Designing an OS for a Warehouse-scale Computer

Malte Schwarzkopf and Matthew P. Grosvenor

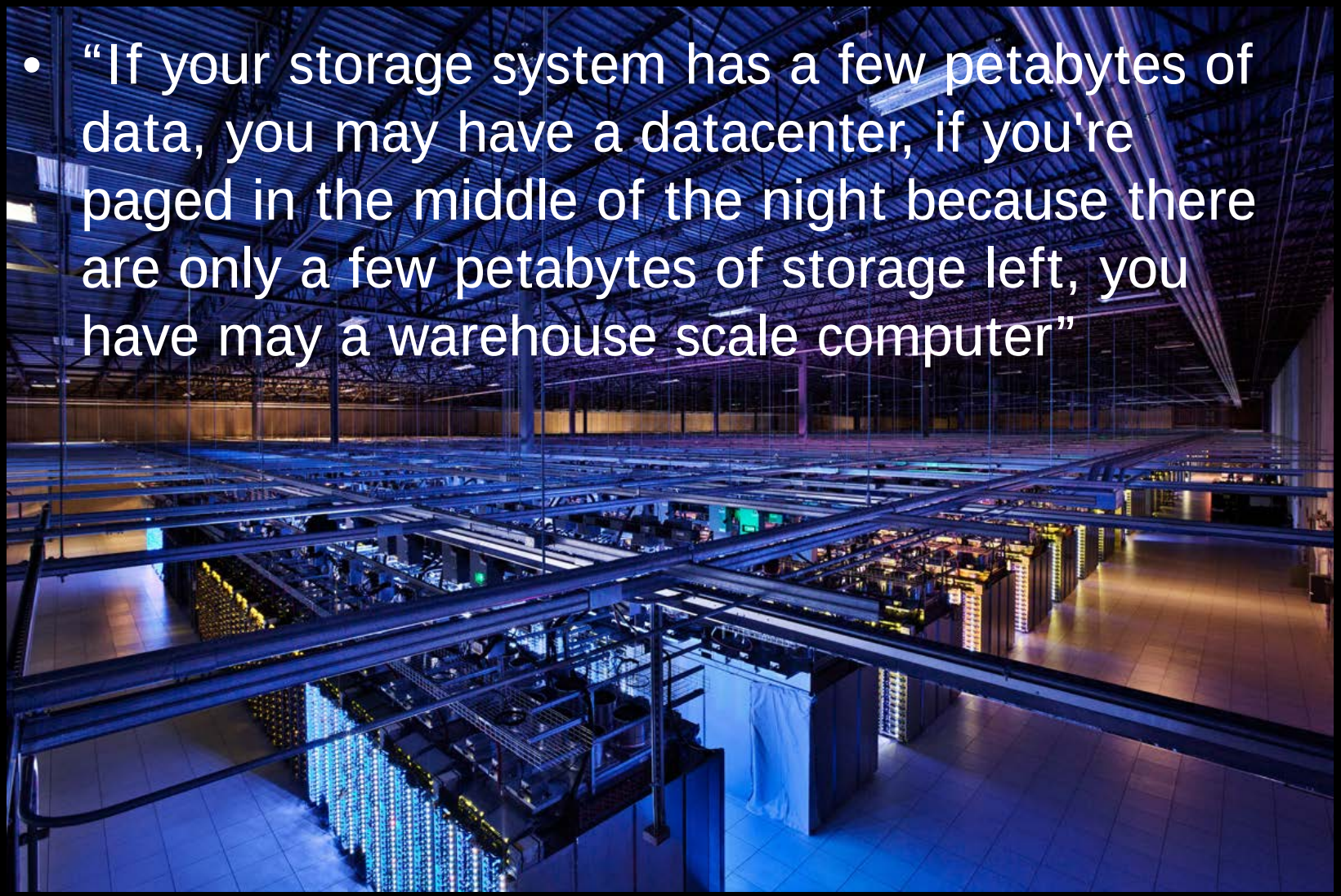


Image liberally borrowed from google.com

Warehouse Scale Computers... (The gory details)

- Current state of the art:
  - Commodity components on custom boards
  - Millions of cores, Petabytes of RAM
  - Exabytes of storage
  - Hosts run a modified Linux kernel
  - User space services provide programming API
    - e.g. BigTable, GFS, Spanner, MapReduce, AppEngine etc.
    - Almost exclusively distributed applications

Warehouse Scale Computers... (The gory details)

- Current state of the art:
  - Commodity components on custom boards
  - Millions of cores, Petabytes of RAM
  - Exabytes of storage
  - Hosts run a modified Linux kernel
  - User space services provide programming API
    - e.g. BigTable, GFS, Spanner, MapReduce, AppEngine etc.
    - Almost exclusively distributed applications

I think I’ve heard that before somewhere...

“A microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system. These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). Traditional operating system functions, such as device drivers, protocol stacks and file systems, are removed from the microkernel to run in user space.” - en.wikipedia.org/wiki/Microkernel

So, what do we actually need?

The µKernel gospel sayeth:

“A microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system. These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). Traditional operating system functions, such as device drivers, protocol stacks and file systems, are removed from the microkernel to run in user space.” - en.wikipedia.org/wiki/Microkernel

Manage Address Spaces

`as_create()` / `as_destroy()`

Run Processes

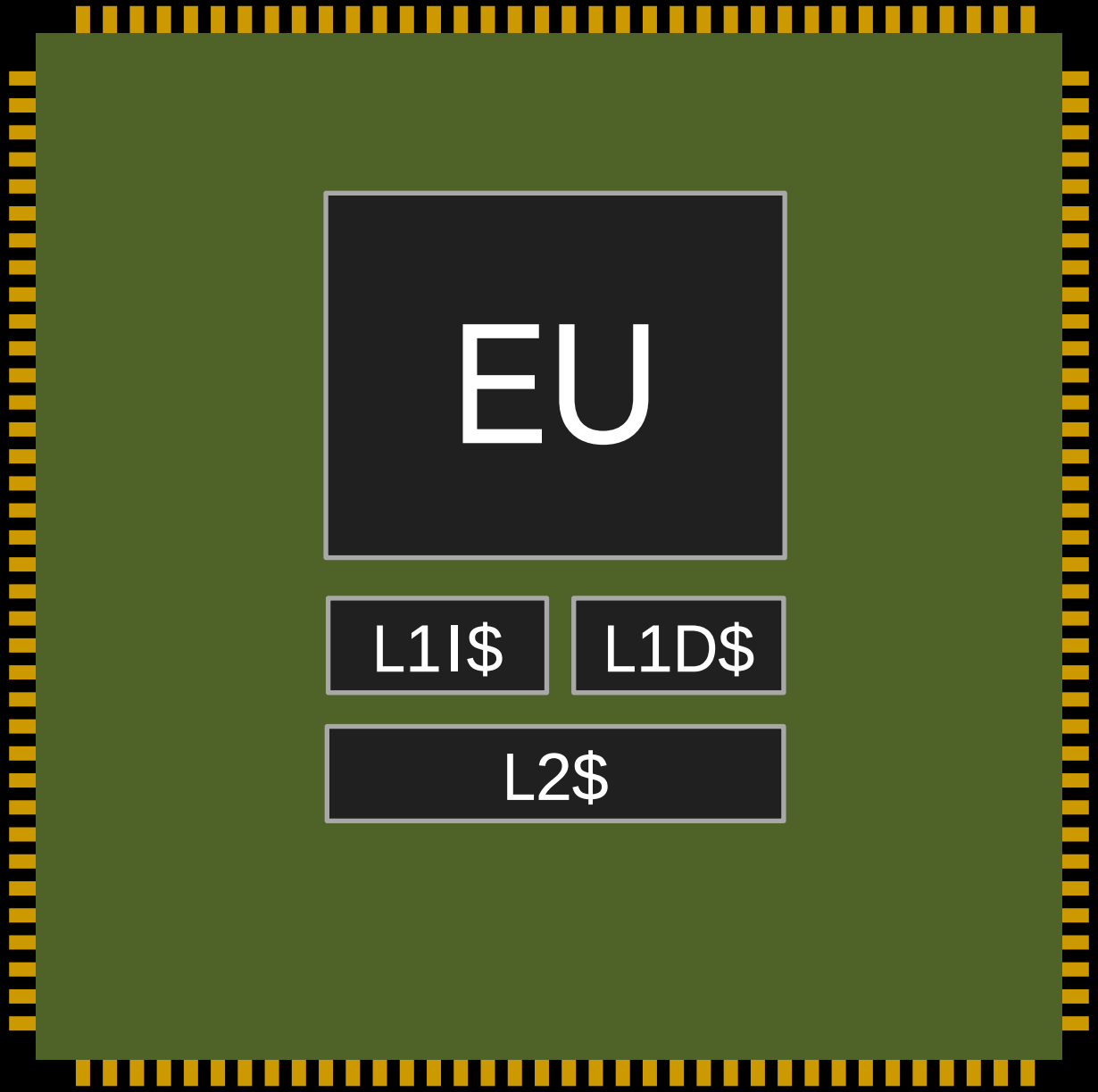
`proc_run()` / `proc_stop()`

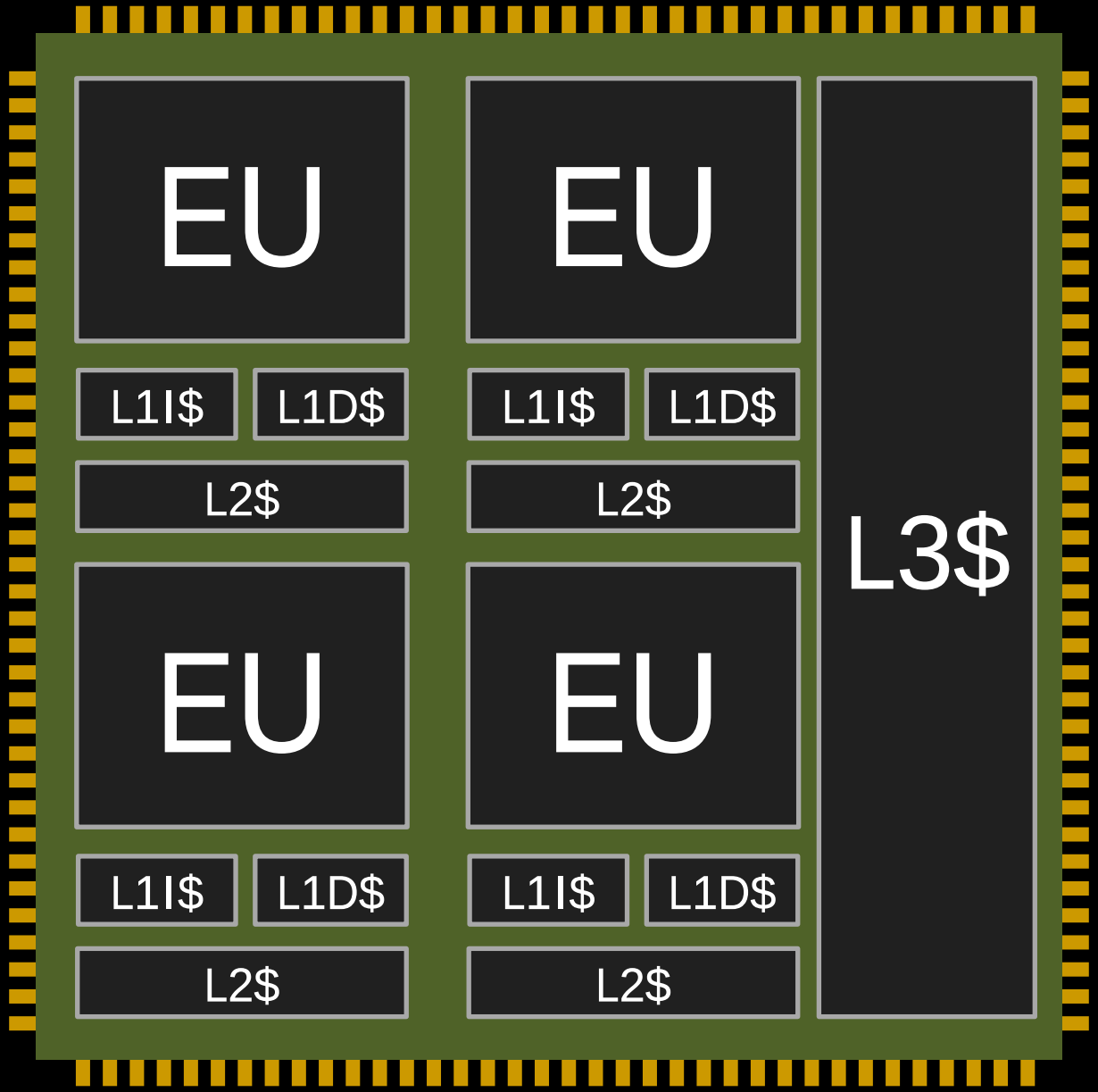
Manage IPC messages

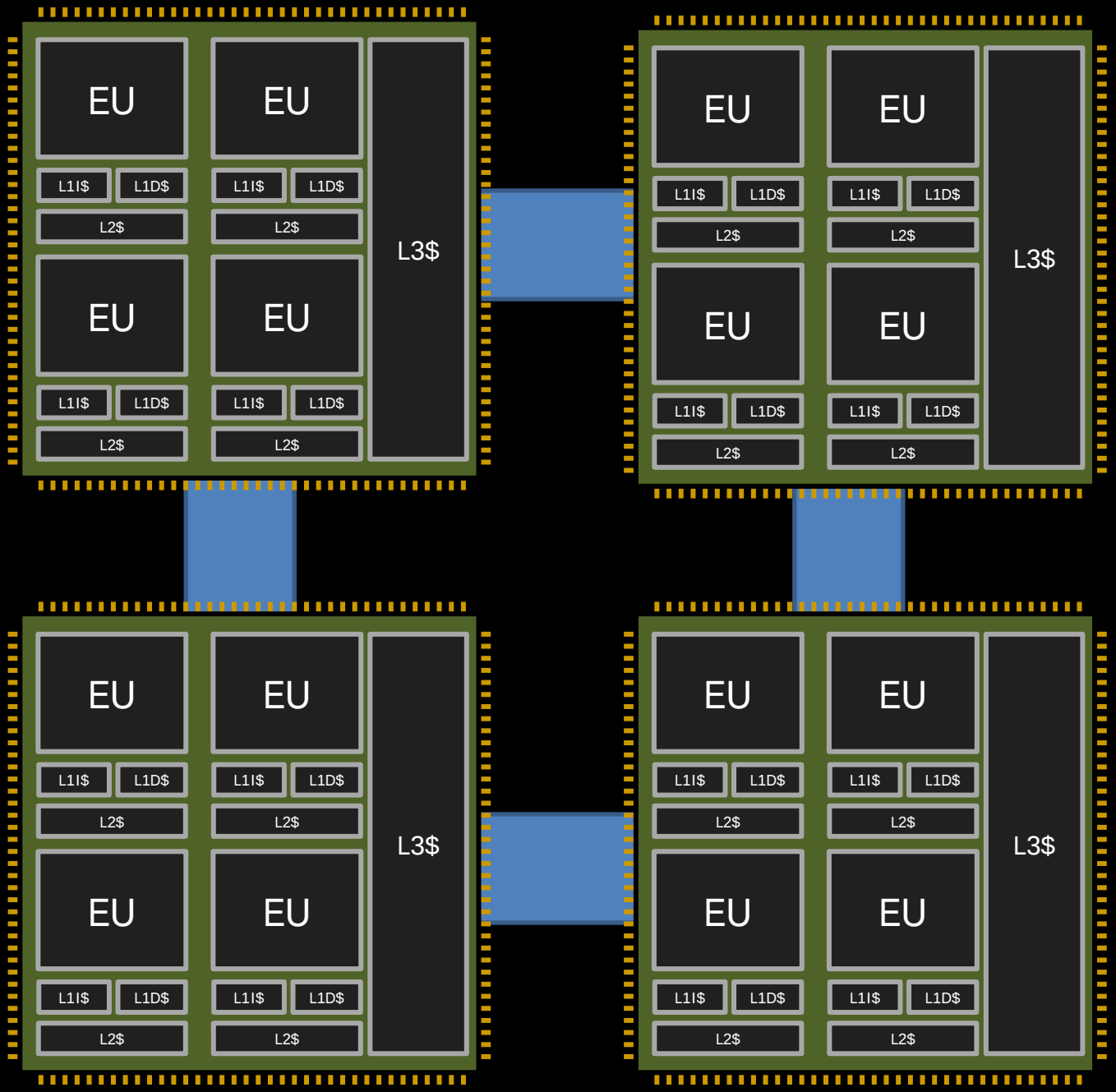
`send_ipc()` / `recv_ipc()`

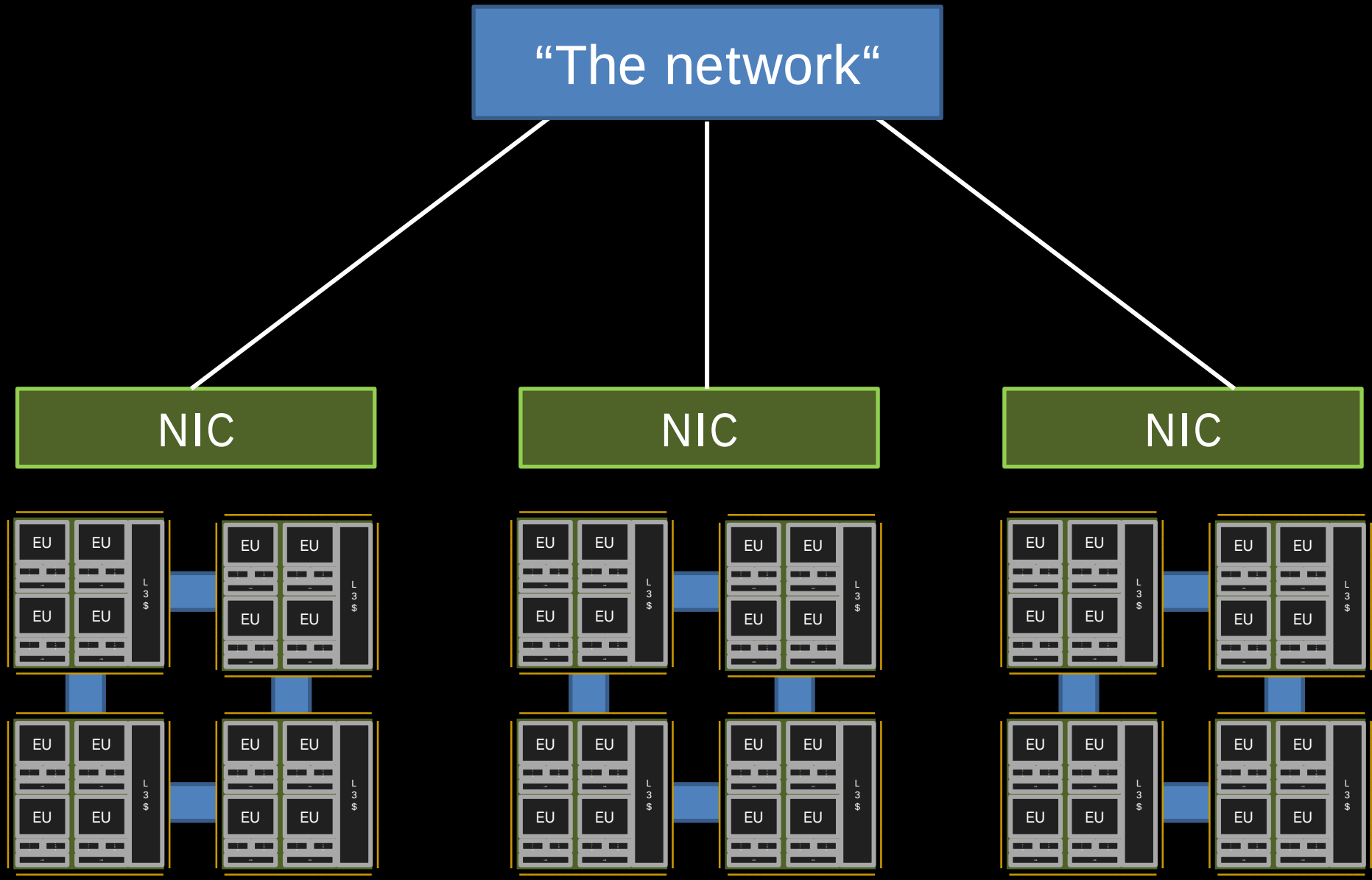
That’s probably not quite enough.

The anatomy of a WSC.


EU
L1I$
L1D$
L2$


EU
EU
L3$
L1I$
L1D$
L2$
L1I$
L1D$
L2$
EU
EU
L1I$
L1D$
L2$
L1I$
L1D$
L2$


EU
EU
L3$
L1I$
L1D$
L2$
EU
EU
L1I$
L1D$
L2$
EU
EU
L3$
L1I$
L1D$
L2$
EU
EU
L3$
L1I$
L1D$
L2$
EU
EU
L3$
L1I$
L1D$
L2$
EU
EU
L3$
L1I$
L1D$
L2$


“The network“
NIC
NIC
NIC
EU
L3$

Wait, are you saying we should have a distributed OS over all of this?!

YES.*)

*) With caveats

Manage Address Spaces

`as_create()` / `as_destroy()`

Run Processes

`proc_run()` / `proc_stop()`

Manage IPC messages

`send_ipc()` / `recv_ipc()`

Network stack

`send_net()` / `recv_net()`

Disk I/O

`read_disk()` / `write_disk()`

Manage Address Spaces

`as_create()` / `as_destroy()`

Run Processes

`proc_run()` / `proc_stop()`

Manage IPC messages

`recv_ipc()` / `send_ipc()`

Network stack

`recv_net()` / `send_net()`

Disk I/O

`read_disk()` / `write_disk()`

READ **WRITE**

But, but, but...

- Reads/writes may not be local – where is my data?
  - IPC to a process on box, across DC, [in another DC?]
  - Disk read on box, across DC, RDMA etc.

- Data consistency is an issue – how do we get it?
  - Replication – where should I get it from?
  - Updates – which one is most recent?
  - Updates – what if an update happens while I'm working?
  - Failure – what if a the node holding data fails?

1. Where is the data?

- All objects in the system (including processes) have a UUID
- Global name service, with local cache (sounds like a TLB)
- If an object name is not in the cache, send a message to figure out where it is (ARP-like).
  [N.B. Object may be in many places]
- Send invalidation message on write to object

- How long do you wait for a response??

1.5. References

- Locally valid identifiers for global names
  - Data objects
  - Processes
  - Device buffers (net/disk)
  - …
- Forged by kernel (≈ capabilities)
- Slim in required, but rich in potential information

```
ref 0x66a67f7e3 {
        max write size:  4K;
        durable: false;
        proto: binary;
        […]
}
```

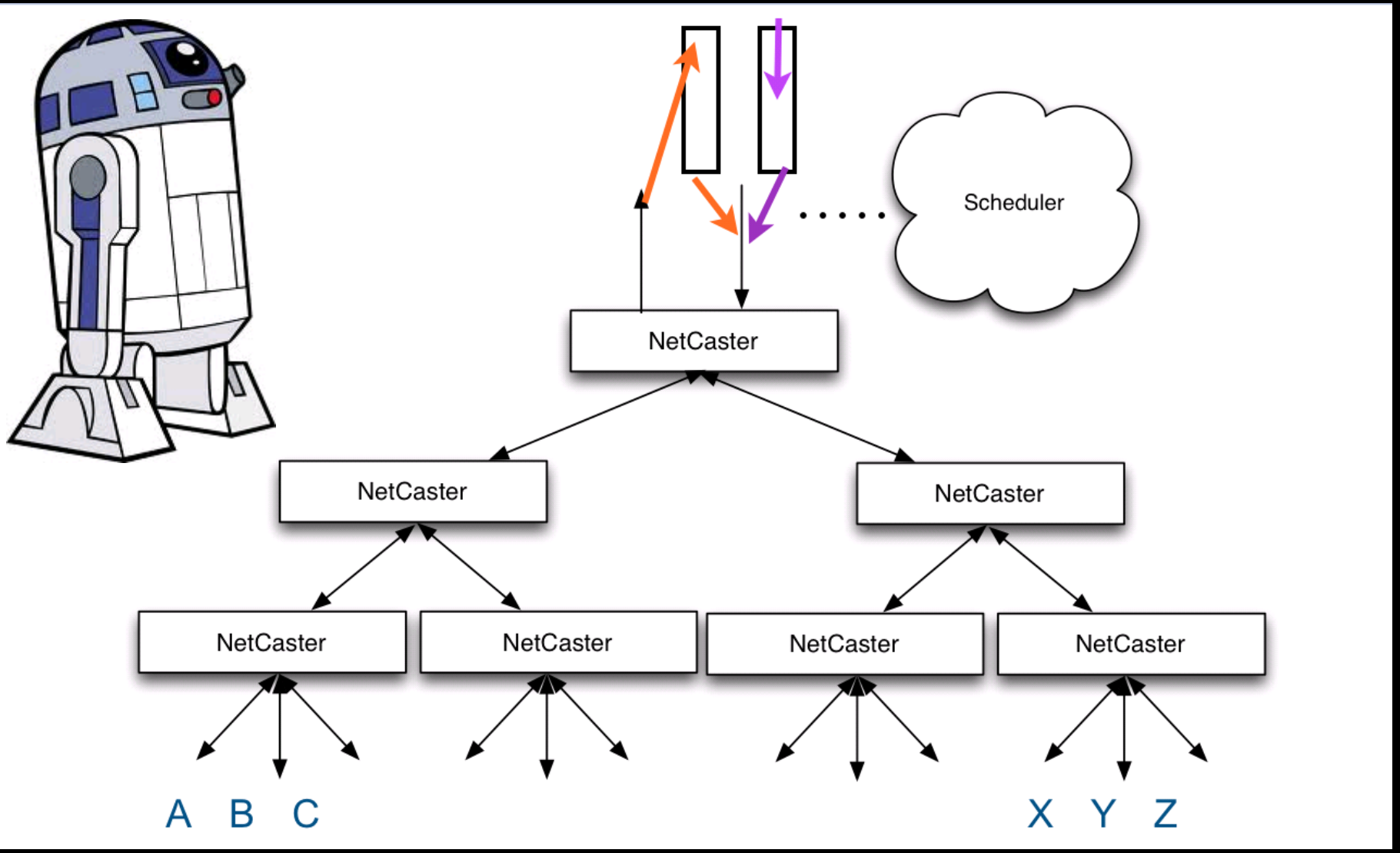
2. Data consistency

- Update the API
- `begin_write()` – take a pointer to the data
- `end_write()` – notify hosts that data has changed (i.e. cache invalidation message)
- `begin_read()` – take a copy of the data to work with
- `end_read()` – has the data changed since I started? (Do I care?)

- How long do you wait?

If only we had a reliable network layer with bounded latency...

But hang on, we do!



Now we just have another cache-coherent layer with a large invalidation cost!

The WSC syscall interface.

- <ptr, size> begin_read(ref)
  bool end_read(ref)

- <ptr> begin_write(ref, size)
  bool end_write(ref)

- bool proc_run(ref)
  bool proc_pause(ref)

- ref create(name)
  ref lookup(name)
  bool delete(ref)

- select([ref])

What’s the goodness?

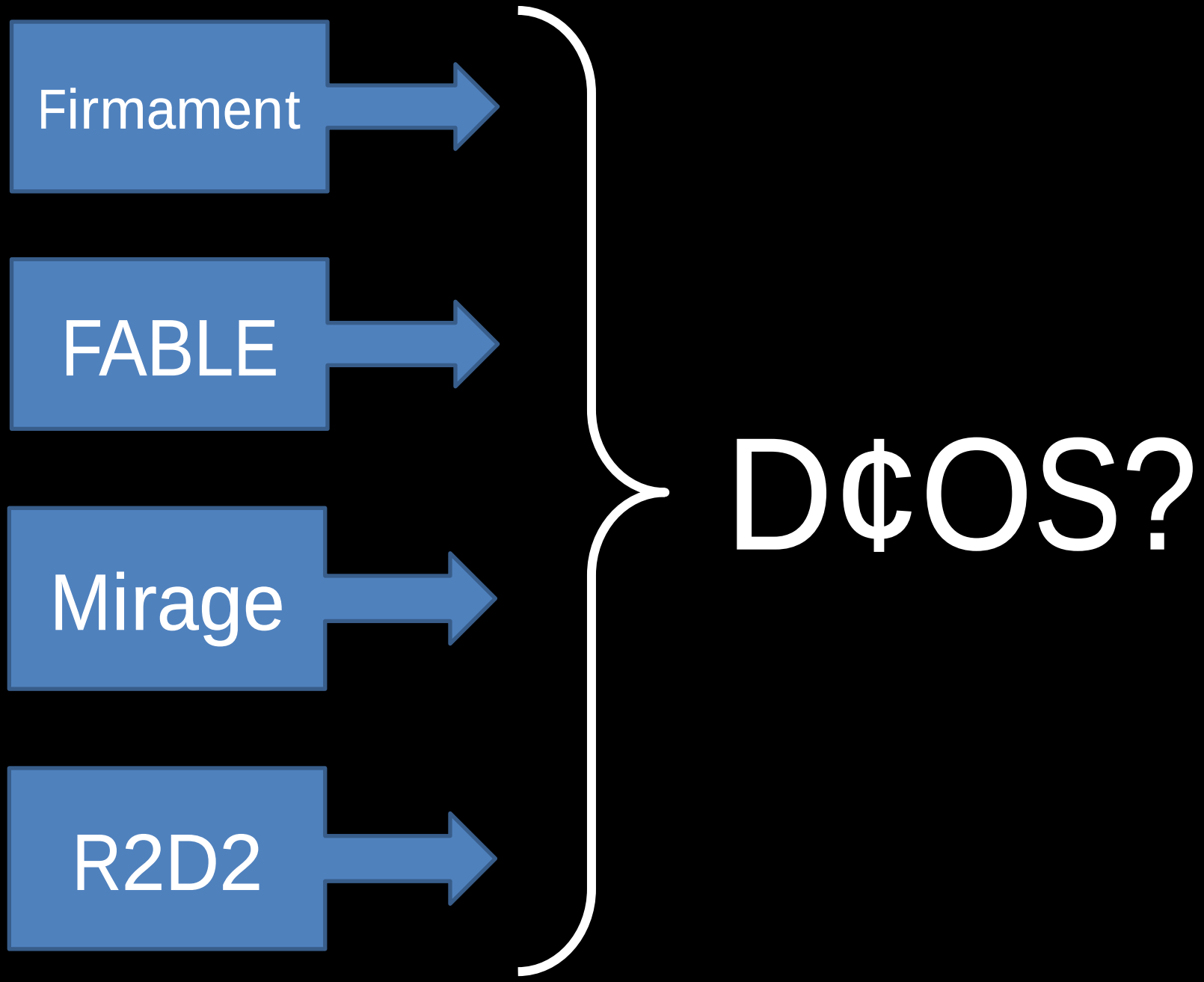
Simple API: “Declarative” programs

- Calls all take a `flags` argument
  - `O_PERSISTENT` / `O_TEMPORARY`
  - `O_REPLICATE=4`
  - `O_WEAKLY_CONSISTENT`
  - …
- Forces the programmer to reason about the properties of their distributed application and its objects!

Reduced complexity

- Easier to debug and reason about
- Can plug in “simulation” layers at syscall interface (to e.g. simulate failures or high bounded latency)
- “Default refs” at `proc_run` can supply useful primitives
  - e.g. timer, network wakeup ref (integrated with R2D2 scheduler)

How can we build this?


Firmament
FABLE
Mirage
R2D2
D¢OS?