Containers and Virtual Machines

Everything old is new again (not really)

Michele Finelli
m@biodec.com
BioDec

Index

Index

Where do they come from ?

- IBM ! IBM invented everything in the 60's.
- Not really, but in a sense . . . it comes from people who wanted to *run away* from IBM.
- Honeywell 200/2000 systems Liberator allowed people to **run a virtual machine that emulated** the IBM 14xx systems.

Where do they come from ?

- IBM ! IBM invented everything in the 60's.
- Not really, but in a sense . . . it comes from people who wanted to *run away* from IBM.
- Honeywell 200/2000 systems Liberator allowed people to **run a virtual machine that emulated** the IBM 14xx systems.

Where do they come from ?

- IBM ! IBM invented everything in the 60's.
- Not really, but in a sense . . . it comes from people who wanted to *run away* from IBM.
- Honeywell 200/2000 systems Liberator allowed people to **run a virtual machine that emulated** the IBM 14xx systems.

Where do they come from ?

- IBM ! IBM invented everything in the 60's.
- Not really, but in a sense . . . it comes from people who wanted to *run away* from IBM.
- Honeywell 200/2000 systems Liberator allowed people to **run a virtual machine that emulated** the IBM 14xx systems.

The glorious 60's

- There were dozens (hundreds ?) of different hardware, all of them incompatible which each other.
- IBM wrote OS/360 to *rule them all* but . . .
- . . . people did not want to rewrite all their (working, legacy) software for the new platform.
- So IBM decided to support **virtual machines** to run its own OSs inside the OS/360.

The glorious 60's

- There were dozens (hundreds ?) of different hardware, all of them incompatible which each other.
- IBM wrote OS/360 to *rule them all* but . . .
- . . . people did not want to rewrite all their (working, legacy) software for the new platform.
- So IBM decided to support **virtual machines** to run its own OSs inside the OS/360.

The glorious 60's

- There were dozens (hundreds ?) of different hardware, all of them incompatible which each other.
- IBM wrote OS/360 to *rule them all* but . . .
- . . . people did not want to rewrite all their (working, legacy) software for the new platform.
- So IBM decided to support **virtual machines** to run its own OSs inside the OS/360.

The glorious 60's

- There were dozens (hundreds ?) of different hardware, all of them incompatible which each other.
- IBM wrote OS/360 to *rule them all* but . . .
- . . . people did not want to rewrite all their (working, legacy) software for the new platform.
- So IBM decided to support **virtual machines** to run its own OSs inside the OS/360.

The glorious 60's

- There were dozens (hundreds ?) of different hardware, all of them incompatible which each other.
- IBM wrote OS/360 to *rule them all* but . . .
- . . . people did not want to rewrite all their (working, legacy) software for the new platform.
- So IBM decided to support **virtual machines** to run its own OSs inside the OS/360.

Hardware level emulation

- To run different computer architectures, with different instruction sets, memory models, etcetera, you need to emulate **the whole hardware**.
- So basically you have **two operating systems** one on top of the other.
- The upper OSs are called “operating system”, sometimes **guest OSs**.
- The lower one is called **hypervisor** — but it is an operating system nonetheless.

Hardware level emulation

- To run different computer architectures, with different instruction sets, memory models, etcetera, you need to emulate **the whole hardware**.
- So basically you have **two operating systems** one on top of the other.
- The upper OSs are called “operating system”, sometimes **guest OSs**.
- The lower one is called **hypervisor** — but it is an operating system nonetheless.

Hardware level emulation

- To run different computer architectures, with different instruction sets, memory models, etcetera, you need to emulate **the whole hardware**.
- So basically you have **two operating systems** one on top of the other.
- The upper OSs are called “operating system”, sometimes **guest OSs**.
- The lower one is called **hypervisor** — but it is an operating system nonetheless.

Hardware level emulation

- To run different computer architectures, with different instruction sets, memory models, etcetera, you need to emulate **the whole hardware**.
- So basically you have **two operating systems** one on top of the other.
- The upper OSs are called “operating system”, sometimes **guest OSs**.
- The lower one is called **hypervisor** — but it is an operating system nonetheless.

Hardware level emulation

- To run different computer architectures, with different instruction sets, memory models, etcetera, you need to emulate **the whole hardware**.
- So basically you have **two operating systems** one on top of the other.
- The upper OSs are called “operating system”, sometimes **guest OSs**.
- The lower one is called **hypervisor** — but it is an operating system nonetheless.

Emulation: hardware or software ?

- Emulating the hardware can be expensive: both for the hypervisor and the guest OS.
- Moreover, if the emulation is entirely done by software there is **always an overhead**.
- To speed things up, people started to tinker to add **hardware support to virtualization**.
- And they **spectacularly succeeded**.

Emulation: hardware or software ?

- Emulating the hardware can be expensive: both for the hypervisor and the guest OS.
- Moreover, if the emulation is entirely done by software there is **always an overhead**.
- To speed things up, people started to tinker to add **hardware support to virtualization**.
- And they **spectacularly succeeded**.

Emulation: hardware or software ?

- Emulating the hardware can be expensive: both for the hypervisor and the guest OS.
- Moreover, if the emulation is entirely done by software there is **always an overhead**.
- To speed things up, people started to tinker to add **hardware support to virtualization**.
- And they **spectacularly succeeded**.

Emulation: hardware or software ?

- Emulating the hardware can be expensive: both for the hypervisor and the guest OS.
- Moreover, if the emulation is entirely done by software there is **always an overhead**.
- To speed things up, people started to tinker to add **hardware support to virtualization**.
- And they **spectacularly succeeded**.

Emulation: hardware or software ?

- Emulating the hardware can be expensive: both for the hypervisor and the guest OS.
- Moreover, if the emulation is entirely done by software there is **always an overhead**.
- To speed things up, people started to tinker to add **hardware support to virtualization**.
- And they **spectacularly succeeded**.

Hardware-based virtualization

To overcome the overhead of running an OS inside another OS, the hardware was changed to accommodate for special instruction sets:

- The Intel VT-x and AMD AMD-V are **processor extensions to the x86 architecture** — they were released in years 2005-2006.
- Hardware-based (or hardware-assisted) x86 virtualization nowadays include:
  1. Linux KVM,
  2. Xen,
  3. VMware,
  4. Microsoft Hyper-V.

Hardware-based virtualization

To overcome the overhead of running an OS inside another OS, the hardware was changed to accommodate for special instruction sets:

- The Intel VT-x and AMD AMD-V are **processor extensions to the x86 architecture** — they were released in years 2005-2006.
- Hardware-based (or hardware-assisted) x86 virtualization nowadays include:
  1. Linux KVM,
  2. Xen,
  3. VMware,
  4. Microsoft Hyper-V.

Hardware-based virtualization

To overcome the overhead of running an OS inside another OS, the hardware was changed to accommodate for special instruction sets:

- The Intel VT-x and AMD AMD-V are **processor extensions to the x86 architecture** — they were released in years 2005-2006.
- Hardware-based (or hardware-assisted) x86 virtualization nowadays include:
  1. Linux KVM,
  2. Xen,
  3. VMware,
  4. Microsoft Hyper-V.

And the Cloud was built

- It is not an overstatement to say that the success of hardware-based virtualization and Linux support for it is demonstrated by the **mere existence of cloud technologies at all**.
- Cloud computing would have been **impossible** without hardware-based virtualization and **free software**.
- So, obviously, hardware-based virtualization must be the best thing that can happen to software development, since Fortran 77, right ?

And the Cloud was built

- It is not an overstatement to say that the success of hardware-based virtualization and Linux support for it is demonstrated by the **mere existence of cloud technologies at all**.
- Cloud computing would have been **impossible** without hardware-based virtualization and **free software**.
- So, obviously, hardware-based virtualization must be the best thing that can happen to software development, since Fortran 77, right ?

And the Cloud was built

- It is not an overstatement to say that the success of hardware-based virtualization and Linux support for it is demonstrated by the **mere existence of cloud technologies at all**.
- Cloud computing would have been **impossible** without hardware-based virtualization and **free software**.
- So, obviously, hardware-based virtualization must be the best thing that can happen to software development, since Fortran 77, right ?

And the Cloud was built

- It is not an overstatement to say that the success of hardware-based virtualization and Linux support for it is demonstrated by the **mere existence of cloud technologies at all**.
- Cloud computing would have been **impossible** without hardware-based virtualization and **free software**.
- So, obviously, hardware-based virtualization must be the best thing that can happen to software development, since Fortran 77, right ?

Index

Where do they come from ?

- History of 4.2BSD shows that Bill Joy on 18 March 1982 added a system call called **chroot**.
- From the man page: *chroot* - change root directory

Where do they come from ?

- History of 4.2BSD shows that Bill Joy on 18 March 1982 added a system call called **chroot**.
- From the man page: *chroot* - change root directory

Where do they come from ?

- History of 4.2BSD shows that Bill Joy on 18 March 1982 added a system call called **chroot**.
- From the man page: *chroot* - change root directory

Why ?

chroot (and its user space command) was created for the following reasons:

- to run untrusted code,
- to *cleanly* build code from scratch,
- to isolate environments.

Why ?

chroot (and its user space command) was created for the following reasons:

- to run untrusted code,
- to *cleanly* build code from scratch,
- to isolate environments.

Why ?

chroot (and its user space command) was created for the following reasons:

- to run untrusted code,
- to *cleanly* build code from scratch,
- to isolate environments.

Why ?

chroot (and its user space command) was created for the following reasons:

- to run untrusted code,
- to *cleanly* build code from scratch,
- to isolate environments.

Jails

> *In 1999 we added a partitioning facility to FreeBSD called jail(2). It reuses the chroot(2) implementation, but prevents well-documented means to escape chroot confinement. Jail offers semi-permeable partitioning of the file system, process, and networking namespaces, and removes all super-user privileges that would affect objects not entirely inside the jail.*

Building Systems to be Shared Securely, Poul-Henning Kamp, ACM Queue July/August 2014.

FreeBSD Jails

> *The FreeBSD "Jail" facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX "root" model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.*

Confining the Omnipotent Root, Poul-Henning Kamp and Robert Watson, Sane 2000.

Jails *are* for virtual machines

- "(. . .) to delegate management capabilities for each **virtual machine** environment."
- "Creating **virtual machines** in this manner (. . .)"
- "(. . .) has many potential uses; the most popular thus far has been for providing **virtual machine** services in Internet Service Provider environments."

It seems like year 2016 !

Jails *are* for virtual machines

- "(. . .) to delegate management capabilities for each **virtual machine** environment."
- "Creating **virtual machines** in this manner (. . .)"
- "(. . .) has many potential uses; the most popular thus far has been for providing **virtual machine** services in Internet Service Provider environments."

It seems like year 2016 !

Jails *are* for virtual machines

- "(. . .) to delegate management capabilities for each **virtual machine** environment."
- "Creating **virtual machines** in this manner (. . .)"
- "(. . .) has many potential uses; the most popular thus far has been for providing **virtual machine** services in Internet Service Provider environments."

It seems like year 2016 !

Jails *are* for virtual machines

- “(. . .) to delegate management capabilities for each **virtual machine** environment.”
- “Creating **virtual machines** in this manner (. . .)”
- “(. . .) has many potential uses; the most popular thus far has been for providing **virtual machine** services in Internet Service Provider environments.”

It seems like year 2016 !

Jails *are* for virtual machines

- “(. . .) to delegate management capabilities for each **virtual machine** environment.”
- “Creating **virtual machines** in this manner (. . .)”
- “(. . .) has many potential uses; the most popular thus far has been for providing **virtual machine** services in Internet Service Provider environments.”

It seems like year 2016 !

Zones

In year 2005, Sun Microsystems launched **Solaris Zones**:

> *Zones provide a means of virtualizing operating system services, allowing one or more processes to **run in isolation** from other activity on the system.*
>
> *This isolation **prevents processes** running within a given zone **from monitoring or affecting processes running in other zones**.*

Virtualization and Namespace Isolation in Solaris (PSARC/2002/174), September 7, 2006.

Zones (continued)

> *A zone is a "sandbox" within which one or more applications can run* ***without affecting or interacting with the rest*** *of the system.*
>
> *It also provides an abstraction layer that* ***separates applications from physical attributes*** *of the machine on which they are deployed, such as physical device paths and network interface names.*

Virtualization and Namespace Isolation in Solaris (PSARC/2002/174), September 7, 2006.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be "ported"; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be "ported"; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be "ported"; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be "ported"; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be "ported"; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Zones had just *one little limitation* . . .

- From the paper:

  > *Zones do not present a new API or ABI to which applications must be “ported”; instead, they provide the standard Solaris interfaces and application environment, with some restrictions.*

- So, you are basically **running Solaris** — which is good . . . if you are Sun Microsystem.
- But people wanted to **consolidate Microsoft Windows**.
- And **run Linux**, too.

Hardware-based virtualization took off, containers stagnated

- Being tied to the same OS of the underlying server meant that by year 2006 the containers adoption **hit a wall**.
- Some kind of support (*i.e.* Solaris' *branded zone*) allowed cross OS support, but without the simplicity of full machine virtualization.
- The market was dominated by Microsoft Windows machines, and Linux-based hypervisors became the *de facto* standard for running them.

Hardware-based virtualization took off, containers stagnated

- Being tied to the same OS of the underlying server meant that by year 2006 the containers adoption **hit a wall**.
- Some kind of support (*i.e.* Solaris' *branded zone*) allowed cross OS support, but without the simplicity of full machine virtualization.
- The market was dominated by Microsoft Windows machines, and Linux-based hypervisors became the *de facto* standard for running them.

Hardware-based virtualization took off, containers stagnated

- Being tied to the same OS of the underlying server meant that by year 2006 the containers adoption **hit a wall**.
- Some kind of support (*i.e.* Solaris' *branded zone*) allowed cross OS support, but without the simplicity of full machine virtualization.
- The market was dominated by Microsoft Windows machines, and Linux-based hypervisors became the *de facto* standard for running them.

Hardware-based virtualization took off, containers stagnated

- Being tied to the same OS of the underlying server meant that by year 2006 the containers adoption **hit a wall**.
- Some kind of support (*i.e.* Solaris' *branded zone*) allowed cross OS support, but without the simplicity of full machine virtualization.
- The market was dominated by Microsoft Windows machines, and Linux-based hypervisors became the *de facto* standard for running them.

In the meanwhile, in Linux-land . . .

Over time a set of technologies entered the kernel:

- Control groups (cgroup).
- Namespaces.
- Copy-on-write storage:
  - AUFS, overlay (file level),
  - device mapper thin provisioning (block level),
  - BTRFS, ZFS (FS level).

In the meanwhile, in Linux-land . . .

Over time a set of technologies entered the kernel:

- Control groups (cgroup).
- Namespaces.
- Copy-on-write storage:
  - AUFS, overlay (file level),
  - device mapper thin provisioning (block level),
  - BTRFS, ZFS (FS level).

In the meanwhile, in Linux-land . . .

Over time a set of technologies entered the kernel:

- Control groups (cgroup).
- Namespaces.
- Copy-on-write storage:
  - AUFS, overlay (file level),
  - device mapper thin provisioning (block level),
  - BTRFS, ZFS (FS level).

In the meanwhile, in Linux-land . . .

Over time a set of technologies entered the kernel:

- Control groups (cgroup).
- Namespaces.
- Copy-on-write storage:
  - AUFS, overlay (file level),
  - device mapper thin provisioning (block level),
  - BTRFS, ZFS (FS level).

Container technologies without containers

From a presentation by Jérôme Petazzoni of Docker's fame:

- *Even when you don't run containers . . .*
- *. . . you are in a container.*
- *Your host processes still execute in the root namespaces and cgroups*

Container technologies without containers

From a presentation by Jérôme Petazzoni of Docker's fame:

- *Even when you don't run containers . . .*
- *. . . you are in a container.*
- *Your host processes still execute in the root namespaces and cgroups*

Container technologies without containers

From a presentation by Jérôme Petazzoni of Docker's fame:

- *Even when you don't run containers . . .*
- *. . . you are in a container.*
- *Your host processes still execute in the root namespaces and cgroups*

Container technologies without containers

From a presentation by Jérôme Petazzoni of Docker's fame:

- *Even when you don't run containers . . .*
- *. . . you are in a container.*
- *Your host processes still execute in the root namespaces and cgroups*

Index

How do you run a container ?

- Running a KVM/QEMU virtual machine is matter of installing libvirtd and running `virsh start`
- How do you do the same, in *container land* ?
- In many different ways.

How do you run a container ?

- Running a KVM/QEMU virtual machine is matter of installing libvirtd and running `virsh start`
- How do you do the same, in *container land* ?
- In many different ways.

How do you run a container ?

- Running a KVM/QEMU virtual machine is matter of installing libvirtd and running `virsh start`
- How do you do the same, in *container land* ?
- In many different ways.

How do you run a container ?

- Running a KVM/QEMU virtual machine is matter of installing libvirtd and running `virsh start`
- How do you do the same, in *container land* ?
- In many different ways.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

Container run-times

Alternatives:

- LXC/LXD.
- `systemd-nspawn`
- The Docker engine.
- `rkt`
- `runC`
- probably something else, released this morning.

At the right time, in the right place

- In March 2013 a company called dotCloud launched a product to run containers in Linux. The product was aimed at **developers** and was meant to be easy.
- They succeeded **quite spectacularly**.
- The company was renamed as **Docker**.

> *This morning (cfr. 23 October 2013), we officially announced that dotCloud, Inc. is changing its name to Docker, Inc.*

At the right time, in the right place

- In March 2013 a company called dotCloud launched a product to run containers in Linux. The product was aimed at **developers** and was meant to be easy.
- They succeeded **quite spectacularly**.
- The company was renamed as **Docker**.

> *This morning (cfr. 23 October 2013), we officially announced that dotCloud, Inc. is changing its name to Docker, Inc.*

At the right time, in the right place

- In March 2013 a company called dotCloud launched a product to run containers in Linux. The product was aimed at **developers** and was meant to be easy.
- They succeeded **quite spectacularly**.
- The company was renamed as **Docker**.

> *This morning (cfr. 23 October 2013), we officially announced that dotCloud, Inc. is changing its name to Docker, Inc.*

At the right time, in the right place

- In March 2013 a company called dotCloud launched a product to run containers in Linux. The product was aimed at **developers** and was meant to be easy.
- They succeeded **quite spectacularly**.
- The company was renamed as **Docker**.

> *This morning (cfr. 23 October 2013), we officially announced that dotCloud, Inc. is changing its name to Docker, Inc.*

The sweet spot

- Docker offered a solution to a problem that was beginning to be very relevant . . .
- . . . the problem of *linking*.

The sweet spot

- Docker offered a solution to a problem that was beginning to be very relevant . . .
- . . . the problem of *linking*.

The sweet spot

- Docker offered a solution to a problem that was beginning to be very relevant . . .
- . . . the problem of *linking*.

A solved problem was no more

- **Linking code together** *per se* was (is) a solved problem — at least since the 70s' but . . .
- . . . it is the software landscape that has changed a lot: a modern (let's say web) project has **dozens** of different components, all of them talking each other over the network.
- The fault could be in part attributed by **HDD** (*Hipster Driven Development*) . . .
- . . . but reality is that there has been a huge driver towards **distributed software components**.

A solved problem was no more

- **Linking code together** *per se* was (is) a solved problem — at least since the 70s' but . . .
- . . . it is the software landscape that has changed a lot: a modern (let's say web) project has **dozens** of different components, all of them talking each other over the network.
- The fault could be in part attributed by **HDD** (*Hipster Driven Development*) . . .
- . . . but reality is that there has been a huge driver towards **distributed software components**.

A solved problem was no more

- **Linking code together** *per se* was (is) a solved problem — at least since the 70s' but . . .
- . . . it is the software landscape that has changed a lot: a modern (let's say web) project has **dozens** of different components, all of them talking each other over the network.
- The fault could be in part attributed by **HDD** (*Hipster Driven Development*) . . .
- . . . but reality is that there has been a huge driver towards **distributed software components**.

A solved problem was no more

- **Linking code together** *per se* was (is) a solved problem — at least since the 70s' but . . .
- . . . it is the software landscape that has changed a lot: a modern (let's say web) project has **dozens** of different components, all of them talking each other over the network.
- The fault could be in part attributed by **HDD** (*Hipster Driven Development*) . . .
- . . . but reality is that there has been a huge driver towards **distributed software components**.

A solved problem was no more

- **Linking code together** *per se* was (is) a solved problem — at least since the 70s' but . . .
- . . . it is the software landscape that has changed a lot: a modern (let's say web) project has **dozens** of different components, all of them talking each other over the network.
- The fault could be in part attributed by **HDD** (*Hipster Driven Development*) . . .
- . . . but reality is that there has been a huge driver towards **distributed software components**.

Distributed software components

- Too many many frameworks / languages / tools.
- Too many many many build and deployment tools, often within a single technology (JS, anyone ?).
- A recipe for snowflake servers which are essentialy un-upgradable.

The only design principle than we can apply is **separate concerns** and **define boundaries** (*i.e.* interfaces, for you old-schoolers).

Distributed software components

- Too many many frameworks / languages / tools.
- Too many many many build and deployment tools, often within a single technology (JS, anyone ?).
- A recipe for snowflake servers which are essentialy un-upgradable.

The only design principle than we can apply is **separate concerns** and **define boundaries** (*i.e.* interfaces, for you old-schoolers).

Distributed software components

- Too many many frameworks / languages / tools.
- Too many many many build and deployment tools, often within a single technology (JS, anyone ?).
- A recipe for snowflake servers which are essentialy un-upgradable.

The only design principle than we can apply is **separate concerns** and **define boundaries** (*i.e.* interfaces, for you old-schoolers).

Distributed software components

- Too many many frameworks / languages / tools.
- Too many many many build and deployment tools, often within a single technology (JS, anyone ?).
- A recipe for snowflake servers which are essentialy un-upgradable.

The only design principle than we can apply is **separate concerns** and **define boundaries** (*i.e.* interfaces, for you old-schoolers).

Distributed software components

- Too many many frameworks / languages / tools.
- Too many many many build and deployment tools, often within a single technology (JS, anyone ?).
- A recipe for snowflake servers which are essentialy un-upgradable.

The only design principle than we can apply is **separate concerns** and **define boundaries** (*i.e.* interfaces, for you old-schoolers).

Index

Where do we go from there

- Right now (April 2016) the ecosystem is complex, fragmented and always changing.
- Death by a thousand frameworks.
- But software systems will always be more and more distributed.
- Containers are here to stay.

Where do we go from there

- Right now (April 2016) the ecosystem is complex, fragmented and always changing.
- Death by a thousand frameworks.
- But software systems will always be more and more distributed.
- Containers are here to stay.

Where do we go from there

- Right now (April 2016) the ecosystem is complex, fragmented and always changing.
- Death by a thousand frameworks.
- But software systems will always be more and more distributed.
- Containers are here to stay.

Where do we go from there

- Right now (April 2016) the ecosystem is complex, fragmented and always changing.
- Death by a thousand frameworks.
- But software systems will always be more and more distributed.
- Containers are here to stay.

Where do we go from there

- Right now (April 2016) the ecosystem is complex, fragmented and always changing.
- Death by a thousand frameworks.
- But software systems will always be more and more distributed.
- Containers are here to stay.

Questions

Thanks !

Email m@biodec.com
Useless social @gaunilone

Thanks !

Email m@biodec.com

Useless social @gaunilone

Thanks !

Email m@biodec.com
Useless social @gaunilone

Thanks !

Email m@biodec.com
Useless social @gaunilone