomorphic IoT system context, the most prominent candidate today is the *Web of Things* (WoT), a set of standards for solving the interoperability issues of different Internet of Things (IoT) platforms and application domains [20]. In essence, WoT makes each 'thing' part of the Web by giving it a URI that can be used for communicating with it. The communication with each thing is supported with a common data model and a uniform API that is recognized by every thing.

For the isomorphic IoT runtime, the Web provides two prominent options: (1) *JavaScript/ECMAScript* [3] and (2) *WebAssembly* [19]. The former is the *de facto* language for web applications both for the web browser and the cloud backend (Node.js); it is currently the most viable option for implementing static isomorphism, i.e., to allow the use of the same programming language throughout the end-to-end system. The latter is a binary instruction format to be executed on a stack-based virtual machine that can leverage contemporary hardware [2], [6]; we see WebAssembly as the best option for providing support for dynamic isomorphism, i.e., the ability to use of common runtime that is powerful but small enough to fit also in low-end IoT devices (Fig. 5). Note that these options are not mutually exclusive, i.e., it would be possible to implement an architecture in which WebAssembly is used as the unifying runtime but in which JavaScript is used as the programming language throughout the end-to-end system.

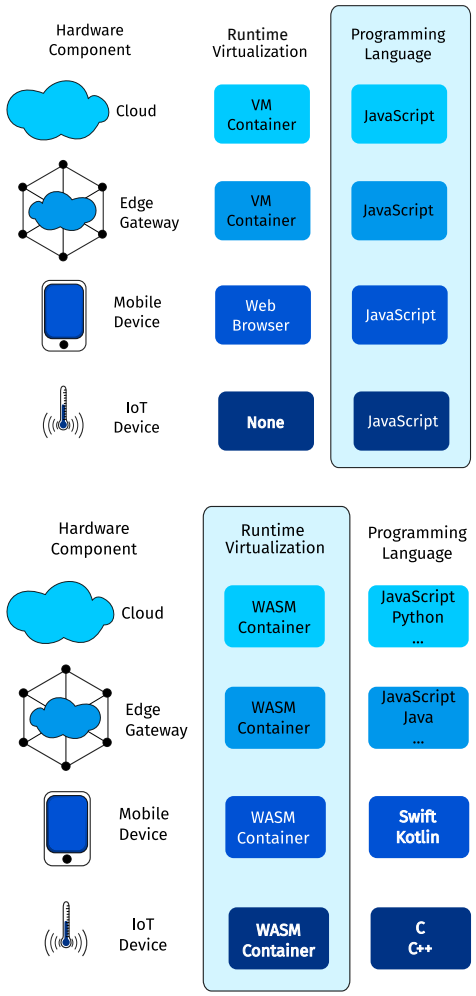

Figure 5. Using Web Technologies to Implement Static and Dynamic Isomorphism – Potential Options. Notice that the options are not mutually exclusive.

Both options have their pros and cons in the context of isomorphic IoT applications. JavaScript offers massive library support (over a million NPM modules), large number of developers familiar with the language, and high-performance virtual machines. However, for isomorphic applications, the dynamic nature of JavaScript may require additional support for packaging the applications into containers. In contrast, WebAssembly programs are organized into modules, which are the unit of deployment, loading and compilation; thus they seem like natural candidates for building lightweight containers [9]. WebAssembly programs can be written in a variety of programming languages and then compiled to WebAssembly for execution. How-

ever, the technology is still relatively immature outside the realm of web browsers. Both technologies can be used to realize a model in which new applications are initialized in the locations where they are needed, as well as realizing the vision of migratory, liquid applications [15].

Ultimately, the definition of a common, isomorphic IoT platform is about *standardization*. While researchers can make relevant contributions and proposals, this area requires collaboration from major industry players to get together and agree on common principles and practices. Alternatively – or in addition – *de facto* standards will surely be established by those companies who manage to create highly successful businesses around their IoT solutions.

Finally, while predicting the rise of isomorphic software, it is important to note that not all software needs to be isomorphic. For instance, low-cost IoT devices such as ambient temperature or air quality sensors are often implemented with ”bare metal” solutions without including any kind of an operating system in the device. Although hardware capabilities are increasing very rapidly (*just 15 years ago, who would have thought microcontrollers to be based on 32-bit architectures or have megabytes of storage memory?*), we do not foresee such devices including support for containers or advanced virtualization capabilities in the next several years. In the same vein, visualization UIs or cloud components that have been developed for monitoring the overall system state usually do not need to be transferable to run in the edge devices. Even though advances in hardware development will probably eventually enable the use of virtualization literally in all types of devices, ultimately these choices will still have to be based on rationally justified use cases rather than blindly trying to make code executable everywhere.

Conclusions

According to a popular saying – often attributed to Alan Kay – in software systems development ”simple things should be simple, and complex things should be possible”. Unfortunately, in modern software development simplicity seems to be a lost virtue [10]. Instead, modern software systems are characterized by plentiful use of virtualization, abundant use of third-party software components from unknown sources, and a cornucopia of overlapping implementation technologies for different parts of the end-to-end system.

When targeting IoT systems, there is currently very little coherence in the development or deployment practices at the level of end-to-end systems. Furthermore, deployment in the large introduces new challenges, especially when one should routinely manage up to millions of devices in a consistent fashion. At the moment, we are still far away from Wasik's prediction, *”In the programmable world, all our objects will act as one”* [17]. We really should not continue programming, installing, and maintaining large-scale IoT systems with the medley of technologies that are in use today.

Our prediction is that IoT development needs isomorphic software architectures, in which subsystems and computational entities can be programmed with a consistent set of technologies, allowing applications and their components to be statically or dynamically allocated, orchestrated and migrated to different entities flexibly. Although fully isomorphic IoT systems are still some years away, their arrival may ultimately dilute or even dissolve the boundaries between the cloud and its edge, allowing computations to be transferred dynamically and performed in those elements that provide the optimal tradeoff between performance, storage, network speed, latency and energy-efficiency.

As most prominent candidates for realizing isomorphism in the context of IoT, we foresee *Web of Things* for APIs, and *JavaScript* or *WebAssembly* for composing flexibly deployable and transferrable application logic.

REFERENCES

1. Ken Arnold and James Gosling. *The Java Programming Language*. 2005.
2. David Bryant. WebAssembly Outside the Browser: A New Foundation for Pervasive Computing. In *Keynote at ICWE'20, June 9-12, 2020, Helsinki, Finland*, 2020.
3. ECMA International. Standard ECMA-262: ECMAScript 2020 Language Specification. June 2020. https://www.ecma-international.org/publications/standards/Ecma-262.htm (visited March 5, 2021).
4. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: the Story of Squeak,

a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, 1997.

5. Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. A World of Active Objects for Work and Play: the First Ten Years of Lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 238–249, 2016.
6. Martin Jacobsson and Jonas Willén. Virtual Machine Execution for Wearables Based on WebAssembly. In *EAI International Conference on Body Area Networks*, pages 381–389. Springer, 2018.
7. Dawid Kurzyniec, Tomasz Wrzosek, Dominik Drzewiecki, and Vaidy Sunderam. Towards Self-Organizing Distributed Computing Frameworks: the H2O Approach. *Parallel Processing Letters*, 13(02):273–290, 2003.
8. Niko Mäkitalo, Timo Aaltonen, Mikko Raatikainen, Aleksandr Ometov, Sergey Andreev, Yevgeni Koucheryavy, and Tommi Mikkonen. Action-Oriented Programming Model: Collective Executions and Interactions in the Fog. *Journal of Systems and Software*, 157:110391, 2019.
9. Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. WebAssembly Modules as Lightweight Containers for Liquid IoT Applications. In *Proceedings of International Conference on Web Engineering 2021*. Springer, 2021.
10. Tiziana Margaria and Mike Hinchey. Simplicity in IT: The Power of Less. *Computer*, 46(11):23–25, 2013.
11. Tommi Mikkonen, Kari Systä, and Cesare Pautasso. Towards Liquid Web Applications. In *International Conference on Web Engineering*, pages 134–143. Springer, 2015.
12. Jason Strimpel and Maxime Najim. *Building Isomorphic JavaScript Apps: From Concept to Implementation to Real-World Solutions*. O'Reilly Media, 2016.
13. Antero Taivalsaari and Tommi Mikkonen. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software*, 34(1):72–80, 2017.
14. Antero Taivalsaari and Tommi Mikkonen. On the Development of IoT Systems. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 13–19. IEEE, 2018.
15. Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. Liquid Software Manifesto: The Era of Multiple Device Ownership and its Implications for Software Architecture. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 338–343. IEEE, 2014.
16. Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Comput.*, 14(6):80–83, 2010.
17. Bill Wasik. In the Programmable World, All Our Objects Will Act as One. *Wired. Available online: http://www.wired. com/2013/05/internet-of-things-2/ (accessed on Oct. 13, 2020)*, 2013.
18. Mark Wilcox, Stijn Schuermans, and Christina Voskoglou. Developer Economics: State of the Developer Nation. Technical report, VisionMobile Ltd, 2016.
19. World Wide Web Consortium. WebAssembly Core Specification, 2019. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf (visited March 5, 2021).
20. World Wide Web Consortium. Web of Things (WoT) Architecture, 2020. https://www.w3.org/TR/wot-architecture/Overview.html (visited March 5, 2021).

Author Bios

Tommi Mikkonen is a Professor of Software Engineering at the University of Helsinki, Finland. He received his PhD from Tampere University of Technology, Finland. His research interests include Web engineering, IoT, and software architectures. Contact him at tommi.mikkonen@helsinki.fi

Cesare Pautasso is a Professor at the Software Institute at USI, Lugano, Switzerland. He received his PhD from ETH Zurich, Switzerland. His research interests include Web engineering, liquid software architectures, and API analytics. Contact him at c.pautasso@ieee.org.

Antero Taivalsaari is a Bell Labs fellow at Nokia Bell Labs. He received his PhD from University of Jyväskylä, Finland. Contact him at antero.taivalsaari@nokia-bell-labs.com.