Let it Crash!

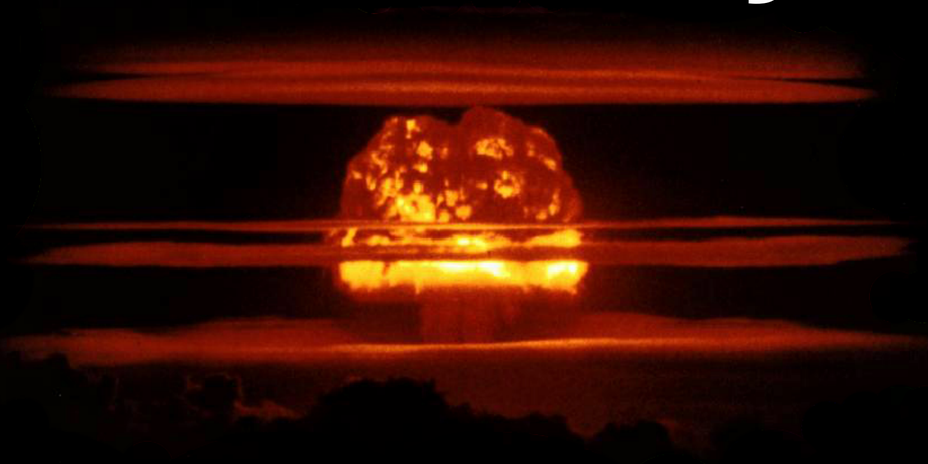
The Erlang Approach to Building Reliable Services

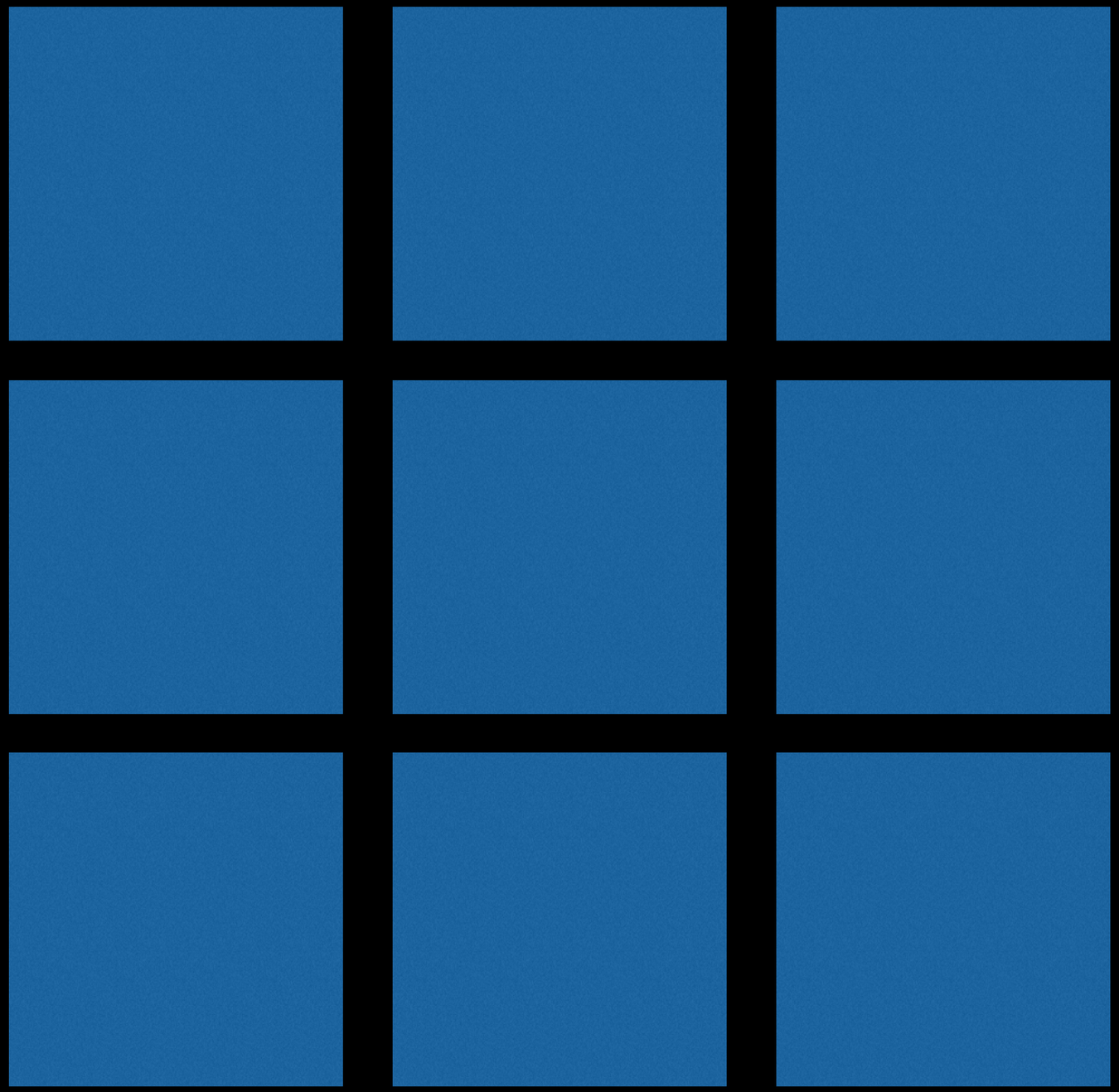
Hi, folks.

This talk is not about convincing Facebook to pay you 19$ billion.

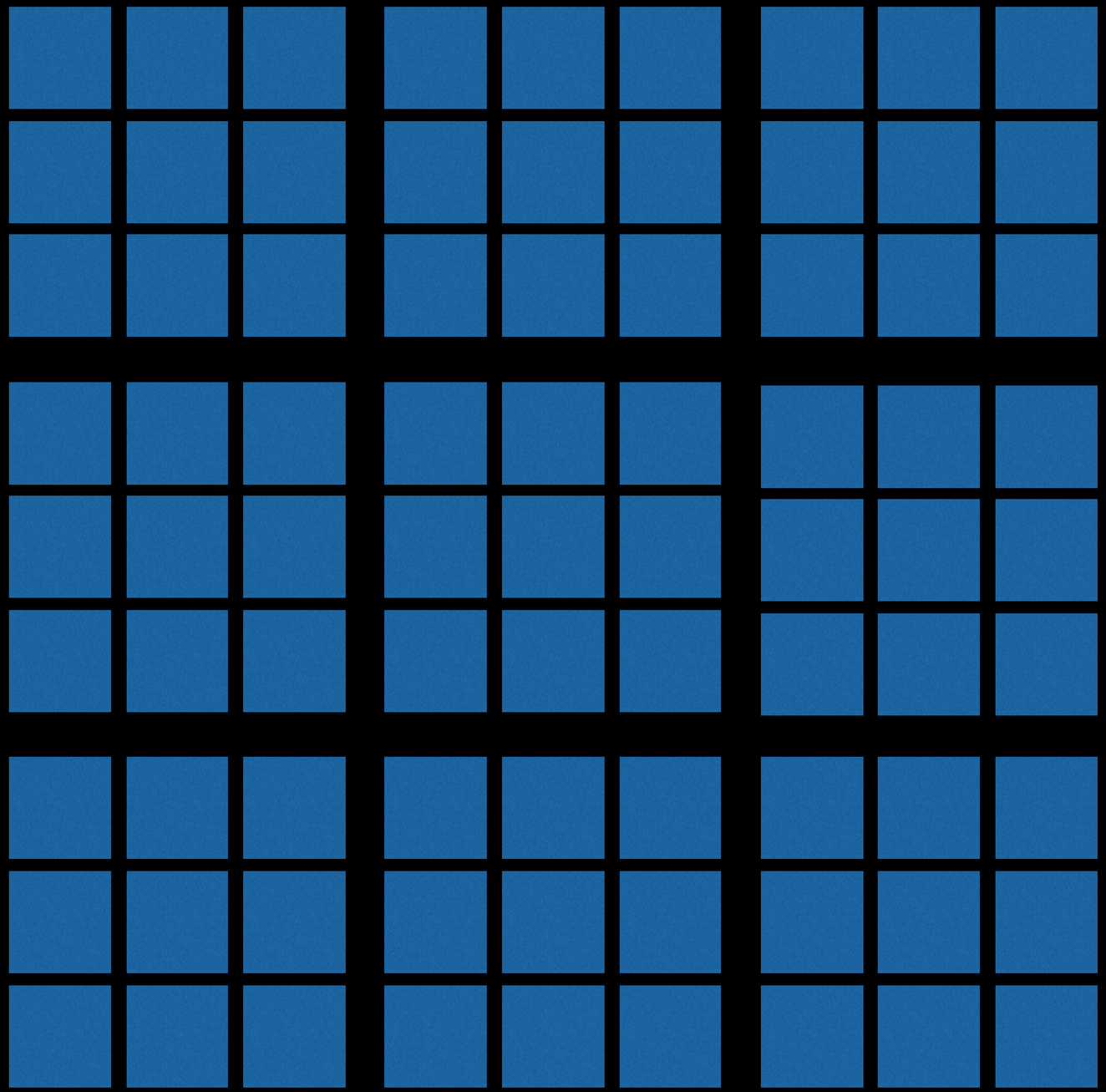
This is a talk about doing good, careful work...

...on systems that might kill people or ruin companies, which is totally rad.

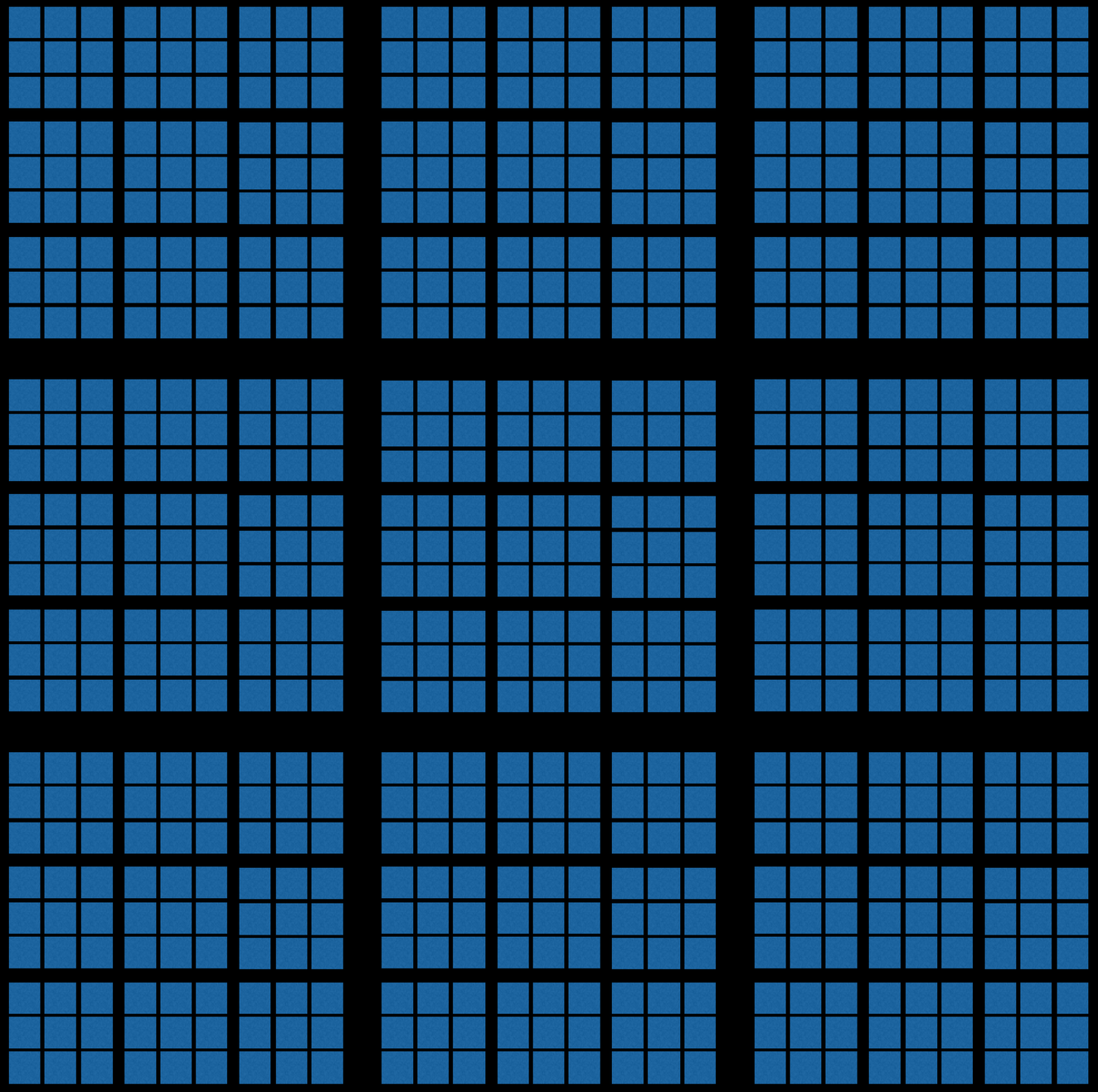




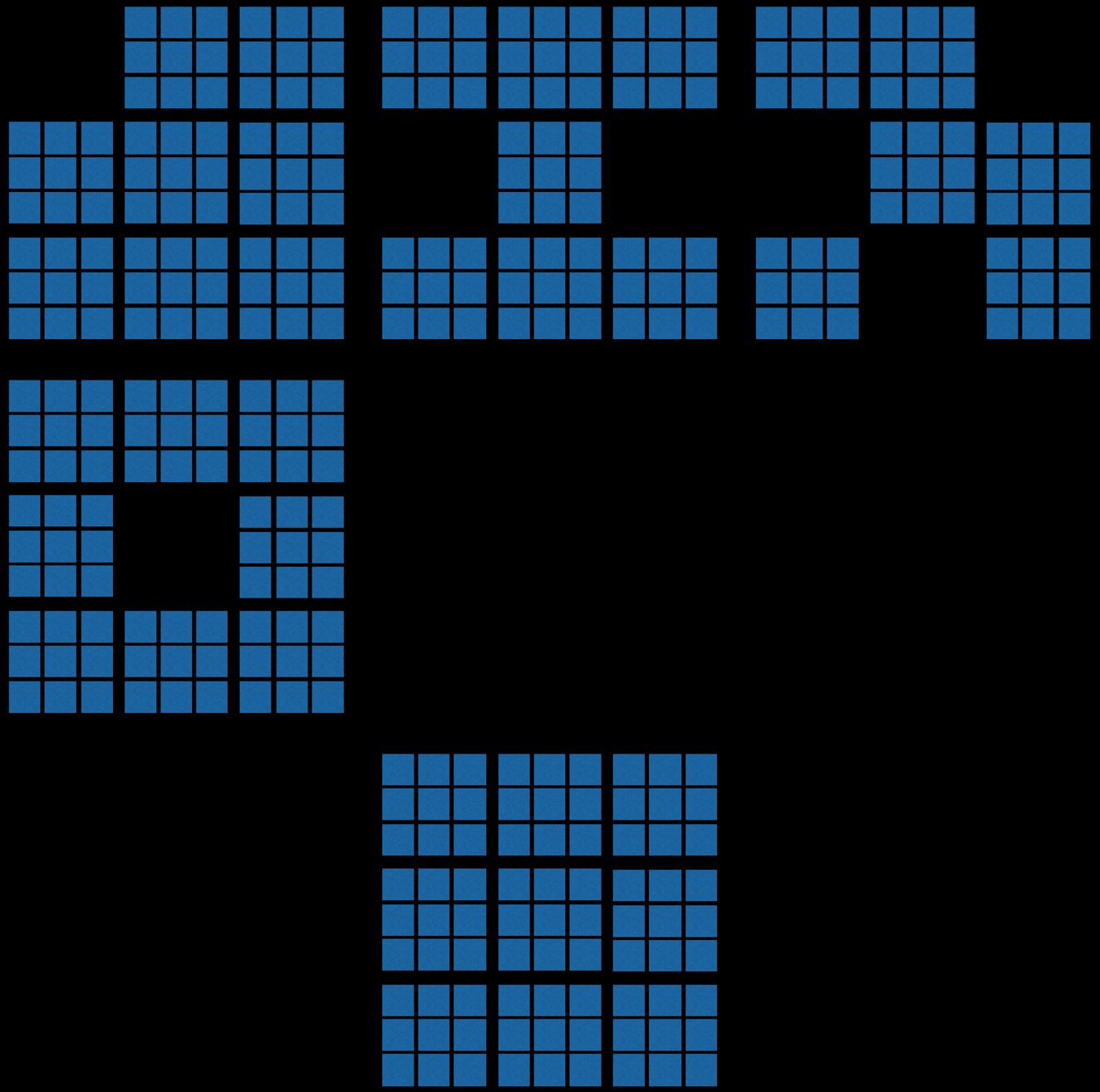
many processors



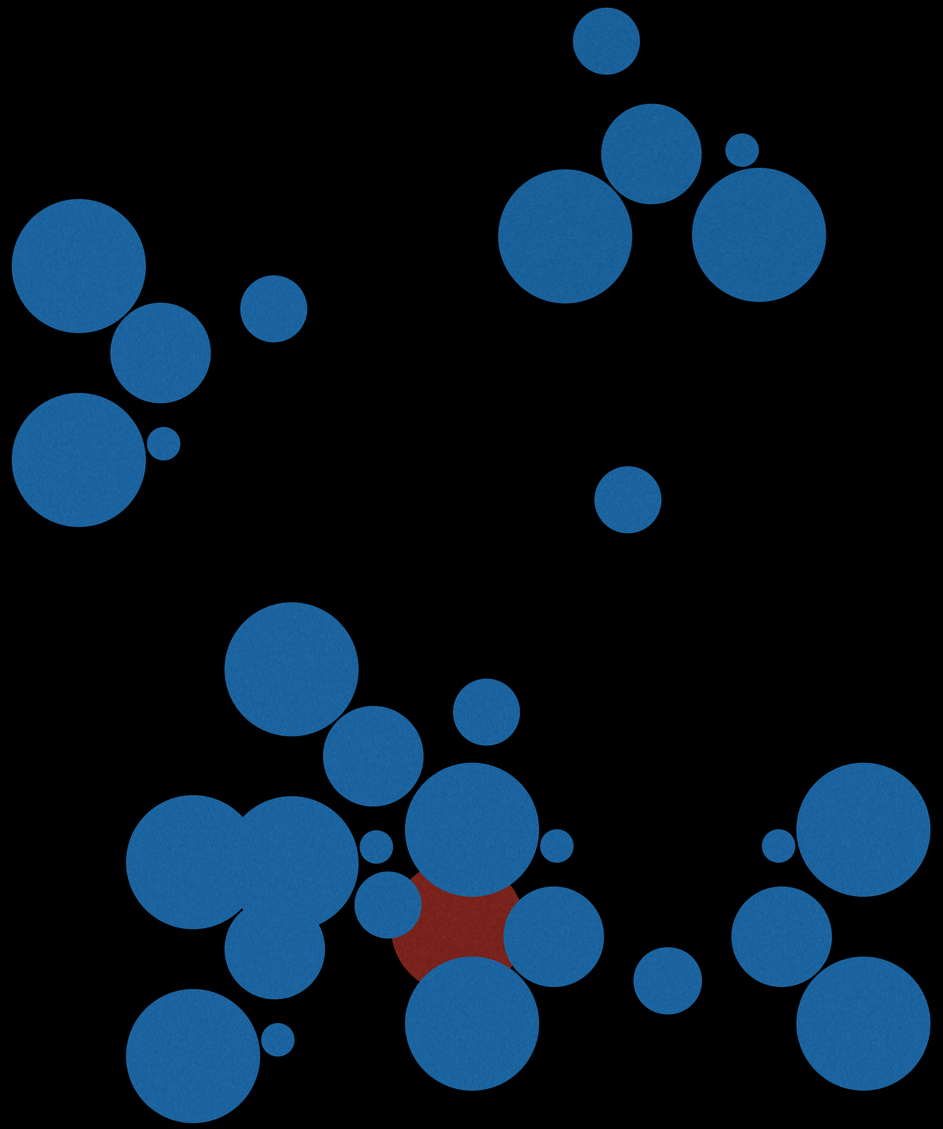
in a network



inside an even bigger network

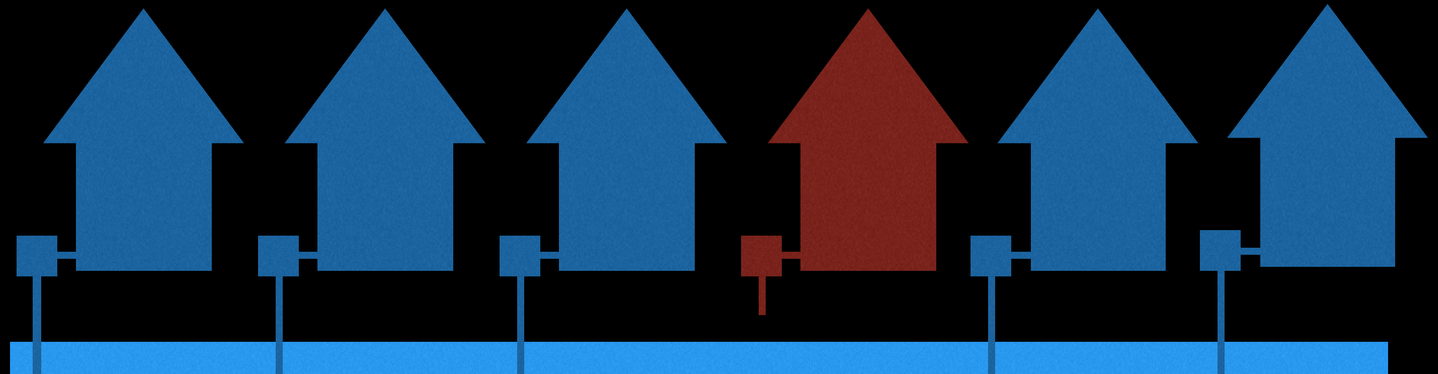


and none of it’s reliable

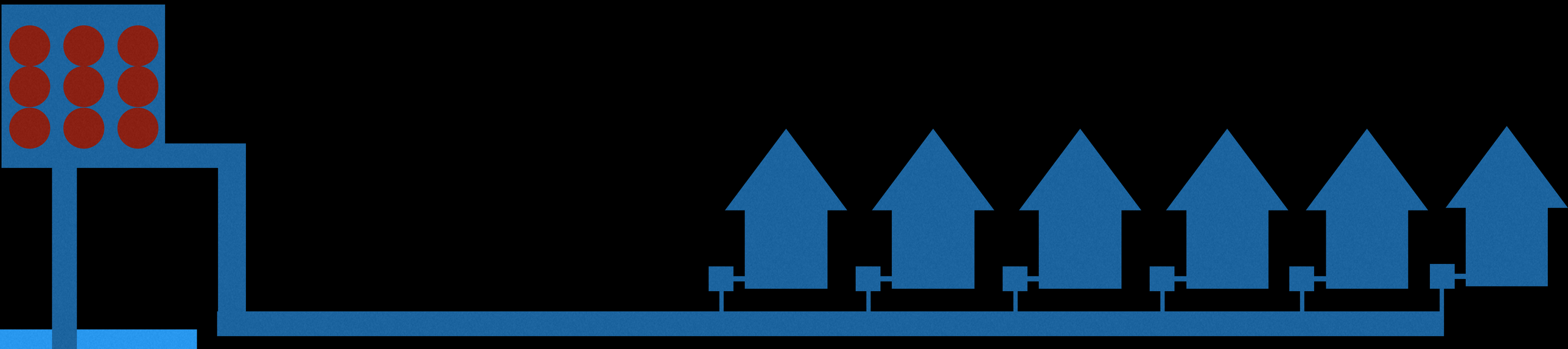


and also it’s spread across the globe

Sometimes we don’t care to make the illusion of whole-system reliability.



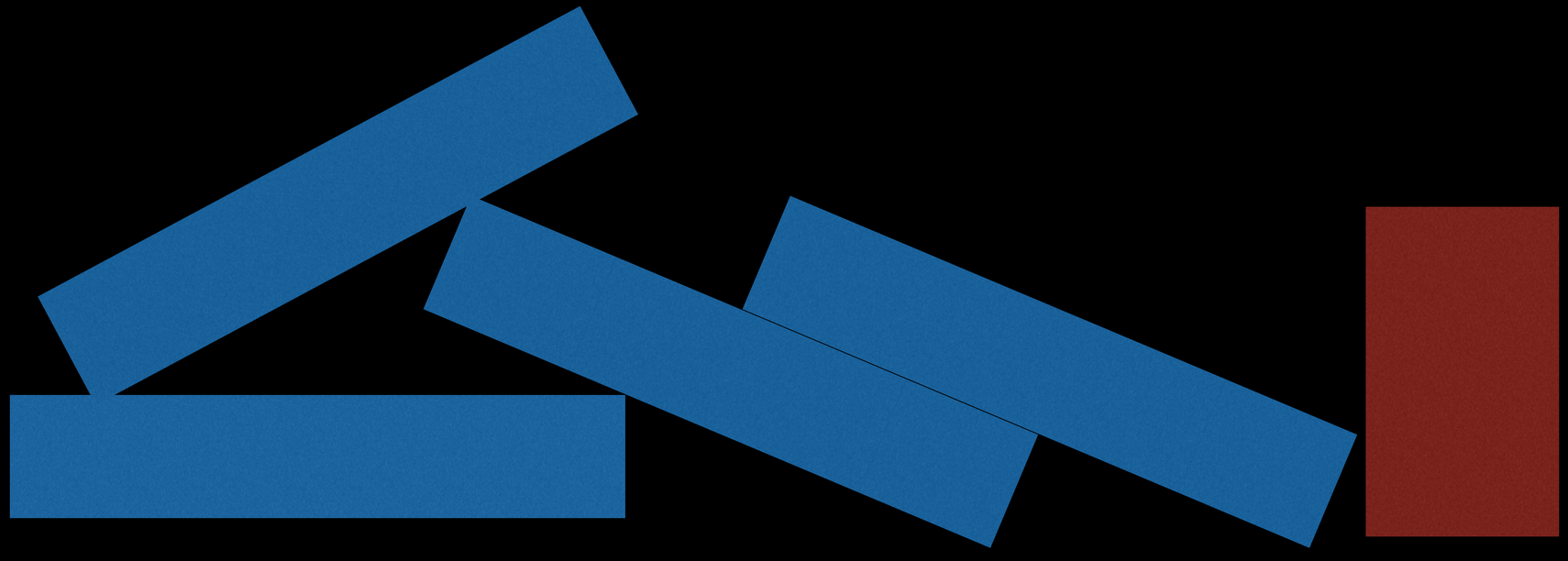
Sometimes we do.



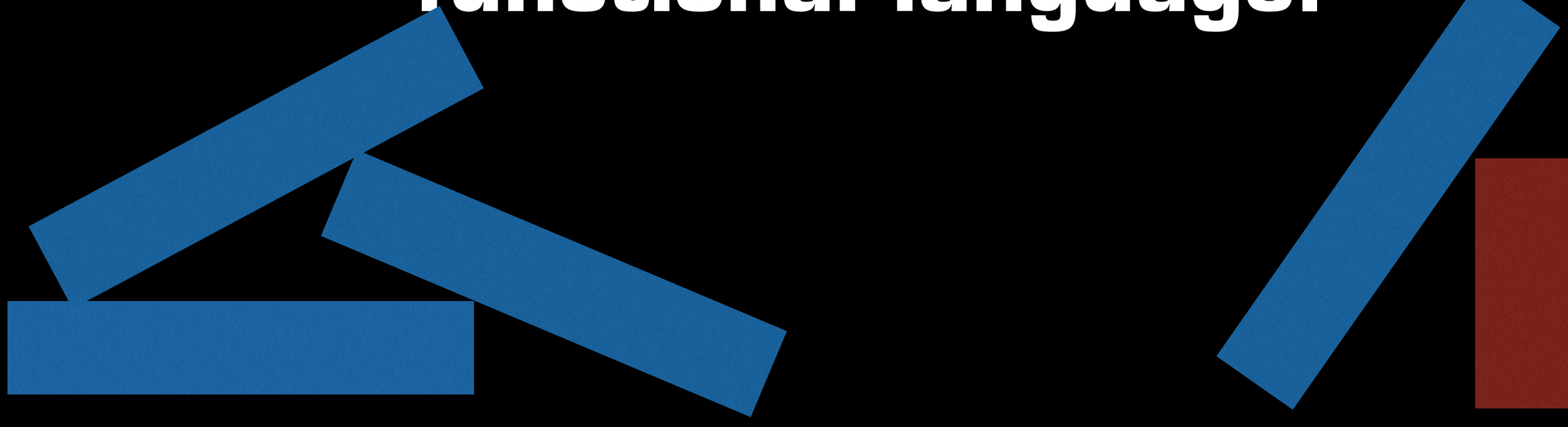
In this talk, we do.


Erlang

A handful simple pieces.

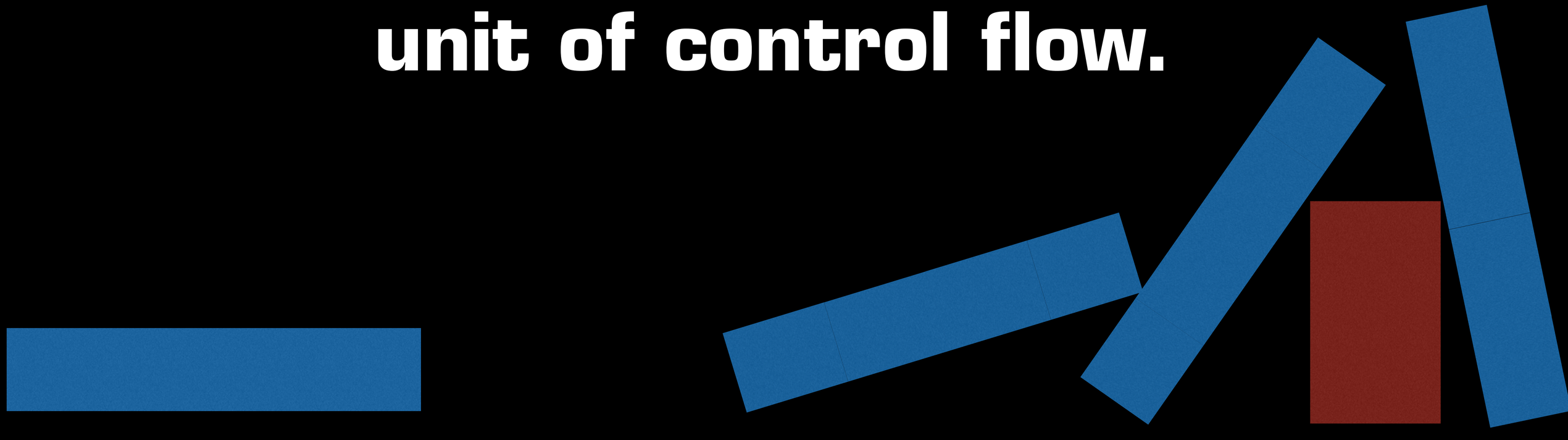


An old-fashioned, explicit functional language.



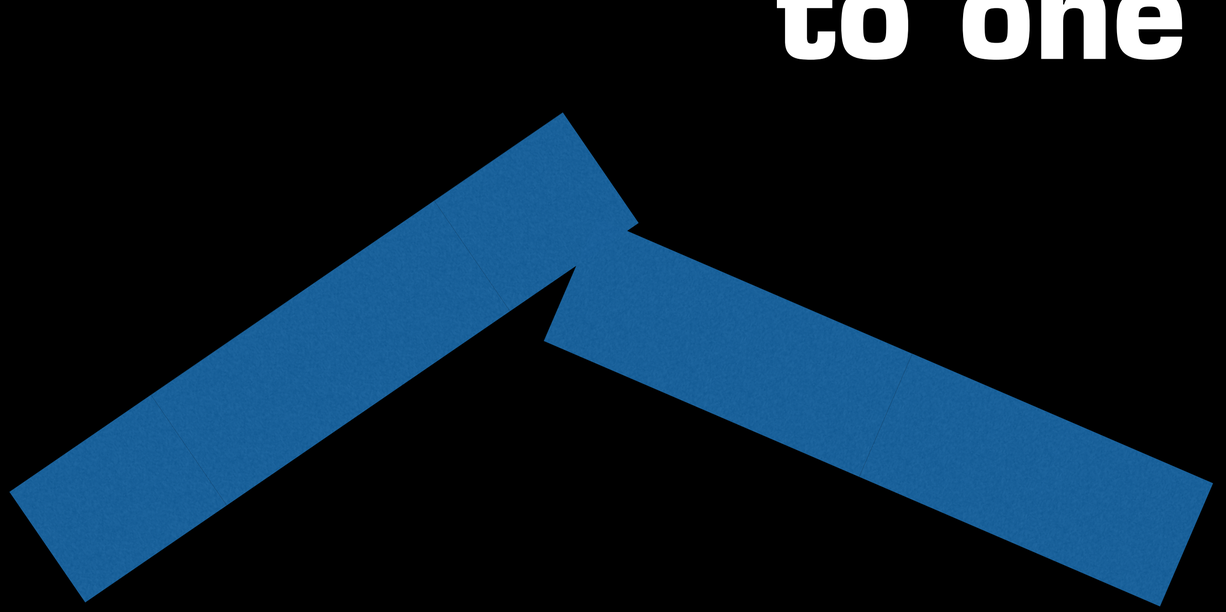
All data is immutable, without exception.

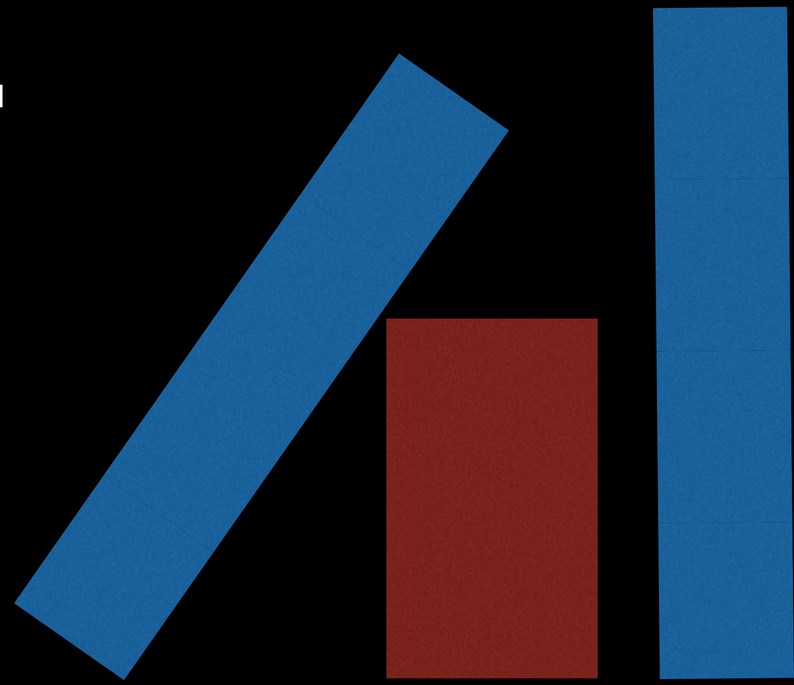
Processes are the smallest unit of control flow.



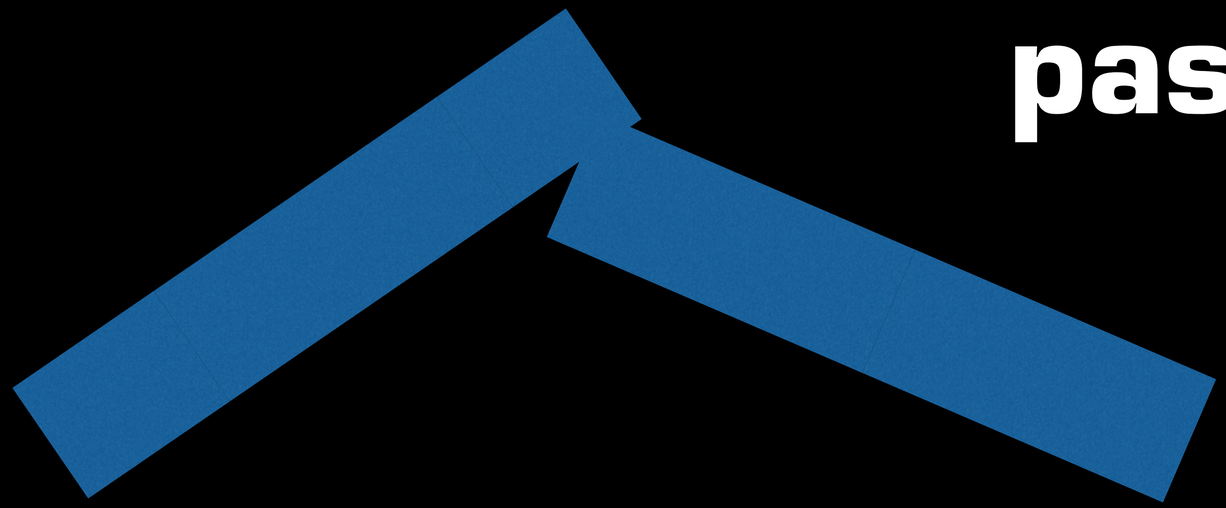
Processes are sequential internally.

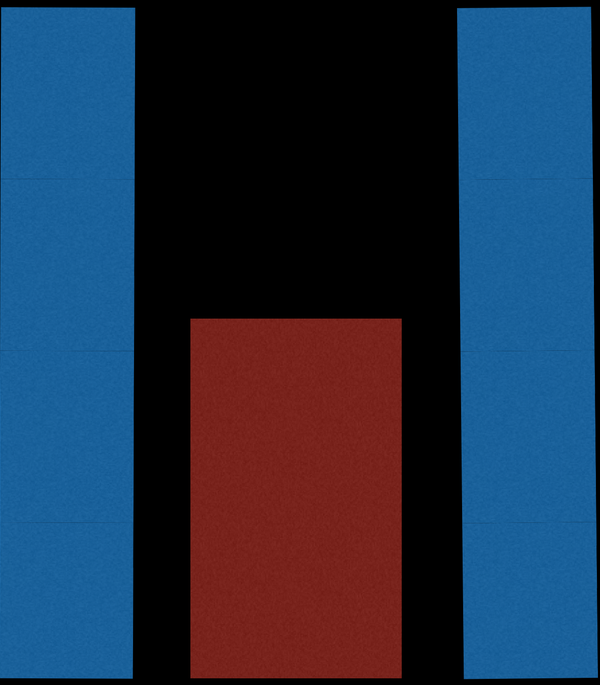
Processes are concurrent
to one another.



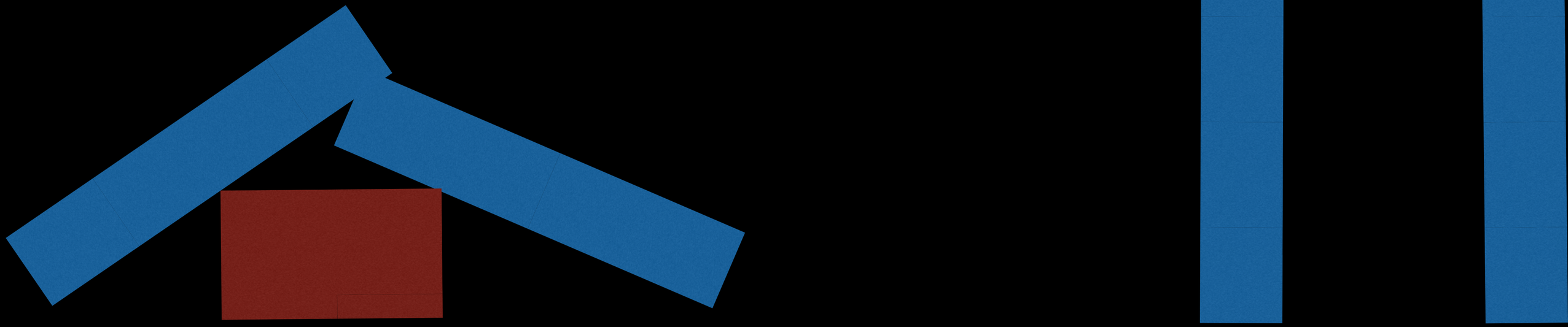


Processes can communicate only through message passing.

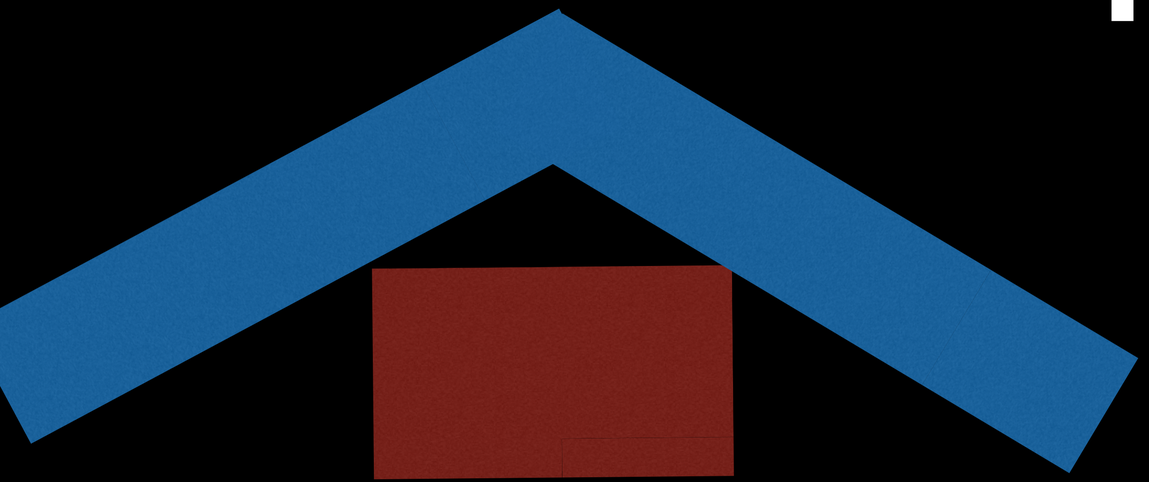


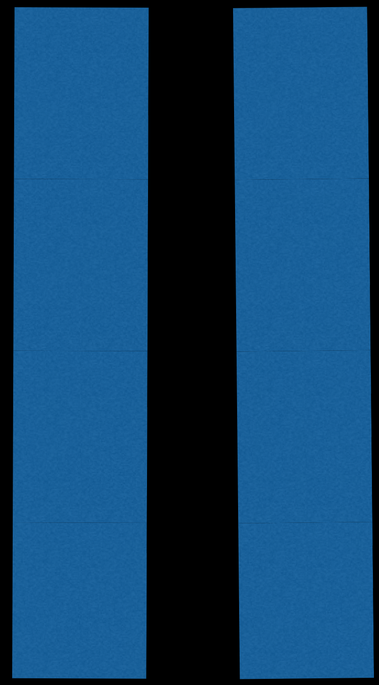


Messages are copied between processes.



Processes can “link” and receive messages about linked pair deaths.





This is called
“trapping exits”.

OR

Processes can “link” and die when linked pairs die.

This is the
default
behaviour.

That’s basically it.

The implications are really fun.

Supervision

Well known process traps exits, restart other processes that fail.

The failed process is restarted from a static specification.

The failed process state is not preserved.

The failed process state is not preserved.

(intentionally)

Registered Names

These are decoupled from the underlying process.

You can change processes without consumers getting wise to it.

Protocols

You can’t manipulate another process’ state.

You have to convince it to change itself.

Network Transparency

Network Transparency

(kinda)

Messages may be addressed to explicit pid, process name. Node may be specified, too.

Erlang hides the network details, leaving just the abstraction.

Hot Code Reloading

Code updates can be inserted at runtime.

New functions and new processes get inserted under old names.

Tricky without VM support.

This is all driving to one end:

Fault Tolerance

Everything in Erlang (and everything not) is in the service of building fault-tolerant systems.

The short-hand for this is:


Let it crash!

Faulty subsystems are difficult to correct.

Don’t bother.
Just restart.


What's unique
to Erlang?

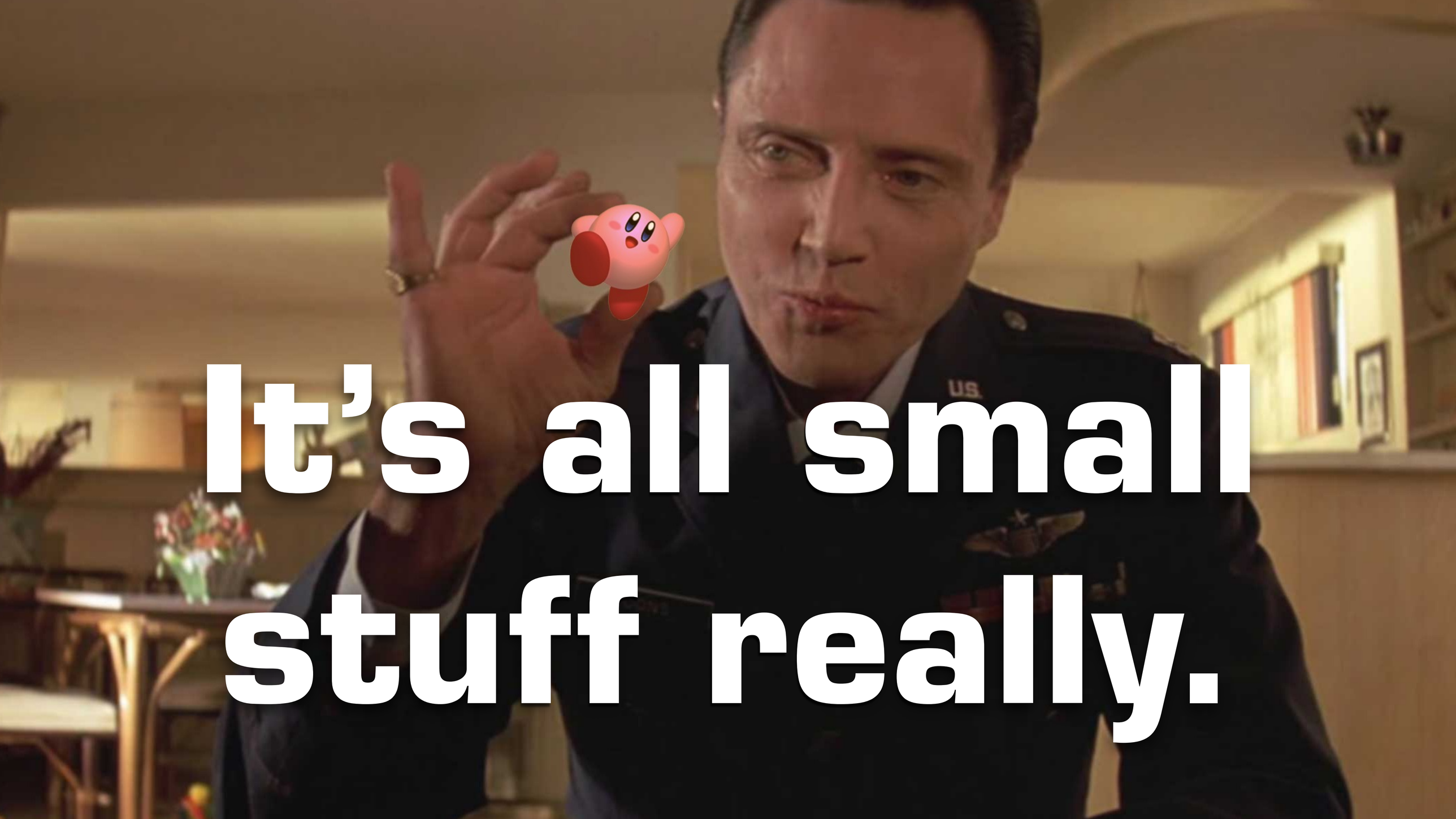
VM support for swapping function implementations.

Cheap processes.

Cheap processes.

(309 words of memory)

Strict process isolation.


It’s all small
stuff really.

What can you do now?

Short of using Erlang, that is...

Immutable Data

This means embracing copying.

Rub some Queue on it.

Queue

+ Async Execution

= Erlang Process

Structure your system as loosely coupled sub-systems.

Sub-systems communicate through a well-known protocol.

They may be co-resident in memory or across machine boundaries.

Decoupling brings many secondary advantages.

Isolate.

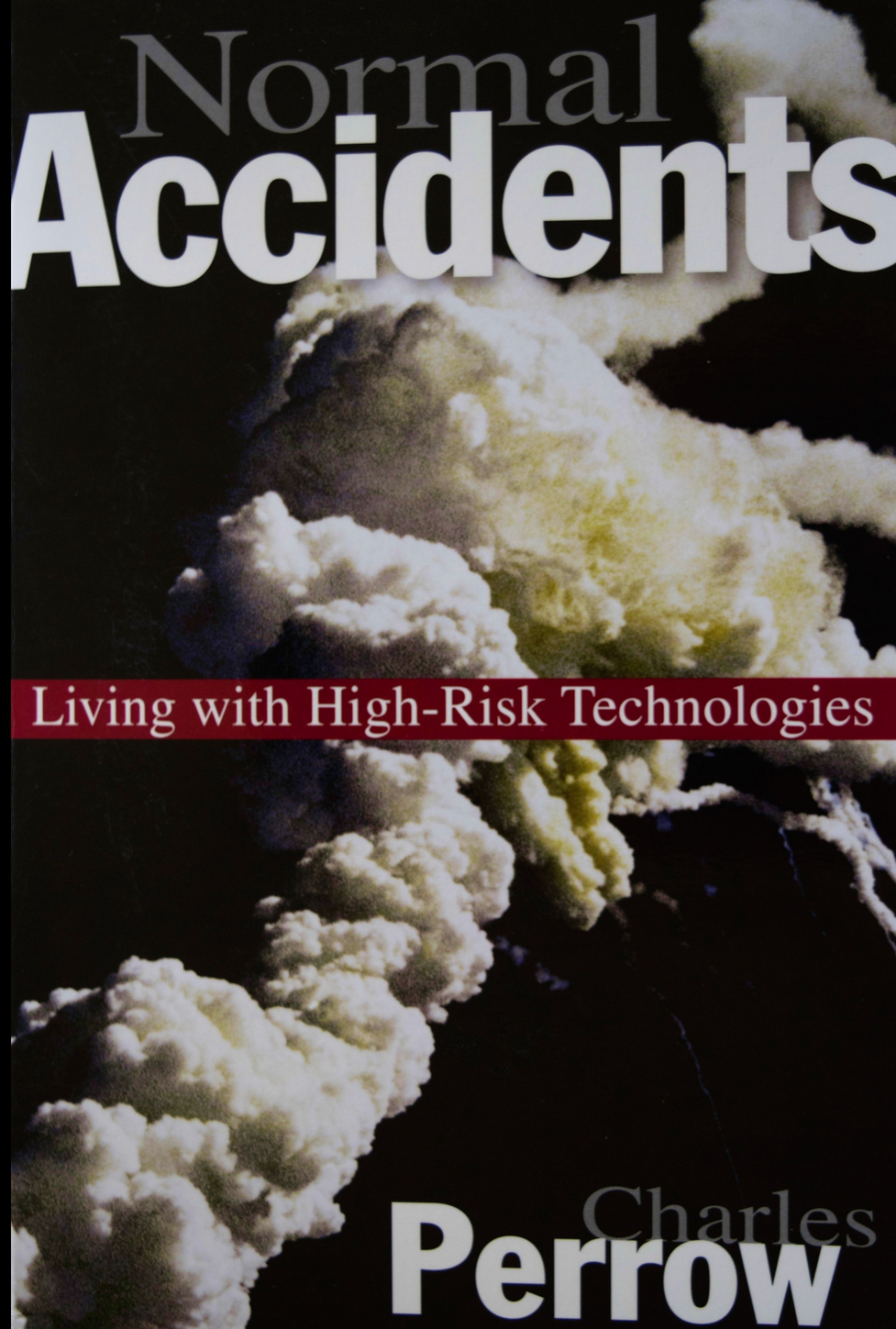
Don’t manipulate memory that isn’t yours.

Critical components get their own machines.

Plan for failure.

If sub-system A depends on B and B fails, what does A need to do?

“Hey, who knew that system C relied on B too?”

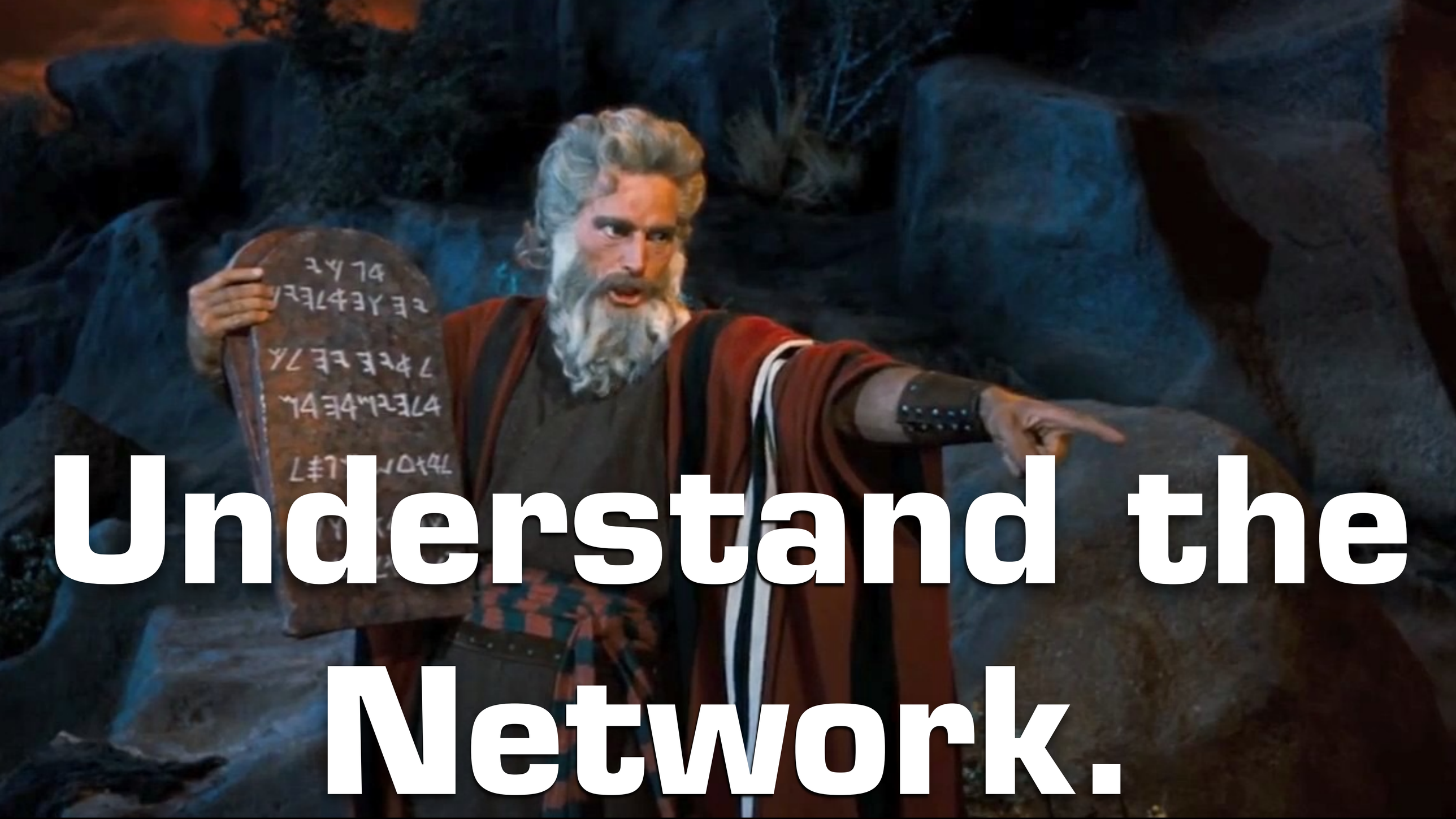


Supervise.

Negotiate the restarts
of failed sub-systems
into well-known states.

Erlang implements all of this for you, but there’s nothing special about it.

Anything else?



Understand the Network.

0. The network is reliable.
1. Latency is zero.
2. Bandwidth is infinite.
3. The network is secure.
4. Topology doesn’t change.
5. There is one administrator.
6. Transport cost is zero.
7. The network is homogenous.

P.S.

This is how the Internet works.

This is how hard real-time systems work.

This is how multi-processors work.

There’s decades
of material to
learn from.

Thanks!

<3

@bltroutwine