Special Topics

Sam Kumar

CS 162: Operating Systems and System Programming

Lecture 27

https://inst.eecs.berkeley.edu/~cs162/su20

Read: None

Everything in this Lecture is Optional

- Won't be on the final

- Topics we'll look at today:
  - More distributed systems (consensus, Byzantine faults)
  - Containers and orchestration
  - Security

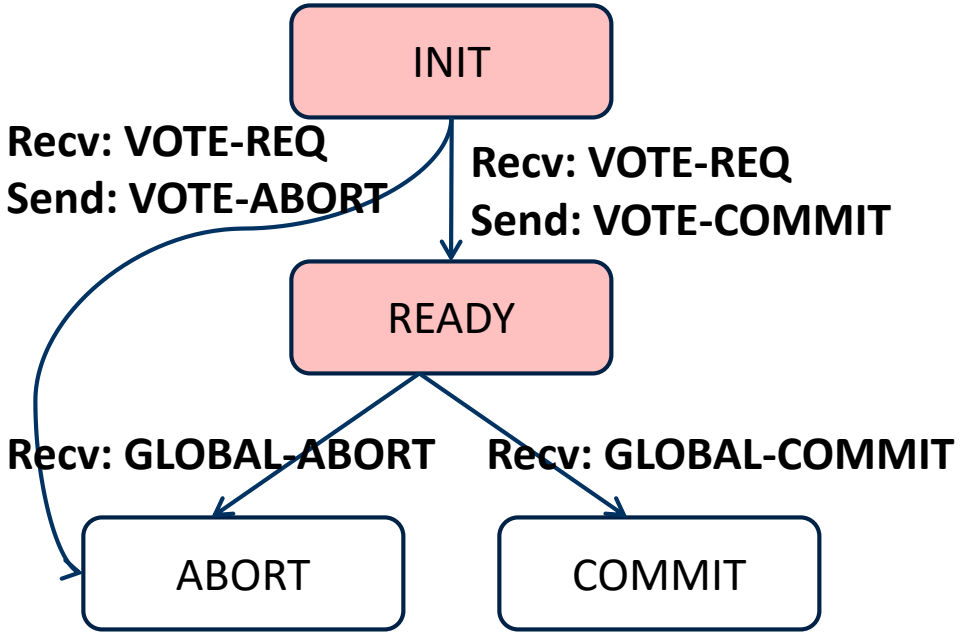
- Tomorrow, we will:
  - Cover Mobile OS
  - Perhaps some techniques for fast address translation
  - Wrap up the class

Recall: Distributed Consensus

- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values
- Distributed Decision Making
  - Choose between “true” and “false”
  - Or choose between “commit” and “abort”
- Equally important (but often forgotten!): make it durable!
  - How do we make sure that decisions cannot be forgotten?

Recall: Dealing with Coordinator Failure in 2PC

- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort
- Worker waits for GLOBAL message in READY
  - Workers must BLOCK waiting for coordinator to recover
  - Workers could try to consult each other
    - If one of them in COMMIT or ABORT state, then they all know the result
    - In another worker is still in INIT, it is safe to ABORT
    - If all of them are in the READY state, then they must still BLOCK
  - What if coordinator and worker both fail?
    - Workers can consult each other, but can't come to any decision
    - This motivates **Three-Phase Commit (3PC)**



Distributed Consensus

- 2PC (and 3PC) make a decentralized decision
  - E.g., changing the value of a key among all replicas for the key

- But they are hardly the only solutions to this problem!

More Robust Agreement…

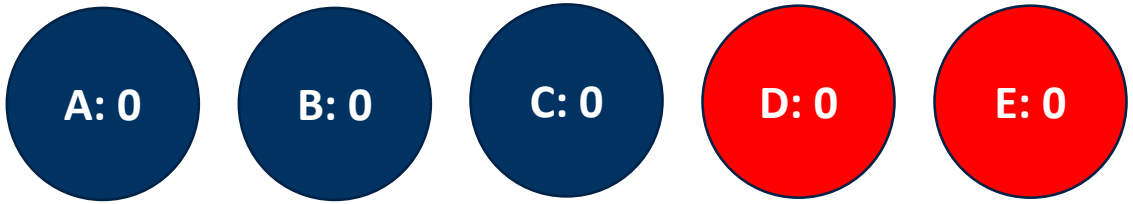
- Is there protocol for distributed consensus that can make progress without depending on any *particular node* being available?

- Yes. **Paxos** and **Raft** are consensus algorithms that can make progress as long as a *majority* of nodes are available

Why a Majority?

- Key property: Overlap
  - Any two majorities have at least one node in common

- Suppose we use transactions to track a value, initially 0



- We run transaction "+2" while D, E are down

Why a Majority?

- Now D, E come back up, and A, B go down

- Our overlap in this case is C
  - Guaranteed by choice of majority
- Overlap prevents us from losing transactions
  - Means every node is responsible for resending missed updates
  - Not just the value, but the transaction ID so we know we're behind

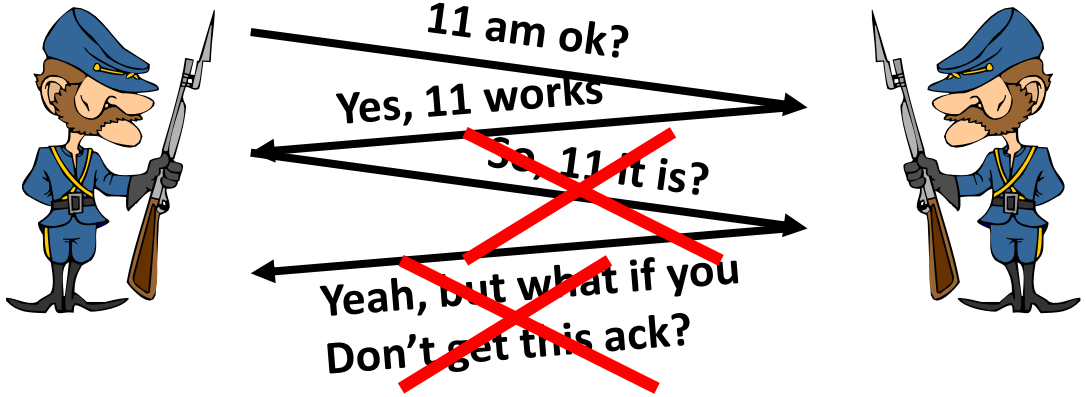
Beyond Fail-Stop Failures

- What if nodes don't just stop talking when they fail?
- What if they send incorrect information?
- Or what if nodes are actively malicious?

- This is called the **Byzantine Failure Model**

Recall: The Two Generals' Problem

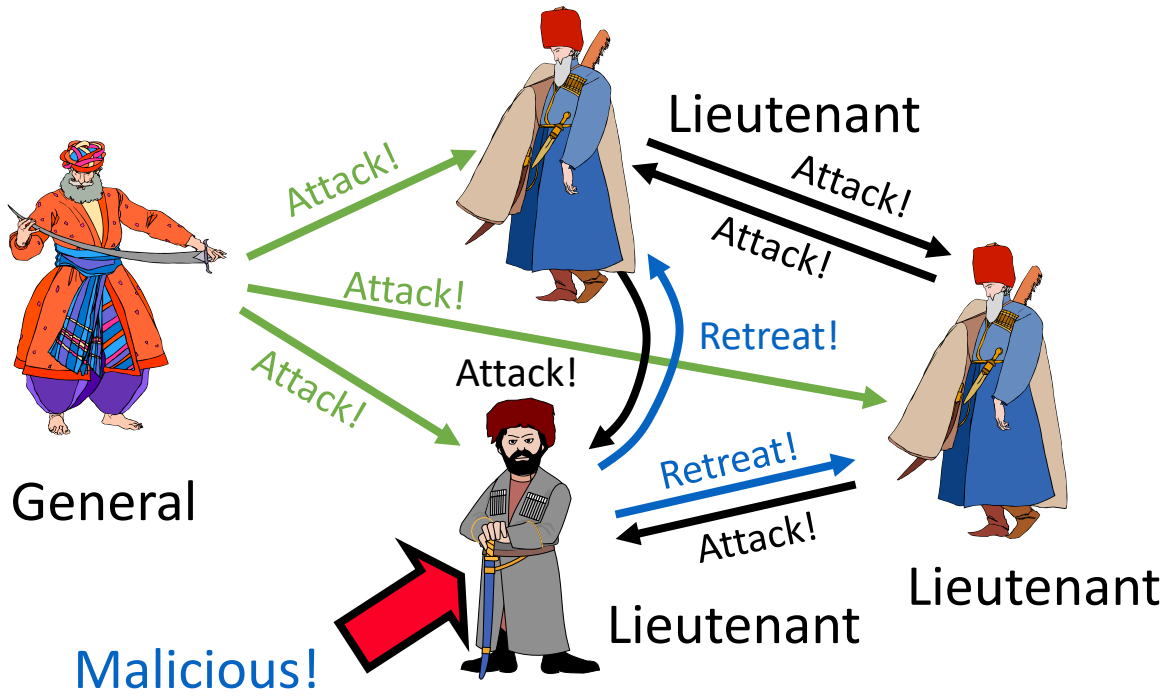
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!
- So, clearly, we need something other than simultaneity!

Byzantine Generals' Problem

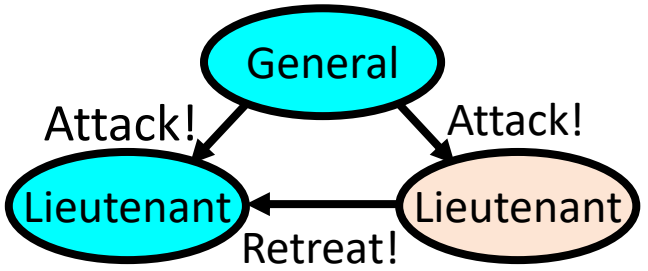
- Byazantine General's Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (f) want chaos
- Want the following Integrity Constraints to apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

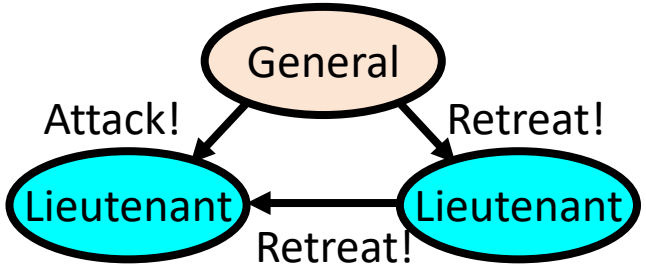


Byzantine Generals' Problem

- Cannot solve with $n = 3$ because one malicious player can mess up things

- With f faults, need $n \geq 3f + 1$ to solve problem
- Various algorithms to solve this
  - e.g., PBFT (Castro and Liskov, 1999)





CAP Theorem

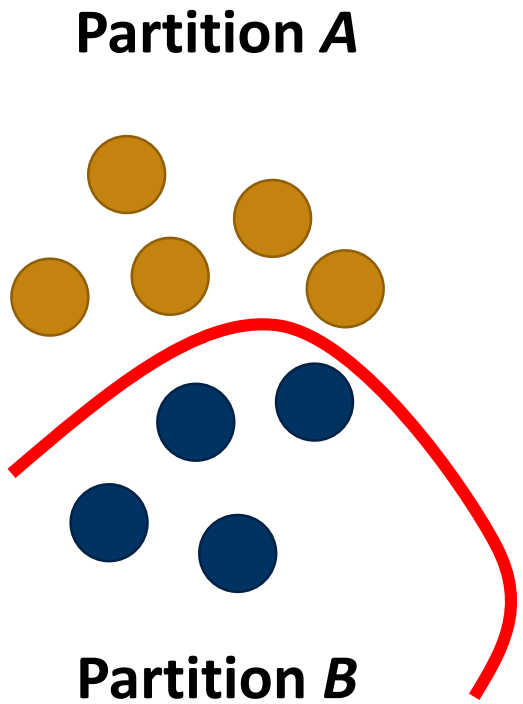
- Originally proposed by Eric Brewer (Berkeley)

1. **C**onsistency – changes appear to everyone in same order (linearizability)
2. **A**vailability – *any* node in the system can process requests
3. **P**artition Tolerance – system continues to work even when one part of network can't communicate with the other

- Impossible to achieve all 3 at the same time (pick two)

CAP Theorem

- If a network partition occurs, what do we do?
- At least one partition stops processing requests so that its history doesn't diverge from other partitions
  - Prefers consistency (CP)
- All partitions keep processing requests, and their histories may diverge; reconcile this later
  - Prefers availability (AP)



Containers and Orchestration

Are Virtual Machine Monitors the "True OS"?

- There was a trend to move "shared drivers" into a Guest domain
- VMMs evolving to look more like an OS
- Stronger isolation can be a plus
- What do you think?

Are Virtual Machine Monitors Microkernels Done Right?

Steven Hand, Andrew Warfield, Keir Fraser,
Evangelos Kotsovinos, Dan Magenheimer†
University of Cambridge Computer Laboratory
† HP Labs, Fort Collins, USA

1 Introduction

At the last HotOS, Mendel Rosenblum gave an 'outrageous' opinion that the academic obsession with microkernels during the past two decades produced many publications but little impact. He argued that virtual machine monitors (VMMs) had had considerably more practical uptake, despite—or perhaps due to—being principally developed by industry.

In this paper, we investigate this claim in light of our experiences in developing the Xen [1] virtual machine monitor. We argue that modern VMMs present a practical platform which allows the development and deployment of innovative systems research: in essence, VMMs are microkernels done right.

We first compare and contrast the architectural purity of microkernels with the pragmatic design of VMMs. In Section 3, we discuss several technical characteristics of microkernels that have proven, in our experience, to be incompatible with effective VMM design.

Rob Pike has irreverently suggested that "systems softdifferences: Microkernels received considerable attention from academic researchers through the eighties and nineties, while VMM research has largely been the bailiwick of industrial research.

2.1 Microkernels: Noble Idealism

The most prolific academic microkernel ever developed was probably Mach [2]. A major research project at CMU, Mach's beginnings were in the Rochester Intelligent Gateway (RIG) [3] followed by the Accent kernel [4]. The key motivation to all of these systems was that the OS be "communication oriented"; that they have rigid, message-based interfaces between system components. Many of the abstractions used in Mach and later systems appeared initially in the RIG, including that of the *port*. However, the communications orientation of these systems originally intended to allow the distribution of system components across a set of dissimilar physical hosts.

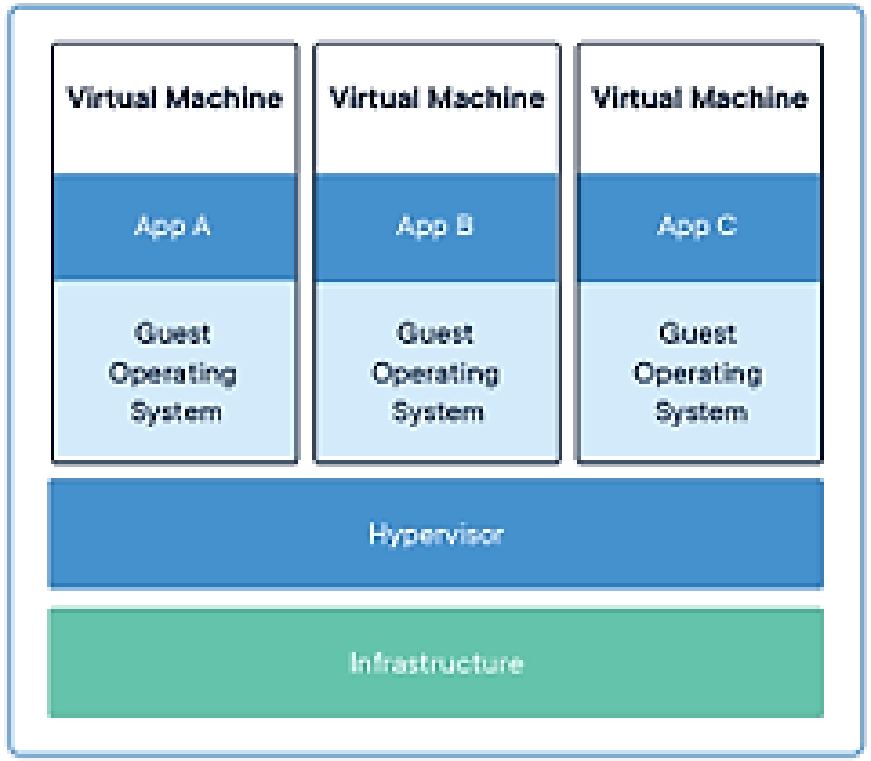
The term "microkernel" was coined in response to the

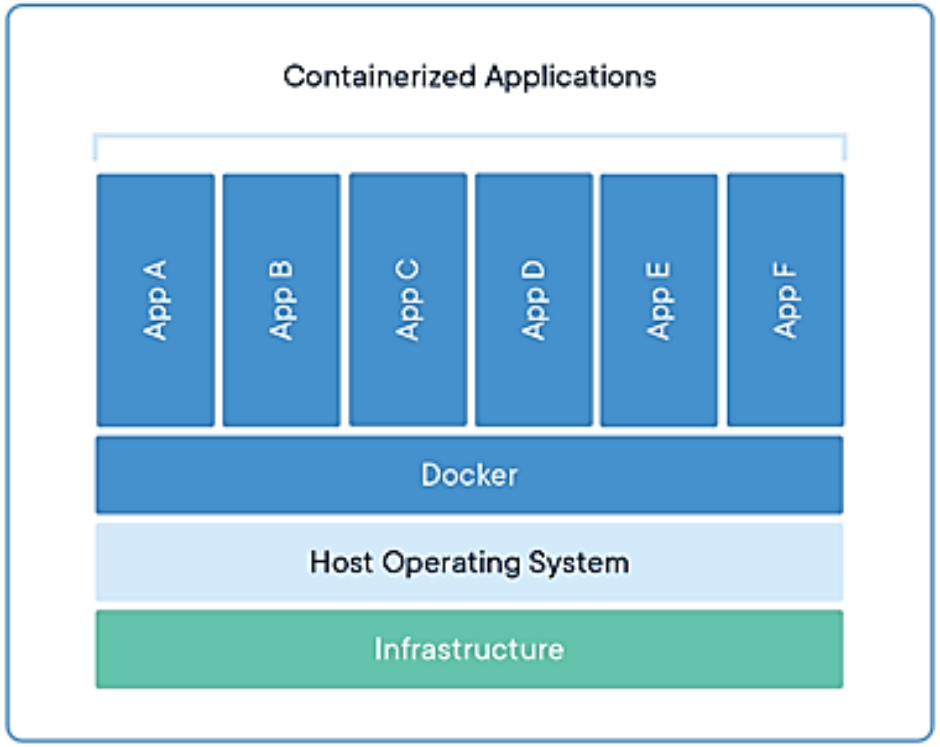
Containers

- Virtual machines: provide each guest with the illusion of its own dedicated hardware

- Containers: provide each guest with the illusion of its own dedicated **operating system**
  - Via resource isolation (cgroups)
  - Via namespace isolation
    - PID namespace
    - Network namespace
    - Filesystem…
  - With its own binaries, libraries, and dependencies

Recall: Virtualization—Execution Environments for Systems

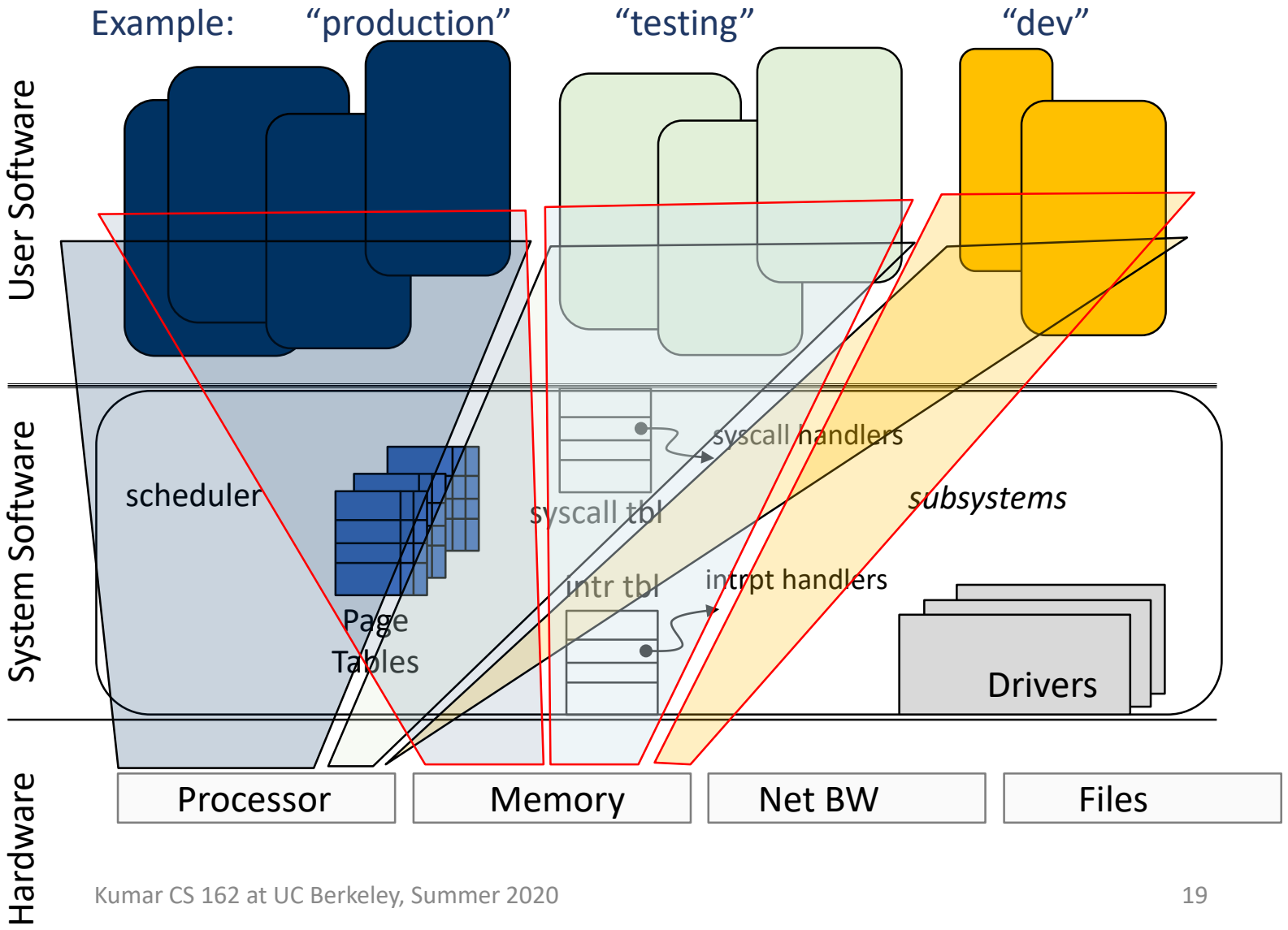




Additional layers of protection and isolation can help further manage complexity

Performance Isolation: CGroups

- Idea: provide greater performance isolation between cgroups than between processes…



CGroups

- Identify collections of processes that will be treated as a *group* for resource allocation
  - Groups can have hierarchical structure
  - i.e., a Group can be comprised of subgroups
  - Process parent-child relationship defines a hierarchy
    - Generalize this beyond `fork()`
- Set of key resource dimensions
  - Processor share (CPU), CPU set (bound to particular cores)
  - Physical memory share,
  - block IO, net priority, net class
  - Namespace (ns), i.e., containers
- Resource limiting, prioritization, accounting and control
- Containers define a collection of libraries and executables that should be a group

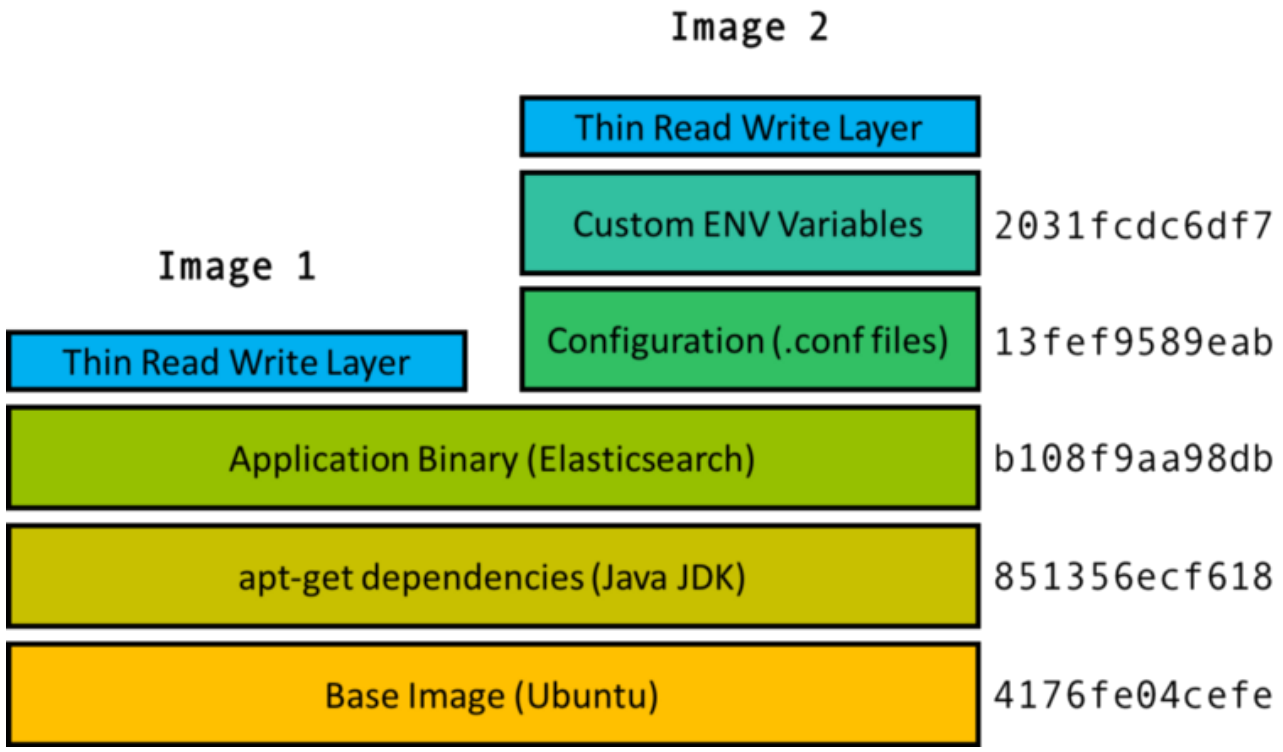
Recording and Manipulating CGroups

- Unix-based systems use `/proc` to represent information about processes
  - `/proc/<pid>` describes process <pid>
- `/proc/cgroups/*`
  - Describes what controllers are implemented
- Each cgroup controller has a directory under `/sys/fs/cgroup/<controller>`
  - `/sys/fs/cgroup/cpu/production`
  - `/sys/fs/cgroup/memory/foo/memory.limit_in_bytes`
- Kernel monitors and controls processes/threads in accordance with cgroup controllers

Docker

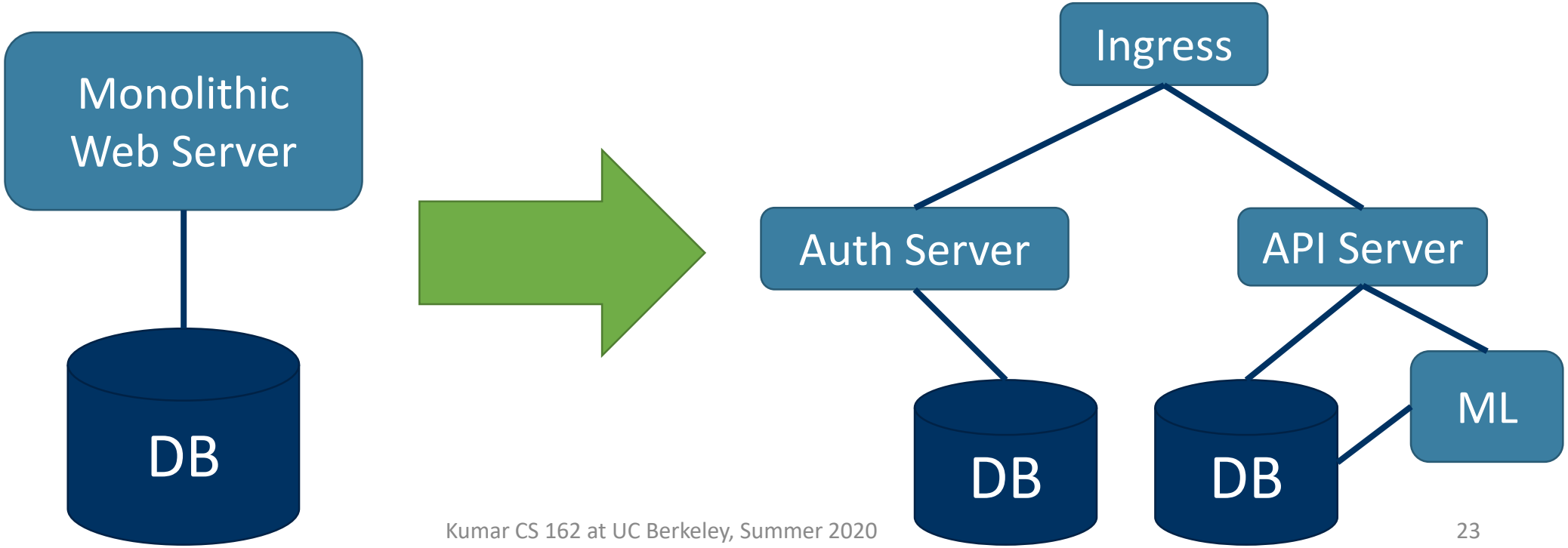
Features on top of OS cgroups:

- Images
  - Describes the starting state of a Docker container (like a snapshot that can be restarted)
- Union file system
  - Image describes file system as a sequence of *layers*, each with some files
  - Overall file system is the *union* of the layers
  - Layers can be reused across images/containers
- Tools and observability
- Container lifecycle management



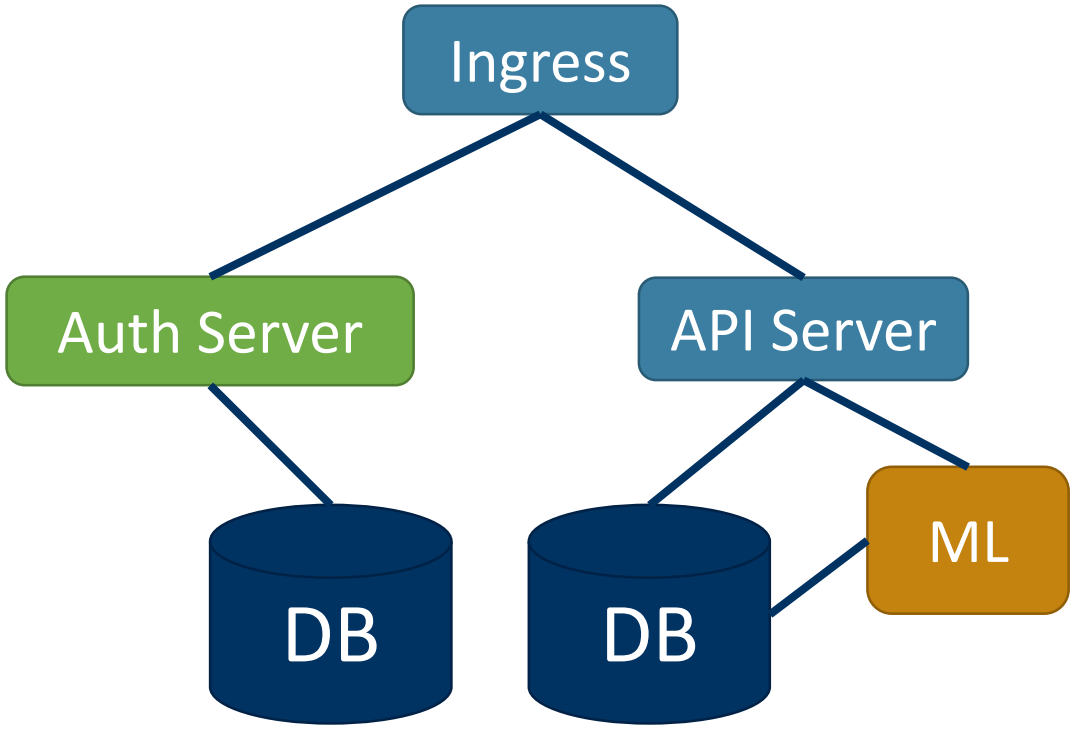
Microservices and Service-Oriented Architecture

- Idea: break a large application into separate services, or microservices
  - Each microservice provides a well-defined interface
  - Each microservice does one thing very well



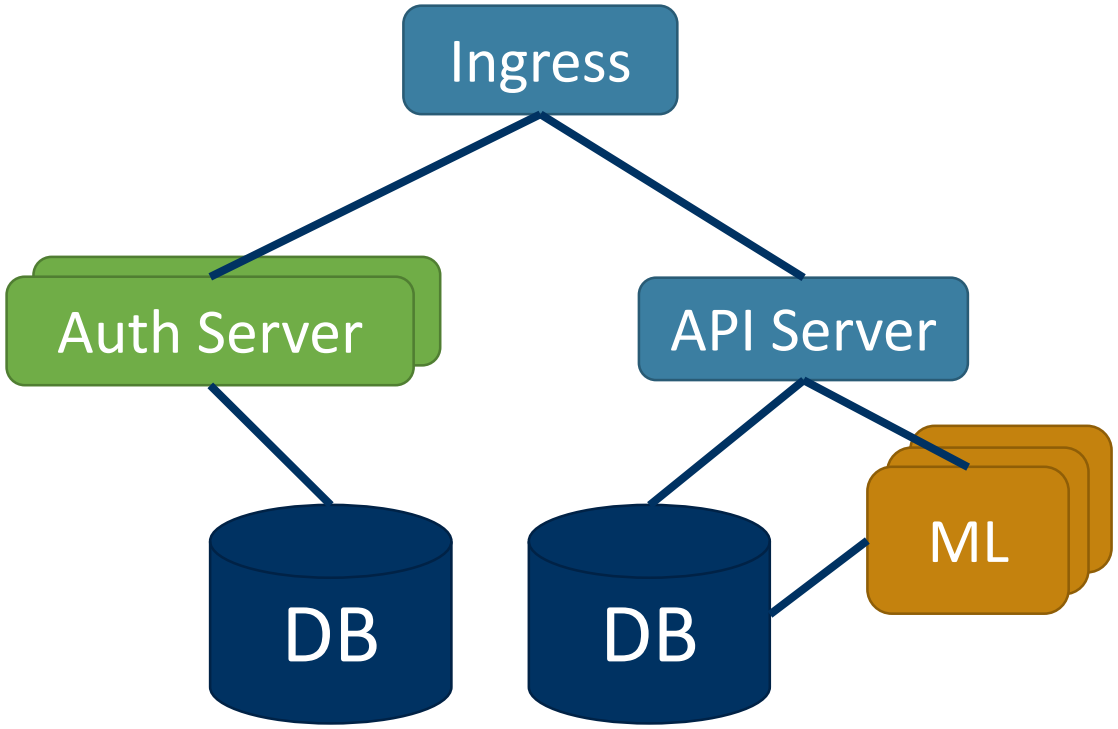
Advantages of Microservices

- Services can be developed independently
  - Only RPC interface matters, not implementation
  - Can even use third party applications



Advantages of Microservices

- Services can be developed independently
  - Only RPC interface matters, not implementation
  - Can even use third party applications
- Horizontal scalability
  - Each microservice can be scaled separately from the rest
- Fault tolerance
  - Failure of one replica of a service should not affect other replicas
  - Failure of an entire service should (ideally) not bring down the entire pipeline



Disadvantages of Microservices

- Much more complex! Not appropriate for simple applications
- More constraints to keep track of
  - E.g., make sure each container is deployed to a VM with enough resources
  - E.g., make sure necessary services are able to communicate
- Potentially more things could go wrong…
  - And more diverse failures to react to

- **Container orchestrators** help us manage this complexity…

Container Orchestration

- A **container orchestrator** is a tool that helps with managing and deploying applications built out of containers

- Manages how containers share the underlying resources
  - Isolation vs. integration, resource requests/limits, …
- Provides common services to microservices and containers
  - Shared persistent storage, authorization, …
- Provides a clean abstraction to the underlying resources
  - To mask details from the developers

Kubernetes (k8s)

- **Nodes** are the machines (possibly virtual) where k8s can scheduler containers
- **Pods** are groups of containers that share some resources. Pods are the unit of scheduling, and containers in a pod are always on the same machine
- **Deployments/ReplicaSets** represent a replicated container (multiple copies of it running)
- **Services** expose pods to the network and act as a load balancer for the pods it contains
  - The “glue” that helps one pod find the others whose services it uses…

k8s Abstraction: Desired State Management

- Desired state is specified *declaratively*
  - Which pods with which resources belong to which service
- Reconciliation: bring the actual state to the desired state

- "Desired state" is stored on a key-value store called *etcd*
  - *etcd* is a replicated, consistent key-value store that uses Raft
  - Guarantees all nodes have a consistent view of the desired state

Announcements

- Project 3 deadline pushed back to Friday
  - But try to get it done soon so you can study for the final
  - Slip day still allowed

- Final Exam on Friday

Computer Security

Computing in the presence of an adversary…

Reliability vs. Security

- Reliability
  - Dealing with Murphy's Law (random failures)

- Security
  - Dealing with a knowledgeable adversary
  - Malice, not mischance

Protection vs. Security

- **Protection**: mechanisms for controlling access to resources
  - Dual-mode operation
  - Address Translation
  - Preemption
  - Users/Permission, Encryption…
- **Security**: use of protection mechanisms to prevent misuse of resources
  - Misuse defined with respect to a policy
  - Need to consider threat model (what the adversary can do)

Useful Security Requirements

- Confidentiality
  - Data only read by authorized users
- Integrity
  - Data is not changed or modified except by authorized users
- Non-repudiation
  - Parties can’t deny having made a statement or sent a message
- Authentication
  - Ensures a user/principal or computer system is what he/she/it claims to be

Encryption: Symmetric-Key

- Sample a random key: $k \leftarrow \{0,1\}^{\lambda}$
  - λ random bits

- Then you can **encrypt** data:
- $E(k, m) \rightarrow c$
- $D(k, c) \rightarrow m$ (recovers original message m)
- Security Guarantee: c reveals nothing about m except its length

- Problem: how to exchange the keys?

Encryption: Public-Key

- Run a special randomized function to obtain $(pk, sk) \leftarrow G()$
  - Carefully structured: can't obtain pk from sk

- Then you can **encrypt** data:
- $E(pk, m) \rightarrow c$
- $D(sk, c) \rightarrow m$ (recovers original message m)
- Security Guarantee: c reveals nothing about m except its length

- Much more expensive than symmetric-key encryption

Digital Signatures

- Run a special randomized function to obtain $(pk, sk) \leftarrow G()$
  - Carefully structured: can't obtain pk from sk

- Then, you can **sign** data:
- $S(sk, m) \rightarrow s$
- $V(pk, m, s) \rightarrow 0$ or 1 (checks if s was generated with sk)
- Security Guarantee: s can't be generated unless you know sk

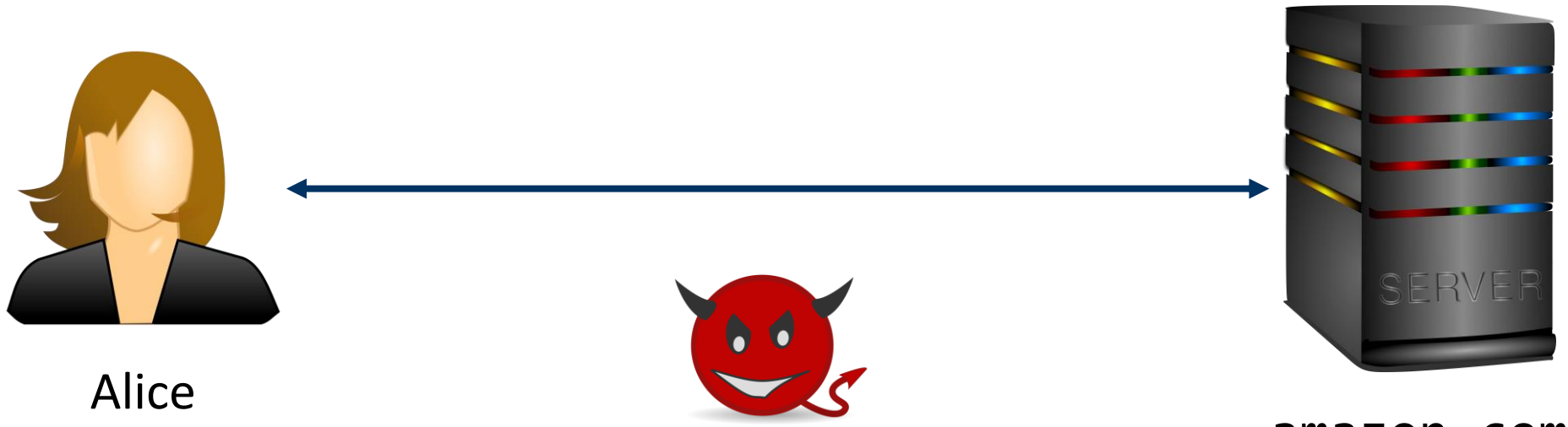
Secure Channels

- How to hide sensitive information from an adversary snooping on the network?

Secure Channels

1. Each party encrypts everything that's sent across the network with recipient's public key (recipient decrypts with secret key)
2. Each party signs everything that's sent across the network with their secret key (recipient verifies with sender's public key)

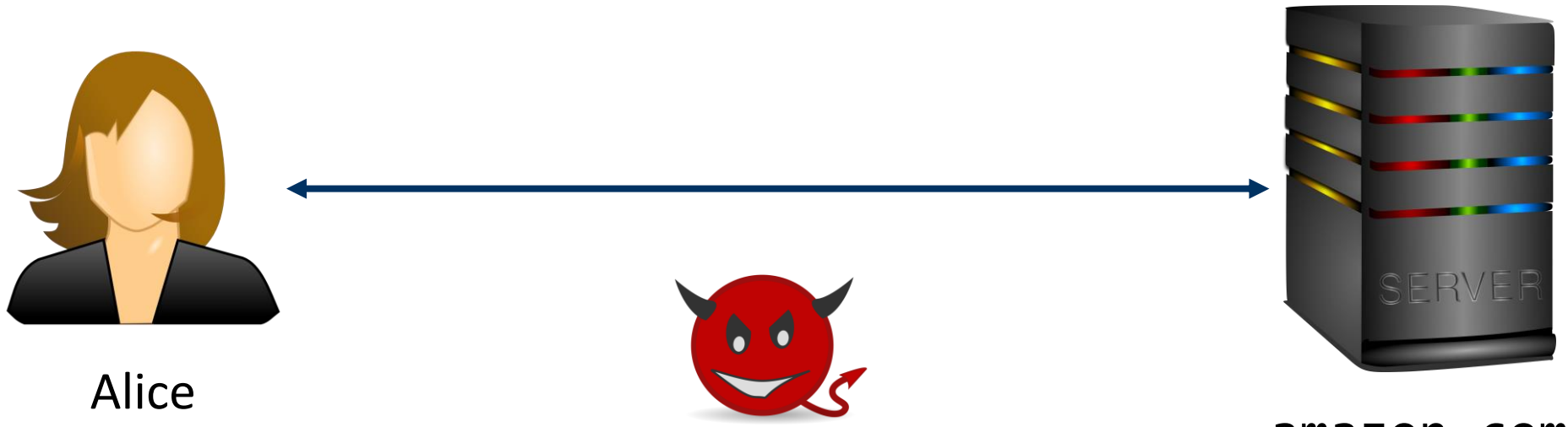
Can be accelerated by agreeing on symmetric keys this way, and then using symmetric keys for the actual data…



Secure Channels: Authentication

How does server know it's really Alice on the other side? And not some adversary impersonating her…

- Alice sends her username and password (requires setting up a shared secret beforehand)
- Or perhaps the server knows Alice's public key in advance, and can ask her to sign something…



Secure Channels: Authentication

How does Alice know it's really the server she wants to access? And that it's not some adversary impersonating `amazon.com`…

- Perhaps the server can also authenticate to Alice (they'd have to set up a shared secret beforehand)
- Perhaps someone Alice trusts has digitally signed a statement saying that "K_A is the public key for `amazon.com`"

Digital Certificate

- How do you know K_A is `amazon.com`'s public key?
- Because some authority that you trust (e.g., Verisign) has signed a statement that K_A is `amazon.com`'s public key
- That statement, together with the signature, is called a **certificate**
- `amazon.com` presents this certificate for incoming connections, so that the user knows they're really speaking to `amazon.com`

HTTPS and TLS

- https = “Use HTTP over SSL/TLS”

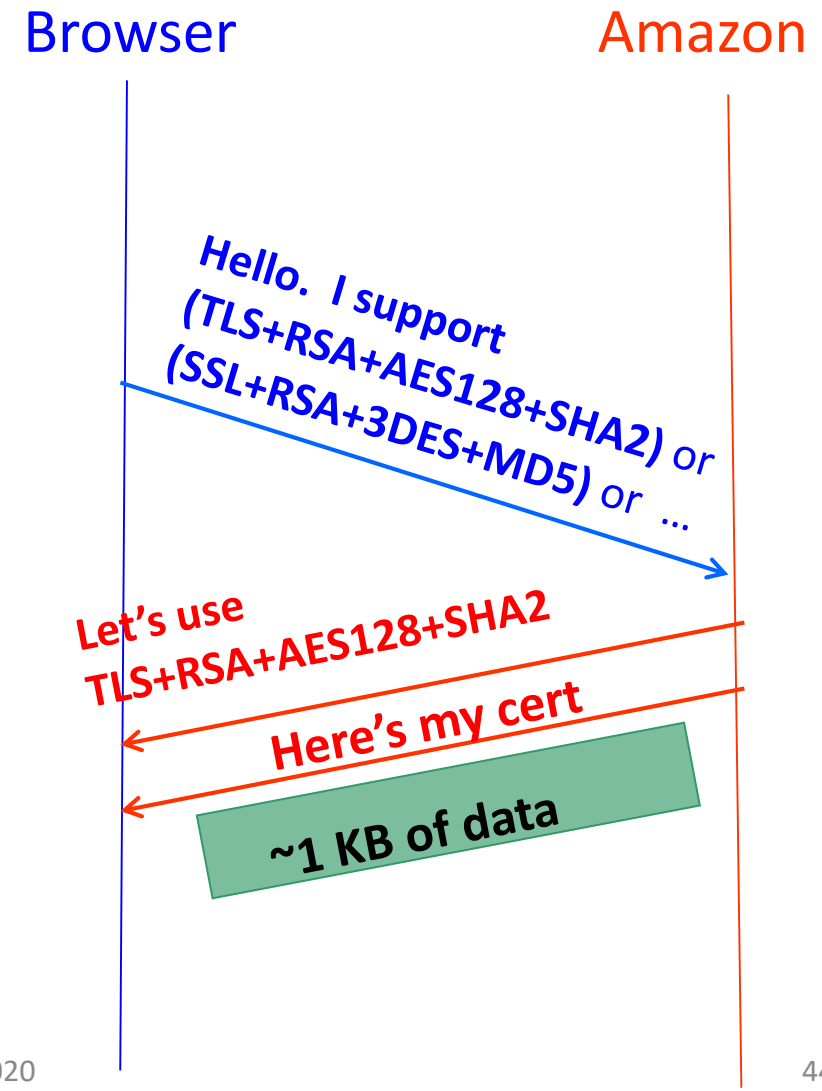
- After establishing a TCP connection, you first establish a secure channel over that TCP session by running a protocol called TLS
  - Server authenticates to user as part of TLS

- All communication happens over TLS
  - Including the user sending username/password to authenticate

HTTPS Connection (Simplified)

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client sends over list of crypto protocols it supports
- Server picks protocols to use for this session
- Server sends over its certificate
- (all of this in the clear)

- THEN browser and Amazon establish shared secret to encrypt/sign data…



Conclusion

- Paxos/Raft: distributed consensus algorithms with majority quorums
- Byzantine Fault Tolerance: nodes may fail in unclean or malicious ways
- Containers: guest has abstraction of its own operating system
  - Microservices: split application into microservices, each in a container
  - Kubernetes: orchestrator for managing containers in a microservice
- Security: using protection mechanisms to prevent misuse of resources
  - Secure channels over the network
  - Security needed at all layers