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

James Kempf for the Nefele Development Team
ER RACSP
July 09, 2018
3rd generation cloud design principles

- Datacenter is abstracted as Single System
- All cloud services are fully distributed
- Datacenter capacity organically adapts and self-configures
- All resources belong to a single resource manager
- Different communication mechanisms for different "distances"
- Set of communication abstractions structured along a latency gradient
Differences from previous SSI work

— No distributed kernel
  — Single server Linux kernel used as a modular system component
    — Provides virtual memory and thread management
  — Other operating system services like process creation and resource management are distributed
— No distributed shared virtual memory or distributed thread management
  — Processes get only as much virtual memory as on a single machine and only as many threads as the local Linux OS can provide
— Monolithic applications are not distributed automatically as SSI suggests
  — Applications need to be designed modularly for distribution

Our work: SSI for Cloud Native

Image source from top to bottom:
https://www.zerostack.com/cloud-native-apps-icons-hover/
http://www.iet.uniipi.it/a.bechini/concur/concur.html
Addressing pain points in existing cloud platforms

— OpenStack
  — Remove lower bound on minimum unit of deployment without impacting scale up potential
  — Processes rather than VM images for execution context
  — Remove need for programmer to deal with virtual networks and infrastructure programming
  — Reduce the administration effort, simplify upgrade and maintenance

— Kubernetes
  — Add native tenant management to improve potential for resource and capacity sharing
  — Replace networking with IPC to simplify service interface and improve performance

— Overall
  — Reduce the number of execution context layers
  — Improve co-ordination between layers
Making 3rd generation cloud real

Implemented now:
- Single System Image for easy development and deployment of applications
- Automatic resource federation for easy setup of datacenter resources
- Message bus for structured, fast control plane messaging
- Blockchain based tenant management with smart contracts

Under development:
- Serverless application runtime with fast key-value store for state handling
- Single touch installation, configuration and upgrade
- Acceleration through domain specific hardware

Remainder of talk

Development environments for cloud native applications
Application runtime
3rd Generation Cloud APIs
Compute
Messaging*
Network
Storage
Accelerator
Linux
Linux
Linux
Linux
Linux
Linux

* Together with RISE and VINNOVA
Nefele SDK

- The SSI abstraction is fronted to the developer through an SDK library
  - Structured like the Linux *libc* OS Syscall library
- SDK library is *libnefele*
  - C library for now (Python and Golang under test)
  - All API calls block until the call completes
- Currently implemented functions
  - Process management (creation, deletion, etc.)
- Legacy
  - Filesystem uses legacy Linux API
  - Linux *libc* calls are also available for single machine operations
Hermoðr Stack

Processes communicate through mailboxes.

Messages in Google protobuf format.

Three addressing schemes:
- Random addresses similar to TIPC ports
- Assigned and distributed location-independent numbered services from TPIC
- Symbolic string based names

Implemented Protocols

- Reliable messaging
- Unreliable messaging
- High-performance RDMA

In process

Implemented Channels

- ZeroMQ
- TIPC
- Transparent IPC
- Rsockets
- RDMA

Implemented Network Protocols

- Ethernet
- IP/UDP
- RDMA
Estimating $\pi$ using Monte Carlo simulation

- Generate points uniformly distributed at random over a 1x1 square.

- Keep track of points inside a circle with radius 0.5 and points outside
  - Blue is outside, red is inside
  - Area of circle is $\pi r^2 = \pi / 4$

- Divide $N_{\text{inner}}$ by $N_{\text{total}}$ should give an estimate of $\pi / 4$

- Multiply by 4 for $\pi$
MonteCarlo simulation to estimate $\pi$.

Parallel simulations to rapidly increase precision.

Sample application

- `nefele_n_spawn()`: spawn as many as tenant has cpu capacity
- `hrmd_new()`: connect to Hermodr message bus
- `Distributed shared folder (/data/pi)`
- `MonteCarlo simulation to estimate $\pi$.`
- `Parallell simulations to rapidly increase precision.`
Monte Carlo main program

```c
int main(int argc, char* argv[]) {
    (...)
    char* worker = "/usr/bin/montecarlo_worker";
    char* printer = "/usr/bin/montecarlo_printer";

    nefele_spawn(&pid_printer, printer, p_arg, ...);
    (...)

do {
    (...)
    nefele_n_spawn(pids, new_req, worker, w_arg, ...);
    nefele_waitpid(0, &status, 0);
} while (pending > 0 || completed < N1);

    nefele_kill(pid_printer, 9);
    printf(...);
}
```
Monte Carlo worker program

```c
int main(int argc, char* argv[]) {
    (...)
    msg.t = monte_carlo_count_pi(N2);
    msg.n = N2;
    (...)

    if (hrmd_service_exists(hrmd,
                           "printer@[8080,1,1]",
                           10000)) {
        hrmd_message(hrmd,
                      "printer@[8080,1,1]",
                      (const byte*) &msg,
                      sizeof(msg));
    }
    (...)
}
```
Assumptions, Open Issues, and Feedback

— Assumptions
  — Designing cloud management software as a *system* to remove the need for infrastructure programming and simplify networking will:
    — Radically reduce the amount of work a developer needs to go through to develop and deploy applications into a cloud
    — Radically reduce the amount of work needed to manage cloud infrastructure
      — Partially this will be due to being able to design management automation in from the start rather than having to patch it in later

— Open Issues
  — Will the Single System approach work without distributed shared virtual memory and distributed thread management?
  — Can our approach better support Kubernetes and other Cloud Native Foundation technologies and serverless programming than running on Openstack or directly on Linux?
  — Can our approach better support emerging application classes like the RiseLab Ray AI application framework?

— Feedback
  — IPC v.s. IP for communication and exposing latency to the developer?
  — How to incorporate policy?
  — Management of storage?
Single System Image for Cloud Native: Key Ideas

- Use processes instead of VMs for execution context
- Use IPC instead of IP for communication locally, expose latency to the developer to allow them to compensate in their designs
- Design cloud systems management to be automated rather than automate it after designed