Systems that run forever self-heal and scale

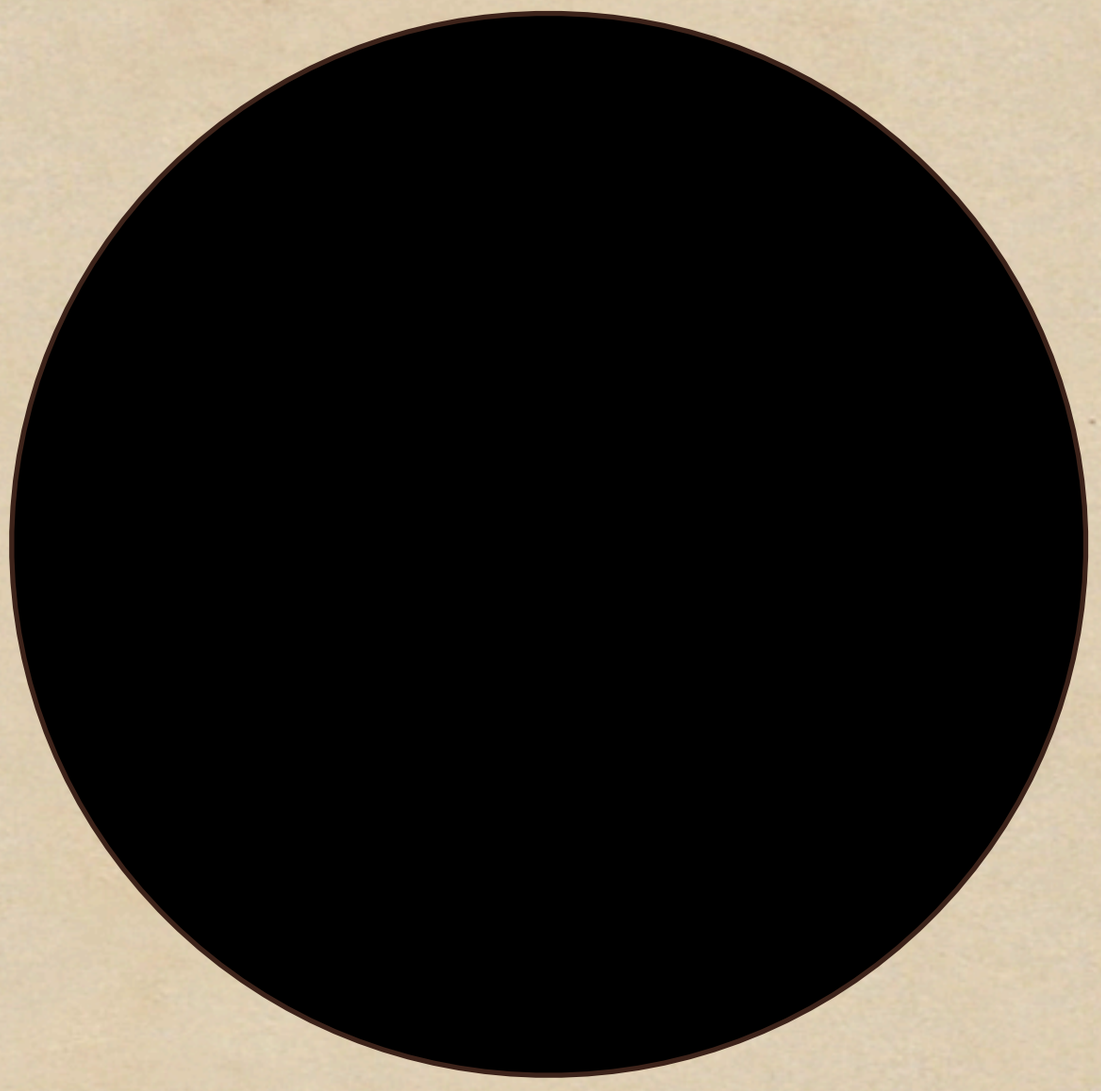
Joe Armstrong

Overview

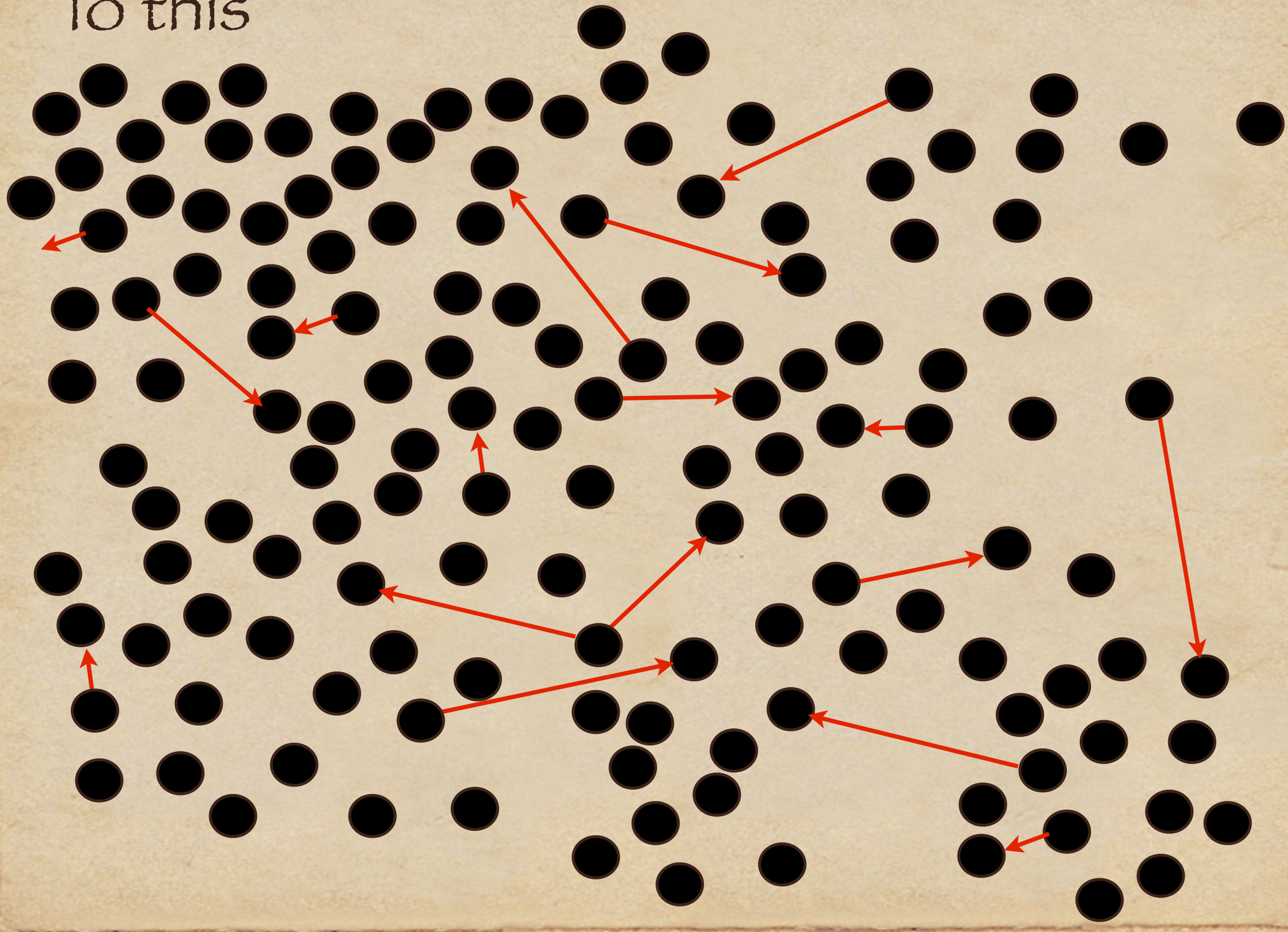
- Problem domain
- Architecture/Algorithms
- The six rules for building HA systems
- Quotes on system building
- How the six rules are programmed in Erlang

Domaín

Go from



To this



- Lots of computers
- Distributed programming
- Parallel programming
- Concurrency
- Fail detection
- Failure repair
- Consistency
- Live Code upgrade

5 nines reliability

9,999 nines?

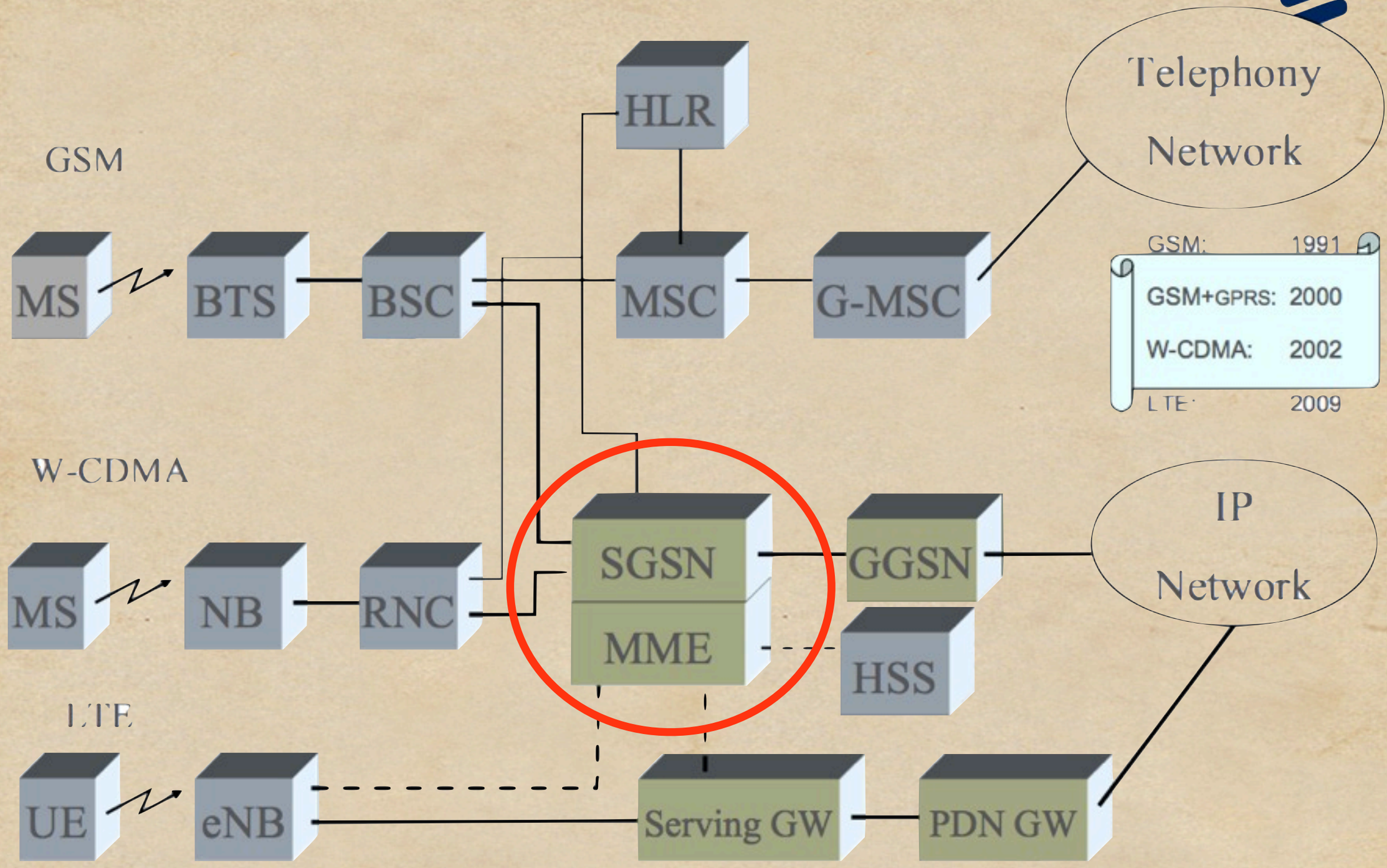
$$P(\text{fail}) = 10^{-3}$$

3333 independent machines

$$P(\text{all fail}) = (10^{-3})^{3333}$$

$$= 10^{-9999}$$

3GPP MOBILE SYSTEMS – GSM, W-CDMA & LTE



Source: Urban Boquist - The Ericsson SGSN-MME


Millions of Smart Phones
Erlang
Internet

- SGSN = Serving GPRS Support Node
- GPRS = General Packet Radio Service
- MME = MME (Mobility Management Entity): The MME is the key control-node for the LTE access-network.

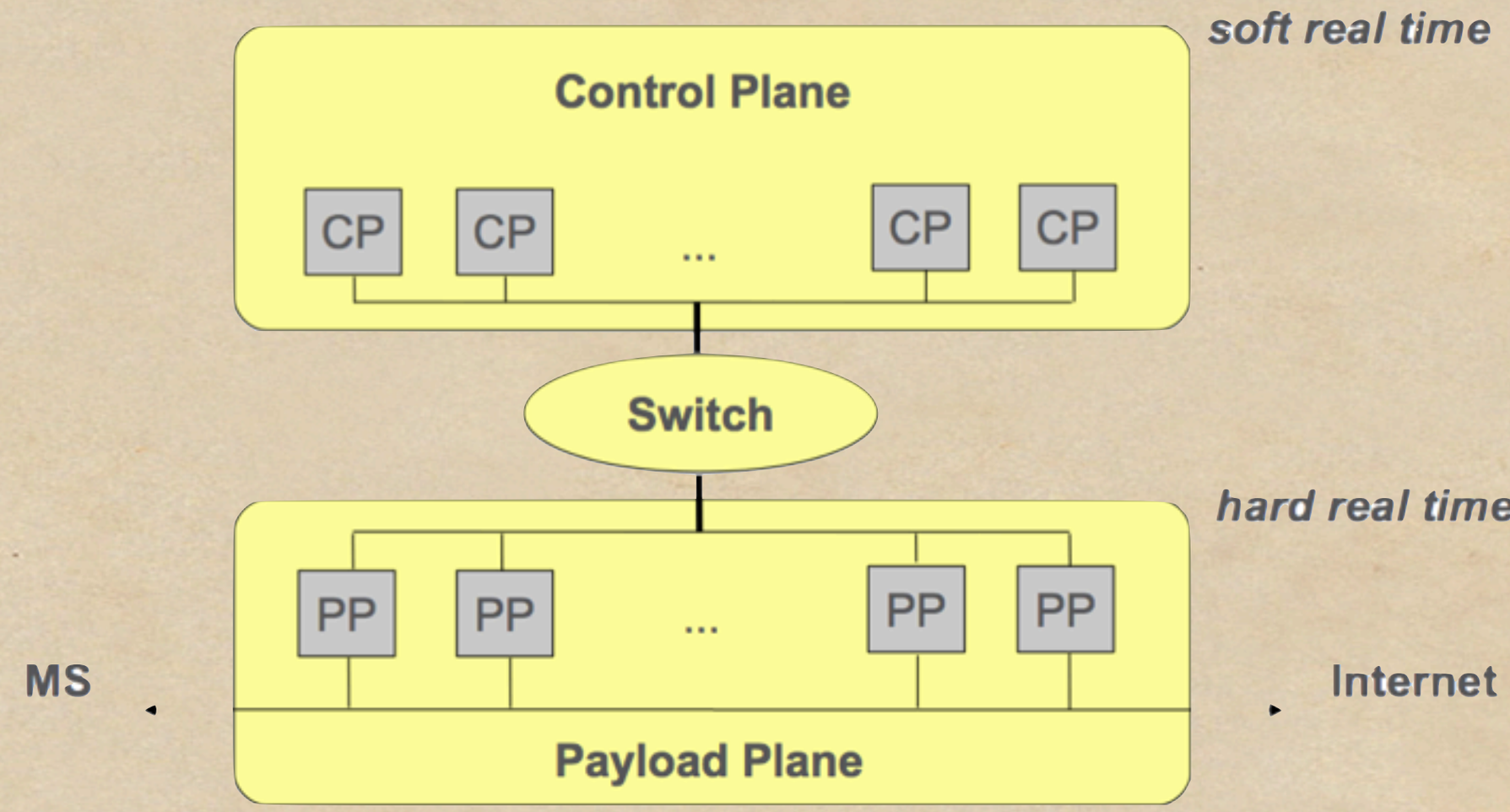
We have unparalleled experience from the deployment and operations of commercial LTE/EPC networks. We launched the world's first commercial LTE/EPC network in Europe and the world's largest network in the US. We have also built and are operating a large number of commercial and pre-commercial LTE/EPC networks across the globe, all including the SGSN-MME. - Ericsson

A serving GPRS support node (SGSN) is responsible for the delivery of data packets from and to the mobile stations within its geographical service area. Its tasks include packet routing and transfer, mobility management (attach/detach and location management), logical link management, and authentication and charging functions - wikipedia.

Building on the proven software and functionality, the new SGSN-MME MkVIII platform on Ericsson Blade System extends the capabilities to meet even the most aggressive growth predictions. It offers superior scalability that supports 18 million users per node and up to 1152 million users when deployed in a pooled configuration. - Ericsson

source:http://www.ericsson.com/ourportfolio/products/sgsn-mme

ARCHITECTURE



Source: Urban Boquist - The Ericsson SGSN-MME

Properties of SGNS/MME

- Goal - 6 nines reliability (31 seconds downtime/ year)
- 18 Million Simultaneous Active Users
- Small footprint (0.0024 square meters per 100k attached users)
- Low power consumption (~50 W per 100k attached users)

"Internet" HA

- Always on-line
- Soft real-time
- Code upgrade on-the-fly
- Once started never stopped - evolving
- Very scalable (one machine to planet-wise)

Distributed Programming is hard

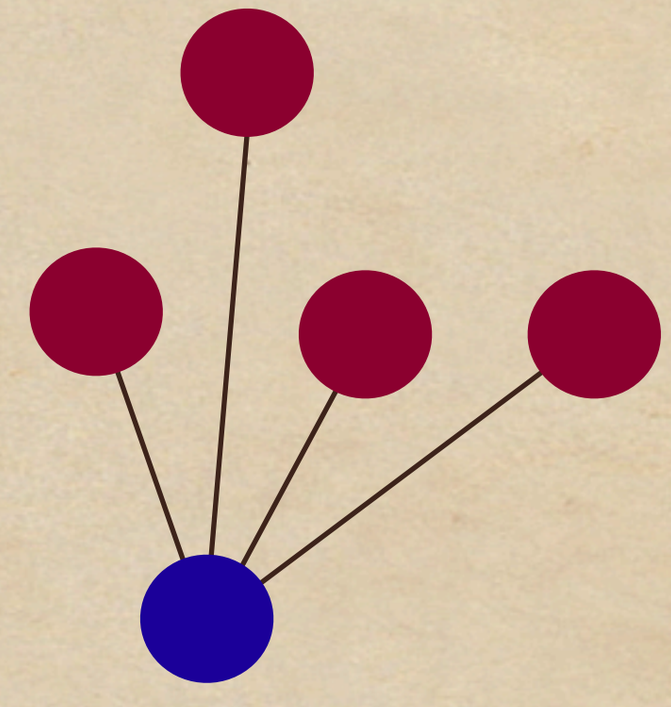
1. Difficulty in identifying and dealing with failures.
2. Achieving consistency in data across processes.
3. Heterogeneous nature of the components involved in the system.
4. Testing a distributed system is quite difficult.
5. The technologies involved in distributed systems are not easy to understand .

5 reasons why Distributed Systems are hard to program - Rajith Attapattu

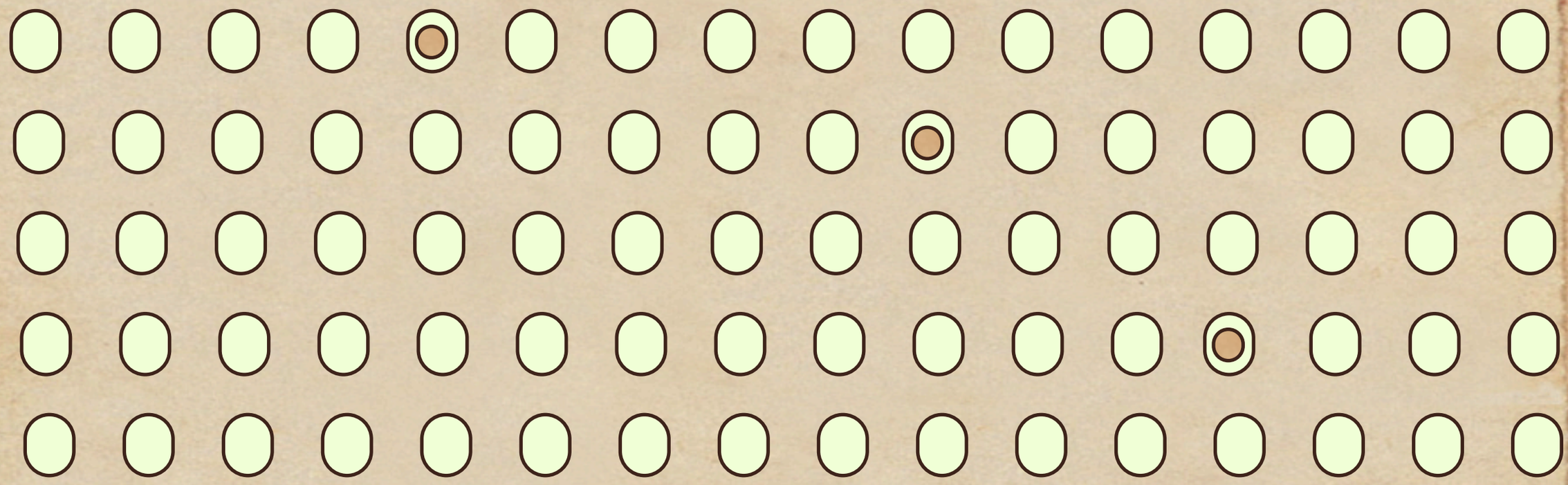
http://rajith.2rlabs.com/2008/07/23/5-reasons-why-distributed-systems-are-hard-to-develop/

Highly available data

- Data is sacred - but we need multiple copies with independent paths to the data.
- Computation can be performed anywhere



Where is my data?



data

Computer

Imagine 10 million computers.

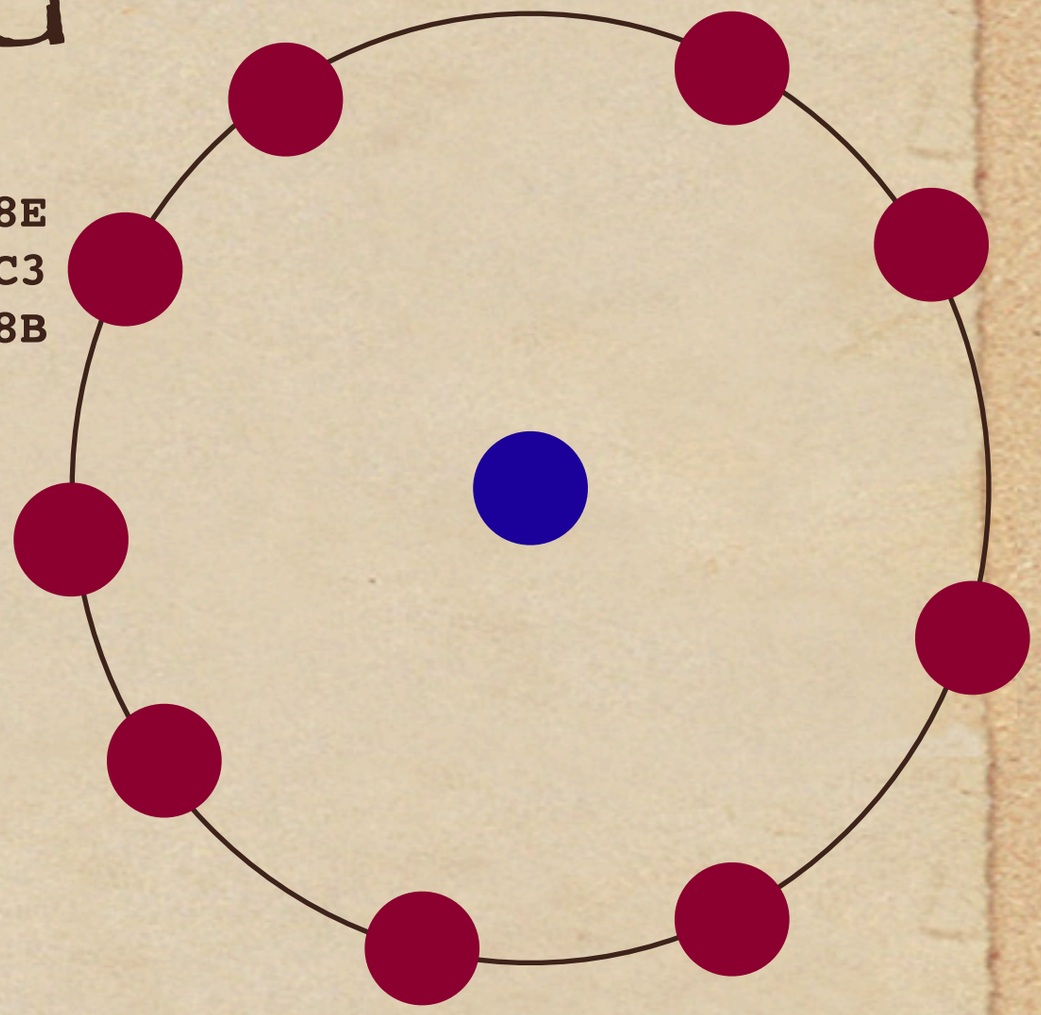
My data is in ten of them.

To find my data I need to know where it is

Key = [5,26,61,...]

Chord

```
S1 IP = 235.23.34.12
S2 IP = 223.23.141.53
S3 IP = 122.67.12.23
..
md5(ip(s1)) = C82D4DB065065DBDCDADFBC5A727208E
md5(ip(s2)) = 099340C20A42E004716233AB216761C3
md5(ip(s3)) = A0E607462A563C4D8CCDB8194E3DEC8B

Sorted
099340C20A42E004716233AB216761C3 => s2
A0E607462A563C4D8CCDB8194E3DEC8B => s3
C82D4DB065065DBDCDADFBC5A727208E => s1
...

lookup Key = "mail-23412"
md5(“mail-23412”) =>
  B91AF709D7C1E6988FCEE7ADF7094A26

So the Value is on machine s3 (first machine
with Md5 lower than md5 of key)

Replica
md5(md5(“mail-23412”)) =>
   D604E7A54DC18FD7AC70D12468C34B63

So the replica is on machine s1
```



Main idea

Hash keys & IP addresses into the same namespace

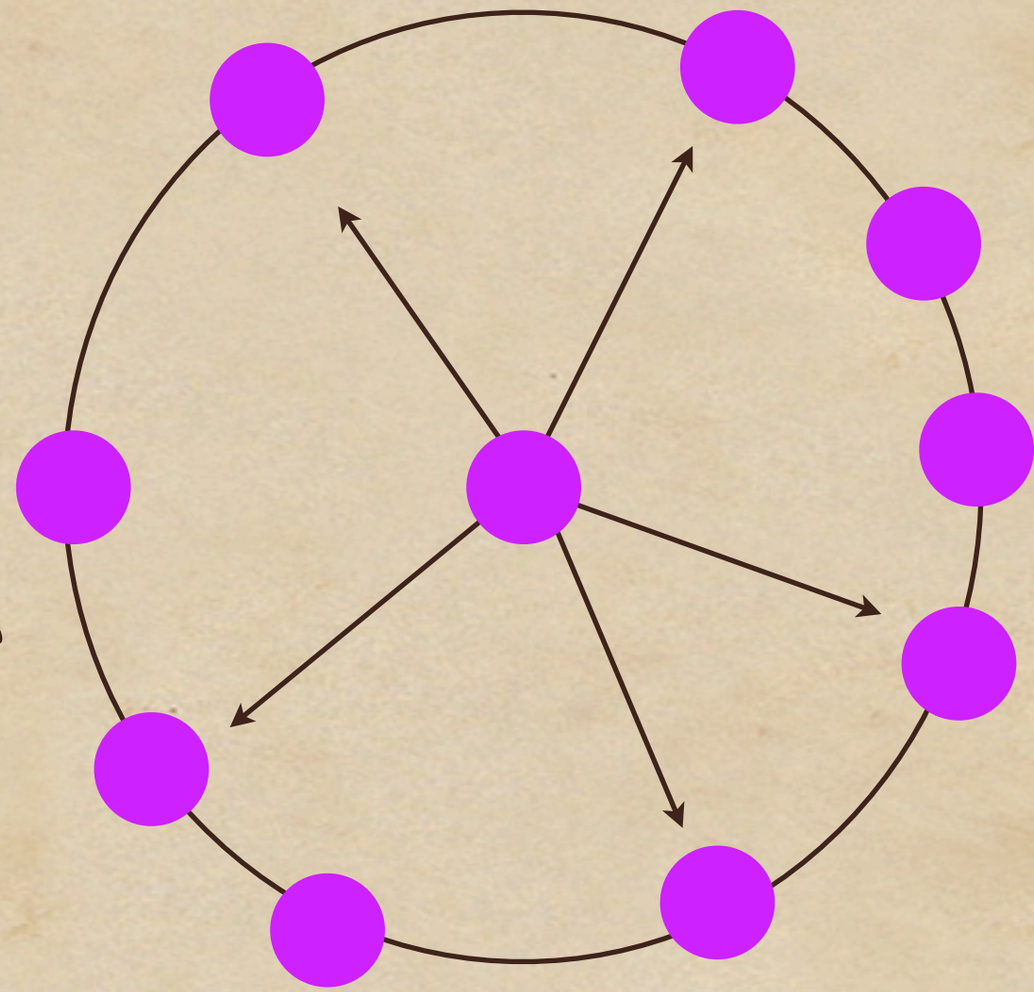
Replicas

- Keep an odd number of replicas
- Vote
- Simple majority wins

Collect five copies in parallel

- Peer

Making 5 replicas takes the same time as two

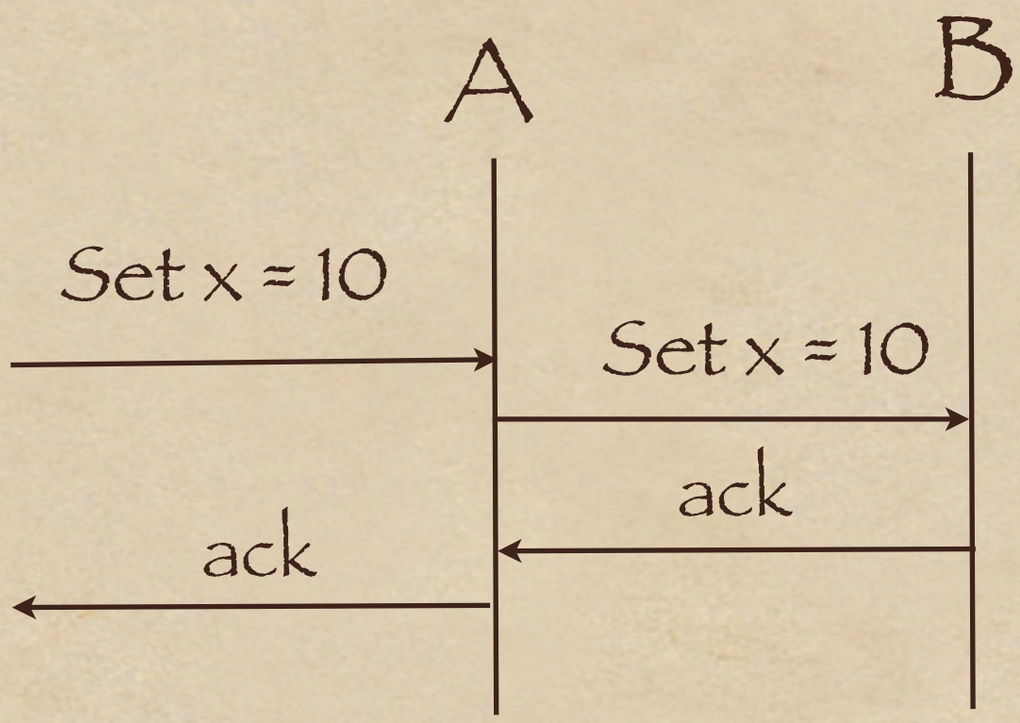


"P2P is the new client-server"

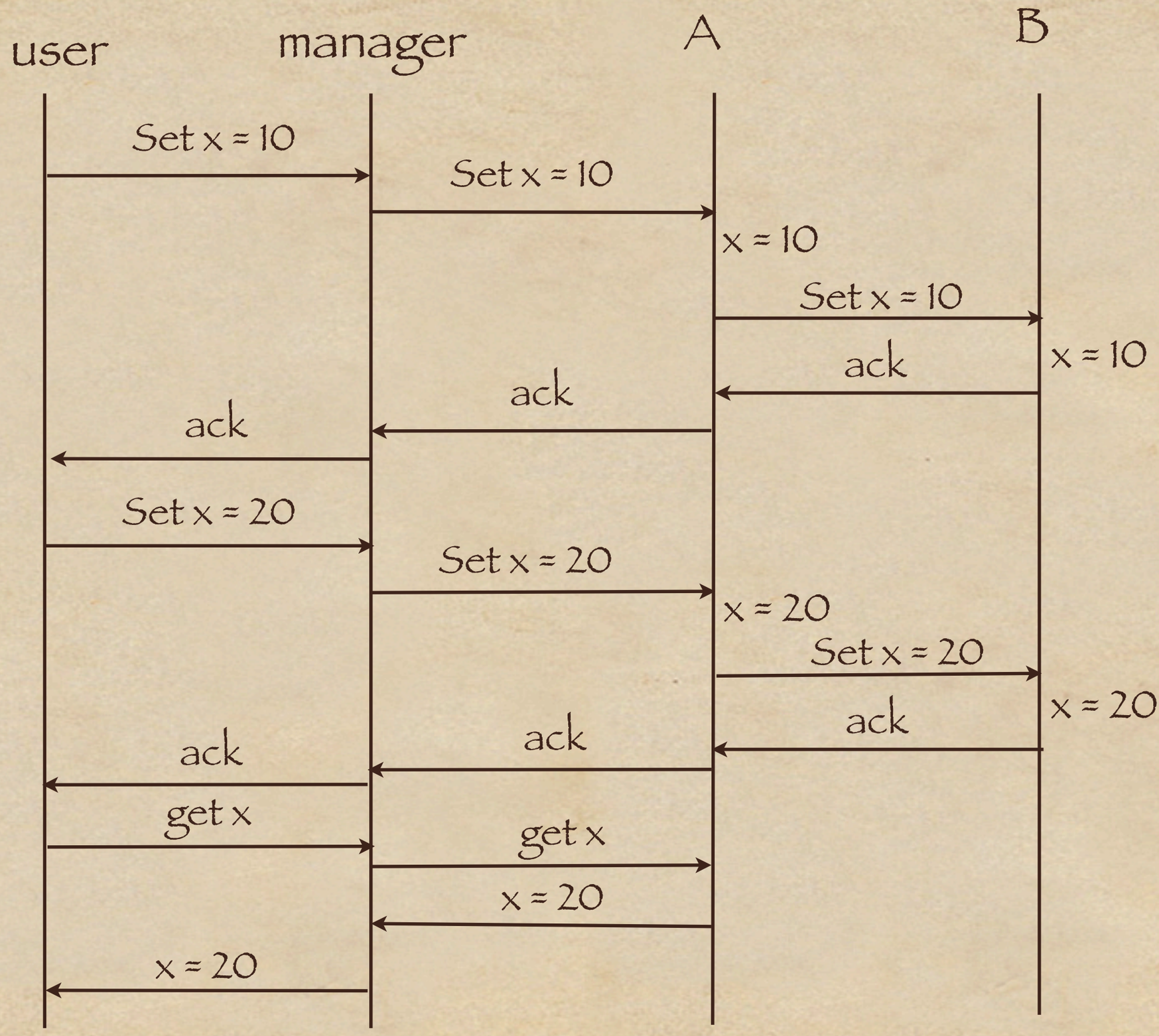
Making replicas is easy

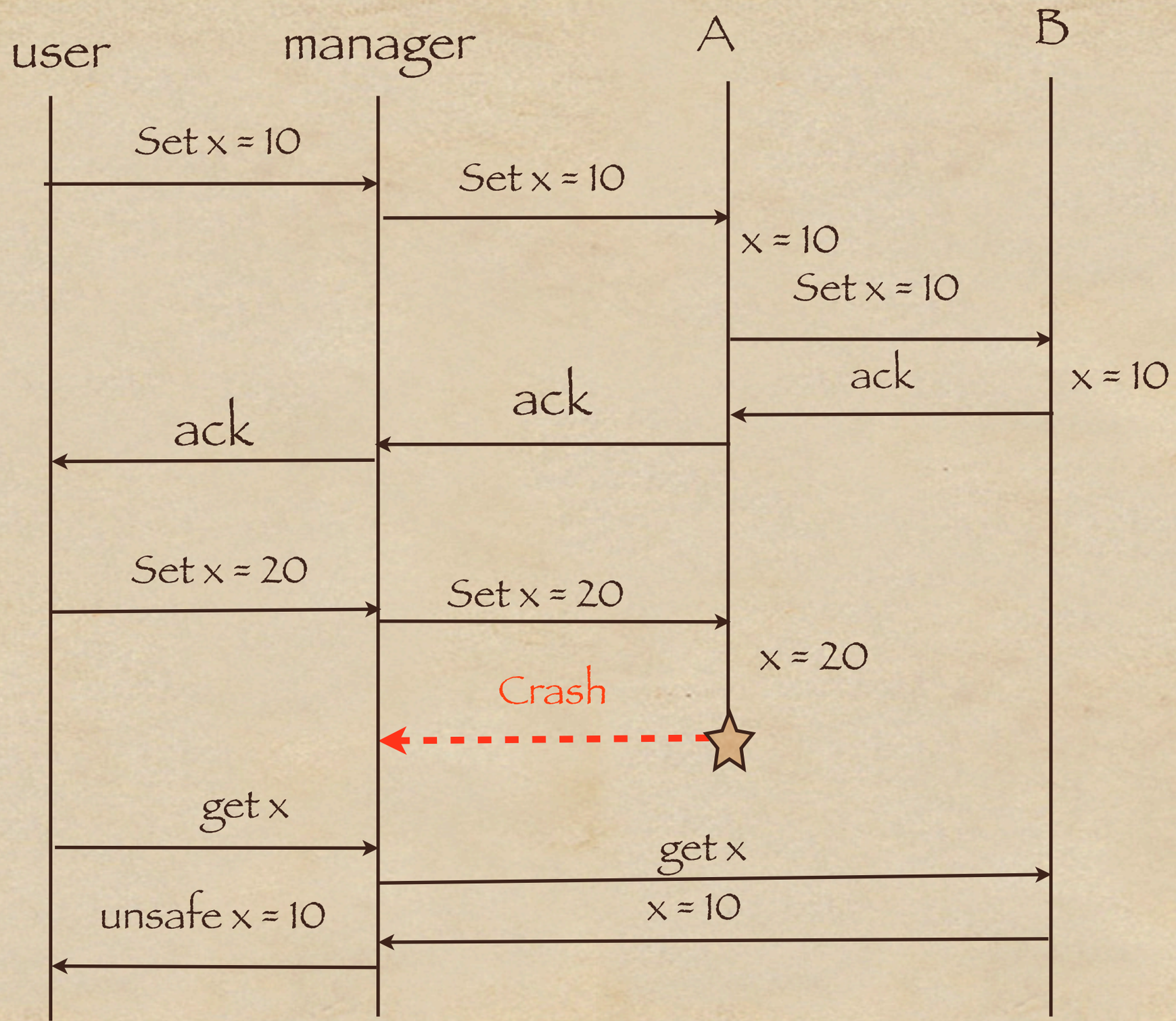
or ...

Replicas



- A knows that x is replicated on B
- B does not know if X is replicated

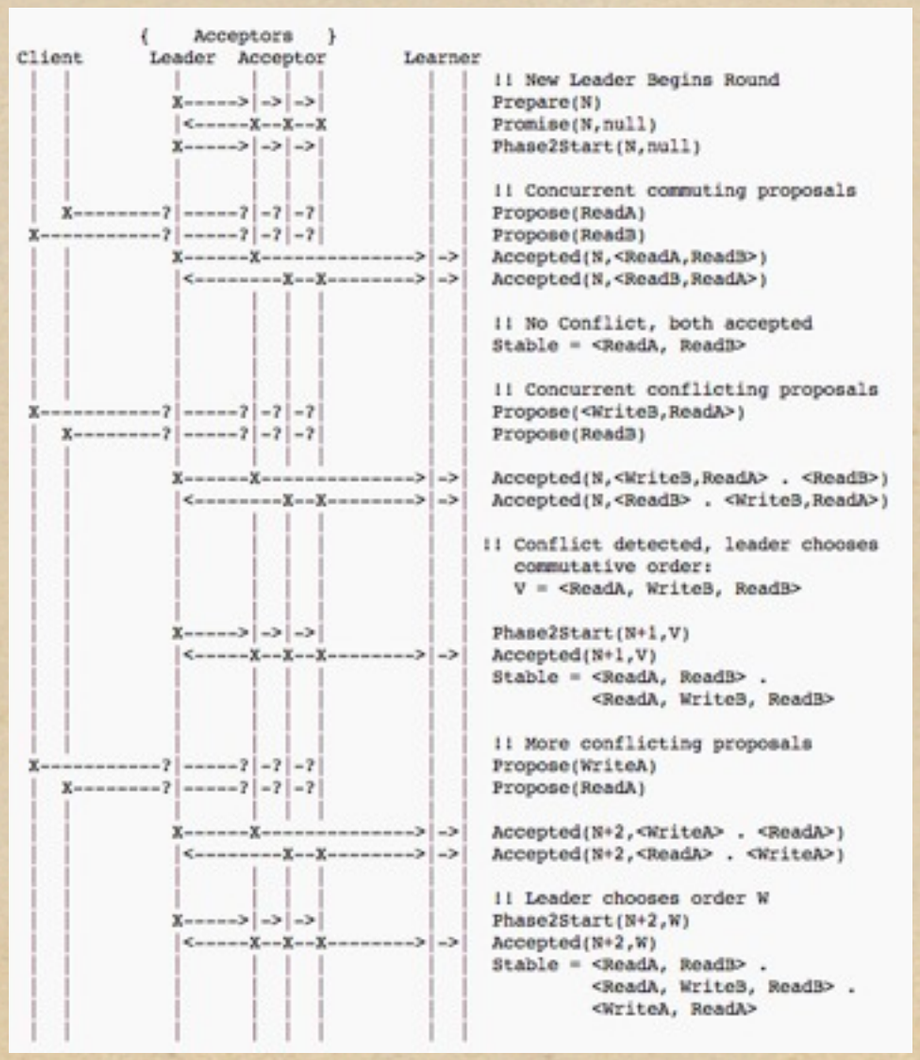

user
manager
A
B
Set x = 10
Set x = 10
x = 10
Set x = 10
ack
x = 10
ack
ack
Set x = 20
Set x = 20
x = 20
Set x = 20
ack
x = 20
ack
ack
get x
get x
x = 20
x = 20


user
manager
A
B
Set x = 10
Set x = 10
x = 10
Set x = 10
ack
x = 10
ack
ack
Set x = 20
Set x = 20
x = 20
Crash
get x
get x
unsafe x = 10
x = 10

What happens if the master dies?

Life get a tad tricky

- Paxos
- Distributed leadership election
- Distributed consensus



Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [9] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [10, 12, 13]. These explanations focus on single-decree Paxos; they are still challenging, yet they leave the reader without enough information to build practical systems. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the original paper [9] until after reading several simplified explanations and designing our own alternative protocol, a process that took several months.

In search of an Understandable Consensus Algorithm
- Diego Ongaro and John Ousterhout

The problem of reliable storage of data has been solved*

* OTP libraries: mnesia gen_leader
Third party: Riak

SIX RULES

ONE ISOLATION

Isolation enables

- Fault-tolerance
- Scalability
- Reliability
- Testability
- Comprehensibility
- Code Upgrade

TWO

CONCURRENCY

Concurrency

- World is concurrent
- Many problems are Embarrassingly Parallel
- Need at least TWO computers to make a non-stop system (or a few hundred)
- TWO or more computers = concurrent and distributed

THREE

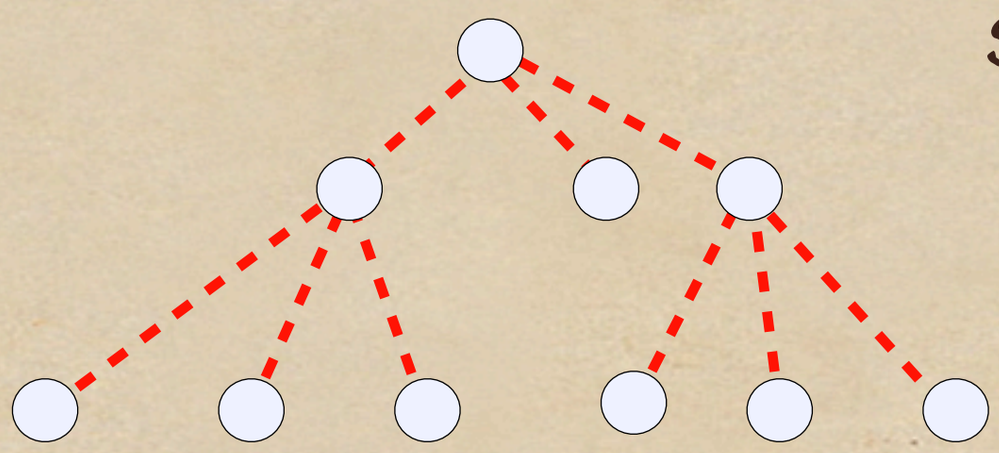
MUST

DETECT FAILURES

Failure detection

- If you can't detect a failure you can't fix it
- Must work across machine boundaries
  the entire machine might fail
- Implies distributed error handling,
  no shared state,
  asynchronous messaging

Supervision trees



supervisors

workers

FOUR

FAULT IDENTIFICATION

Fault Identification

- Fault detection is not enough - you must no why the failure occurred
- Implies that you have sufficient information for post hock debugging

FIVE

LIVE CODE

UPGRADE

Live code upgrade

- Must upgrade software while it is running
- Want zero down time
- Once a system is started we never stop it

SIX

STABLE

STORAGE

Stable storage

- Must store stuff forever
- No backup necessary - storage just works
- Implies multiple copies, distribution, ...
- Must keep crash reports

QUOTES

GRAY

As with hardware, the key to software fault-tolerance is to
hierarchically decompose large systems into modules, each module being
a unit of service and a unit of failure. A failure of a module does
not propagate beyond the module.

...

The process achieves fault containment by sharing no state with
other processes; its only contact with other processes is via messages
carried by a kernel message system

-

-

-

Jim Gray

Why do computers stop and what can be done about it

Technical Report, 85.7 - Tandem Computers,1985

GRAY

- Fault containment through fail-fast software modules.
- Process-pairs to tolerant hardware and transient software faults.
- Transaction mechanisms to provide data and message integrity.
- Transaction mechanisms combined with process-pairs to ease exception handling and tolerate software fault
- Software modularity through processes and messages.

Fail fast

The process approach to fault isolation advocates that the process software be fail-fast, it should either function correctly or it should detect the fault, signal failure and stop operating.

Processes are made fail-fast by defensive programming. They check all their inputs, intermediate results and data structures as a matter of course. If any error is detected, they signal a failure and stop. In the terminology of [Christian], fail-fast software has small fault detection latency.

Gray
Why ...

Fail early

A fault in a software system can cause one or more errors. The latency time which is the interval between the existence of the fault and the occurrence of the error can be very high, which complicates the backwards analysis of an error ...

For an effective error handling we must detect errors and failures as early as possible

Renzel -

Error Handling for Business Information Systems,

Software Design and Management, GmbH & Co. KG, München, 2003

ALAN KAY

Folks --

Just a gentle reminder that I took some pains at the last OOPSLA to try to remind everyone that Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

The big idea is "messaging" -- that is what the kernel of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase)....

http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

SCHNEIDER

Halt on failure in the event of an error a processor
should halt instead of performing a possibly erroneous operation.

Failure status property when a processor fails,
other processors in the system must be informed. The reason for
failure must be communicated.

Stable Storage Property The storage of a processor should be
partitioned into stable storage (which survives a processor crash)
and volatile storage which is lost if a processor crashes.

Schneider

ACM Computing Surveys 22(4):229-319, 1990

“Let ít crash”

Armstrong

Corollary

"all sequential languages get error handling wrong"

Programming

Erlang

- A Functional programming language
- A set of design principles (OTP)
- A Virtual Machine (BEAM)

How do we program our six rules?

- Use a library?
- Use a programming language designed for this

How to implement the six rules in Erlang

Rule 1 = Isolation

- Erlang processes are isolated
- One process cannot damage another
- One Erlang node can have millions of processes
- Process have no shared memory
- Process are very lightweight

Rule 2 = Concurrency

- Erlang processes are concurrent
- All processes run in parallel (in theory)
- On a multi-core the processes spread over the cores

```
Pid = spawn(fun() -> ... end)

Pid ! Message

receive
   Pattern1 -> Actions1;
   Pattern2 -> Actions2;
   Pattern3 -> Actions3;
   ...
end
```

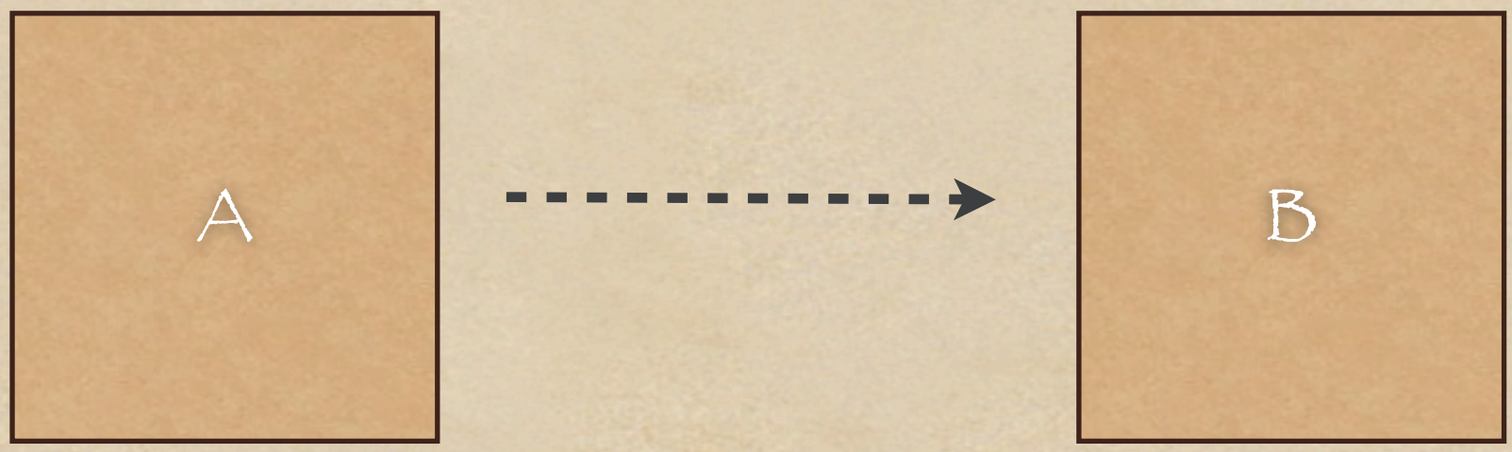
Rule 3 = Failure detection

- Erlang processes can detect failures

```
Pid = spawn_link(fun() -> ... end),
process_flag(trap_exit, true)

receive
    {'EXIT', Pid, Why} ->
        ...
end
```

- Can link to a remote process

Fix the error somewhere else



A is a black box.

It might be an entire machine

If an entire machine crashes

another machine must fix the problem

Rule 4 - fault identification

- Erlang error signals contain error descriptors

```
Pid = spawn_link(fun() -> ... end),
process_flag(trap_exit, true)

receive
   {'EXIT', Pid, Why} ->
       error_log:log_error({erlang:now(),Pid,Why})
       ...
end
```

Rule 5 - live code upgrade

- Erlang can be modified as it runs

```
-module(foo).

...

f1(X) ->
   foo:bar(X),     %% Call the latest version of foo:bar
   bar(X).         %% Call this version of bar

bar(X) ->
   ...
```

- Applications can be upgraded as they run (this is a large part of OTP)

Rule 6 - Stable storage

- Use mnesia - highly customizable - can store data on disk + RAM, can RAM replicate etc.
- Use third-party storage - Riak, CouchDB etc

Fault tolerance implies scalability

- To make things fault-tolerant we have to make sure they are made from isolated components
- If the components are isolated they can be run in parallel
- Things that are isolated and can be run in parallel are scalable

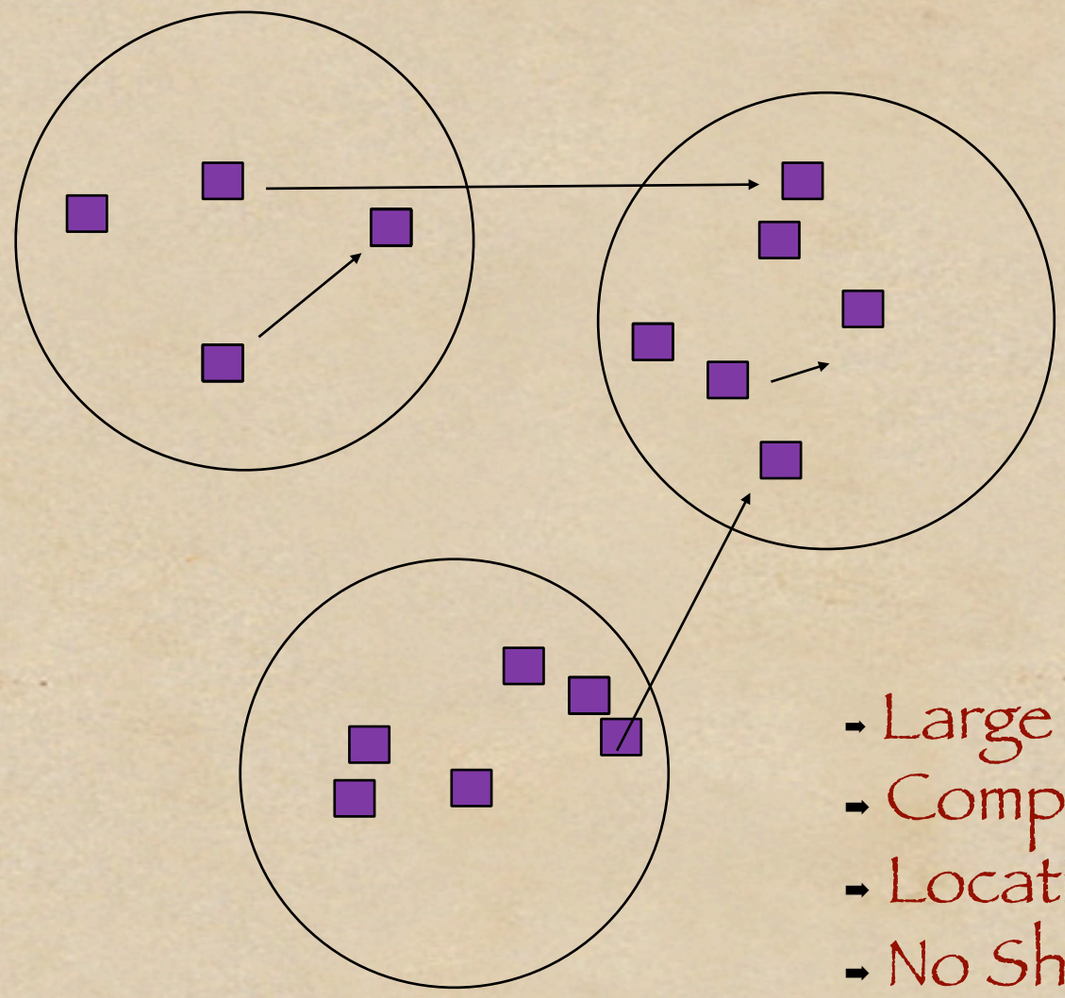
Erlang

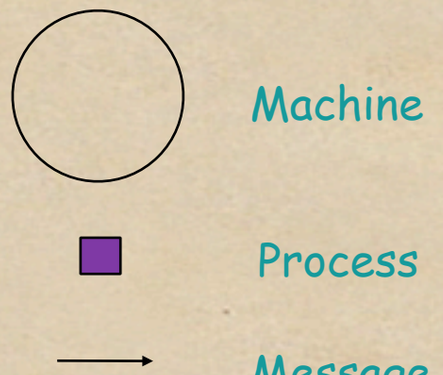
- Very light-weight processes
- Very fast message passing
- Total separation between processes
- Automatic marshalling/demarshalling
- Fast sequential code
- Strict functional code
- Dynamic typing

Properties

- No sharing
- Hot code replacement
- Pure message passing
- No locks
- Lots of computers (= fault tolerant scalable ...)

What is COP?





- Large numbers of processes
- Complete isolation between processes
- Location transparency
- No Sharing of data
- Pure message passing systems

No Mutable State

- Mutable state needs locks
- No mutable state = no locks =

programmers bliss

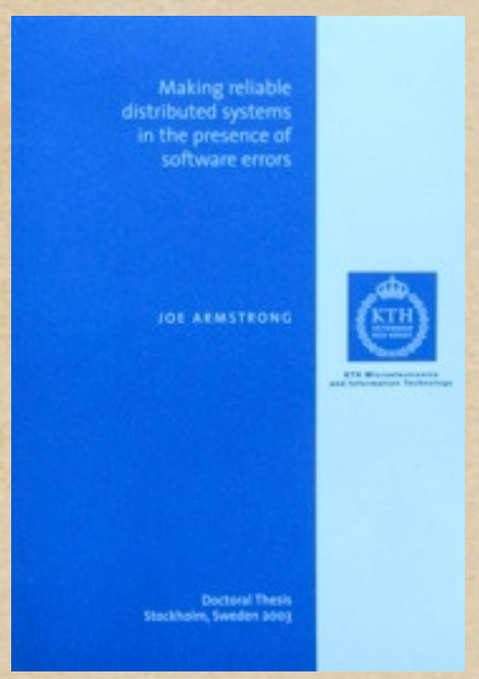
Projects

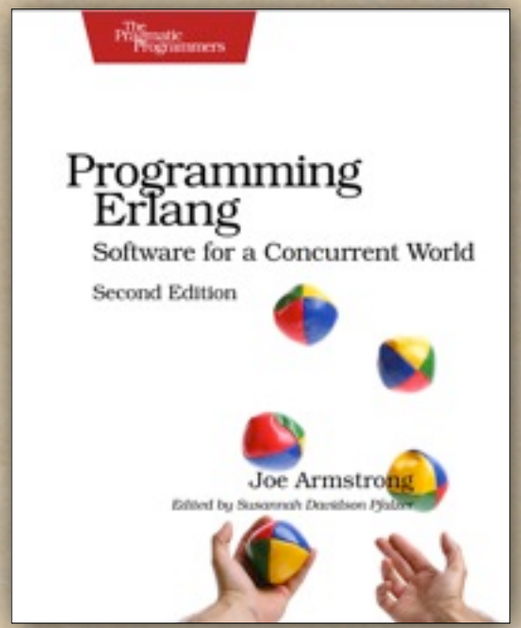
- CouchDB
- Riak
- Chef11
- Mochiweb/cowboyScalaris
- Nitrogren/Chicago Boss
- Ejabberd (xmpp)
- Rabbit MQ (amqp)

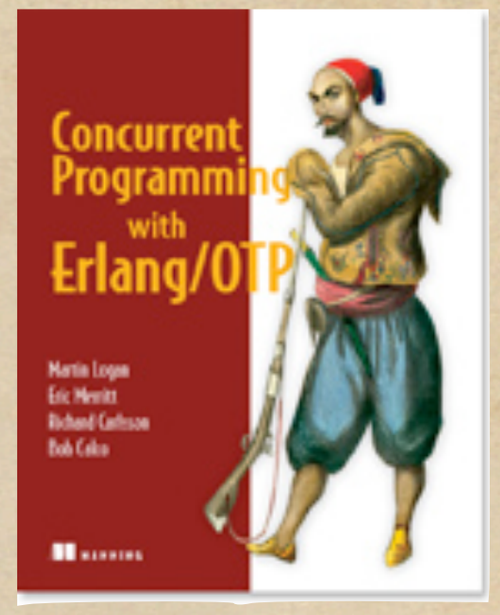
Companies

- Ericsson
- Basho
- Tail-f
- Klarna
- Facebook
- WhatsApp
- ...

Books







http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf

Fun stuff

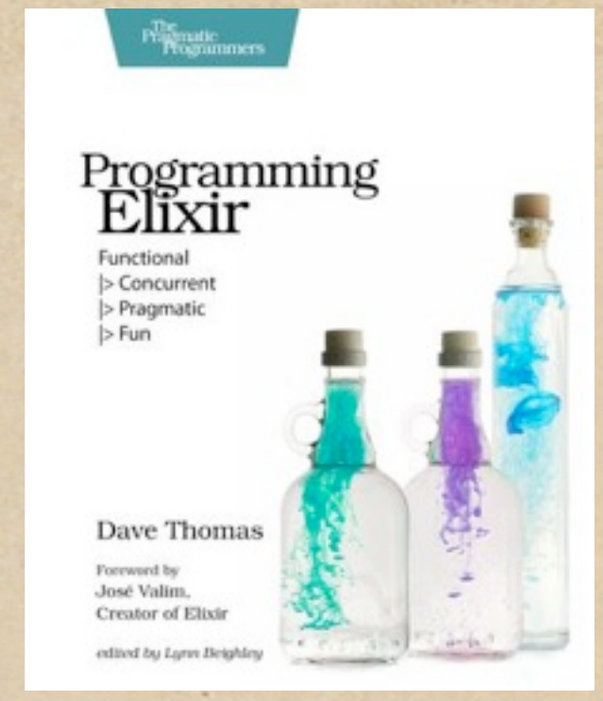
- Elixir

  Same VM

  Wrappers to OTP libraries

  Ruby like syntax

  Fancy metaprogramming



Video from concurix

QUESTIONS