Runtime Dependency Inversion For Scalable Programming

Position Paper

Areg Melik-Adamyan
Intel Corporation
areg.melik-adamyan@intel.com

Kshitij Doshi
Intel Corporation
kshitij.a.doshi@intel.com

***Abstract*—For decades, programs have been written with the implicit notion that the hardware on which they are to be executed is fixed in quantity and capabilities, and that required adaptation to those quantity and capabilities of hardware. Virtualization and cloud computing made it possible to have elasticity of demand and this fostered the emergence of the cloud-native programming models. The developers now focus on APIs, resiliency, extensibility, service level functionality, etc. Applications run mainly as functions and containers on top of some largely standardized runtime and the runtime capabilities represent the underlying computational resources in a predetermined manner, to which the application software adapts. This limits proper use of programming models, as underlying layer abstractions leak to the upper layer and forces application to adapt to underlying layer the same way as with bare hardware. In this paper, we introduce a Runtime Dependency Inversion Principle, an idea in which the infrastructure-as-code paradigm also covers runtimes, thus allowing an application to compose, on demand, the runtime best matched to its execution requirements, so that an application uses the programming model that best fits the intent instead of being constrained by the runtime supported model of programming.**

***Index Terms*—cloud-native, programming model, runtimes, scalability, software defined everything.**

I. Introduction

Infrastructure provisioning and complex application deployment in the cloud is a demanding and complicated task. Traditionally programmers were writing software on machines very similar in type to those on which programs were meant to be deployed. It was one of the fundamental advantages, to develop and debug a program on a desktop and then deploy it on an architecturally congruent server, that helped standardized high volume architectures to rapidly displace expensive purpose-built machines.

The emergence of cloud-native programming at scale places this model in tension in several key respects. It is not practical to assume that the environment for code development matches that of production scale testing and deployment. It is challenging to mock-up an end production deployment for testing to be sure that a microservice does not fail during startup. In general, individual developers or small teams get to complete unit testing only of a part of a final system as a microservice wrapped in a container—without a way to replicate or mimic a full environment locally.

Thus the environment that a programmer faces has many moving parts, dependencies, CI/CD pipelines, orthogonal components such as tools and languages for realizing infrastructure as code accompanying the application without a guarantee to be compatible with each other. The developer fundamentally has little control over a significant amount of software and platform infrastructure that is set up and set running, on which their application depends. Further, each change requires extra effort, and frequently a change that applies well to the whole infrastructure is hard to reflect in the local environment.

Several helper tools provide partial solutions:

- Kubernetes [1] – serves as a Swiss army knife in automating a great many infrastructure setups. A key limitation of it, however, is that it is hard in Kubernetes to define a complex overlay network topology, incorporate storage for stateful applications, and use heterogenous computational units.
- Terraform [2] – is the de-facto standard for defining infrastructure-as-a-code, but it has a low-level of abstraction. While it is intended to ease infrastructure deployments across clouds, at its core it is basically an automation tool.
- Apache Brooklyn [3] – provides for considerable support for intent-based computing; however, it is a very complex language, and currently lacks primitives for expressing control over hardware and service level objectives (SLOs).
- Some standards – e.g., CAMP [4], TOSCA [5]. Not very popular, low-level and complicated for humans.

The approaches mentioned above suffer from two key limitations today: the abstractions they employ remain at a low-level, producing code bloat and vendor lock-in. To circumvent, a high-level declarative means are needed to deploy the needed runtime with needed capabilities.

II. Requirements For Runtime Inversion Through Intents

Developers often know what combinations of hardware and communication capabilities and software frameworks are well-matched to the cloud design patterns that they are designing into their applications. Today they can communicate these requirements in low-level directions such as what should be

the hardware and software compositions for the containers, and how those containers are interconnected with one another and with various third party services. To express and achieve granular and situation-adaptive SLOs a developer needs help with declaring an intent at a high level and to have the resource provisioning, runtime composition, and scheduling infrastructure come together in an elastic manner to satisfy the intent. This is similar to intent-based networks (IBN) [6], which let network operators and application programmers to express what is required from a network and internally work-out the mechanism details of "how it is accomplished". A second parallel to draw from IBN is the maintaining of the intended goal unless the goal changes – thus removing complexity both in setup and continued operation. When developers similarly achieve the runtime and hardware compositions that flex similarly, enterprises can reduce expenses while improving efficiency and reliable operation. The next evolution of runtimes should therefore include abilities to:

- Control and provision hardware resources in a vendor-agnostic way
- Declaratively define requirements for the hardware that the application will run
- Define application constraints and SLOs in a declarative manner
- Deploy different types of applications: stateful/stateless, service/batch, repetitive/event-driven, etc.
- Have a single toolchain
- Interact with the provisioning system dynamically through an APIs directly from the application.

This comprises the essence of the Runtime Dependency Inversion (RDI) Principle — runtimes to be constructed on-demand to track and maximize developers' objectives, adapt to the underlying hardware, and minimize the need for applications to change their logic to adapt to operating hardware and runtime.

III. Runtime Dependency Inversion Principle in Action

Under the RDI principle, we propose that runtimes be (1) software-defined, (2) satisfy application SLOs for operation and (3) be deployable by/from the application. We envision a domain-specific Infrastructure Control Language, for embedding directives that a preprocessor translates into instructions for a hardware infrastructure processing layer (IPL). An IPL may operate at multiple levels (node, rack and datacenter). The directives include: (1) Generic descriptions of requestable hardware resources – such as, CPU, GPU, TPU, FPGA and others, with SLOs over them, (2) Definitions of computing entities, including whether they are stateful(actor-like) or stateless (tasks) and onto which resources they may be deployed, (3) Deployment directives, which convey the mapping between SLOs and provisioning of resources (4) Connect directives which are used to provide a graph of connections among resources, along with applicable data access SLOs (5) Definitions of a long-term external persistent storage and virtual overlay network with its specific SLOs, and (6) Generic runtime requirements that span multiple application structures – from legacy applications to cloud-native applications. Listing 1 shows an example of construction under the RDI principle.

Listing 1. ICL example

```
@resource dcluster {
  CPU: { num: 128, RAM: >128GB }
  Network: { topology: any, throughput: > 40
      Gbps, protocol: any }
}

@resource ccluster {
  CPU: { num: 16384 }
  GPU: { gang: 512}
  TPU: { num: 64, type: habana}
  Network: { topology: all-to-all, latency: <
      20mcs,  protocol: QUIC }
}

@resource web {
  Network: { protocol: https; requires: mtls }
}

@connect net1 { network: d_cluster, c_cluster,
web; security: mtls; encryption: AES128 }

    # omitting other actors

@actor DL { storage: NFS }
{
@run {binary: /nfs/models/model1, path: /
    horovod/nfs/input/*}
}

@deploy {service: DL, resource: c_cluster,
runtime: torch-horovod}
{
    @on: scheduling -> static
}
```

IV. Conclusion

Runtime Dependency Inversion principle changes the relation of the application and underlying hardware and runtime by allowing one to express the requirements of the application in a high-level, software defined way. It allows the same application to be deployed into diverse environments by clearly separating the concerns of the application logic from the structure and mechanisms under the runtime interfaces.

References

[1] Kubernetes, https://kubernetes.io/.
[2] Terraform, https://terraform.io/.
[3] Apache Brooklyn, https://brooklyn.apache.org/.
[4] Cloud Application Management for Platforms (CAMP), https://www.oasis-open.org/committees/download.php/47278/CAMP-v1.0.pdf.
[5] OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA), http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.pdf.
[6] B. Saha et al, Intent-based Networks: An Industrial Perspective, FICN '18: Proceedings of the 1st International Workshop on Future Industrial Communication Networks, October 2018.