Computing in the Continuum

EDOARDO TINTO, University of Padova, Italy
TULLIO VARDANEGA, University of Padova, Italy

Compute devices may be found scattered around the physical world nowadays. In the periphery of the network, the so-called Edge, this phenomenon is particularly evident. Here, the decreasing cost of hardware components and increasing bandwidth availability encourage the proliferation of computing units. The rise of the Internet-of-Things exemplifies this trend. This context, characterized by the coexistence of large data centers, providing abundant virtualized resources, and constrained cyber-physical systems with comparatively scarce hardware provisions, creates interesting opportunities for applications capable of reaping the benefits of both.

- Computing locally to the data source dispenses with the need to move data to the center of the Cloud, gaining in latency, privacy, and security.
- Moving computation closer to the Edge also brings benefits due to access to local facilities, such as physical actuators, and to user proximity.

However, heterogeneity and fragmentation hinder the realization of this vision. The former affects both the hardware and software stacks, worsening the closer one gets to the Edge and producing silos of un-communicating technologies, while the latter is the effect of a layered view of Cloud and Edge, promoting a fragmented view of the computing resources. To overcome those challenges, a new vision should be developed, considering both Cloud and Edge as part of a seamless infrastructure where computations can transit while pursuing efficiency (and user-defined) metrics. Crucial to the success of said infrastructure is a common runtime, catering for the execution of application components essentially anywhere across the networked compute devices. This model we identify as the Compute Continuum and, in this document, we present our vision of it, alongside three application scenarios that might take advantage of this novel paradigm.

1 THE INFRASTRUCTURE

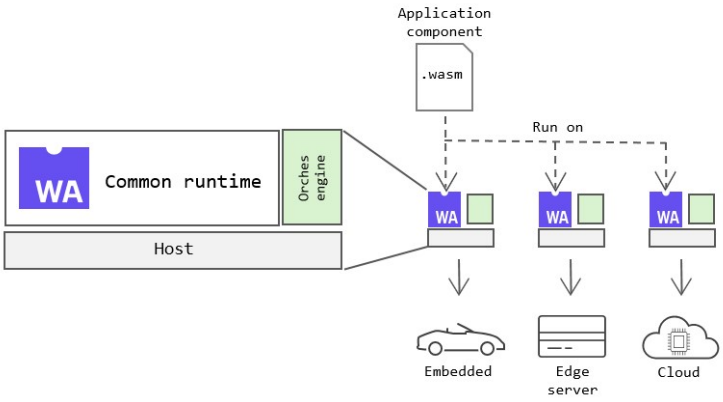

Fig. 1. A WebAssembly runtime, acting as a common runtime, could be used to permit the same application components to run in any node of the system, regardless of the hosting node architecture and software stack.

Authors' addresses: Edoardo Tinto, edoardo.tinto@phd.unipd.it, University of Padova, Padova, Italy; Tullio Vardanega, tullio.vardanega@unipd.it, University of Padova, Padova, Italy.

To develop a unified runtime, as depicted in Fig. 1, one that can be embedded in any type of compute node of the Continuum, and that allows one and the same application component to be executed anywhere across them, regardless of the underlying host, we should be mindful of the execution contexts in which application components might be run into. Here, we consider three of those contexts, which, as a whole, are representative of the industry standard. A proposal aiming to be truly exemplary of all possible code deployment and execution scenarios has to encompass the following three:

(1) Fully virtualized environments within resourceful Cloud datacenters.
(2) Edge devices hosting an RTOS (for instance, a Linux-based RTOS, which is a growing trend in cyber-physical systems).
(3) Devices with no OS at all, for the more resource-constrained systems, for whom an OS would be prohibitive or unsuited.

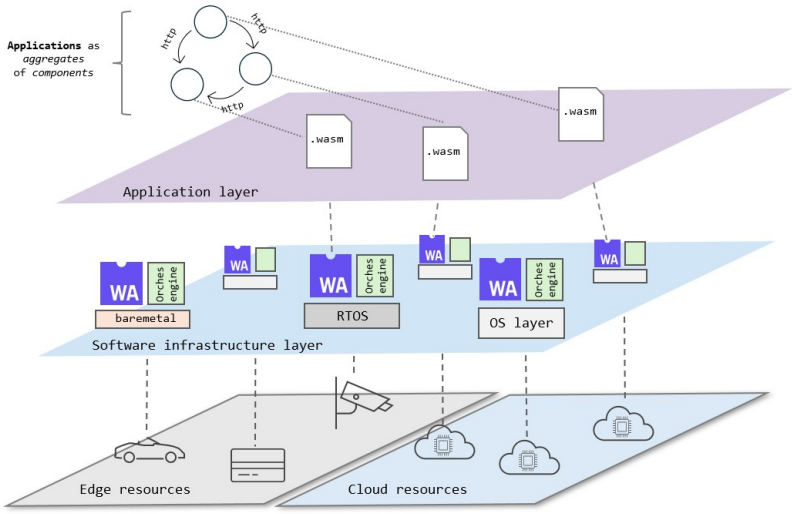

Fig. 2. The envisioned infrastructure, representing the three execution contexts (OS, RTOS, and bare metal), and applications as aggregates of orchestrated components

Noticeably, the execution contexts presented here are devoid of containers, as depicted in Fig. 2: the envisioned model is entirely container-less. Having shed light on where we intend to compute, the expectations for the novel computing model should be addressed.

1.1 Interactions on top of (streamlined) Web protocols

Various manners of interaction exist among distributed software components, ranging from socket-based communications to structured RESTful API. Some of these mechanisms predate the birth of the Web and handle resources, in the form of remote objects, through language-specific features as part of complex frameworks (e.g., CORBA). Therefore, with the aim of electing a single approach to interface components, we argue that Web-level protocols, such as HTTP/3, represent the most suited choice at the present state of the art. Accessing remote resources through Web-level protocols, as partook in the case of Cloud resources, is evidently more attractive than its alternatives, thanks to clients

simplicity in obtaining access to them in HTTP interactions. However, the existing runtime libraries offering Web-level communication mechanisms tend to be highly dependent on general-purpose OS facilities, disqualifying themselves from being candidates for all our execution contexts. Developing new, self-contained libraries is therefore paramount.

1.2 Live migration of components

Computation should migrate in a seamless manner to any node in the Continuum, and the occurring migration has to be live, hence stateful and connection-preserving. Although common in Cloud systems, offloading might result in unsustainable runtime costs due to re-execution; live migration represents a better alternative. It brings several advantages, as it allows computation to move to where it is more convenient to run without re-execution. Assessing where to execute can not be done statically, beforehand, but is a decision that has to be taken at running time. Also, migration-inspired facilities have immediate benefits to the running applications. For instance, they encourage snapshotting the state of critical components, to allow performing roll-back procedures in the face of faults.

1.3 Dynamic resource pooling

Computing resources, now both in the Cloud and in the Edge (in the form of devices with spared computing power), should be dynamically grouped into federations, temporarily, to complete application-specific objectives. Executing components on top of these resource pools requires overcoming the heterogeneity that affects its Edge. Namely, a common execution environment should be available to each and every node.

2 ENABLING TECHNOLOGIES

Two technologies, both emerging from the ranks of web development tools, promise to play a crucial role in the construction of the infrastructure for the Continuum.

WebAssembly (Wasm) has shown that (1) compiler-enforced sandboxing, which means, limiting the points of contact, both as memory access and external interfaces, between a component and its surrounding environment, is already viable (hence, there is no reason for not enforcing sandboxing at compile-time). Also, (2) just-in-time (JIT) compilation of an application component could be done effectively and locally. We argue that this approach might be of interest in embedded and real-time systems. In fact, it enables a component in Wasm bytecode, for now, to migrate within the system and then be compiled where it is scheduled to execute, for that specific target only. This approach thwarts the need for compiling beforehand and storing each application component for each and every (compilation) target in the system, which would severely constrain migration. Finally, (3) this approach could reach near-native execution performance (which is better than what containerization offers today). Once the idea of running the same component anywhere is accepted, migrating that same computation seems a worthwhile endeavor.

Rust instead improves safety of software components thanks to an ownership-based memory management approach. Thanks to its statically assessable safety properties, this technology has a future in the embedded and real-time domains. Adopting a memory-safe language for the infrastructural element of the Continuum is a key attribute of our proposal.

3 THE COMPILING TOOLCHAIN

To this day, we can imagine executing an application component in two manners, via interpretation or after (cross-)compilation. While interpreting a Wasm component, its source code passes through several stages, as shown in the case of an LLVM-based compilation toolchain:

Programming language > IR (LLVM) > Wasm (ready for interpretation)

The source code of the component is turned into Wasm bytecode through several passes. Then, the Wasm bytecode could be directly interpreted without further processing.

Conversely, when wanting to compile, in a JIT fashion, a Wasm component, we need to perform additional steps, as shown in the next line, where a simplified compilation pass operated by Cranelfift, the Wasmtime built-in compiler, is shown:

Programming language > IR (LLVM) > Wasm > IR (Clif) > ELF

The whole procedure (from source code to machine instructions), passes through two compilation stages, and, in so doing, attains the full benefits of Wasm. This is already a substantial improvement, and much could be done from this milestone.

However, would it be possible to shorten the compilation procedure, still maintaining the advantages mentioned earlier (sandboxing, for one)? In the future, this latter scenario might be characterized by a single compilation stage:

Language > IR (Clif) > ELF

This question arises while thinking about the realization of a migration mechanism, transparent to the application.

4 RUNTIME-MANAGED LIVE MIGRATION

Achieving live migration requires statefulness, hence the capacity to preserve any advancement in the migrating computation after the migration occurs, and connection preservation. To attain the former, two challenges need to be addressed: (1) how to checkpoint and then restore a computation, and (2) how to support different processor architectures. Recent research outcomes in the field of intermittent computing, where computation is expected to suffer interruptions due to frequent energy shortages, suggest viable approaches on how to checkpoint the state of a computation. In those proposals, the compiler inserts the runtime procedures required to take a component snapshot, as a compile-time optimization pass. This is an interesting starting point for building a stateful migration mechanism. Defining a state representation, meaningful above different host architectures, is still an open question. The case of interpreter-based migration poses less of a challenge because stateful migration could be already achieved, with minimal interventions on the instruction fetch loop.

Preserving existing connections is necessary to avoid the burden of re-establishing connections upon migration, which would cause a run-time penalty. Existing network protocols such as QUIC provide a valuable starting point for reasoning on a client and server-side migration mechanism, capable of preserving existing connections.

5 THE ORCHESTRATION SCENARIO

In the Continuum, applications must be intended as cohesive aggregates of relatively small computations, capable of being moved and executed in any node of the system. These aggregates require orchestration, in a manner that is rather distant from its Cloud-native counterpart. Characteristics of the latter are the lack (or difficulty) in building federations and the high footprint of its agents, designed for resourceful virtualized machines, and thus unsuited for the leaner Edge. Instead, from a Continuum-native orchestration, we should expect:

(1) Lightweight orchestration engine, suited for even the most resource-constrained Edge devices.
(2) Context awareness, in a way that allows scheduling decisions to be made with respect to user-defined performance indicators, such as latency.

(3) Migration resulting from orchestration-level decisions, where a control plane, being distributed or centralized, decides if a component has to move.
(4) Federation of clusters, in a hierarchical fashion.
(5) Possibly, try to minimize, if not utterly remove, the use of configuration files for providing the orchestrator with information about components. Much of that same information could be embedded, in the form of annotation, in the application source code itself.

6 APPLICATION LAYOUT

Our proposed model enforces the following application-level features:

(1) Applications should be composed of migratable components.
(2) The migration and restore procedures should be entirely managed by the runtime. Deciding when to interrupt the computation is a delicate step, which should be addressed by a compiler, not a developer.
(3) Interaction between a component and the outside world should happen through a single interface, such as a Web-level protocol. Runtime libraries with minimal external dependencies, exposing HTTP-like interfaces might be particularly effective to this end.

7 SCENARIOS OF USE

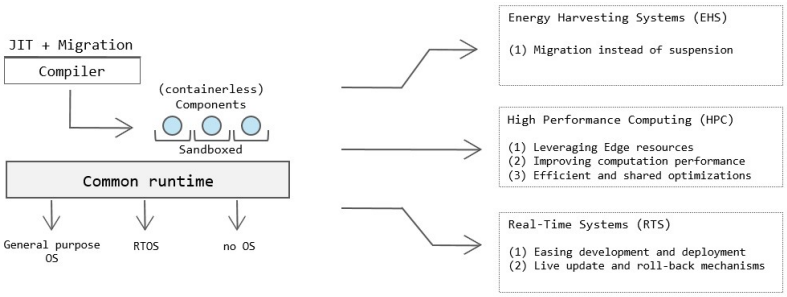

Fig. 3. Some key traits of the proposed runtime, and the advantages they offer to three application domains, namely energy harvesting systems (EHS), high-performance computing (HPC), and real-time systems (RTS).

A common and migration-capable runtime could benefit many application scenarios, as shown in Fig. 3. The already mentioned intermittent computation, found in energy harvesting systems (EHS) is one of those. Here, instead of suspending a computation upon the arrival of a low-energy phase, we allow it to move to another more energy-rich location, with a potential gain in performance.

High-performance computing (HPC) might also benefit from the envisioned model, mainly due to the following three facts. The Continuum offers to HPC a possibly vast pool of computing resources, those in the Edge, which, at the moment, are almost entirely detached from it. In addition to that, the proposed model could attain near-native speed while executing application components, which is a notable improvement when compared to traditional containerized HPC applications. Finally, a common runtime offering an internal compiler will encourage the development of (possibly) highly efficient optimization passes, transversal to many projects, instead of application-specific optimization with scant portability.

To conclude, the real-time domains might find some valuable contributions in the Compute Continuum. The ease of development and deployment are two of those. A common runtime infrastructure would simplify writing applications due to the well-defined interfaces between components and the environment, while also facilitating component deployment due to its homogeneity, being the runtime common to all the hosting nodes. Migration-related facilities would also contribute to improving performance, support live updates of application components (while the system remains operational), and offer new roll-back mechanisms to address transient software faults.

Revised: September 2024