Making Cloud Easy: Design Considerations and First Components of a Distributed Operating System for Cloud

Wolfgang John, Joacim Halén, Xuejun Cai, Chunyan Fu, Torgny Holmberg, Vladimir Katardjiev, Tomas Mecklin, Mina Sedaghat, Pontus Sköldström, Daniel Turull, Vinay Yadhav, and James Kempf

Ericsson Research

Abstract

Cloud offerings have over the years been transformed from bare-metal to virtual machines, to containers, and most recently to serverless functions. Each of these execution context abstractions has been accompanied by a new layer of centralized management, but these extra management layers have led to a confusing and fragile system, that wastes developer time on managing execution contexts with little contribution to the application. We argue that it is time to revisit ideas of Single System Image (SSI) concepts to simplify the management of Cloud execution contexts. An SSI abstraction for Cloud provides easy and convenient developer access to resources without recourse to programming multiple levels of execution contexts. We propose a novel set of design principles inspired by earlier distributed operating system and SSI research. We also present a first corresponding service, realizing fully decentralized resource management, folding multiple layers of centralized management stacks into a single layer spanning across datacenter resources. As a result, developers never see individual execution environments, but deal with processes and IPC familiar from local development machines.

1 Introduction

Cloud management systems have gone through two generations of evolution since the introduction of virtualization technology in the early 2000's. The first generation lifted servers up into virtual machines (VMs) and placed them on virtualized hardware. Most management was done with custom built scripts, or proprietary tools such as VMware's VCenter [1], designed specifically for single tenant enterprise datacenters (DCs). Through server consolidation, enterprise DCs were able to achieve substantial OPEX and CAPEX reductions. However, the demand for SLAs and multi-tenant isolation required more advanced resource management solutions.

The second generation, represented in the open source arena by OpenStack [2], features centralized resource managers across the DC for computation, networking, and storage, and a collection of services for autoscaling, service discovery, health monitoring, etc. Framework and container management systems such as Mesos [3] and Kubernetes [4] have been layered on top of such DC management systems, increasing the level of complexity. Recently, serverless computing or function as a service (FaaS) introduced yet another layer of execution context abstraction, both in public and open-source Clouds [5].

As new abstractions are introduced for each execution context, a new layer of management is required, often with different APIs. For developers, accustomed to developing and deploying applications on an operating system (OS), dealing with multiple management systems adds complexity unrelated to their core applications. We believe that next generation Cloud platforms must simplify the developer's interactions with the DC.

In addition, DC management has its own set of pain points. Adding or removing servers as well as upgrading the platform, all require hand-editing, susceptible to human error. Furthermore, datacenter staff must run a 'development' deployment for testing next to their production deployment. Networking is another source of complexity and time sink for developers. Virtual networks need managing, even as performance like latency often disappoints. We argue that networks inside a DC should be like a bus in a server: fast and invisible, with its management hidden by default.

In this paper, we present a decentralized approach to designing Cloud management systems. Our approach, based on a set of fundamental design principles, aims to simplify DC operation and facilitate developer interaction with the Cloud platform. With our principles, we argue that it is worthwhile to revisit pioneering work on Single System Image (SSI) [6] and distributed OS ideas in a Cloud context, enhanced with recent advancement in

Corresp. authors: [daniel.turull | joacim.halen]@ericsson.com
At the time of writing, Pontus Sköldström was with RISE Acreo.

hardware and software technology and methods. Sec. 2 presents a brief overview of this work, and discusses its applicability to the modern Cloud DC. Sec. 3 presents our design principles and and their advantages. Sec. 4 discusses our corresponding initial Cloud stack. Our first proof-point, Nefele Compute, is outlined in Sec. 5, and Sec. 6 summarizes the paper and discusses open issues.

2 Distributed Operating Systems Research

We take inspiration from the a collection of academic work on distributed OSs in the 1980's. Architecturally, a distributed OS consists of two components:

i) A kernel, which handles access to the resources on a single machine. Historically, this kernel was a microkernel, keeping the OS size to a minimum.

ii) A collection of distributed agents that implement the OS services as a distributed system. They access the local microkernel for resources, co-ordinate amongst themselves, and reflect those resources out to the users of the OS as a single network-wide resource.

Schwarzkopf, et al. [7] describes how modern DCs can benefit from a distributed design, highlighting the historical obstacles to a distributed OS, and how these obstacles are not relevant anymore. The goal of the distributed OS was to transparently place an application and hide the physical location from developers. The same goal holds in today's DC management systems. Transparency typically extended down into the kernel, so kernel operations like virtual memory (through distributed shared memory) and thread execution were distributed, as well as higher level services such as file systems[1]. One goal of distributed OS approaches was to maintain an SSI of a cluster of machines, i.e. presenting the distributed computing resources via an interface that maintains the illusion of one single system [8].

One reason why the distributed OS work lost traction was due to the disparity between CPU and network performance [7]. While CPU speeds increased dramatically from the mid-1980's through 2000, networking performance increased more slowly during this period. This disparity made transparent distributed virtual memory and thread execution very expensive. Moreover, the emphasis on micro-kernels during the rise of Linux prevented the distributed OSs from gaining widespread deployment and the distributed OS work lost traction.

Fast forwarding to 2018, the same performance issues still exist for distributed shared memory. A page fault on a modern, multicore server takes on the order of 100 nsecs to resolve, while a remote memory copy using a fast network protocol like RDMA takes on the order of 10 usecs. In the end, what the developer really wants is transparency of service location without paying a performance penalty. This suggests a way to bring distributed OS concepts into the modern Cloud DC context, namely:

i) OS services like resource management or process creation are implemented through a set of coordinated distributed agents, offering an abstraction of a DC consisting of many servers as an SSI for Cloud. These services hide the complexity of infrastructure management.

ii) Processes are the basic unit of execution in the OS and the basic way of communication are IPC and messaging. The latter hides most of the complexities with networking from developers.

iii) Resources on a server are handled by a single machine kernel. This includes virtual memory and threads, so the system has no distributed shared memory, and virtual memory and threads are single machine only.

As a consequence, processes cannot have sizes larger than a single server can support, and cannot create more threads than a single server can handle. We believe that these tradeoffs are justified given the performance cost applications would otherwise incur, and that few applications will be negatively impacted by these limitations, given modern multicore servers and terabyte RAM.

3 Design Principles for making Cloud easy

Considering the lessons derived from distributed OS research, we suggest a set of design principles [9]. These principles are intended to outlive the changing layer cakes of functional architectures and become the ground on which we base our Cloud management system:

All Cloud services are fully distributed. We believe that next generation Cloud systems must be designed along the lines of a distributed OSs such as Ameoba [10] and Saguaro [11]. Unlike these early distributed OSs, services that are performance critical, such as virtual memory and thread management, should remain local. Thus, a local kernel becomes a kind of per server module providing performance sensitive services for the OS, while other services, such as process creation, are implemented in a distributed fashion across the DC.

No unit of management abstraction below the datacenter. We believe a next generation DC OS has to provide a management abstractions that presents a DC as an SSI for the Cloud. Specifically, we consider an SSI for distributed applications running in the Cloud, providing transparent process placement as well as a single process, file, and IPC space in a multi-tenant environment. This SSI should provide an abstraction of DC server resources, i.e. compute including hardware accelerators, network, and storage, through an API that neither distinguishes where workloads are placed nor how to communicate with physical locations To address known is-

[1]Distributed shared memory and distributed thread execution are inter-twined since a common design pattern for inter-thread communication is via shared memory protected by memory based locking.

sues with the SSI and distributed OS concepts, the implementation of such a system has to be based on recent advancement in hardware and software technology and methods such as RDMA for fast state synchronization, or analytics techniques to mitigate suboptimal application placement [7, 8].

Datacenter capacity grows organically and self-configures. Human intervention should not be required to add or remove servers except for physical installation and maintenance. Furthermore, the management system should not require a footprint of several servers, but should scale down to even a single server. This principle has three important advantages: i) it reduces the possibility of human error; ii) it eliminates the role of centralized databases as single points of failure; and iii) it enables management of much smaller clusters and facilitates the agile deployment of micro datacenters.

All resources belong to a single resource manager. A *resource* is a physical/logical component in limited supply, necessary to provide a service to a tenant. Cloud resources typically have dedicated managers per resource type (specifically compute, network, storage) that allow tenants to perform computational or communication tasks. In existing management systems (e.g., OpenStack), these managers own the resources, so other managers and tenants must negotiate with all of them in order to obtain access, complicating resource management. We propose a single resource manager that provides a single point of contact for server resources, allowing aggregations of different resource types to be allocated in one atomic request. Our experience from earlier work shows that the lines of control in such an architecture are cleaner and allow better resource usage [12].

Communication abstractions along a latency gradient. One of the main concerns with having a distributed control plane is the communication overhead and the time for the various distributed functions in the system to converge. While high-throughput, low-latency network technologies reaching up to 100 Gbs/s are established in DCs, communication overhead and latency remain a serious challenge. For developers, we believe that the abstraction mechanisms provided must expose, rather than hide, latency. The reason for this is that developers often need to accommodate higher latencies with various implementation schemes (e.g. caching) whereas for a low latency connection such schemes are unnecessary.

Different communication mechanisms for different topological distances. It must be apparent for developers that communication over different topological distances, e.g. DC internal or external, has different characteristics and might come with different costs. For ease of use, it is at the same time desired to use the same communication interface. An analogy would be a country prefix code when making a international phone call compared to making a local call on the same line.

We are aware that contemporary container management systems, most prominently Kubernetes, share some of the vision behind our design principles. While we acknowledge the good reasons for their wide adoption in research and industrial communities, we want to point out some advantages of our proposed design, particularly with respect to ease of use.

First, developers and tenants do not have to deal with programming of execution context abstractions such as VMs or containers, but can focus on their application logic through an abstraction native for most developers, namely processes and inter-process communications

Second, networking details can be fully hidden from the developers. In current container management systems such as Docker Swarm or Kubernetes, containers on different hosts typically communicate using overlay networks. Each container is assigned an IP address and processes in the container bind/connect to specific ports. The addresses may be location-independent (e.g. IPv6 ILA [13]) and persistent, human-readable names can be mapped to addresses using DNS. Communication patterns often have to be implemented by the processes themselves and/or using external means e.g., anycast and round-robin through DNS, IP multicast, or message brokers. In our SSI for Cloud, each communicating process has a location-independent mailbox, addressed by the global PID or a name, so ports do not need to be coordinated. Additionally, the messaging API provides a number of common communication patterns such as pub/sub, any-, multi-, and broadcast, and various load-balancing and failover mechanisms. These built-in mechanisms simplify development of distributed applications but still allow users to implement custom mechanisms.

Finally, following our proposal, groups of agents are formed and optimized automatically to limit control plane communication overhead. This is done through a boot federation process, with little need for explicit developer involvement. A distributed approach scales automatically without a single point of failure.

4 Initial Platform Architecture

Based on the above design principles, our initial platform is shown in Fig. 1. The primary use case for the proposed platform is to support Cloud native development environments. Having the development environment actually running in the Cloud allows developers to build as well as deploy applications directly in the Cloud. In a sense, we would like to redefine "Cloud native" from "born in the Cloud" to "gestated in the Cloud", erasing the difference between the developer's laptop and the Cloud. The platform is especially well suited for FaaS because the

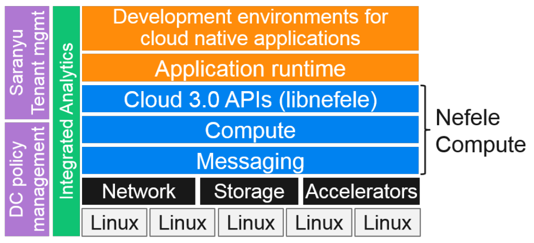

Figure 1: Proposed Cloud Architecture

application deployment model of FaaS inherently treats the entire DC like a single compute resource.

To support developers with familiar and well established abstractions known from Linux, the new platform's API represents the available Cloud resources in the form of an SSI. At the core of the SSI is a distributed computation service which realizes a fully distributed control plane. The control plane is responsible for placing and managing processes across the datacenter's servers. This distributed design avoids single points of failure and makes incremental scaling possible, both down and up. Communication between processes within the datacenter is exposed through common IPC rather than sockets and TCP. IPC is implemented using a brokerless messaging service, effectively removing networking inside the DC from the developer's concern. For communication with external services, e.g., on the Internet or in other DCs in a distributed Cloud, developers can interact with the same interface, and standard IP networking will be used through a proxy. The communication between client processes and the proxy will use the low latency, intra-DC protocol and the proxy will handle communication with external services using IPs, ports, and sockets. The developer will not need to use sockets for microservices and serverless functions that live in the same DC. As increased network performance is key for a distributed OS, we are exploring how to seamlessly integrate recent network hardware and software improvements like RDMA, DPDK, and smart NICs.

5 Nefele Compute Implementation

Nefele Compute is the computation-as-a-service component of our distributed resource management and the first service following our design principles[2]. Nefele provides a fully distributed, multi-tenant DC management system for computation resources, without a single monolithic controller, providing an SSI abstraction of an entire DC. Nefele allows developers to utilize operating system APIs within a DC just as they would on their laptop, while handling resource management and namespace isolation transparently (Fig. 2). For the SSI realized by Nefele, we assume that the DC is homogeneous in terms of computer architecture (i.e. instruction set), but supports heterogeneity when it comes to, e.g., CPU speed, number of cores, and memory size.

The distributed control plane is built upon a collection of *Nefele Agents*, each running self-contained on a single server. Communication between agents is asynchronous through IPC, which provides a high degree of concurrency and parallelization of control, resulting in faster decisions. Although there are advantages with a distributed control plane, it is inherently more sensitive to errors as tasks are distributed among many components that can fail. In order to make Nefele more reliable we use well established techniques and mechanisms on identified sensitive paths, e.g., supervisors to monitor and restart agents on failure [15], redundant control-plane functions, and appropriate consistency mechanisms for handling of OS states like process tables.

In our first deployment, Nefele Agents consist of a set of subagents realizing the following services:

i) *Boot federation*: Nefele agents automatically federate into groups using the boot federation service. These groups allow for optimized communication paths.

ii) *Process placement*: Informed by analytics and policy, process placement decides where to deploy a process. For instance, there is no parameter in the API to indicate on which server to place a process. The decision about where to place server workloads is made by the resource manager based on the current DC load, constraints and policy as specified by the developer and DC owner, as well as historical execution analytics.

iii) *Distributed namespace management*: Tenant isolation is provided by Linux namespaces that are extended across all machines where a tenant is present, maintaining the SSI abstraction.

In the boot federation procedure, agents federate into small broadcast groups to keep communication as local as possible. These groups then assemble themselves into hierarchies until a top group spanning all agents is created. The boot federation procedure automatically rearranges groups when new servers are added or existing servers are removed or fail. This transforms the datacenter into a self-organizing, self-healing, masterless distributed system. To further enhance flexibility, the boot federation process allows agents providing different services such as placement and analytics to specify the parameters of their group organization, for example the number of agents per group.

Processes are the smallest execution unit in Nefele. They are based on a Linux process, using Linux namespaces for namespace isolation and virtual memory for memory isolation between processes on the same machine. Nefele automatically provides isolated namespaces for a tenant on servers where the tenant's processes

[2]Saranyu, the second service, is a distributed tenant and service management system based on smart contracts, and is described in [14].

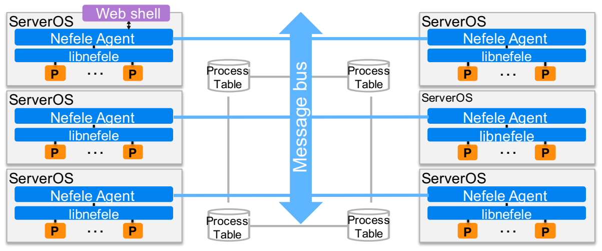

Figure 2: Nefele Distributed Agent Architecture

are placed. A tenant process is given a global Nefele id and the Linux process id is never seen outside of the OS on the server where the process is running. The tenant can only see and query its own processes from the distributed control plane. For instance, the commonly used Linux command *ps* will list all processes associated with the respective tenant in the entire DC.

The library *libnefele* is the entry point for Nefele system calls. This library contains functionality for process creation and deletion, retrieving process information, interprocess communication, and other OS functions.

```
// Create a collection of processes in one atomic call

int nefele_c_spawn(
 nef_pid_t *pids[],        // Pids of new processes
 int n,                    // # of processes to spawn
 char *paths[],            // References to executables
 char *argv[],             // Arguments
 char *envp[],             // Environment variables
 affinity_t *affinities[],// Affinity rules
 policy_t *policies[],     // Resource limitations
 profile_t *profiles[]     // Workload profiles
) ;
```

We have added functions that reflect common operations in a Cloud environment. Examples are `nefele_n_spawn()` which creates *n* identical new processes atomically, running the same code on different data, and `nefele_c_spawn()` (see listing) which creates a cluster of dissimilar processes running different code.

A call to the latter is turned into an asynchronous request to the local Nefelel Agent. This agent fills in values to parameters not provided in the call and may augment others. These values can be default values, specified globally by the tenant, derived by a static analysis, or derived by an analytics system based on earlier invocations of the specified binaries. The extended request is then sent asynchronously to the local placement agent, which communicates with neighbouring placement agents according to the group created during the boot federation. Each contacted placement agent decides which of the requested workloads it can take based on its current set of workloads and how they are expected to evolve over time, and sends an resource offer back to the requesting agent. By specifying values for affinity or other policies, the developer can provide fine-grained intents to the placement agents. When the consolidated reply comes back from the local placement agent, the local Nefele Agent sends a request to create the corresponding processes to the selected remote agents as specified in the reply. These agents reply with the assigned Nefele process id once the processes are created. After which, the local Nefele Agent originally receiving the `nefele_c_spawn()` call, returns the array of Nefele process ids to the calling user process.

6 Summary and Open Issues

We described the design of a new Cloud management system that is explicitly intended to simplify developer and operator interactions with a DC. We base its design on a novel set of architectural principles derived from the distributed OS literature. The proposed management system differs by redefining certain OS services to span across the entire DC rather than adding yet another management layer. As a result, developers never see individual servers, but rather the entire DC looks like an image of a single system - an SSI for Cloud. This provides easy access to DC resources without recourse to execution context management programming.

We believe making Cloud easier to use along the lines of our design principles will translate into several benefits: i) Simplified automation and higher resource utilization leading to lower OPEX, as automation becomes pervasive and the use of processes as rightsized execution units minimizes DC resource stranding; ii) Simplified development through focus on application code to the exclusion of extraneous configuration tasks, leading to lower development costs and a better overall developer experience; and iii) Increased usability and scalability through a distributed approach, allowing computation clusters to automatically federate and to scale down by the same mechanism as they scale up.

An alpha release of our platform based on Nefele Compute and Saranyu Tenant Management [14] is currently up and running in our DC at Ericsson Research. Right now, we are performing validation and performance tests. In addition, we plan to improve the mechanisms for tenant isolation by experimenting with new namespace extensions, to reduce communication latency by fully utilizing RDMA, to employ analytics to support process placement through self-learning, and to enrich the *libnefele* API with calls supporting development of distributed systems at the lowest level.

Acknowledgments

The authors want to thank the anonymous reviewers for their helpful comments and John Wilkes for his efforts and time to shepherd this paper with a lot of constructive feedback. In part, this work has been supported by the VINNOVA K5 project.

References

[1] VMware, "VCenter Server Data Sheet," 2018. [Online]: http://www.vmware.com/se/products/vcenter-server.html (Accessed: 2018-05-14).

[2] OpenStack Foundation, "OpenStack Open Source Cloud Computing Software," 2018. [Online]: https://www.openstack.org (Accessed: 2018-05-13).

[3] Apache Foundation, "Apache Mesos," 2018. [Online]: https://mesos.apache.org/documentation/latest/ (Accessed: 2018-05-14).

[4] Linux Foundation, "Kubernetes: Production grade orchestration," 2018. [Online]: http://kubernetes.io (Accessed: 2018-05-14).

[5] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research," *CoRR*, vol. abs/1708.08028, 2017. [Online]: http://arxiv.org/abs/1708.08028 (Accessed: 2018-05-09).

[6] R. Buyya, T. Cortes, and H. Jin, "Single System Image," *Int. J. High Perform. Comput. Appl.*, vol. 15, pp. 124–135, May 2001.

[7] M. Schwarzkopf, M. P. Grosvenor, and S. Hand, "New Wine in Old Skins: the Case for Distributed Operating Systems in the Data Center," in *Proceedings of APSys*, 2013.

[8] P. D. Healy, T. Lynn, E. Barrett, and J. P. Morrison, "Single system image: A survey," *J. Parallel Distrib. Comput.*, vol. 90-91, pp. 35–51, 2016.

[9] J. Halén, W. John, and J. Kempf, "Cloud 3.0: Making Cloud Easy," 2018. [Online]: https://www.ericsson.com/research-blog/cloud-3-0-making-cloud-easy/ (Accessed: 2018-05-13).

[10] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A Distributed Operating System for the 1990s," in *IEEE Computer*, pp. 44–53, May 1990.

[11] G. Andrews, R. Schlichting, R. Hayes, and T. D. M. Purdins, "The Design of the Saguaro Distributed Operating System," in *IEEE Transactions on Software Engineering*, pp. 104–118, Jan 1987.

[12] J. Halén, S. Hellkvist, S. Baucke, F. Wuhib, and Y. O. Yazir, "Wind: Management and Orchestration in the Distributed Heterogeneous Cloud," in *IEEE World Congress on Services (SERVICES)*, 2015.

[13] T. Herbert and P. Lapukhov, "Identifier-locator addressing for IPv6," Internet-Draft draft-herbert-intarea-ila-01, Internet Engineering Task Force, Mar. 2018. Work in Progress.

[14] S. Nayak, N. Narendra, A. Shukla, and J. Kempf, "*Saranyu*: Using Smart Contracts and Blockchain for Cloud Tenant Management," in *IEEE Cloud Conference*, 2018.

[15] Erlang/OTP, "OTP Design Principles User's Guide," 2018. [Online]: http://erlang.org/doc/design_principles/users_guide.html (Accessed: 2018-05-03).