Episode 3. Principles in Network Design

Part 2

Baochun Li

Department of Electrical and Computer Engineering
University of Toronto

Designing a real system is a **creative** process, and like anything else creative, there are some **ideas** that help getting good designs

Recall: Designing the network as a system

Every complex computer system involves one or more communication links, usually organized to form a network

- Identified challenging properties of a network
- The layering principle: the three-layer reference design
- The end-to-end argument (which we just covered): applications know the best!

But there are more of these principles (techniques) in design

Reading: Keshav 6.1 — 6.5

What is system design?

A computer network provides computation, storage and transmission resources

System design is the art and science of putting together these resources into a harmonious whole

Ultimately, you wish to extract the most from what you have

Where do we start from?

Simplicity?

Three rules of work:

Out of clutter find **simplicity.**

From discord find **harmony.**

In the middle of difficulty lies **opportunity.**

Albert Einstein

~~**Simplicity?**~~ It can be an overarching philosophy, but it is too high level.

Performance metrics and resource constraints

In any system, some resources are more freely available than others

- Think about a high-end laptop connected to Internet by a DSL modem
- The constrained resource is link bandwidth
- CPU and memory are unconstrained

We wish to maximize a set of performance metrics given a set of resource constraints

Explicitly identifying constraints and metrics helps in designing efficient systems

- Maximize reliability (mean time between failures) for a car that costs less than $10,000 to manufacture

Real-world system design should be

Scalable, modular, extensible, and elegant

Future-proof

- Rapid technological change
- Market conditions may dictate changes to design halfway through the process
- International standards, which themselves change slowly, also impose constraints

Most resources are a combination of time, space, computation, money, labor, and scaling

Let’s think about a few of these in turn

Time

Shows up in many constraints

deadline for task completion, time to market, mean time between failures

Metrics

response time: mean time to complete a task

throughput: number of tasks completed per unit time

degree of parallelism = response time $\times$ throughput

20 tasks completed in 10 seconds, and each task takes 3 seconds

$\rightarrow$ degree of parallelism = $3 \times 20 / 10 = 6$

Space

Example: a limit on the memory available for a buffer to hold packets in switches and routers

We can also view bandwidth as a **space** constraint

> A T3 link has a bandwidth of 44.768 Mbps. If we use it to carry video streams with a mean bit rate of 1.5 Mbps, we can fit at most 29 streams in this link.

Scaling

A design constraint, rather than a resource constraint

Minimizes the use of centralized elements in the design

- Yet, forces the use of complicated distributed algorithms

Hard to measure

but necessary for success

Think about resource bottlenecks

Bottlenecks are the most constrained elements in a system

System performance improves by removing the bottlenecks

- But inevitably creates new bottlenecks

In a balanced system, all resources are simultaneously bottlenecked

- This is optimal, but nearly impossible to achieve
- In practice, bottlenecks move from one part of the system to another
- Historical example: Ford Model T

Time for design ideas!

Idea #1: Multiplexing and virtualization

Multiplexing

Another word for sharing

Trades time and space for money

Users see an increased response time, and take up space when waiting, but the system costs less

economies of scale make a single large resource cheaper

Multiplexing

Examples

- Multiplexed communication links
- Servers in cloud computing

Another way to look at a shared resource

- Unshared virtual resource — the telephone network with time-division multiplexing

Server controls access to the shared resource

- uses a **schedule** to resolve contention
- choice of scheduling: critical in proving quality of service guarantees — think about boarding a flight

Statistical multiplexing

Suppose resource has capacity **C**

Shared by **N** identical tasks

Each task requires capacity **c**

If $N \cdot c \leq C$, then the resource is underloaded

If at most 10% of tasks active, then $C \geq N \cdot c / 10$ is enough

- We used **statistical knowledge** of users to reduce system cost
- This is the statistical multiplexing gain

Two types of statistical multiplexing

Spatial

we expect only a fraction of tasks to be simultaneously active

Temporal

we expect a task to be active only part of the time — its **average** resource consumption is less than its peak

e.g. silence periods during a voice call; video streams with variable bit rates

Idea #2: Batching

Batching: trading response time for throughput

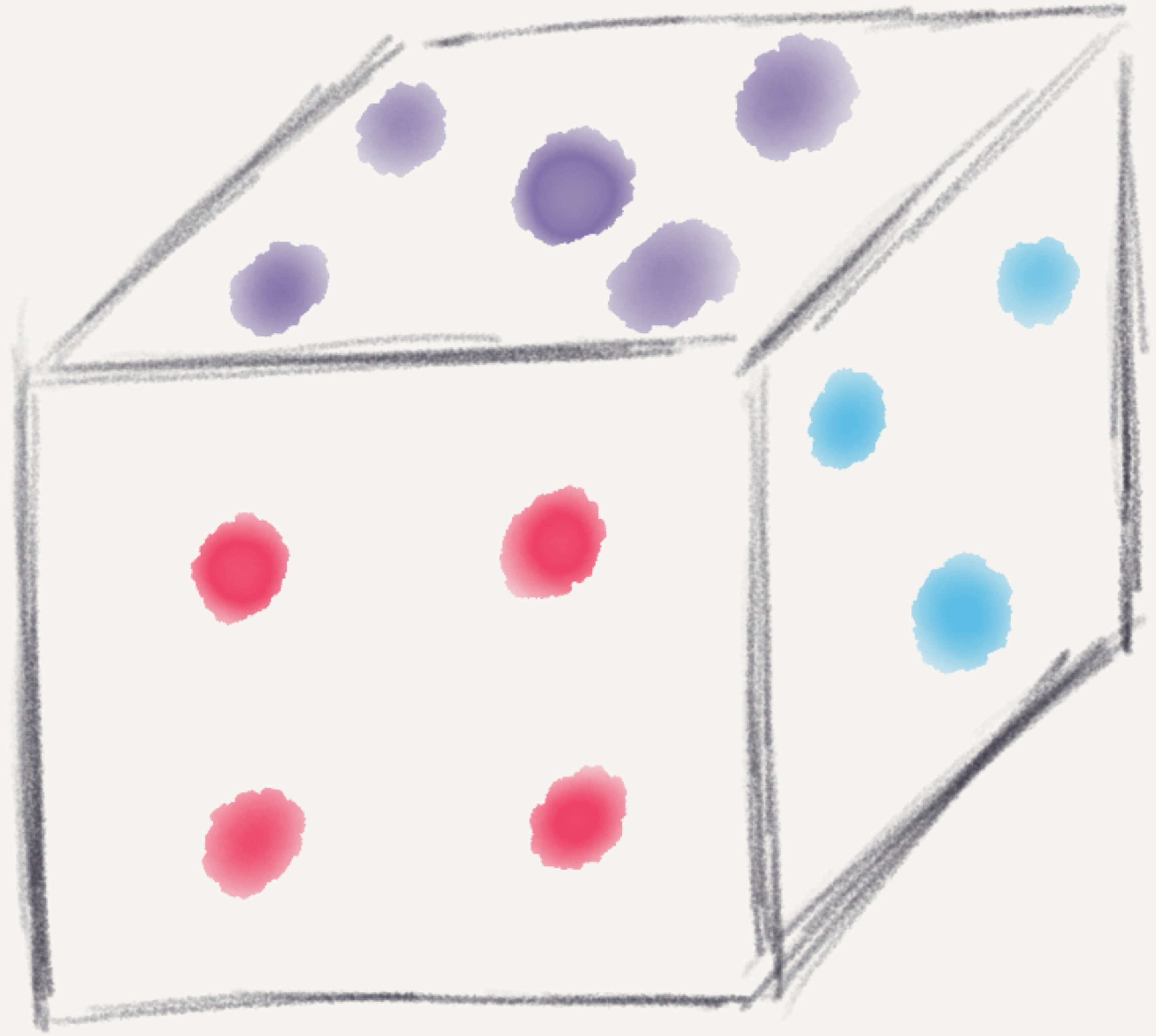
Group tasks together to amortize overhead

Only works when overhead for N tasks < N time overhead for one task

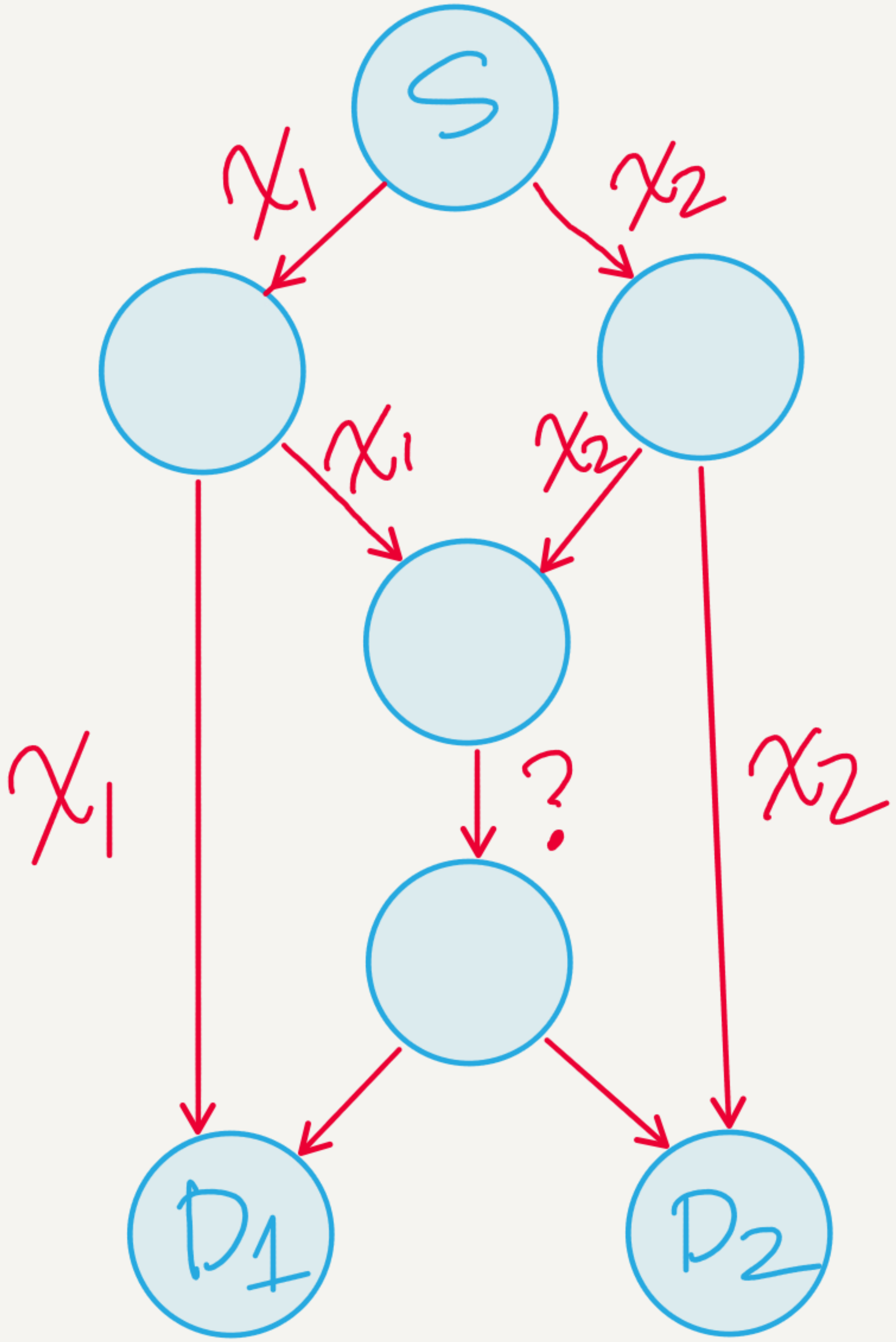
Also, time taken to accumulate a batch shouldn't be too long

We're getting reduced overhead and increased throughput, but suffering from a longer worst case response time

Idea #3: Randomize!

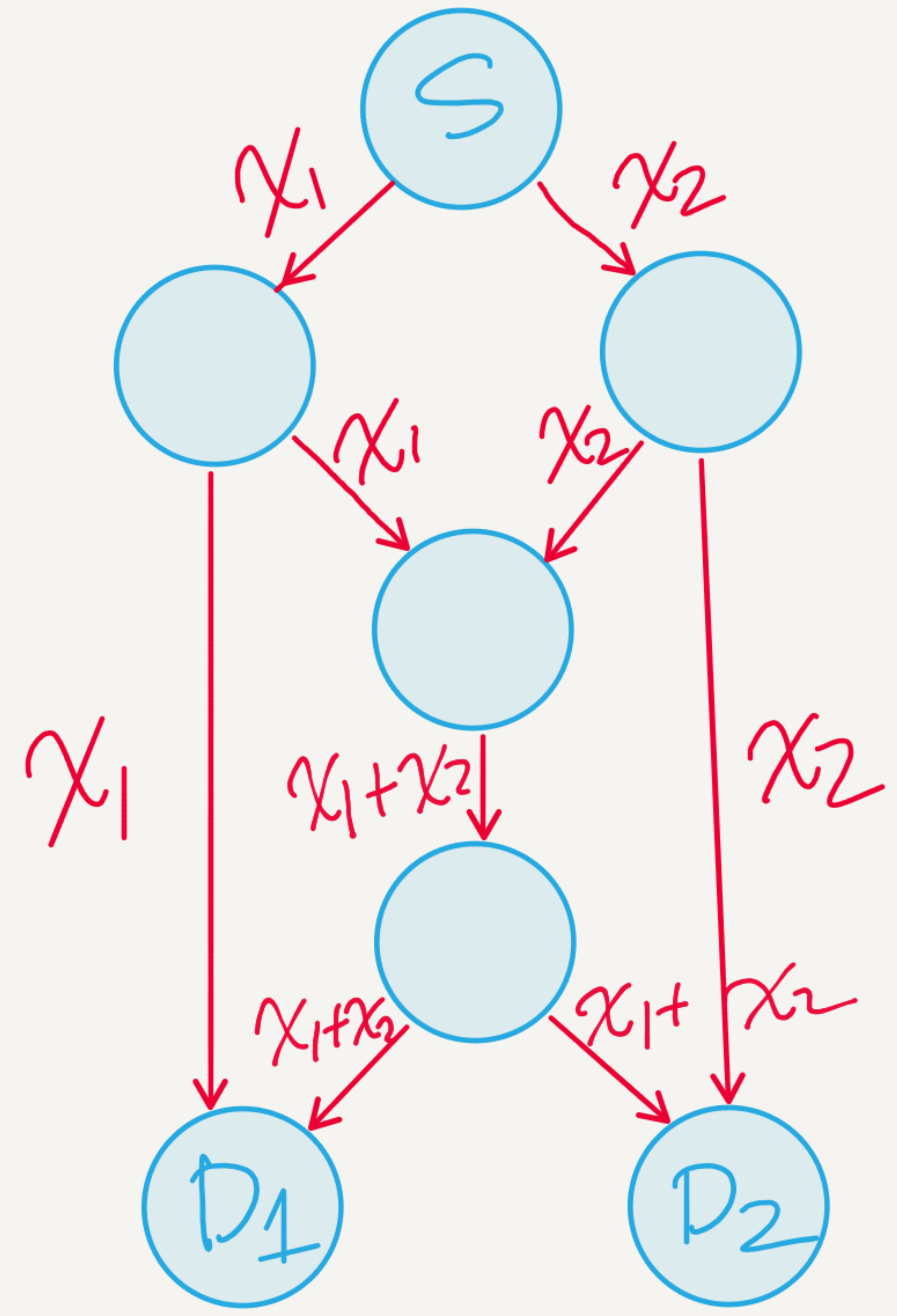


Multicast in a directed network graph



Achievable Throught of 1.5?

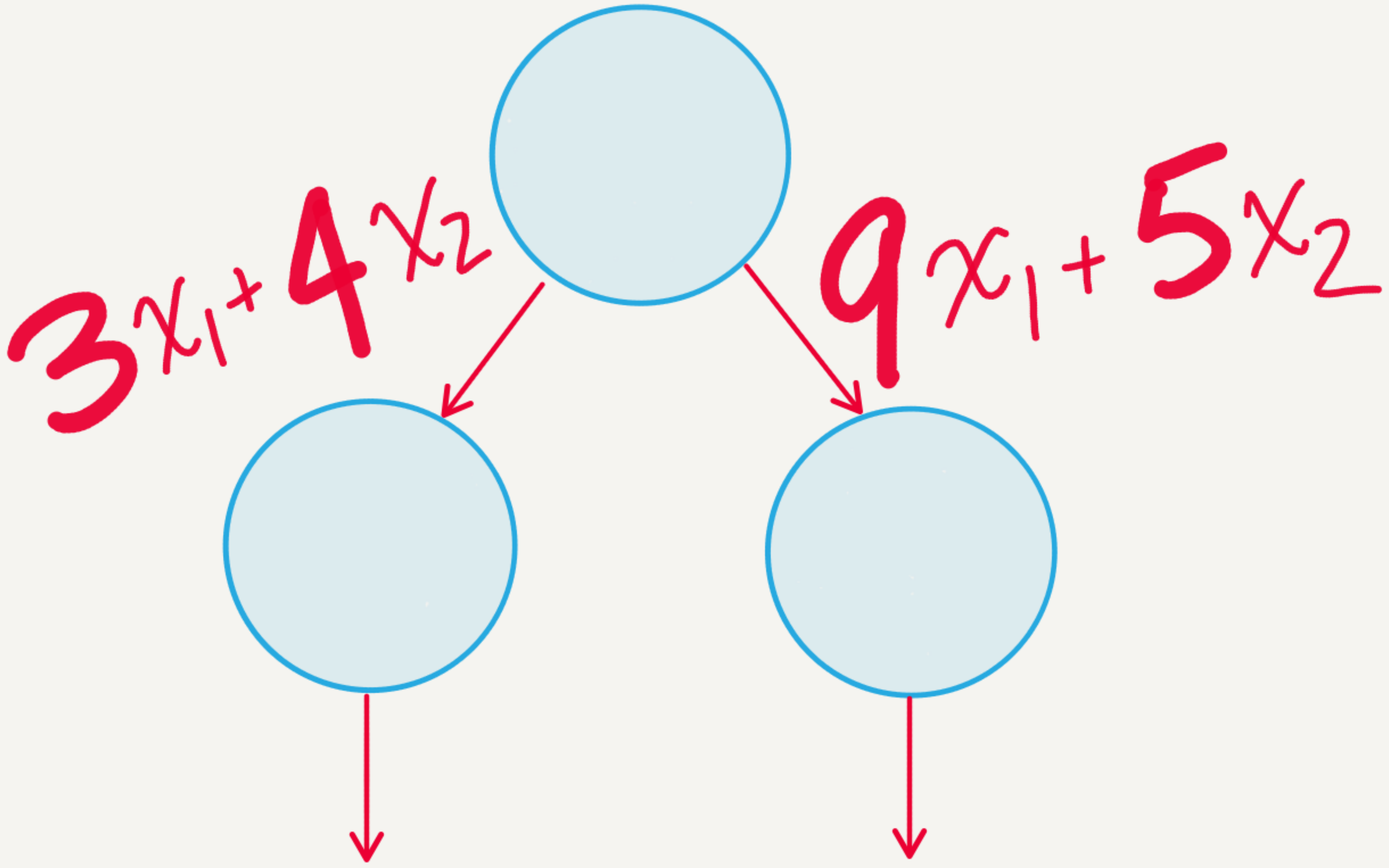
Linear network coding achieves the maximum throughput in any directed network graph!



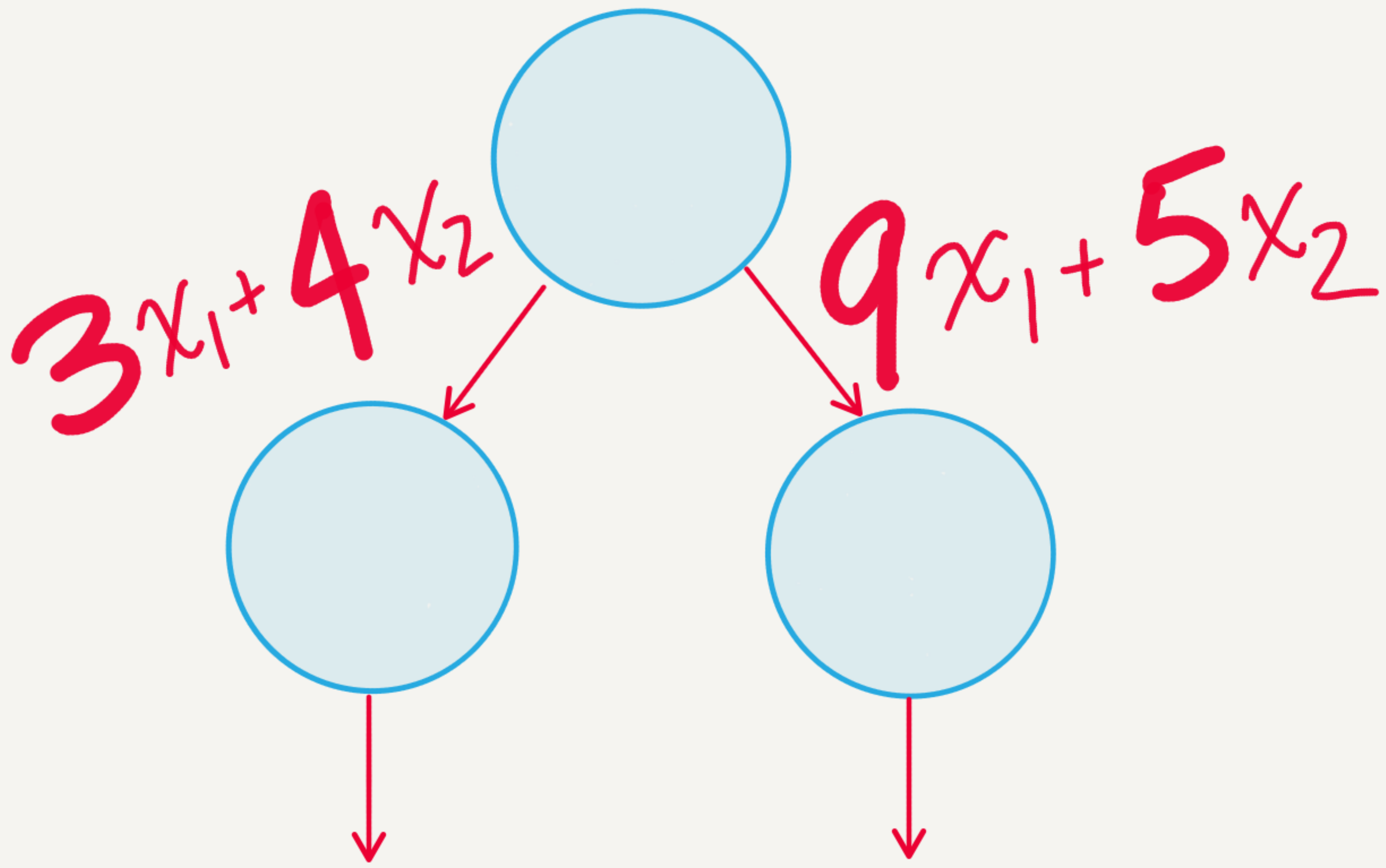
Achievable Throught of 2!

+ Exclusive OR

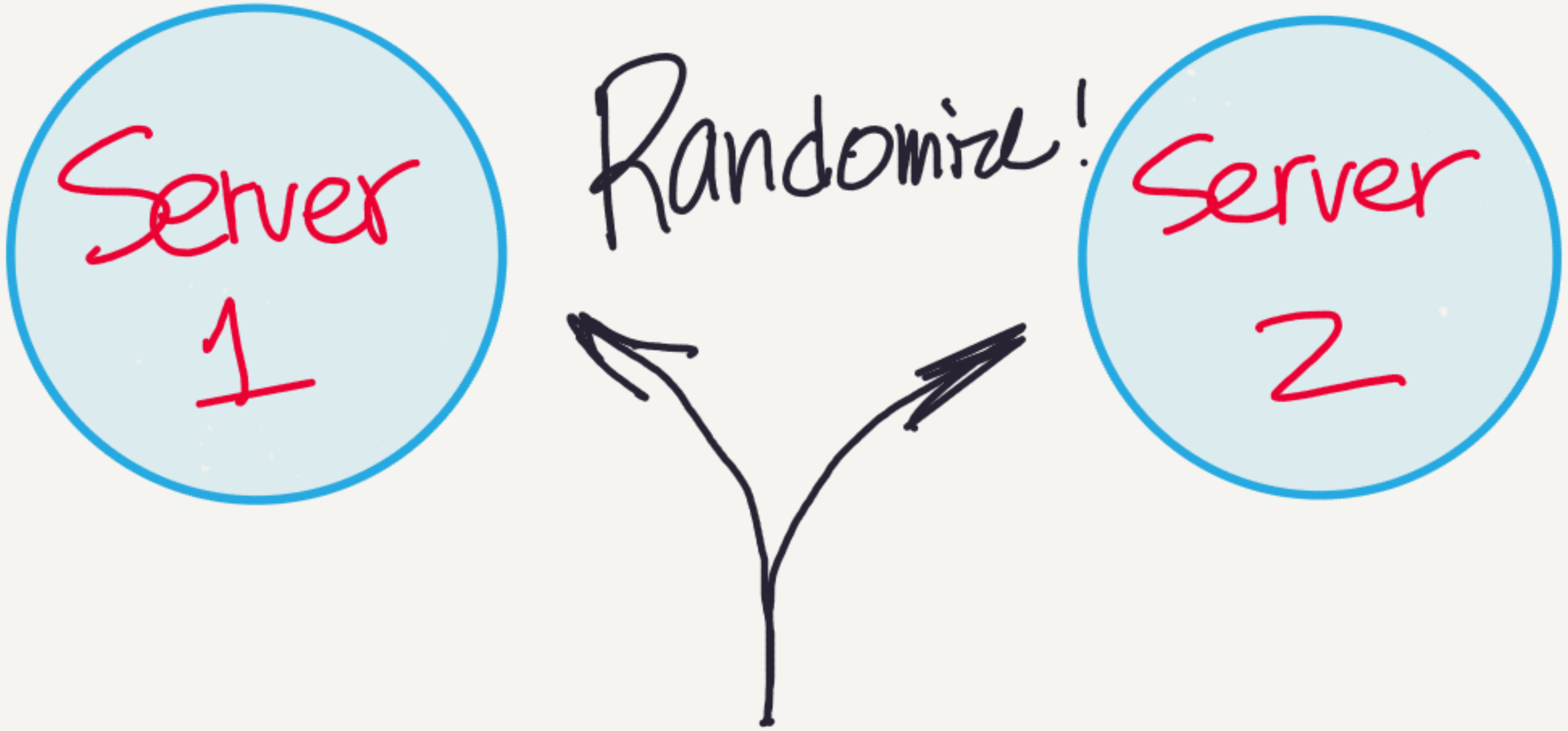
But how do we compute the coefficients to achieve optimality?



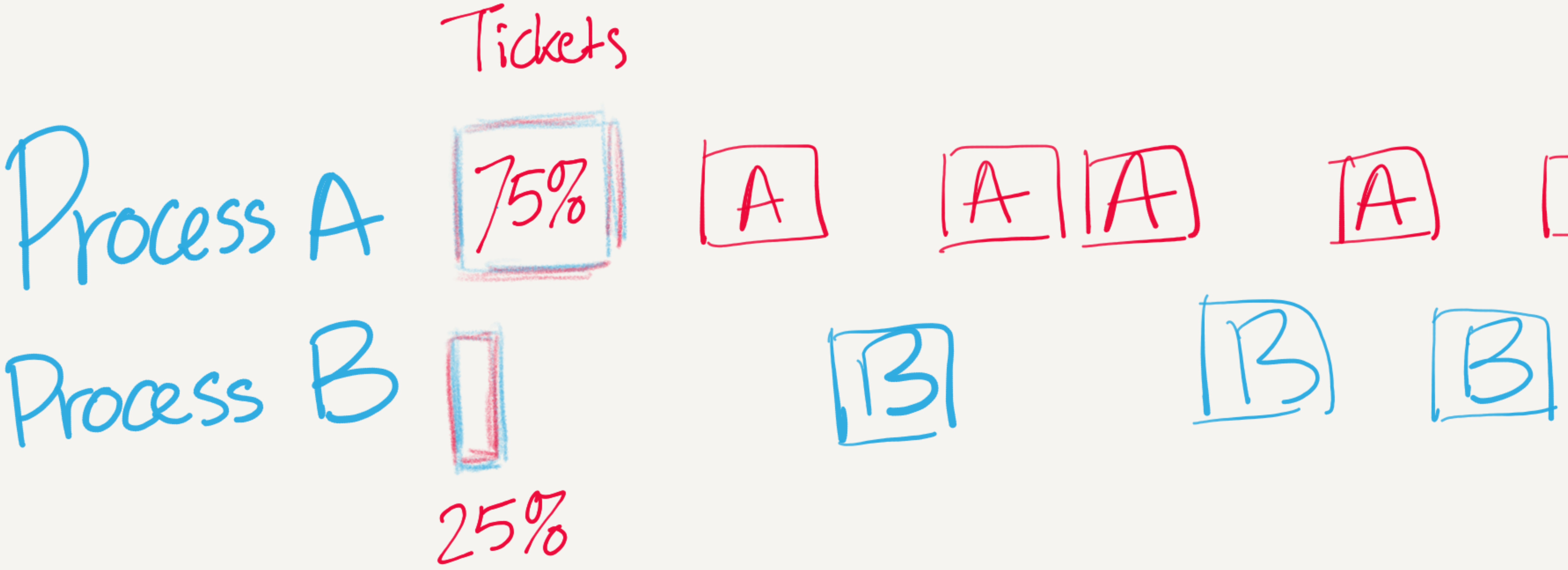
Random network coding: generate coefficients randomly



Randomized load balancing



Lottery CPU Scheduling

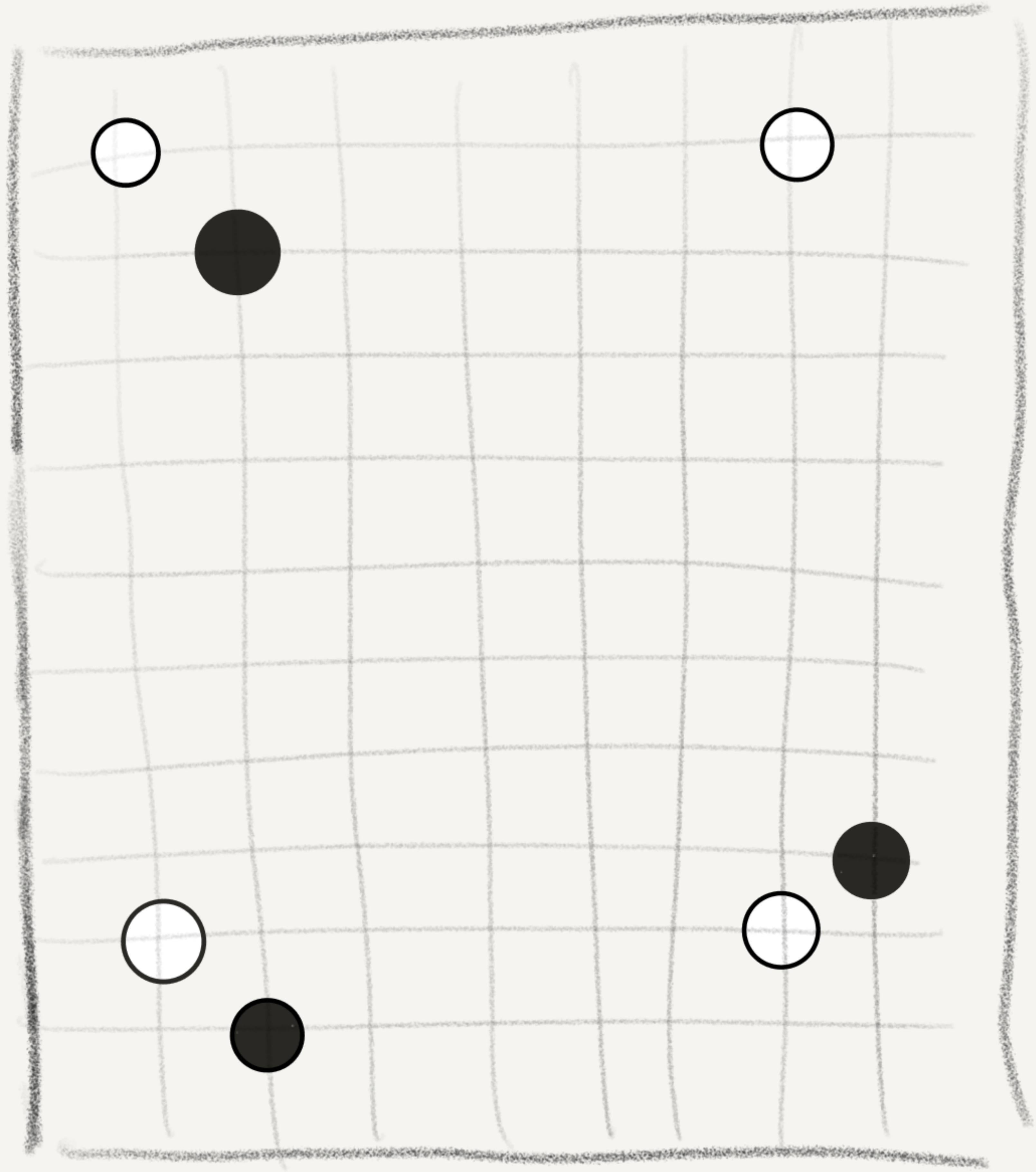


Randomized algorithms may avoid maintaining states and improve performance

QuickSort: O(n *log* n) time if we select pivot elements uniformly at random

The **Monte Carlo method** uses randomness for deterministic problems that are difficult to solve

The advent of **Monte Carlo Tree Search** in 2006 dramatically improves the ability for computers to play Go



Many other real-world examples

Resolving contention in broadcast medium by backing off randomly

Randomized routing

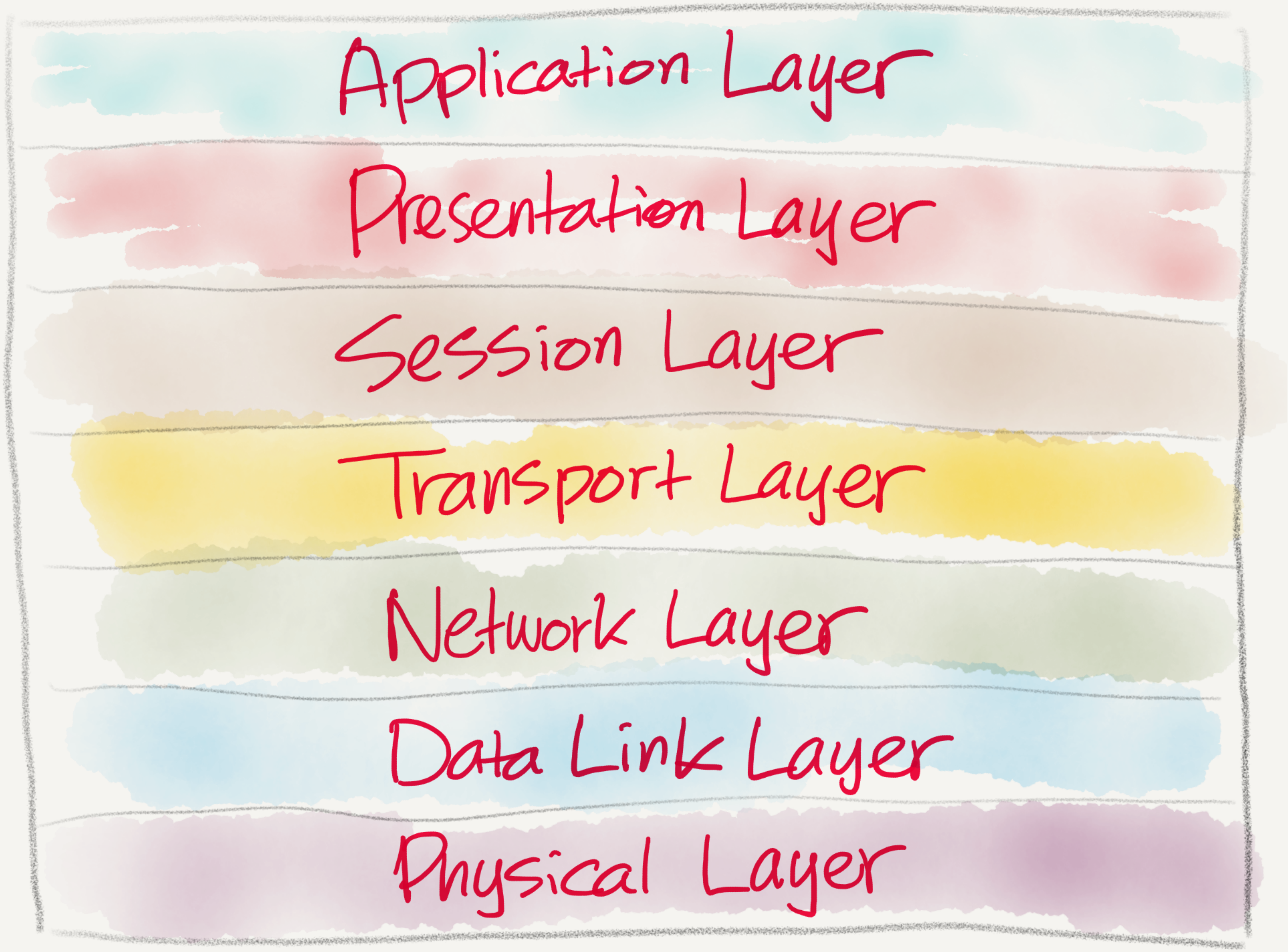
So, randomize and avoid maintaining states as much as possible

But what if I have to remember and maintain *some* states?

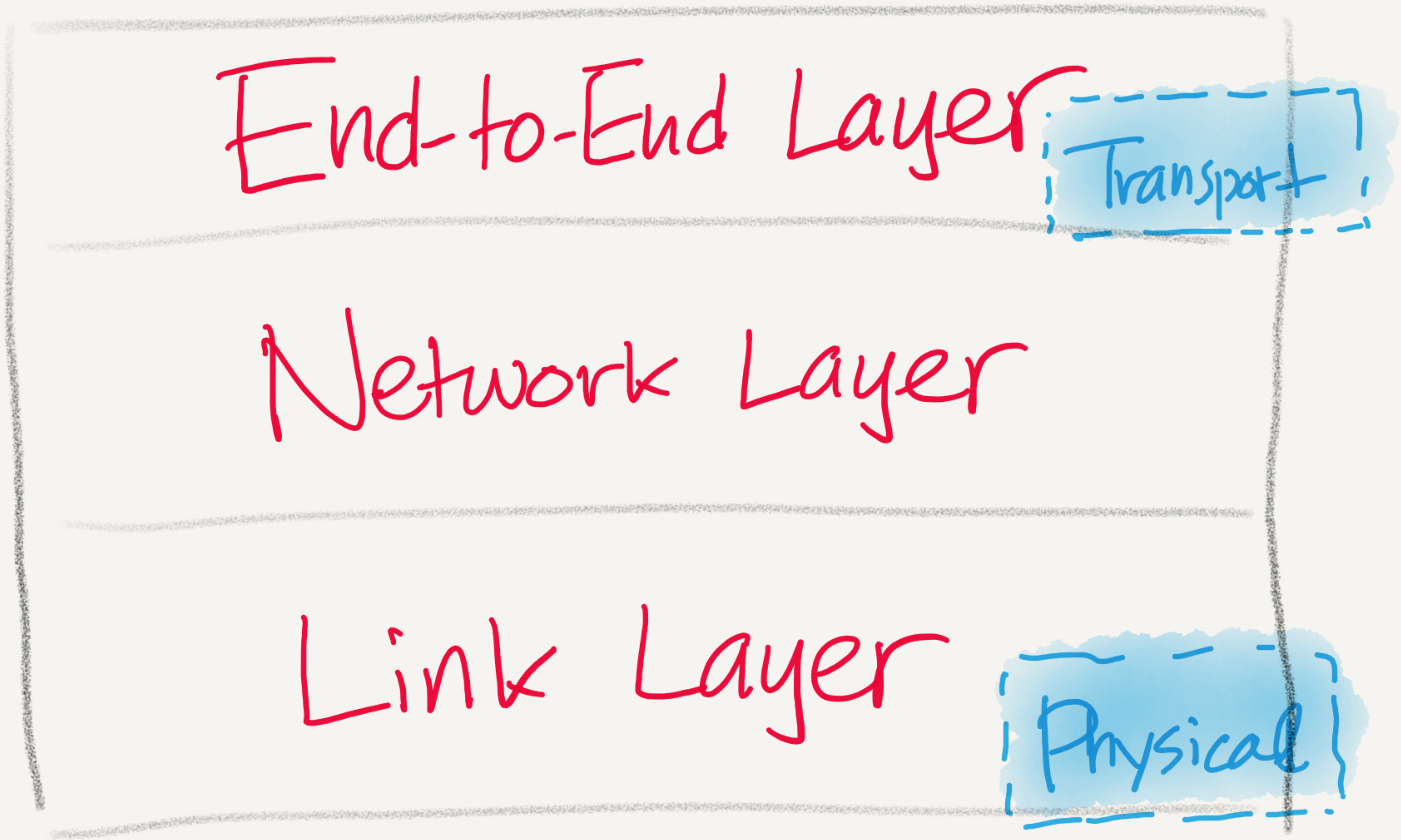
Use **soft states**, which expire after some time without a refresh

Idea #4: Use layers, but no more than 3

The (In)famous ISO/OSI Model



What you really need is...



Ch. 7, Principles of Computer System Design: An Introduction
J. Salzer, M. Frans Kaashoek, MIT

But why?

How do you draw a line between functions?

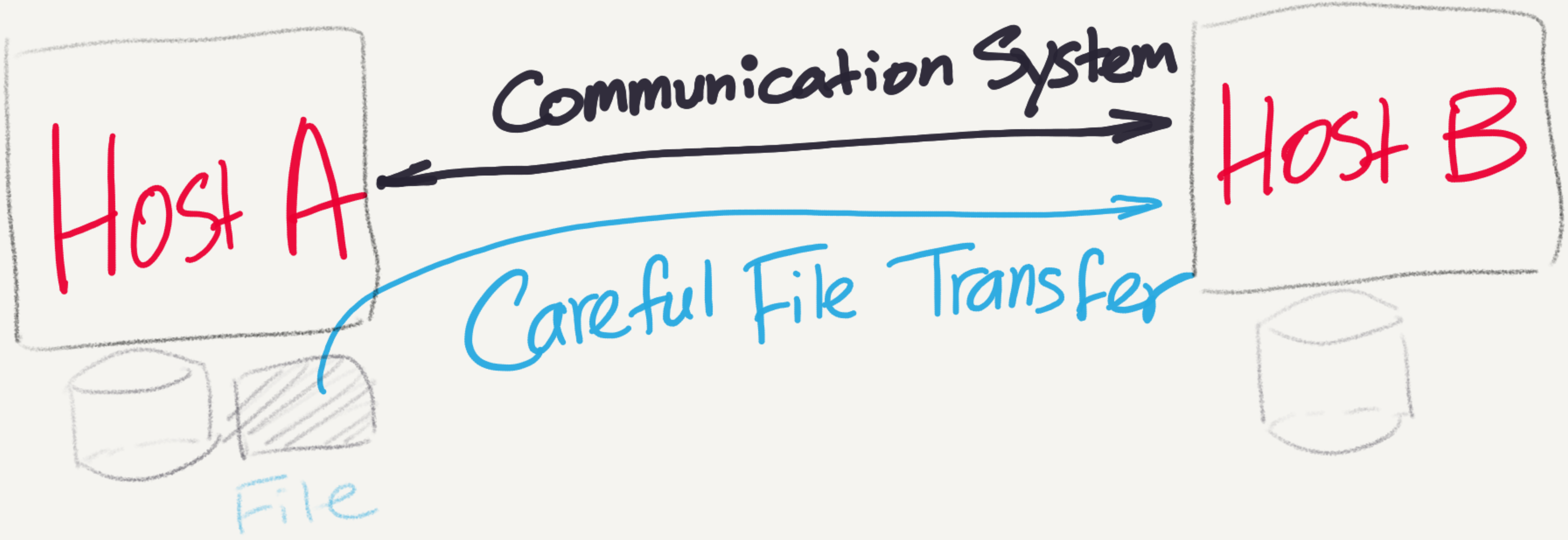
The function in question can completely and correctly be implemented only with the knowledge and help of the **application**, standing at the end points.

Sometimes, an incomplete version of the function provided by the lower layers may be useful as a **performance enhancement.**

End-to-End Arguments in System Design, 1984

J. Salzer, D. Reed and D. Clark

The end-to-end argument: example



The end-to-end argument can be made elsewhere

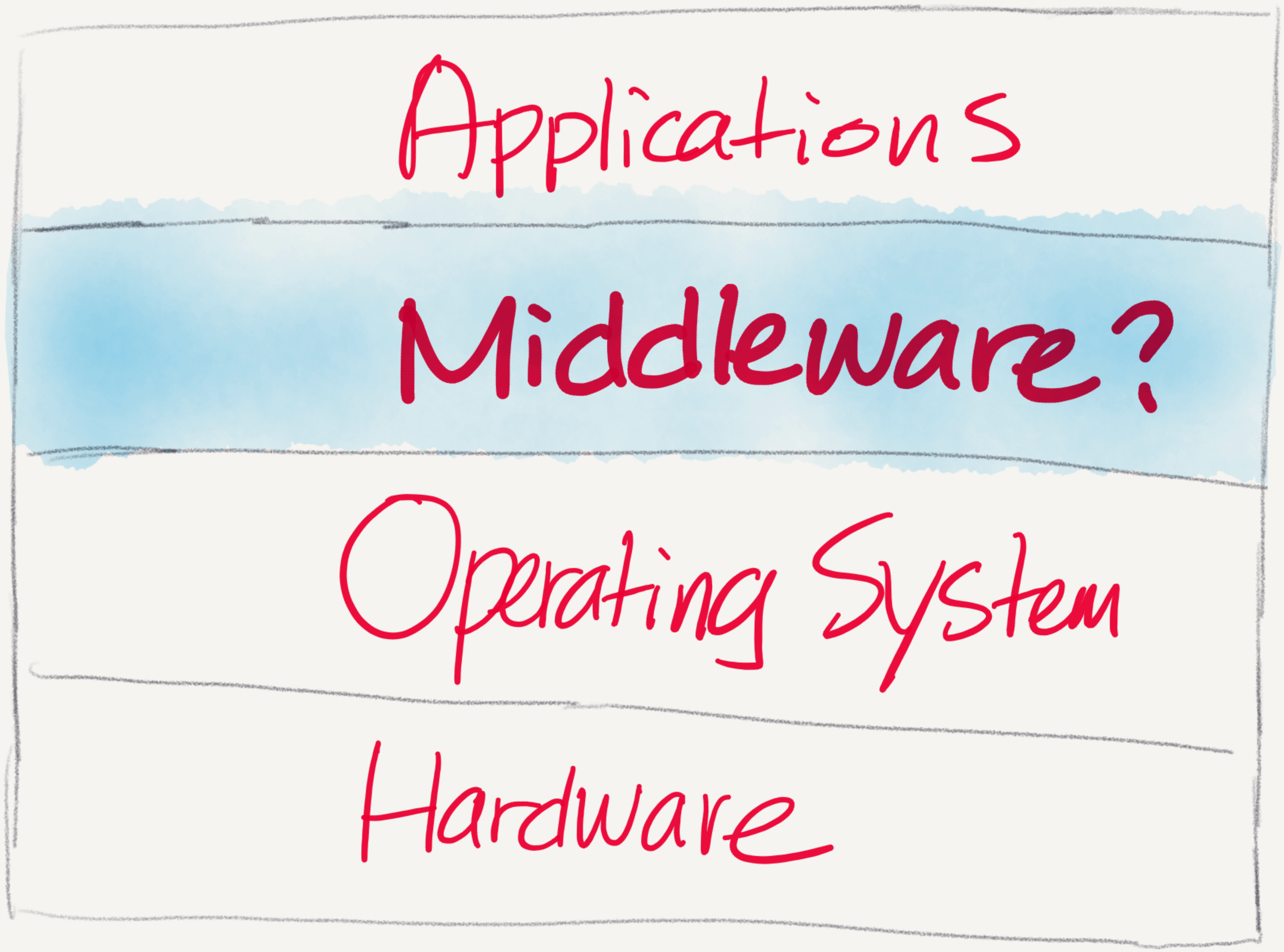
Error control: best done in the applications

Encryption (Branstad, 1973): Diffie and Hellman; Needham and Schroeder

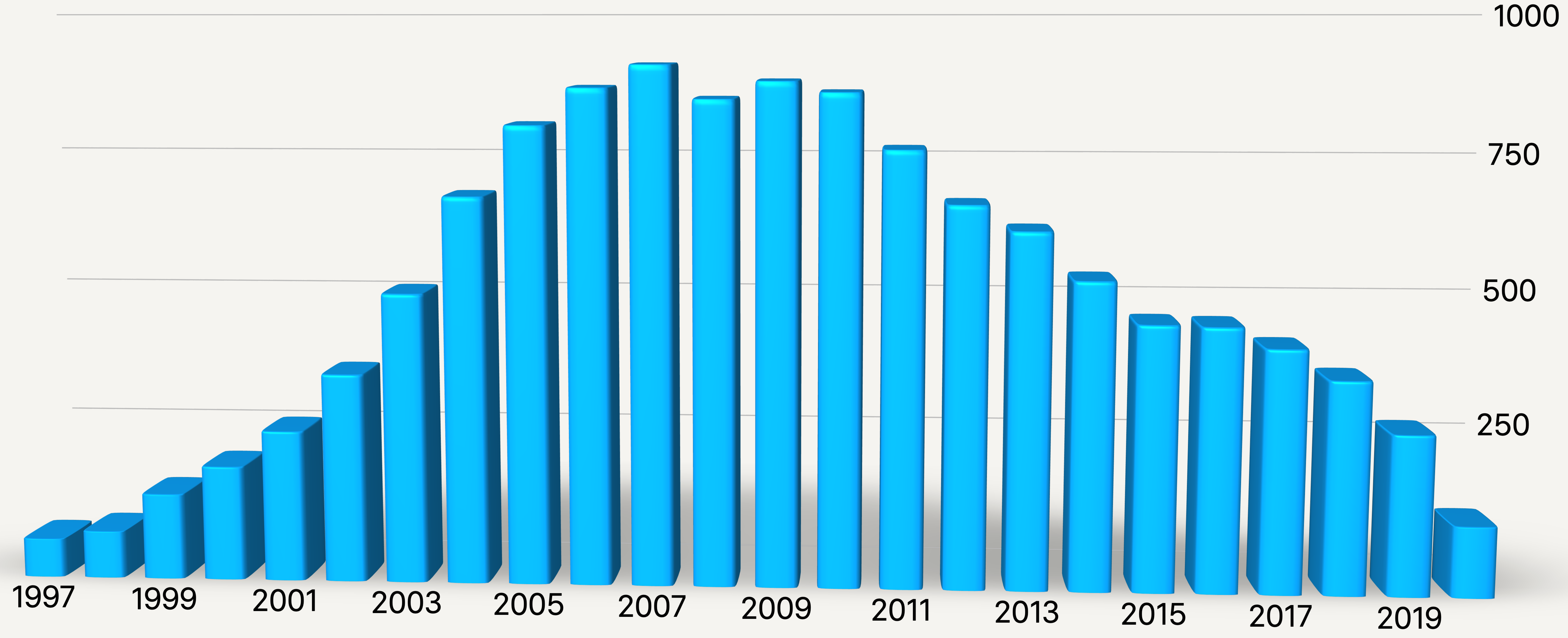
Two-phase commit protocols: do not depend on reliability, FIFO sequencing, or duplicate suppression for their correctness

RISC architectures: no need to anticipate application requirements for an esoteric feature

Layers in a computer system: 3 or 4?

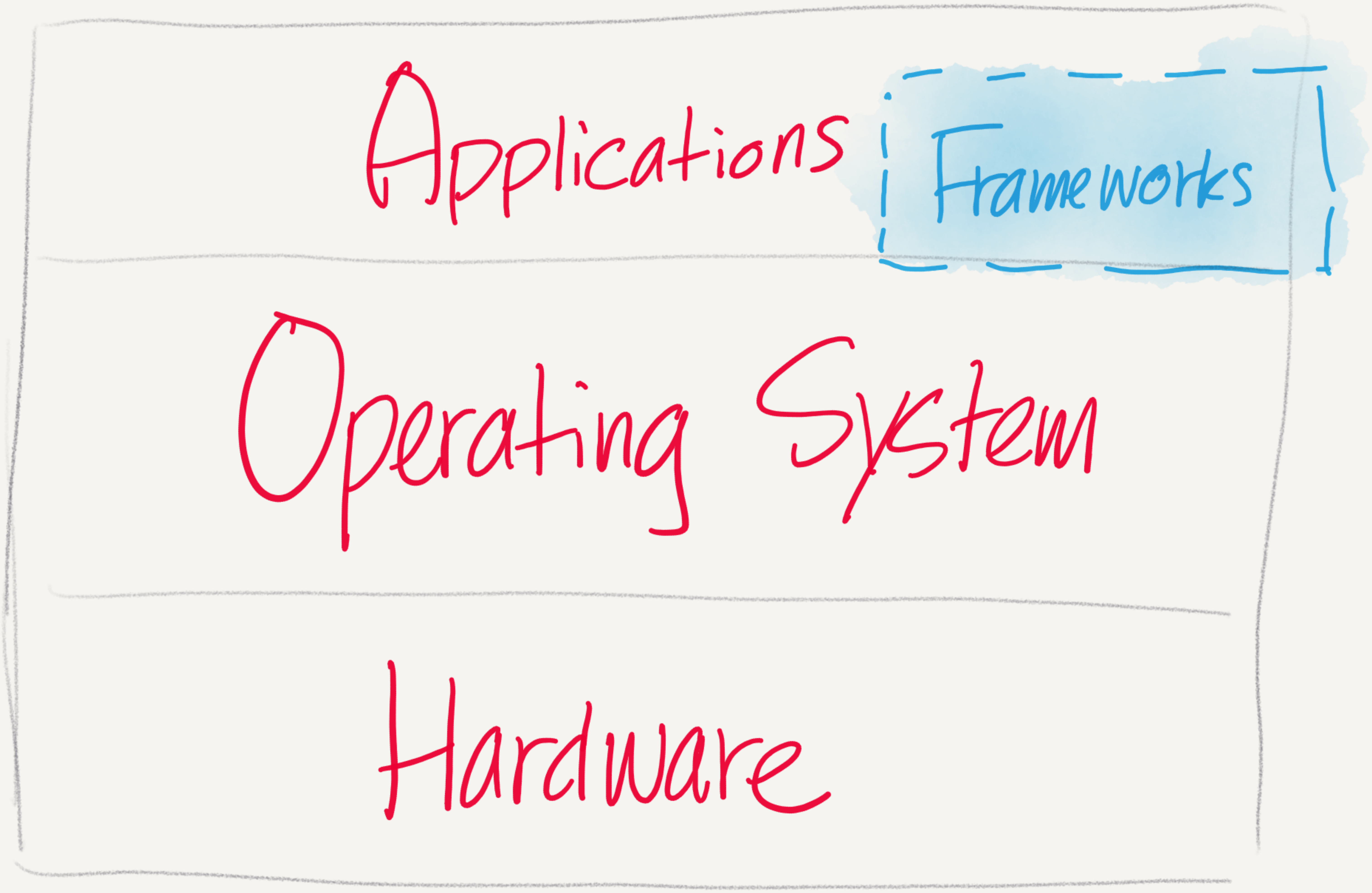


The rise and fall of middleware research



Source: Google Scholar search, “middleware” in titles only

Layers in a computer system



Corollary to idea #4: Use software, unless performance is not good enough

Case in point: machine learning

Software on multiple CPUs

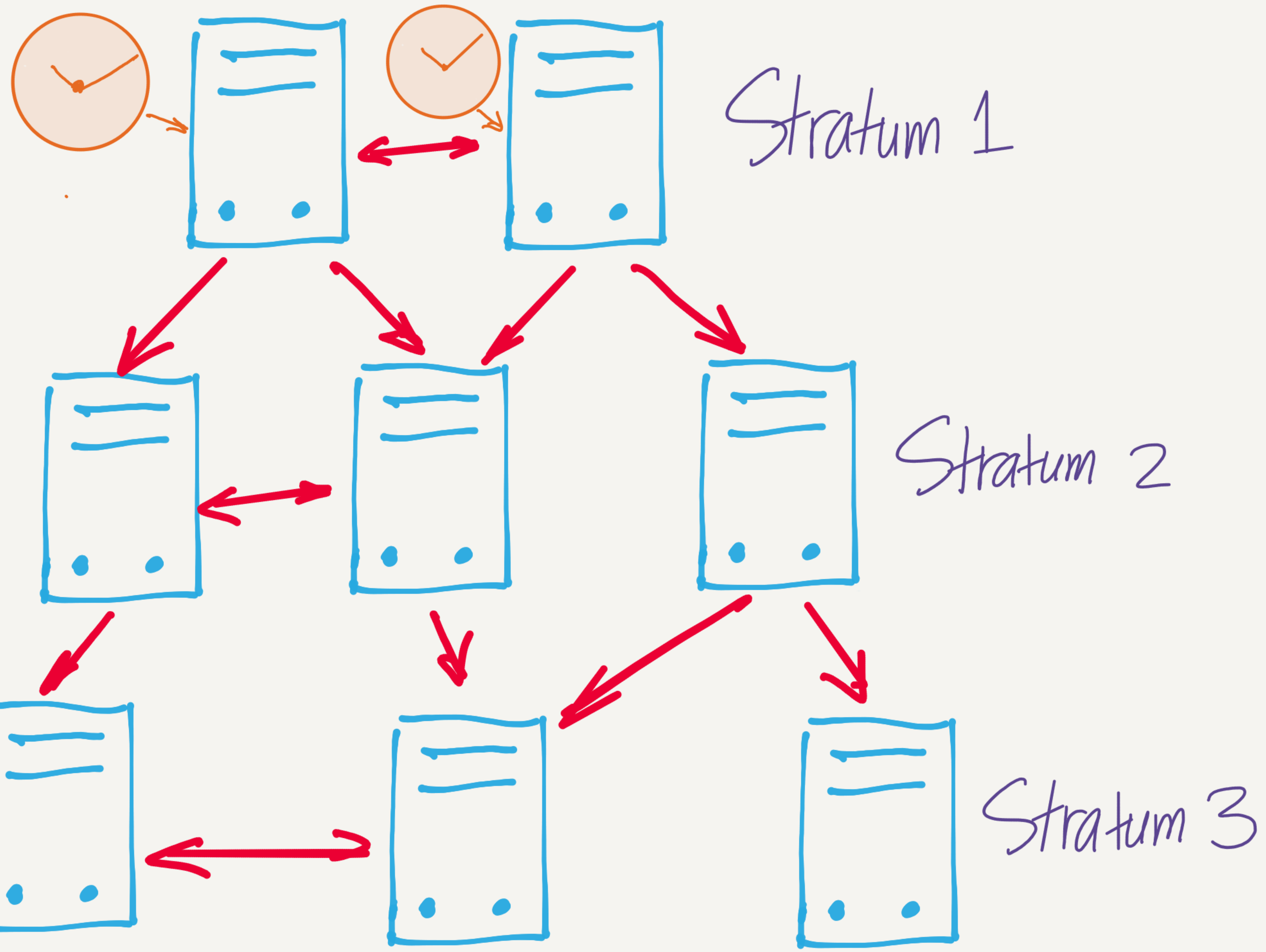
Software on multiple GPUs

Software on multiple TPUs

Implement more functions in hardware, but only when they definitely help improve the performance of the applications we run.

Idea #5: Use hierarchies to be more scalable, but no more than 3

Network Time Protocol (NTP)



Other examples using hierarchies

Domain name system (DNS): root servers — organization servers — authoritative name servers

Certificate authorities (CA): root certificates — intermediate certificates — certificates

Web service: original servers — edge servers in CDNs — clients

Why do we use only 3 levels in the hierarchical design?

Well, 3 is scalable enough based on real-world experiences — the complexity from more levels is not necessary.

Hierarchical designs are conceptually easy, but difficult to implement correctly

Ideas towards implementing hierarchies

Cache aggressively in leaf nodes to avoid congesting the root

Nodes in each hierarchy should depend only on their parents for proper execution

Leaf-to-leaf communication is expensive to support

- peer-to-peer vs. client-server?

Peer-to-Peer vs. Client-Server?

Permissionless Blockchain Cloud Computing

P~~eer-to-Peer~~ vs. ~~Client-Server?~~

Permissionless Blockchain

Cloud Computing

~~Peer-to-Peer~~ vs. ~~Client-Server~~?

Slow

Fast

Traditional Routers Software-Defined Networking

~~Peer-to-Peer~~ vs. ~~Client-Server~~?

Traditional Routers Software-Defined Networking

~~Peer-to-Peer~~ vs. ~~Client-Server~~?

Fault-tolerant

Central point of failure

Corollary to idea #5: Use the cloud as much as possible, only use peer-to-peer when necessary

#1: Multiplexing and virtualization: use statistical knowledge of the users

#2: Batching: trading response time for throughput

#3: Randomize and avoid maintaining states as much as possible

#4: Use layers, but no more than 3

#5: Use hierarchies to be more scalable, but no more than 3

#6: Use pipelines

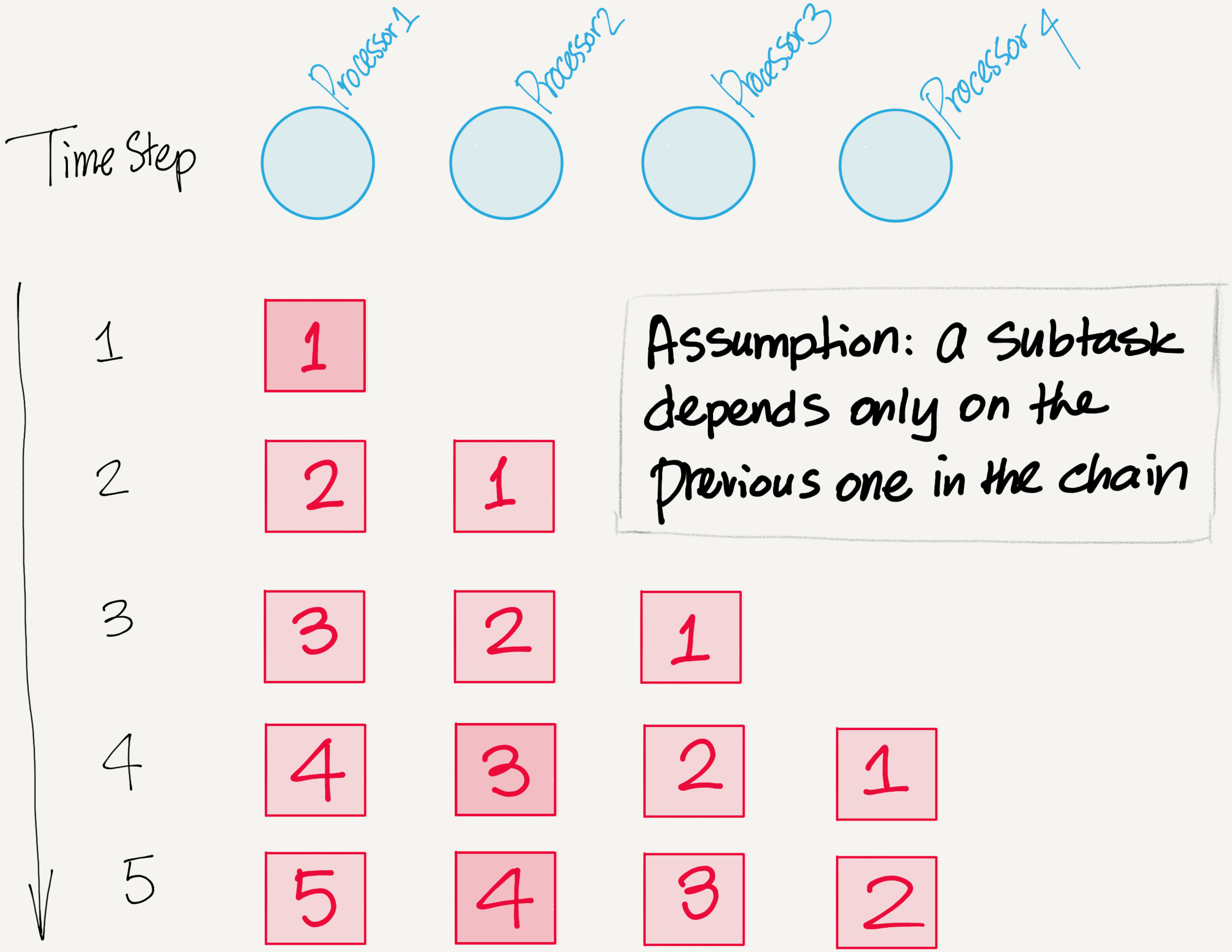
Idea #6: Use pipelines

Traditional parallelism with more processors requires breaking up a task into multiple independent subtasks

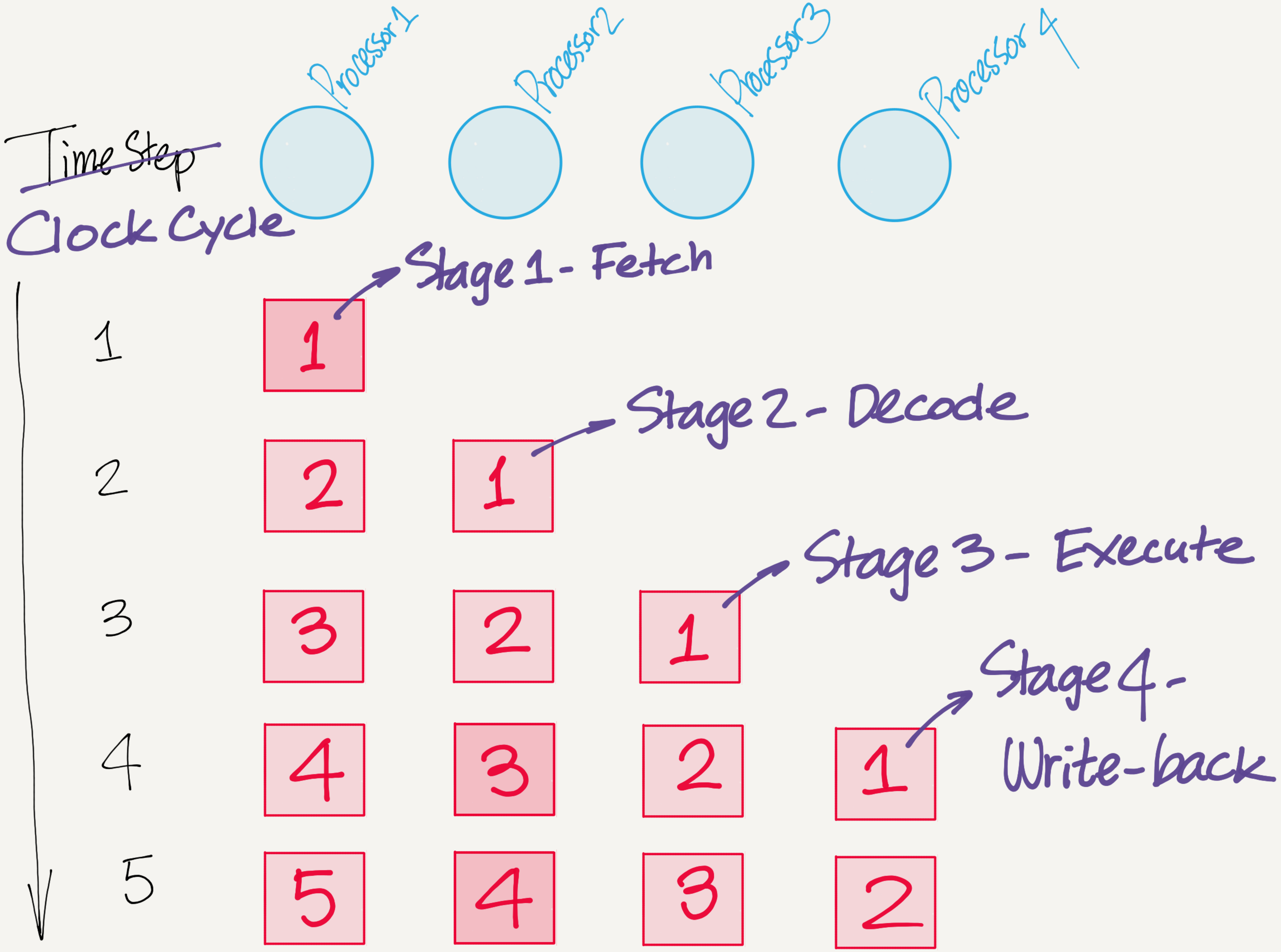
Example: downloading images into a web browser

But what if the subtasks are dependent — one cannot start until another ends?

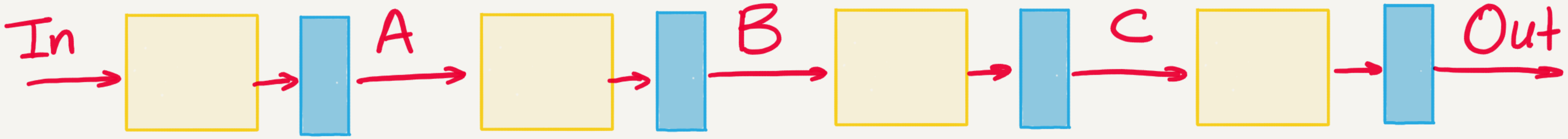
Pipelining parallelism increases throughput

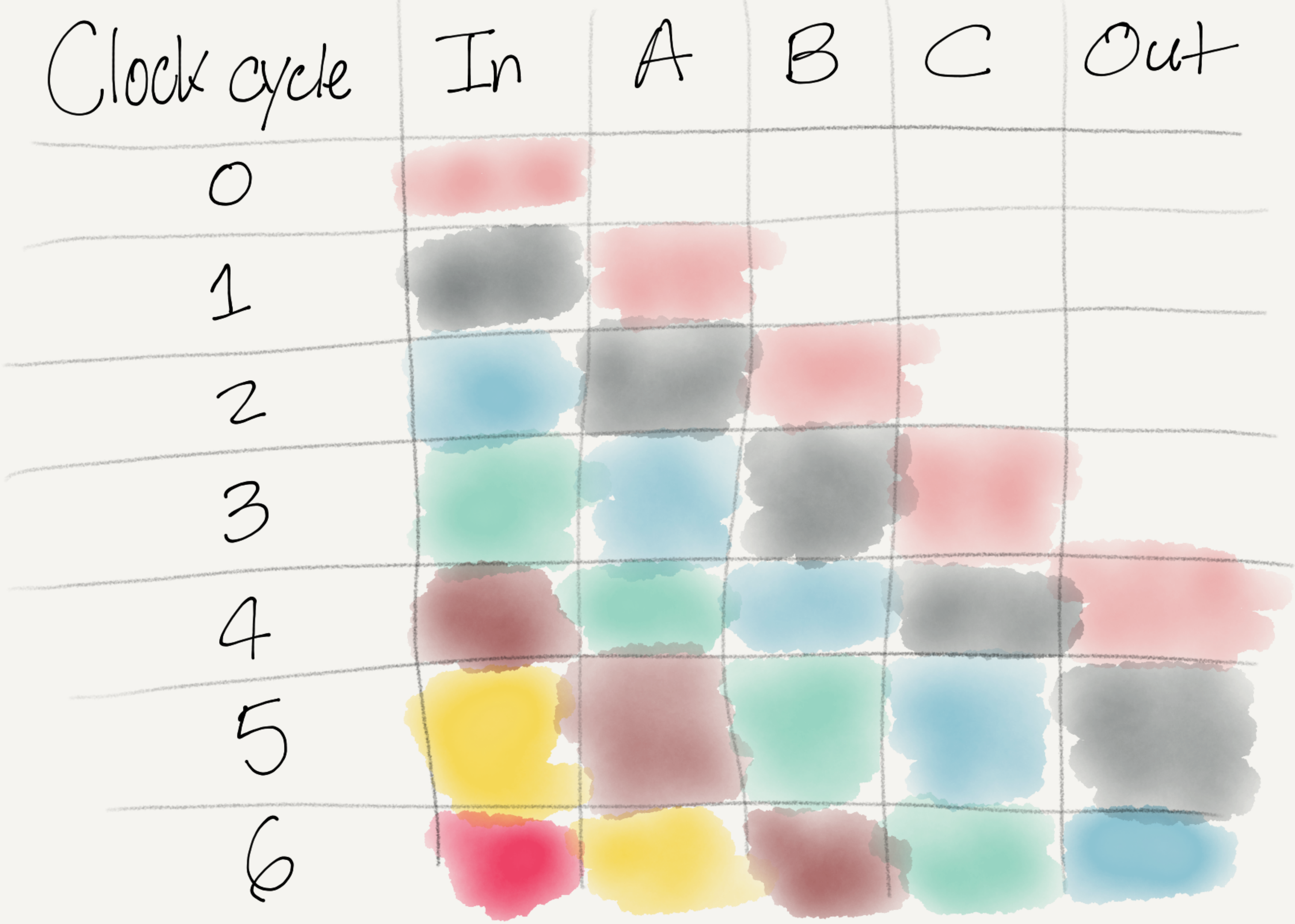


Instruction pipelining with a 4-stage pipeline

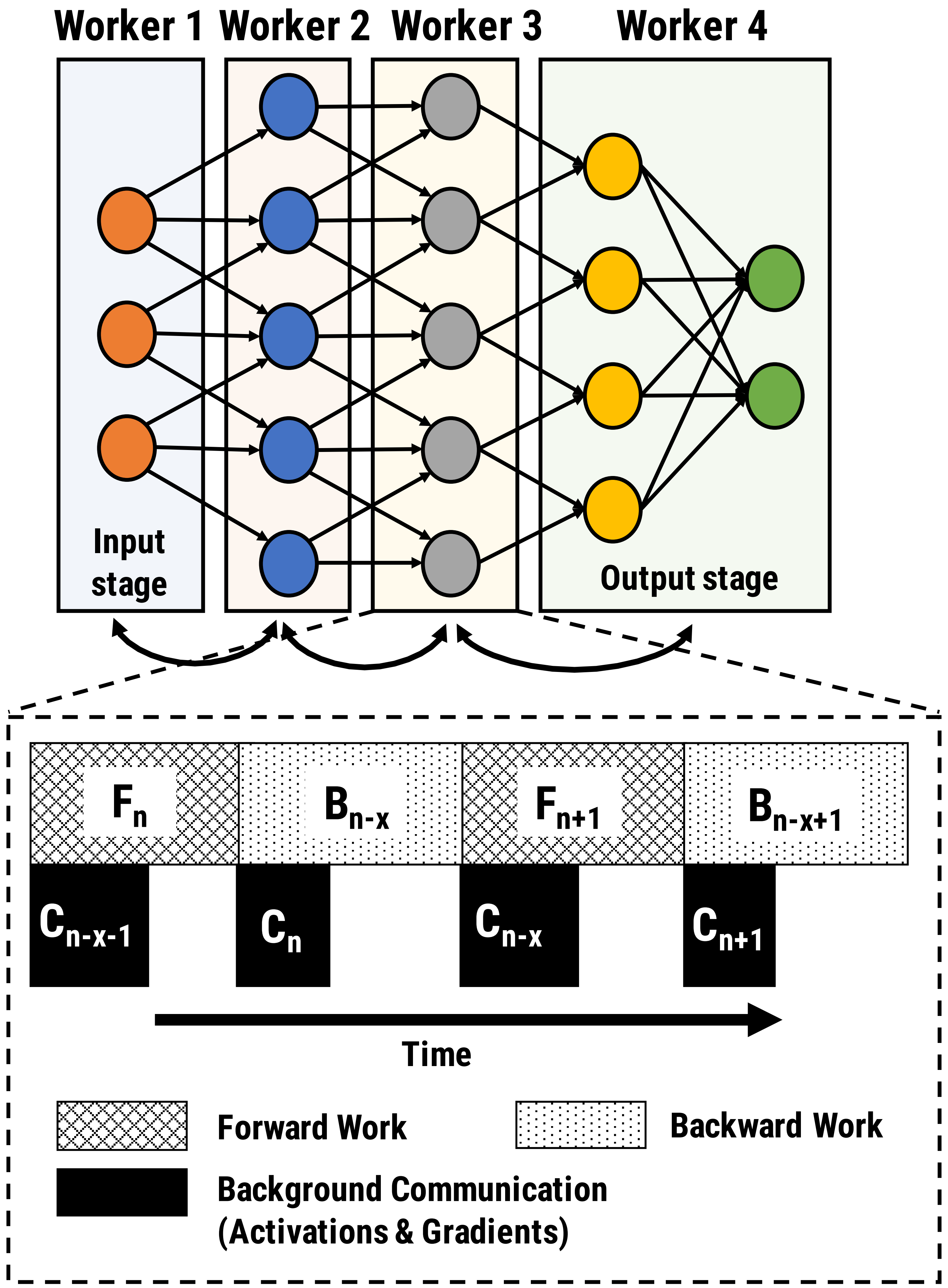


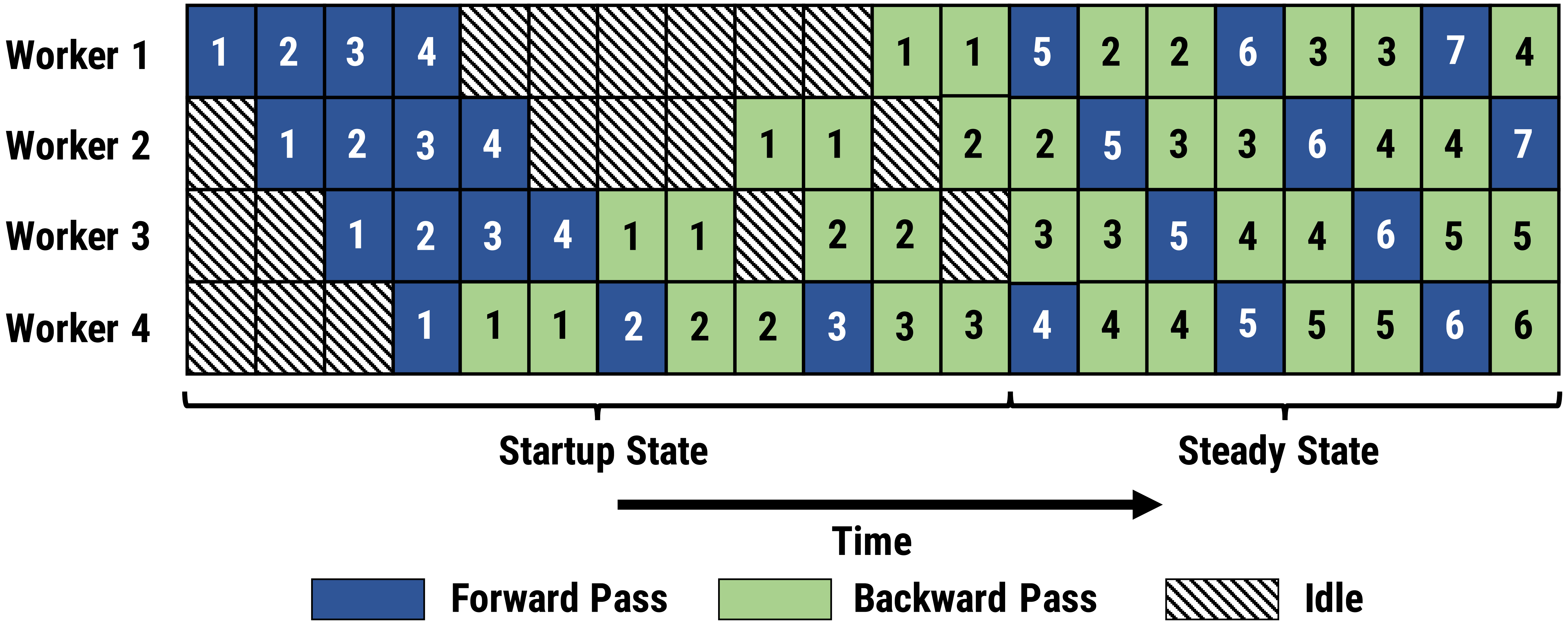
A pipelined adder in a cloud or edge TPU





Pipeline parallelism for DNN training





Narayanan, et al., "PipeDream: Generalized Pipeline Parallelism for DNN Training," ACM SOSP 2019.

Fundamentally, system design is about trade-offs

Trade-offs between more abundant resources and scarce resources at the bottleneck

Trade-offs in system design

Multiplexing and virtualization: more time, more space consumed, but costs less

Parallelism: more computation, less time to complete

Batching: trading response times for system throughput

Reading: Keshav 6.1 — 6.5