Runtime Load-Shifting of Distributed Controllers Across Networked Devices

Runtime load-shifting of distributed controllers across networked devices*

Angelo Filaseta[1][0009−0004−6797−6814] and Danilo Pianini[1][0000−0002−8392−5409]

Alma Mater Studiorum—Università di Bologna, 47522 Cesena (FC), Italy
angelo.filaseta@studio.unibo.it, danilo.pianini@unibo.it

Abstract. The ability to monitor and steer the behaviour of complex distributed systems is an increasingly hot research topic, fostered by the growing adoption of hybrid cloud-edge technologies that constitute a computational continuum. One key feature of these systems is the ability to scale in size, embracing a wide number of heterogeneous devices and applications. This complexity, in turn, impacts the monitoring and control systems that need, at the same time, to be able to deal with high complexity and computational load and be available on all kinds of devices. In this paper, we introduce an architecture that allows for shifting the computational load of monitor systems at runtime across different devices in the cloud-edge continuum. We show the feasibility of the proposed approach by providing a reference implementation integrated with an existing simulation platform, leveraging Kotlin multiplatform to allow interoperability among different runtimes.

Keywords: Runtime load-shift · Distributed Monitoring · Distributed control · Interoperability.

1 Introduction

Recent trends in the development of distributed systems are pushing towards constructing a cloud-edge continuum, where services can migrate opportunistically across very diverse devices [19]. Monitoring and controlling the behaviour of such systems is paramount [7], and although initial studies on distributed monitoring exist [3], it is often achieved by aggregating the information provided by the devices in a sub-portion of the system (often, a single monitoring service) [28]. One problem in this context is the computational load of the monitoring service: if the system is large and complex, it may need to perform heavy-duty computations, such as rendering the relations among the monitored system's components. The problem is exacerbated by the fact that the monitoring/control service must be available on any device an administrator may have access to, hereby including low-power and battery-equipped devices such as smartphones, which may not be able to perform the required computations (or may do so an unacceptable cost in battery life). In general, the monitoring/control service

* Artifacts available in https://zenodo.org/record/7817433

must be able to use the available resources efficiently: if the monitoring device is powerful enough, it can host the computation, freeing the shared resource from the duty; otherwise, it should delegate the computation to a more suitable device. Crucially, the monitoring service must be able to do so at runtime, as the availability of devices may change over time, as well as the actual resource availability; for example, a smartphone kept under charge may well take care of the whole computation, but as soon as it gets disconnected from the power grid, battery consumption concerns apply. Similar considerations can be made for other performance metrics, such as networking issues: mobile devices may be located where the network connectivity is poor, and thus switch at runtime into an operation mode that optimises for low data rate.

Contribution In this paper, we propose an *architecture* that allows for shifting the computational load of monitoring and control systems *at runtime*. It requires the capability to identify in advance the components that may be moved across devices, and a shared technology or runtime that can execute on all the devices that might need to host the heavy-duty part of the monitoring: consequently, technologies capable of targeting *multiple runtimes* through *multi-target compilation* are particularly well-suited to implement the proposed system.

The remainder of this manuscript is as follows: Section 2 introduces the problem we are addressing and shows examples from the industry; Section 3 describes the architecture we propose to tackle the problem; Section 4 describes a proof-of-concept implementation of the proposed architecture; Section 5 exercises the proof-of-concept and draws lessons on the architecture; Section 6 concludes the paper and outlines future work.

2 Problem statement

Consider a (possibly large) distributed system composed of multiple devices and processes that needs to be monitored (namely, information on the system's state needs to be collected, aggregated, and displayed on a monitoring device, generating a directional information flow) and controlled (which, in addition to monitoring the system, can act on it, generating a bidirectional information flow). Assume that the system is monitored and controlled by a single service, which can equivalently be a single process hosted on a single device or a distributed system, as far as it exposes a single entry point. Note that this definition is loose enough to include atypical monitoring and control systems such as simulators, which are often used as development and debugging support when the system is being built. Assume the monitoring/control service to have a hefty computational load in some of its parts; although no specific kind of computation is assumed, a typical example can be the rendering of the relations among the monitored system's components, which may involve the computation of and organisation in space of large graphs whose edges are frequently reshaped (e.g., if the system includes mesh-networked parts and/or mobile devices) and whose node set evolves in time (e.g., if the system is open and new nodes join or leave).

Finally, the monitoring service must efficiently use the available resources on any device an administrator may have access to, ranging from well-equipped workstations to low-power devices such as smartphones (or even more resource-constrained devices, including wearables such as smartwatches). Consequently, the monitoring service must be able to *dynamically* shift its computational load across different devices, balancing the load considering the available resources and the current needs of the system, and supporting scenarios such as *moving the computationally expensive part* on or off a handheld device when the device is connected to the power grid or disconnected from it. To achieve the result, the system needs a state transfer protocol to be in place for the reconfiguration to happen at runtime; different algorithms come with their own properties and guarantees, which the system will inherit. In this work, we focus on the software architecture of the overall load-shifting service, leaving the specific state transfer protocol out of the scope of this contribution.

2.1 Analogies with systems in the literature and in the industry

The idea of moving the computational load of a distributed system across nodes is not new. In particular, *load balancing* is a hot theme [2] in cloud systems, where tasks must be allocated to the available resources in a way that optimises for the system's performance [20]. However, in most cases, tasks running in the cloud are not designed to be portable to the network leaves: load balancing in this context happens at the level of the cloud provider, with certain guarantees of homogeneity (often obtained through virtualisation) [18]. An interesting take on the subject has been proposed by the community working on agent-based programming. Mobile agents are indeed designed to be portable across devices, however, the proposed solutions typically rely on a *shared runtime* or *middleware* capable of executing the agents' specifications [9,5,6]. In this work, we try instead to provide an architecture that allows for shifting the computational load across diverse runtimes.

The problem at hand is akin to systems existing in industry, except that, to the best of our knowledge, none supports dynamic relocation of the heavy-duty part across the edge-cloud continuum. The problem of load-shifting is already relevant and visible by non-expert observers in the videogame industry: from the right abstraction level, a videogame is a controller/monitor of a virtual world whose evolution requires complex logic and audio-video elaboration.

As per many load-intensive applications, the traditional way to play high-end videogames is to use (powerful) personal computers or consoles; however, recent trends have seen the rise of cloud gaming services, which allow users to play games with essential devices and low-performance computers [15]. Several major players proposed their platforms (including, but not limited to, Google Stadia, Microsoft xCloud, Amazon Luna, and Nvidia GeForce Now), which are based on the same principle: at the core, the idea is to transmit the game's inputs to a dedicated cloud server rendering the game and sending back the resulting video and audio streams. This type of architecture works similarly to a streaming service, except that data is bidirectional, with inputs flowing from the player and

the provider streaming AV data, which directly results from these inputs. As a result, the server is the only component that actively processes game data.

On the opposite side of the spectrum, in the same industry, we have in-browser games [30]. In these games, typically, client browsers receive the game data from the server and the client machine executes the received game logic and related audio-video rendering. Although this option looks similar to the classic setup, as the client machine is responsible for the heavy-duty computation in both cases, the case of in-browser games adds an important technological constraint: the game code must be *portable*, as the browser can access a set of technologies much stricter than those available on a general-purpose operating system. In fact, most of the games available as native executables for consoles or PC are not available for browsers: the problem here is more related to the technology stack than to the device capabilities.

Framing the problem definition of Section 2 into the videogame industry, we would like our game to be able to be played on a portable console, then be moved to a cloud or edge server when the battery gets low, and finally move into a desktop PC web browser—everything without the need for a restart of the application. Although such flexibility could be overkill for a gaming application (which would also have to find strategies to mitigate the impact of the latency introduced by the load shifting to the user, for instance by presenting a load screen), we believe it is not for the monitoring and correction of the behaviour of a large-scale distributed system—where similar issues apply.

3 Proposed architecture

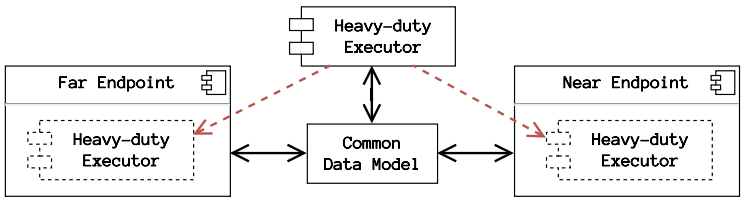

Fig. 1: Abstract architecture of the proposed system. Dashed lines indicate that the actual component (drawn with solid lines) can be in any of the potential states. In short, the proposal is to isolate the data model in order to have a common shared language, and have a mobile component that can be moved across devices (either because it is pre-installed and enabled on demand, or implemented with actual mobile code) capable of performing the heavy-duty computation.

The proposed architecture, summarised in Figure 1 would be composed of four modules:

– **Far Endpoint ($\mathcal{F}$)**—*software* component, usually non-local, that provides means to perform the primary operations required on the target system through a well-defined API;

- **Near Endpoint ($\mathcal{N}$)**—*software* component the user interacts directly with, hence, running on a device the user has direct access to, whose goal is to interact with the Far Endpoint;
- **Common Data Model ($\mathcal{M}$)**—a formal (and serialisable) description of the data exchanged among all the components of the system;
- **Heavy-duty Executor ($\mathcal{H}$)**—software module responsible for performing resource-intensive computational tasks; this is the component that can be executed on either a Far or Near Endpoint instance, and, crucially, it can be moved from and to these endpoints at runtime, according to the model and assumptions presented in Section 2.

At the core of the idea is the isolation of the $\mathcal{H}$ component from the rest of the system and the definition of a common data model $\mathcal{M}$ that allows the $\mathcal{H}$ component to hop from one endpoint to another. Communication that would have happened in the form $\mathcal{N} \rightleftharpoons \mathcal{F}$ is actually translated into $\mathcal{N} \overset{\mathcal{M}}{\rightleftharpoons} \mathcal{H} \overset{\mathcal{M}}{\rightleftharpoons} \mathcal{F}$. Notice that $\mathcal{F}$ and $\mathcal{N}$ can, in principle, be as many as needed, as far as they can communicate and a single $\mathcal{H}$ is operational at a time for each $\mathcal{N}$ instance. Multiple $\mathcal{F}$ instances require more care, as $\mathcal{H}$ would need to be moved $\mathcal{F}$-to-$\mathcal{F}$ (possibly mediated by $\mathcal{N}$ instances): although possible in principle, we do not explicitly cover the case of multiple $\mathcal{F}$ in the architecture.

The architecture does not mandate the protocol used to move the $\mathcal{H}$ component between the $\mathcal{F}$ and $\mathcal{N}$ endpoints, but two abstract strategies are possible: (i) **copy and enable**: both the $\mathcal{F}$ and $\mathcal{N}$ endpoints have a copy of the $\mathcal{H}$ component, and only one of the two is active at a time; and (ii) **mobile code**: the $\mathcal{H}$ component is mobile code that is actually moved along the network. In the copy/enable strategy, the $\mathcal{H}$ component should be entirely separable from its state, constituting (part of) $\mathcal{M}$. Indeed, on a load shift, only the state is sent from the component hosting it previously to the new one. This strategy is the most straightforward to implement and the one that should have the better performance in terms of reactivity, but it requires careful design of $\mathcal{H}$ component, burdens both $\mathcal{F}$ and $\mathcal{N}$ with the duty to host a quiet copy of the software, and does not allow for runtime updates of $\mathcal{H}$. On the other hand, the mobile code strategy is more flexible, allows in principle for the runtime injection of updated versions of $\mathcal{H}$ (although this operation raises the question of how to ensure the integrity of the ongoing computation), but it is more complex to implement and is expected to impose more stress on the networking infrastructure when load-shifting.

3.1 Load-shifting spectrum: an example

In this section, we briefly discuss how the proposed architecture may support the scenario described at the end of Section 2.1. We assume three devices in our system: (i) a battery-powered handheld console; (ii) a remote server located on the cloud or the edge; and (iii) a desktop PC. For the scenario to work as we expect, we need two instances of $\mathcal{N}$ located on the devices the user has direct access to (the handheld console and the desktop PC), and an instance of $\mathcal{F}$

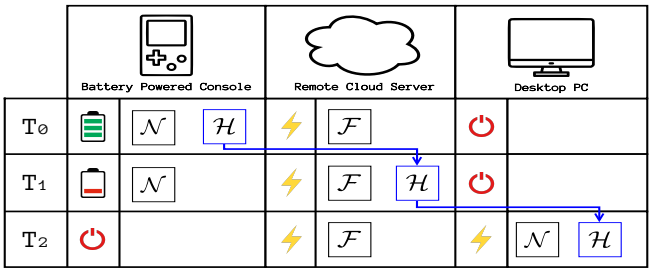

Fig. 2: Configuration and evolution of the example: the heavy-duty component $\mathcal{H}$, initially hosted on a handheld device, gets opportunistically shifted to the cloud, and then to a desktop PC-hosted web browser. Rows represent subsequent time steps, columns represent devices. Every column is split into two parts, on the left is the device status (off, battery level if turned on but off the power grid, on if on the power grid); on the right are the software components being executed.

located on the remote server. The situation and the evolution of the system are summarised in Figure 2.

At the beginning of the scenario, the user is playing the game on the handheld console with a full battery, and the $\mathcal{H}$ component is located in the $\mathcal{N}$ component running on the handheld console. When the battery lowers, the load gets shifted to the remote server, and $\mathcal{H}$ is moved to the $\mathcal{F}$ component running remotely. Finally, if the user turns on the desktop PC and prefers to play the game there for higher responsiveness, they can start the local $\mathcal{N}$ component, then require a load-shift moving the $\mathcal{H}$ component locally.

3.2 Limitations and technological constraints

The first constraint we need to consider concerns $\mathcal{M}$: we notice that it must be serialisable and deserialisable in a way independent of the underlying runtime. Although cross-platform serialisation formats capable of representing most data structures across languages exist both in textual/human-readable form (e.g., JSON [17], YAML[1], etc.), and binary/efficiency-driven [25] form (e.g., Protocol Buffers [11], BSON[2], etc.) the requirement of being able to interoperate across possibly diverse runtimes imposes a clear and unambiguous specification. In turn, this implies that although the proposed architecture can be adopted for brand-new systems (in which the effort for a new design is due anyway), a significant effort would be required to retrofit many existing systems to the proposed architecture. Ultimately, for this architecture to be applicable, a pre-identification of all mobile parts is essential to maintain a consistent set of available resources and optimise the allocation of computational load. The need for such pre-identification stems from the fact that, in the general case, mobile parts

[1] `http://yaml.org/spec/1.2/spec.pdf`
[2] `https://bsonspec.org/spec.html`

may join or leave the system at any time, resulting in a constantly changing network topology. Consequently, a mechanism must be implemented to update the available resources in response to such changes.

However, the most significant constraint imposed by the proposed architecture is technological in nature and concerns the $\mathcal{H}$ component. By its very nature, $\mathcal{H}$ must be able to run as a module of $\mathcal{N}$ and $\mathcal{F}$ for which, however, we did not impose any constraint on the runtime or technology. This situation leads us to three potential cases.

Shared technology/runtime By sheer luck or careful design, $\mathcal{N}$ and $\mathcal{F}$ share the same technology stack and runtime, and thus $\mathcal{H}$ can be realised once with a compatible technology and consumed by both $\mathcal{N}$ and $\mathcal{F}$. (note: they do not need to share the same programming language, as far as the executable form is portable across devices). This is the least interesting case, as in most cases modern runtimes provide means to move load between different network nodes. In these cases, the proposed architecture may not be needed (unless trying to anticipate changes).

Porting For a more interesting case, we assume that $\mathcal{N}$ and $\mathcal{F}$ are based on entirely different and incompatible stacks and runtimes. For instance, $\mathcal{F}$ may be a Java/JVM or a native application running on a server, while $\mathcal{N}$ is a web application intended to run in-browser. One way to tackle this problem is to port $\mathcal{H}$ to both runtimes. This solution is adopted in some cases, with entire applications rewritten from their original runtime in JavaScript or TypeScript to target the browser[3]. Having multiple copies (one per runtime) of the same core application, however, is hardly ideal. Maintaining the code in sync is a tedious, expensive, and error-prone task, and while maintaining a porting may be worth it for a single application whose development is completed or slow-paced, the approach cannot scale to applications such as the primary target of this study: monitors for distributed systems, that are modern pieces of software running on the bleeding edge of technology and have a very fast-paced development.

Multi-target technology For the same case of the previous paragraph, a second option exists: selecting a technology for $\mathcal{H}$ capable of targeting both the runtimes of $\mathcal{N}$ and $\mathcal{F}$. This way, a single codebase exists for $\mathcal{H}$, with a multiple-target build process that produces two separate executables. Despite the simplicity of the approach on paper, it does present its fair share of challenges. The first one is, trivially, that the technology must be able to target both runtimes. This alone restricts the pool of suitable technologies, and the more the possible runtimes of $\mathcal{N}$ and $\mathcal{F}$, the more difficult it is to find a technology that can target all of them. A second relevant concern regards libraries and ancillary components, as they must be compatible with both runtimes. The second concern is often

[3] One notable example is the porting of the ioquake3 engine (`https://ioquake3.org/`) in JavaScript (`http://www.quakejs.com/`) as a proof-of-concept of the feasibility of running complex applications in the browser.

overlooked, but it is a significant one: if two different libraries are required for the same task to be achieved on two different runtimes, the host language and tooling must provide means to abstract over the differences and to select the specific implementation at runtime.

4 Proof of concept

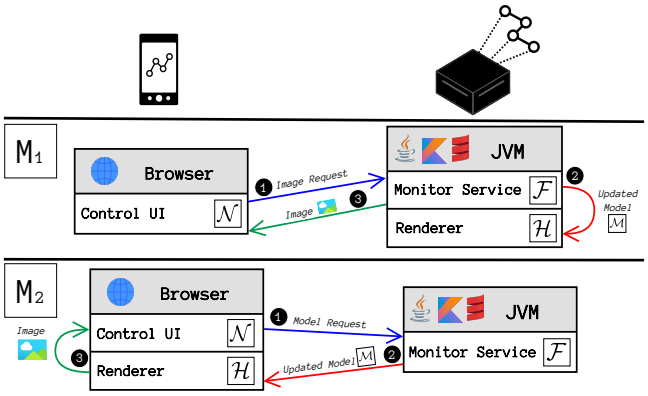

Fig. 3: Idea of the proof of concept. The $\mathcal{H}$ component can be moved at runtime to be hosted on the same host and platform of the $\mathcal{F}$ (in M_1) or to the same host and platform of the $\mathcal{N}$ (in M_2).

To exercise the proposed architecture and demonstrate its feasibility, we show a proof-of-concept implementation in the context of distributed monitoring. We want to build a monitor system for a distributed system that is composed of a set of nodes whose geographical distribution and logical relationships can change with time, thus whose rendering and control require significant computational effort. To stress the issues induced by incompatible runtimes, we choose a monitor service $\mathcal{F}$ implemented as a JVM-running application, and we want the monitoring and control access point $\mathcal{N}$ to be instead hosted in a web browser. Thus, our renderer component $\mathcal{H}$ must be able to run on both the browser and the JVM. In the former case, we want $\mathcal{N}$ to receive an instance of $\mathcal{M}$ and leverage $\mathcal{H}$ to render in-browser. In the latter case, instead, we want $\mathcal{N}$ to receive a pre-rendered image of the system built by $\mathcal{F}$ by running $\mathcal{H}$ on the JVM and displaying it in the browser. In this case $\mathcal{M}$ is already located on $\mathcal{F}$ and does not need to be moved. Figure 3 summarises the architecture of the proof-of-concept implementation.

4.1 Technology selection

The presence of incompatible runtimes (the browser and the JVM) makes the trivial solution introduced in Section 3.2 infeasible. At the same time, given the problem at hand, we deemed the second solution (porting $\mathcal{H}$ to both runtimes)

too expensive in terms of development resources and error-prone, as the same process would have needed to be written twice. Consequently, we were left with the third solution (multi-target technology).

An inspection of the mainstream technologies capable of targeting both the JVM and the browser with the same codebase restricted the pool of candidates to two languages and related frameworks: Scala [22] (with Scala.js [13,14]) and Kotlin-Multiplatform [16]. Both frameworks are currently maintained and have been used to develop interoperable tools that are found in the literature; two examples of interest to the community of distributed systems are, for instance, 2p-kt [10], a Prolog engine inspired by tuProlog [12] and written in Kotlin-multiplatform, and Scafi [8], a Scala implementation of the aggregate computing semantics [4] that also features a web-based playground [1]. An analysis of the two frameworks showed that they are similar from the point of view of the available features and documentation, and that the choice between them is mostly a matter of personal preference. In our proof-of-concept, we chose to use Kotlin-Multiplatform, mostly driven by the existence of a multiplatform library for serialisation developed and maintained by the Kotlin team (*kotlinx.serialization*[4]). We expected the serialisation to be critical in the development of the proof-of-concept, as the initial analysis highlighted the serialisation as a relevant issue in the implementation of the $\mathcal{M}$ component. In any case, we believe that the same proof-of-concept could be implemented in Scala as well, relying on third-party serialisation libraries.

Kotlin-multiplatform Kotlin features a mechanism for sharing pure Kotlin code across multiple platforms, enabling the development of platform-agnostic software modules, which are then compiled for a large variety of targets (including the JVM, Javascript, Android, and native binaries for several architectures and operating systems). The application or library code is split into multiple source sets, the root of which is the *common* one, consisting of core libraries and essential tools, enabling code to be platform-independent and functional on all systems. The other source sets contain platform-specific code variants targeting specific platforms. These variants (Kotlin/JVM, Kotlin/JS, and Kotlin/Native), offer platform-specific language extensions, libraries, and tools.

Of course, all dependencies (libraries) used in the common code must be available for all the target platforms. However, some functions may not be available for all targets, or may require a platform-specific implementation (for instance, a graphical component may require to be implemented using different toolkits). In these cases, Kotlin exposes a specific mechanism that allows declaring that some types or functions will be implemented in a platform-specific way: the `expect` keyword in the platform-agnostic code will mark the target as something not implemented yet, but that will be once a platform is selected. In the platform-specific code, the corresponding feature will be prefixed by the `actual` keyword (and the compiler will check that every platform-specific implementation provides all the `expect`ed types and functions).

[4] `https://github.com/Kotlin/kotlinx.serialization`

4.2 System to be controlled: $\mathcal{F}$

Building an entire cloud-edge continuum ecosystem was well beyond the scope of this work, and we thus opted for controlling a system that could, in principle, abstract away the underlying system and expose the entities typical of the problem we wanted to tackle: a simulator supporting the modelling of a situated distributed system deployed on the cloud-edge continuum, and running within an instance of the Java Virtual Machine (to comply with the requirements of Section 4). Ideally, the target simulator must have a clean separation between the renderer and the control components, so that the former can be easily replaced with a different implementation. Additionally, to ease the development process, we wanted the simulator to be open source and covered by a permissive license.

A natural choice was the Alchemist Simulator [24], which is well-known in the DAIS community [23] and has been used in the past multiple times to simulate systems akin to the ones we want to control. Alchemist comes with its own Java-based rendering engine, but it also provides a clean separation between the rendering and the control components, as witnessed by the existence of two separate modules implementing two different graphical interfaces (one based on Java Swing [21] and one based on JavaFX [26]).

In our proof-of-concept, the Alchemist simulator has been used as a Java library, we created an additional module using Kotlin that translated the entities exposed by the simulator in a multiplatform-friendly format. Such a component, written in Kotlin-JVM, exposes the core simulator controls as a REST API to allow for external control. The API also defines the protocol for load-shifting: the $\mathcal{N}$ component is responsible to select whether it wants to run or offload $\mathcal{H}$, and selects the appropriate route on $\mathcal{F}$.

4.3 Common Data Model: $\mathcal{M}$

The choice of the system to be controlled mandated the construction of the common data model. We selected a subset of the Alchemist model that we deemed relevant for the purpose of the proof-of-concept, and implemented it in pure Kotlin, in a format friendly to the serialisation library. In particular, we had to create pure-Kotlin surrogate classes for the entities exposed by the simulator that we needed to serialise, such as nodes and their position in the environment. We notice that it is vital for the data model to be as minimal as possible: it represents the abstractions that are allowed to circulate between the components, the more they are, the more complex the overall API and the more demanding the overall system becomes.

4.4 Monitor/Controller: $\mathcal{N}$

Our monitor/controller, adhering to the requirements of Section 4, is a web-based application with a simple UI displaying a rendering of the controlled the system. The canvas is populated by the $\mathcal{H}$ component, which, depending on the position

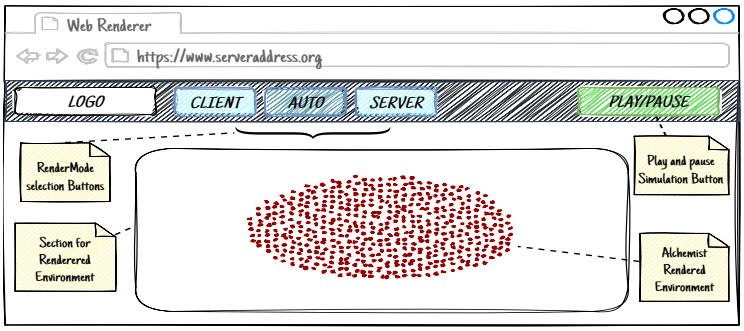

Fig. 4: Sketch of the monitor/controller UI for the proof-of-concept.

of a switch, must migrate from the browser to the JVM-hosted $\mathcal{F}$ component. To exemplify the work modes that we expect would be implemented on a real system, we design the interface to support three modes:

1. $\mathcal{H}$ **forcibly on** $\mathcal{N}$: $\mathcal{H}$ remains on $\mathcal{N}$, or migrates to $\mathcal{N}$ if it was on $\mathcal{F}$;
2. $\mathcal{H}$ **forcibly on** $\mathcal{F}$: $\mathcal{H}$ remains on $\mathcal{F}$, or migrates to $\mathcal{F}$ if it was on $\mathcal{N}$;
3. **Automatic**: depending on the system status, $\mathcal{H}$ migrates dynamically on $\mathcal{N}$ or $\mathcal{F}$. In the current proof-of-concept, the implementation is a very simple policy that migrates $\mathcal{H}$ depending on the available CPU cores. Real systems could adopt much more refined policies.

Provided the simplicity of the UI at hand, we decided to implement the monitor/controller from scratch based on the sketch depicted in Figure 4.

In our proof-of-concept, the web application was developed using Kotlin/JS, and more specifically the React.js [29] framework port. The library provides a way to create reusable and self-contained components, encapsulating both the visual appearance and internal logic of a specific part of the application. To provide a responsive user experience, the application leverages components available as Javascript libraries on public repositories, such as React-Bootstrap [27]. As an implementation note, we add that we had to create custom adapter components to allow the typed use of the aforementioned libraries, since JavaScript is dynamically typed and Kotlin is statically typed.

4.5 Renderer: $\mathcal{H}$

In this section and in the remainder of the manuscript, we abuse the term "renderer" to refer to the component responsible for both rendering the system to be controlled and for interpreting and sending the command. The reason is that the most computationally-intensive operation is the rendering itself, and, in the spirit of load-shifting the most computationally-intensive operation, we tend to identify the heavy-load component with the most demanding operation it supports.

This component, which must be able to run both in the browser and the JVM, must be written in Kotlin multiplatform and use solely libraries available

for both platforms. Although the ecosystem of multiplatform libraries is still in its infancy, we found a library suitable for rendering the system to be controlled (KorLibs/KorIM[5]). In this proof of concept, $\mathcal{H}$ is a renderer producing a graphical representation of a $\mathcal{M}$ representing the simulation environment.

4.6 Final design

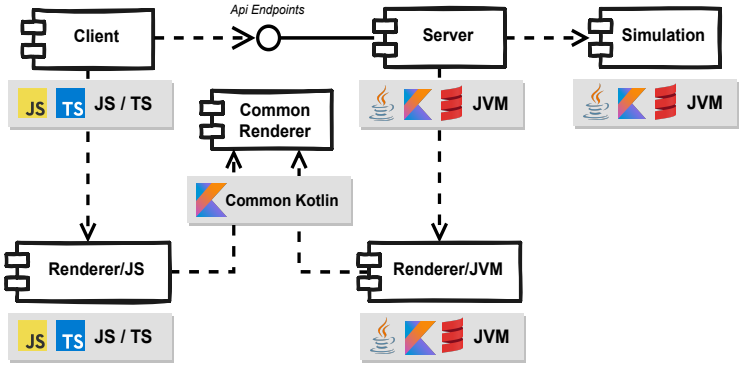

Fig. 5: UML component diagram of the implemented proof-of-concept.

The final incarnation of the proposed architecture is depicted in the UML Component Diagram of Figure 5. The Server component represents $\mathcal{F}$ in the architecture, serving as a Web Server that provides API Endpoints. These endpoints are capable of executing commands on the Simulation components and retrieving data, which are then converted into structures compatible with the $\mathcal{M}$. The Client component represents $\mathcal{N}$, and is intended to communicate with the Server component via the previously mentioned API Endpoints. Both the $\mathcal{F}$ and $\mathcal{N}$ components rely on the platform-specific version of the Renderer component, which is the implementation of $\mathcal{H}$ in the architecture. Both $\mathcal{F}$ and $\mathcal{N}$ also need a platform-specific version of $\mathcal{M}$ to make the execution of $\mathcal{H}$ possible in every scenario. As mentioned, the implementation of $\mathcal{M}$ in pure Kotlin allows the serialization and deserialization operations to assure consistency when moving data between nodes. The state transfer protocol of the proof of concept is as follows: $\mathcal{F}$ always exposes *two* API endpoints, one for obtaining a representation of the model as an image, and the other to obtain the model as a serialised Kotlin object; when $\mathcal{N}$ decides to shift the load, it changes the API endpoint to which commands are sent. The proof-of-concept has been integrated within the main Alchemist repository[6], and is available within the official distribution.

[5] `https://docs.korge.org/korim/`
[6] `https://github.com/AlchemistSimulator/Alchemist`

Fig. 6: Snapshots of the monitor/controller UI for the proof-of-concept, taken subsequently during the same experiment. Initially (left) the render (i.e., $\mathcal{H}$) is running within the client's browser (i.e., $\mathcal{N}$), then (right), at runtime, it is migrated to the server (i.e., $\mathcal{F}$), with no interruption, and no perceivable difference in the rendered image. In both snapshots, the warmer the color, the more the corresponding node had resource availability for itself.

5 Evaluation

In this section, we perform an evaluation of the proof-of-concept, discussing the viability of the proposed architecture to support interoperability and load-shifting. We initially perform a qualitative assessment, verifying that the proof-of-concept is able to render the system to be controlled and to shift the load between the browser and the JVM. We then perform a performance evaluation of the proof-of-concept, with the goal of comparing the operating conditions of the system when $\mathcal{H}$ is running on $\mathcal{N}$ and $\mathcal{F}$, and investigate how the system scales with larger and larger systems to monitor.

5.1 Test environment and qualitative assessment

We perform our evaluation by observing the behaviour of the proof-of-concept in a simulation of an existing reference system. We select an example from the Alchemist tutorial [23] in which a network of nodes coordinates for the use of a shared resource set. Resource usage tokens are generated and exchanged among neighbouring nodes ensuring that every resource is used by at most one node at a time. In the monitoring system, we investigate how many times every node had access to the resource, as the algorithm is not designed to guarantee fairness. For the sake of reproducibility, inspection, and to enable further research on the prototype, we provide an open-source repository with code, instructions, and support scripts for executing the experiment[7].

Figure 6 depicts the current aspect of the monitor and the rendered monitored system, and shows that $\mathcal{H}$ can be moved dynamically back and forth from $\mathcal{N}$ to

[7] `https://github.com/AngeloFilaseta/DAIS-2023-alchemist-web-renderer`

$\mathcal{F}$ without any interruption in the rendering process, thus complying with the requirements the architecture is designed to satisfy.

5.2 Performance evaluation

Free variables We consider two free variables: (i) the number of nodes participating in the system, N ($N \in \mathbb{N}^+$), a proxy for the size of the monitored system; and (ii) the device hosting $\mathcal{H}$, either the same hosting $\mathcal{F}$ or hosting $\mathcal{N}$.

N has direct impact on the model size, and thus on the payload size when $\mathcal{H}$ is hosted on $\mathcal{N}$ (otherwise, the actual network payload is a rendered image, whose size is approximately constant). In our experiment, the serialized size of the model was at its minimum 46827 bytes ($N = 1600$) and 46827 bytes at its maximum ($N = 14400$).

Metrics We measure four metrics to evaluate the performance of the system: (i) **rendering time**: the time required by $\mathcal{H}$ to complete its execution, this metric is meant to compare the raw performance of $\mathcal{H}$ across the available platforms and loads, we expect the execution on the JVM (i.e., $\mathcal{H}$ on $\mathcal{F}$) to be faster than on the browser (i.e., $\mathcal{H}$ on $\mathcal{N}$).); (ii) **serialisation time**: the time required by $\mathcal{F}$ to serialise $\mathcal{M}$ (if $\mathcal{H}$ is hosted on $\mathcal{N}$) or the rendered image (if $\mathcal{H}$ is being hosted on $\mathcal{F}$); this is a proxy metric for the load on the device hosting $\mathcal{F}$; (iii) **deserialisation time**: the time required by $\mathcal{N}$ to deserialise $\mathcal{M}$ (if $\mathcal{H}$ is hosted on $\mathcal{N}$) or the rendered image (if $\mathcal{H}$ is being hosted on $\mathcal{F}$), this is a proxy metric for the load on the device hosting $\mathcal{N}$; (iv) **total time**: the time required to complete an entire iteration, which includes all the previous metrics plus the network delay. We measure each metric five times, and we present the average result.

Results

We execute the experiment using two isolated containers on the same machine, an Intel® Core™ i5-2520M CPU and 8GiB RAM. The results are presented and summarised in Figure 7. As expected, allocating the load of $\mathcal{H}$ to $\mathcal{F}$ results in more consistent system performance with the growth of the monitored system. This is mainly driven by the higher efficiency of the JVM compared to the browser: besides being consistently faster in rendering the system (i.e., $\mathcal{H}$), it is also interestingly quicker in serialising the model as an image than it is in serialising it as plain JSON. We notice, however, that this last consideration is strictly dependent on the specific serialisation formats and libraries used, and may change with different implementations. We observe that hosting the $\mathcal{H}$ on the same node of $\mathcal{N}$ is pretty efficient for small systems, but it scales worse than hosting it on $\mathcal{F}$ with the monitored system size. This behaviour is evident both in the graph showing the deserialisation time and in the chart showing the overall system time.

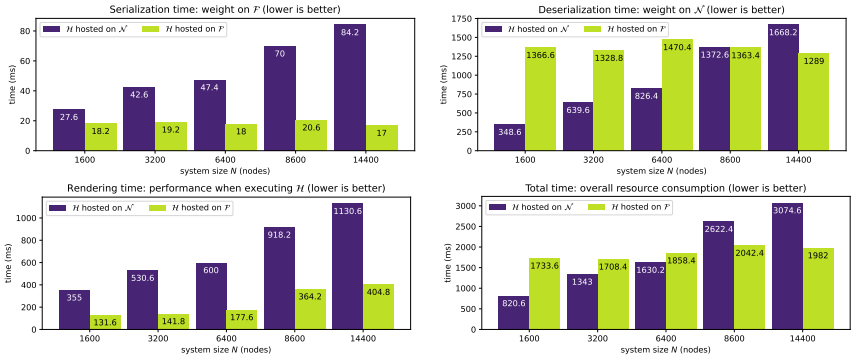

Fig. 7: Performance evaluation of the proof-of-concept. We measured proxy metrics for the load on $\mathcal{F}$ (top left), the load on $\mathcal{N}$ (top right), the cost of running $\mathcal{H}$ in either runtime (bottom left), and the overall cost of the system (bottom right).

6 Conclusion and future work

This paper introduces a novel architecture for developing monitoring and control systems for distributed systems deployed on a heterogeneous infrastructure in which the heavy-load part of the process can be moved across the devices. The architecture has been exercised by developing a proof-of-concept implementing the proposed architecture as an in-browser monitoring and control system for a simulator running on the JVM. The proof-of-concept shows that the proposed architecture is viable and can dynamically shift the load across different target runtimes, as far as the load that needs shifting can be executed on both platforms (or, suboptimally, get rewritten in multiple languages). The performance evaluation suggests that being able to shift the load dynamically across devices can be beneficial on heterogeneous systems.

Of course, although the proof-of-concept has been integrated with the Alchemist simulator, it is still very early in its development, which we plan to continue in the future. Additionally, we plan to study how the architecture can be adapted in case of multiple load sources, possibly impacting different metrics and thus requiring a dynamic assessment of the best location for the specific job. Finally, we believe that a lot of interesting research can be done in the area of automation of the load-shifting process: the current architecture and the prototype are designed to be used by a human operator, and show a single very simple strategy for the automated load-shift. However, much more complex mechanisms can be devised, possibly learning-enabled; we notice, in fact, that having multiple load sources scattered across a networked system at runtime may lead to pretty complex scenarios which could benefit from sophisticated automated approach.

Acknowledgements The authors thank Gianluca Aguzzi for the fruitful discussions on languages, frameworks, and tools for multi-platform programming.

References

1. Aguzzi, G., Casadei, R., Maltoni, N., Pianini, D., Viroli, M.: Scafi-web: A web-based application for field-based coordination programming. In: COORDINATION 2021, Lecture Notes in Computer Science, vol. 12717, pp. 285–299. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_18
2. Aslam, S., Shah, M.A.: Load balancing algorithms in cloud computing: A survey of modern techniques. In: 2015 National Software Engineering Conference (NSEC). IEEE (Dec 2015). https://doi.org/10.1109/nsec.2015.7396341
3. Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M.: Adaptive distributed monitors of spatial properties for cyber-physical systems. J. Syst. Softw. **175**, 110908 (2021). https://doi.org/10.1016/j.jss.2021.110908
4. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic **20**(1), 1–55 (jan 2019). https://doi.org/10.1145/3285956
5. Bak, S., Menon, H., White, S., Diener, M., Kalé, L.V.: Multi-level load balancing with an integrated runtime approach. In: 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018. pp. 31–40. IEEE Computer Society (2018). https://doi.org/10.1109/CCGRID.2018.00018
6. Bellavista, P., Corradi, A., Stefanelli, C.: Mobile agent middleware for mobile computing. Computer **34**(3), 73–81 (2001). https://doi.org/10.1109/2.910896
7. Bennaceur, A., Ghezzi, C., Tei, K., Kehrer, T., Weyns, D., Calinescu, R., Dustdar, S., Hu, Z., Honiden, S., Ishikawa, F., Jin, Z., Kramer, J., Litoiu, M., Loreti, M., Moreno, G.A., Müller, H.A., Nenzi, L., Nuseibeh, B., Pasquale, L., Reisig, W., Schmidt, H., Tsigkanos, C., Zhao, H.: Modelling and analysing resilient cyber-physical systems. In: SEAMS@ICSE 2019. pp. 70–76. ACM (2019). https://doi.org/10.1109/SEAMS.2019.00018
8. Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: ScaFi: A scala DSL and toolkit for aggregate programming. SoftwareX **20**, 101248 (Dec 2022). https://doi.org/10.1016/j.softx.2022.101248
9. Chen, B., Cheng, H.H.: A runtime support environment for mobile agents. In: Volume 4a: ASME/IEEE Conference on Mechatronic and Embedded Systems and Applications. ASME (2005). https://doi.org/10.1115/detc2005-85389
10. Ciatto, G., Calegari, R., Omicini, A.: 2p-kt: A logic-based ecosystem for symbolic AI. SoftwareX **16**, 100817 (Dec 2021). https://doi.org/10.1016/j.softx.2021.100817
11. Currier, C.: Protocol buffers. In: Mobile Forensics – The File Format Handbook, pp. 223–260. Springer International Publishing (2022). https://doi.org/10.1007/978-3-030-98467-0_9
12. Denti, E., Omicini, A., Ricci, A.: tu prolog: A light-weight prolog for internet applications and infrastructures. In: Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Proceedings. vol. 1990, pp. 184–198. Springer (2001). https://doi.org/10.1007/3-540-45241-9_13
13. Doeraene, S.: Cross-platform language design in scala.js (keynote). In: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala. p. 1. Association for Computing Machinery (2018). https://doi.org/10.1145/3241653.3266230
14. Doeraene, S., Schlatter, T., Stucki, N.: Semantics-driven interoperability between scala.js and javascript. In: Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala. p. 8594. Association for Computing Machinery (2016). https://doi.org/10.1145/2998392.2998404

15. Domenico, A.D., Perna, G., Trevisan, M., Vassio, L., Giordano, D.: A network analysis on cloud gaming: Stadia, GeForce now and PSNow. Network **1**(3), 247–260 (Oct 2021). https://doi.org/10.3390/network1030015
16. Hagos, T.: Introduction to kotlin. In: Beginning Kotlin, pp. 1–20. Apress (Nov 2022). https://doi.org/10.1007/978-1-4842-8698-2_1
17. Ihrig, C.J.: JavaScript object notation. In: Pro Node.js for Developers, pp. 263–270. Apress (2013). https://doi.org/10.1007/978-1-4302-5861-2_17
18. Kanbar, A.B., Faraj, K.: Region aware dynamic task scheduling and resource virtualization for load balancing in iot-fog multi-cloud environment. Future Gener. Comput. Syst. **137**, 70–86 (2022). https://doi.org/10.1016/j.future.2022.06.005
19. Milojicic, D.S.: The edge-to-cloud continuum. Computer **53**(11), 16–25 (2020). https://doi.org/10.1109/MC.2020.3007297
20. Mishra, S.K., Sahoo, B., Parida, P.P.: Load balancing in cloud computing: A big picture. J. King Saud Univ. Comput. Inf. Sci. **32**(2), 149–158 (2020). https://doi.org/10.1016/j.jksuci.2018.01.003
21. Newmarch, J.: Testing java swing-based applications. In: TOOLS 1999: 31st International Conference on Technology of Object-Oriented Languages and Systems, 22-25 September 1999, Nanjing, China. pp. 156–165. IEEE Computer Society (1999). https://doi.org/10.1109/TOOLS.1999.796479
22. Odersky, M.: Essentials of scala. In: Langages et Modèles à Objets, LMO 2009, Nancy, France, 25-27 mars 2009. vol. L-3, p. 2. Cépaduès-Éditions (2009)
23. Pianini, D.: Simulation of large scale computational ecosystems with alchemist: A tutorial. In: DAIS 2021, Lecture Notes in Computer Science, vol. 12718, pp. 145–161. Springer (2021). https://doi.org/10.1007/978-3-030-78198-9_10, https://doi.org/10.1007/978-3-030-78198-9_10
24. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27
25. Popic, S., Pezer, D., Mrazovac, B., Teslic, N.: Performance evaluation of using protocol buffers in the internet of things communication. In: 2016 International Conference on Smart Systems and Technologies (SST). IEEE (Oct 2016). https://doi.org/10.1109/sst.2016.7765670
26. Robillard, M.P., Kutschera, K.: Lessons learned while migrating from swing to javafx. IEEE Softw. **37**(3), 78–85 (2020). https://doi.org/10.1109/MS.2019.2919840
27. Subramanian, V.: React-bootstrap. In: Pro MERN Stack, pp. 315–376. Apress (2019). https://doi.org/10.1007/978-1-4842-4391-6_11
28. Taherizadeh, S., Jones, A., Taylor, I.J., Zhao, Z., Stankovski, V.: Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. J. Syst. Softw. **136**, 19–38 (2018). https://doi.org/10.1016/j.jss.2017.10.033
29. Thakkar, M.: Introducing react.js. In: Building React Apps with Server-Side Rendering, pp. 41–91. Apress (2020). https://doi.org/10.1007/978-1-4842-5869-9_2
30. Vanhatupa, J.M.: Browser games for online communities. International Journal of Wireless & Mobile Networks **2**(3), 39–47 (Aug 2010). https://doi.org/10.5121/ijwmn.2010.2303