Planes, Raft, and Pods

A Tour of Distributed Systems Within Kubernetes

@boluptuous

Craftsy

Distributed Systems?

> “Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers.”
>
> - Mikito Takada

?

> “Open-source platform for automating deployment, scaling, and operations of application containers around clusters of hosts, providing container-centric infrastructure”
>
> - Kubernetes Documentation

???????

Flexible platform for running containerized apps!

How does Kubernetes leverage distributed systems?

What is a container?

Pod = 1 or more containers

Deployments manage pods

Kubernetes is distributed

Master Components

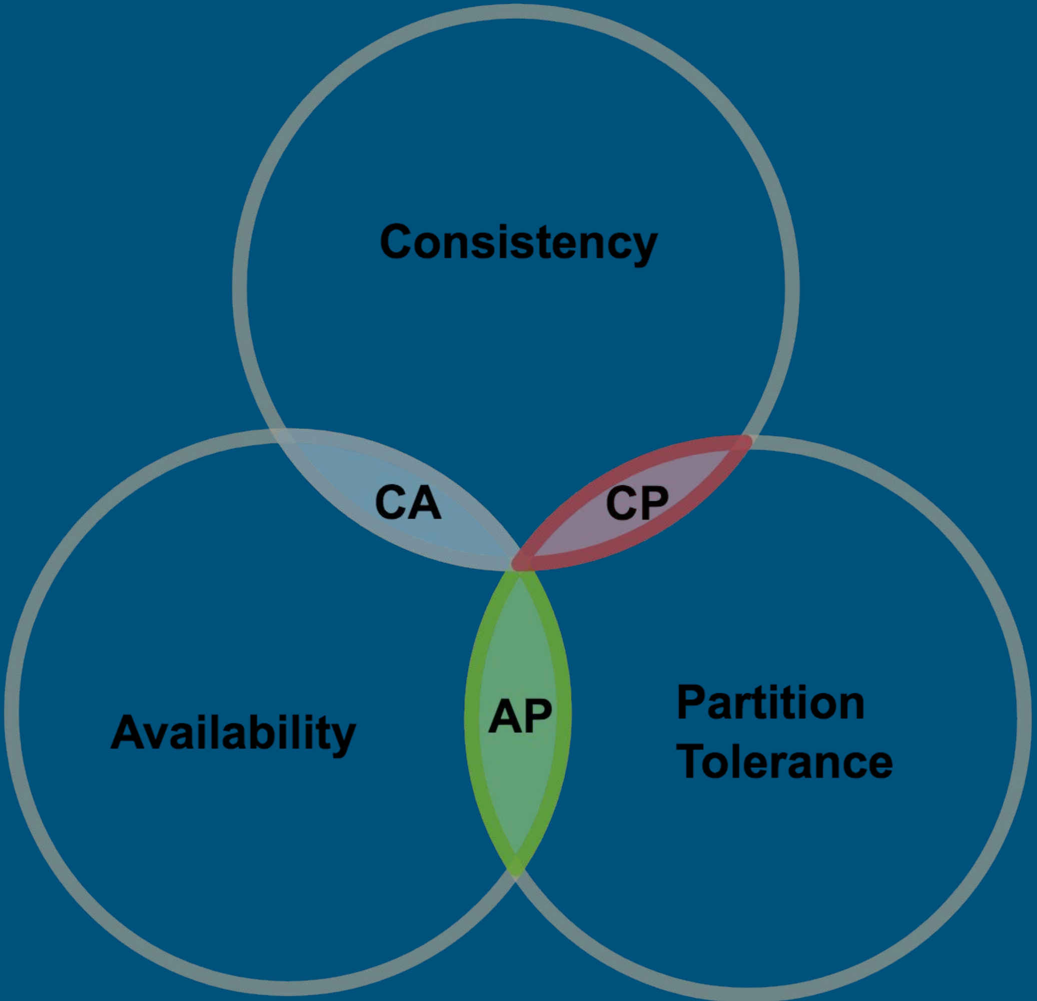
- etcd
- API Server
- Controllers
- Scheduler

Node Components

- Kubelet
- Kube-proxy
- Container runtime

etcd!

multiple etcds > one etcd


Consistency
CA
CP
AP
Availability
Partition Tolerance

Why etcd?

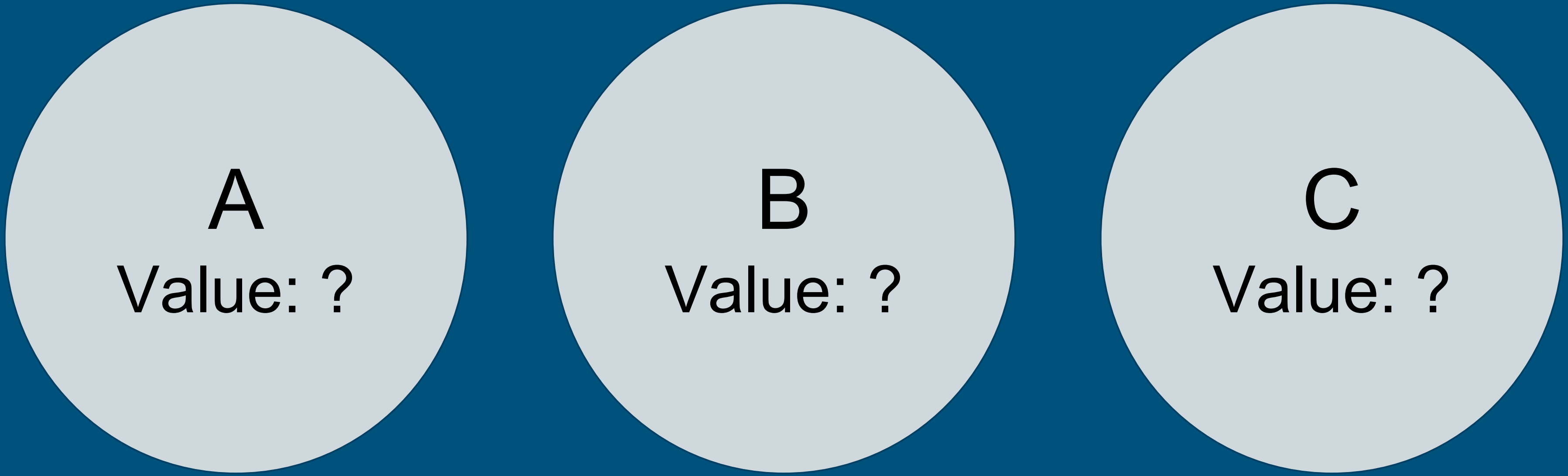
etcd is designed for “large scale distributed systems… that never tolerate split brain behavior and are willing to sacrifice availability” to achieve it

- etcd Documentation

Simple interface hides complex problems

raft.github.io
raft

Let’s look at a Not Raft system



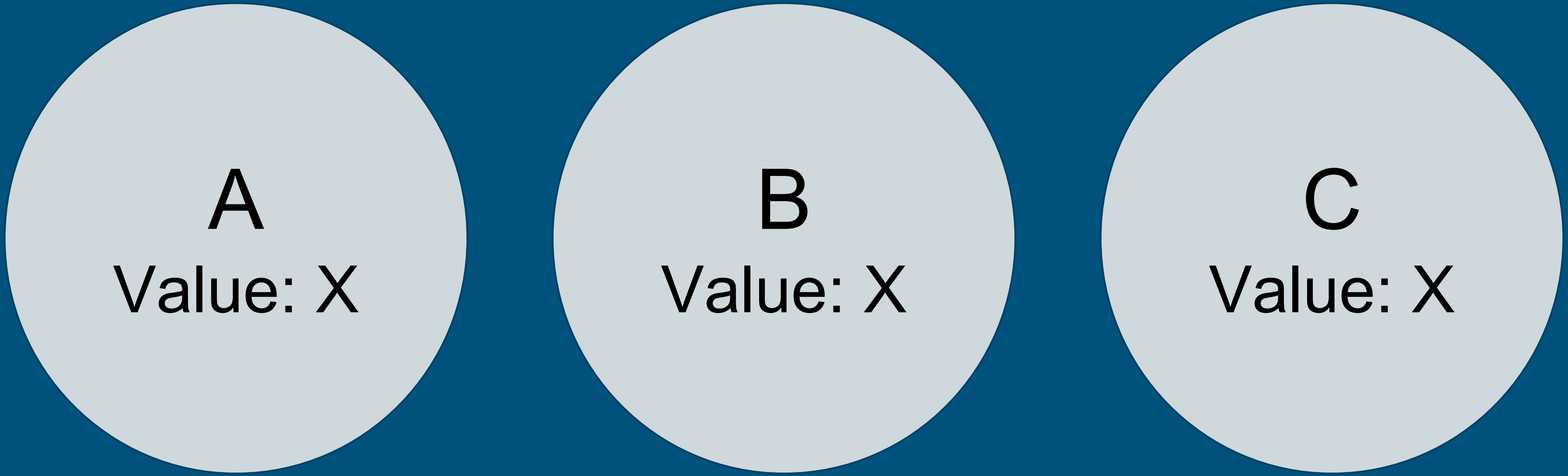
A
Value: ?

B
Value: ?

C
Value: ?

Nodes receive requests, write to disk, and then broadcast new value to all other nodes

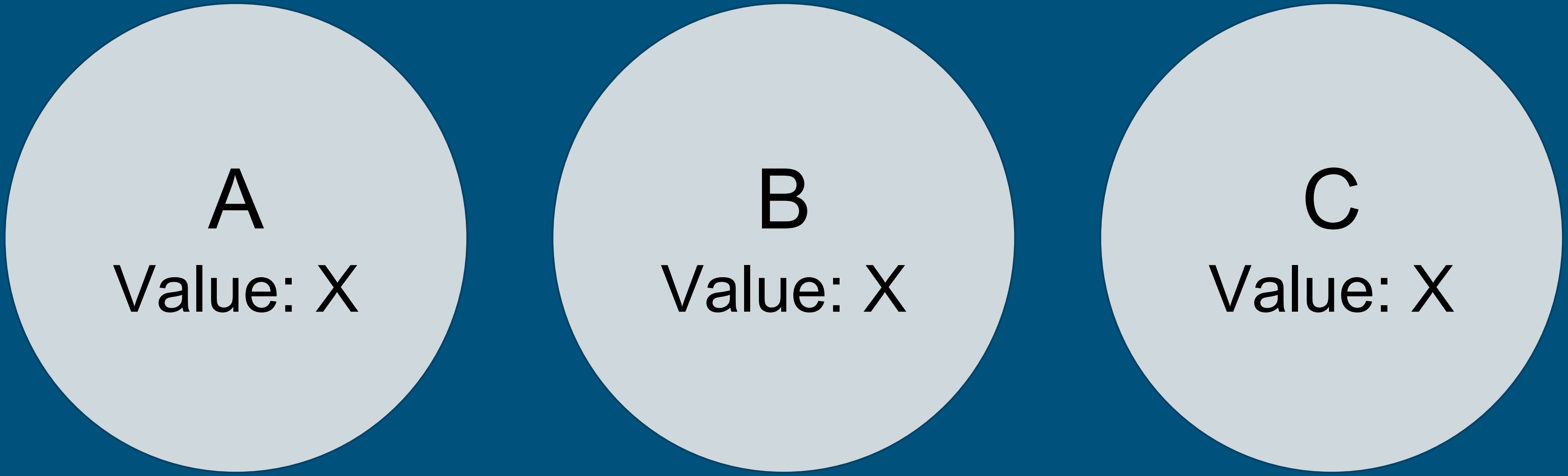
What happens if there’s multiple updates to the value at the same time?



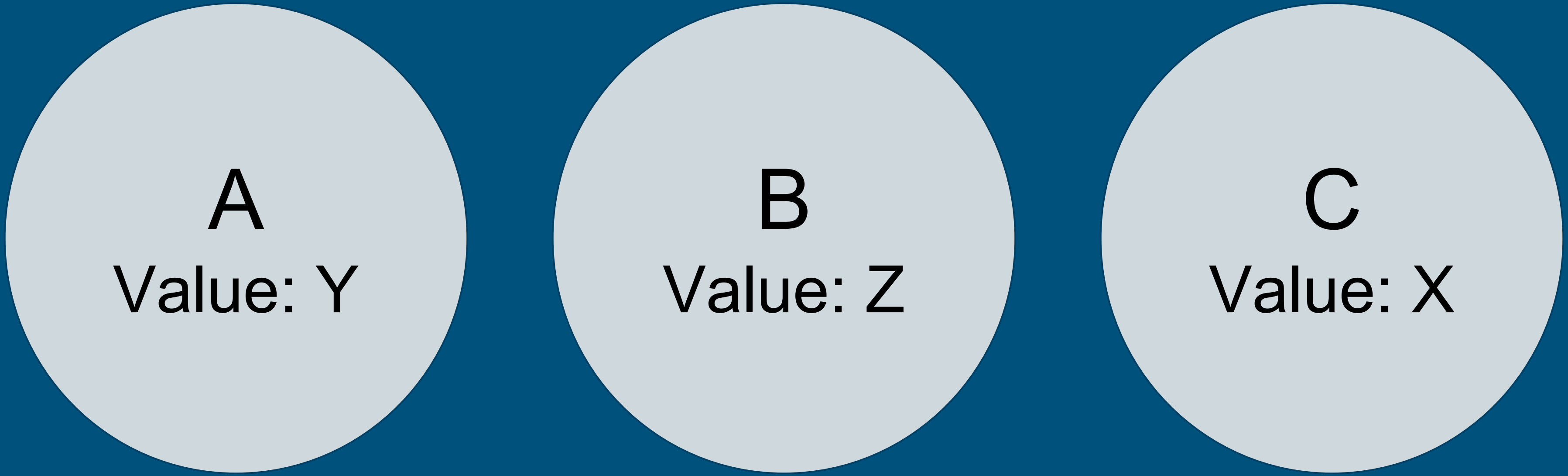
A
Value: X

B
Value: X

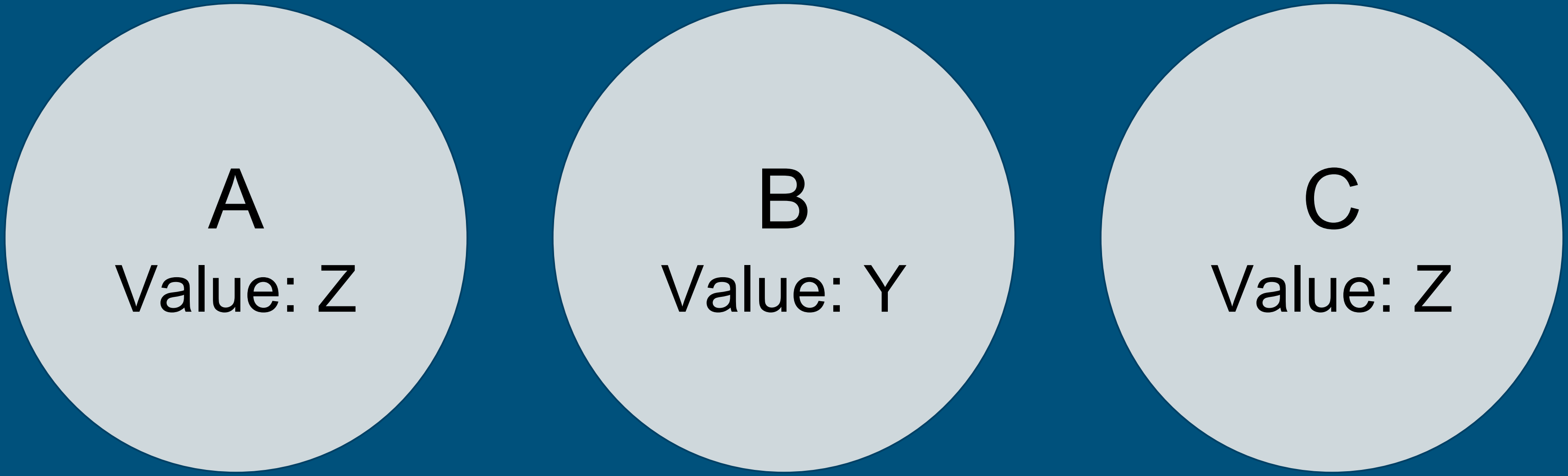
C
Value: X



Client 1 tells Node A that the new value is Y
Client 2 tells Node B that the new value is Z



Node A writes Y to disk
Node B writes Z to disk



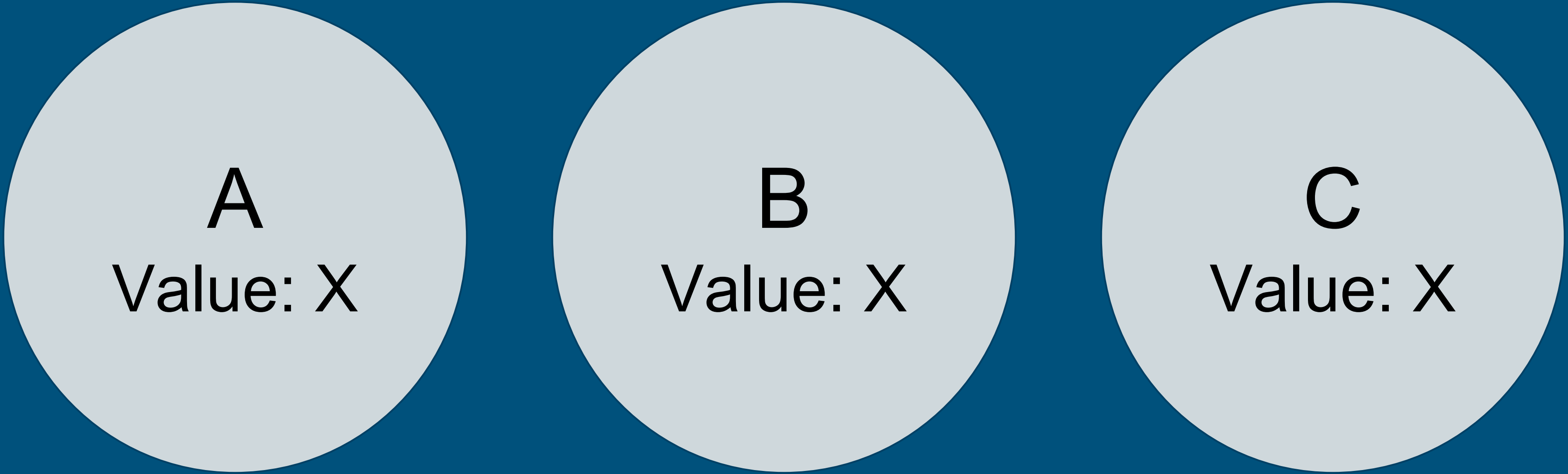
A
Value: Z

B
Value: Y

C
Value: Z

Node A broadcasts the new value Y
Node B broadcasts the new value Z
(we assume A’s messages arrive before B’s)

New Scenario!



A

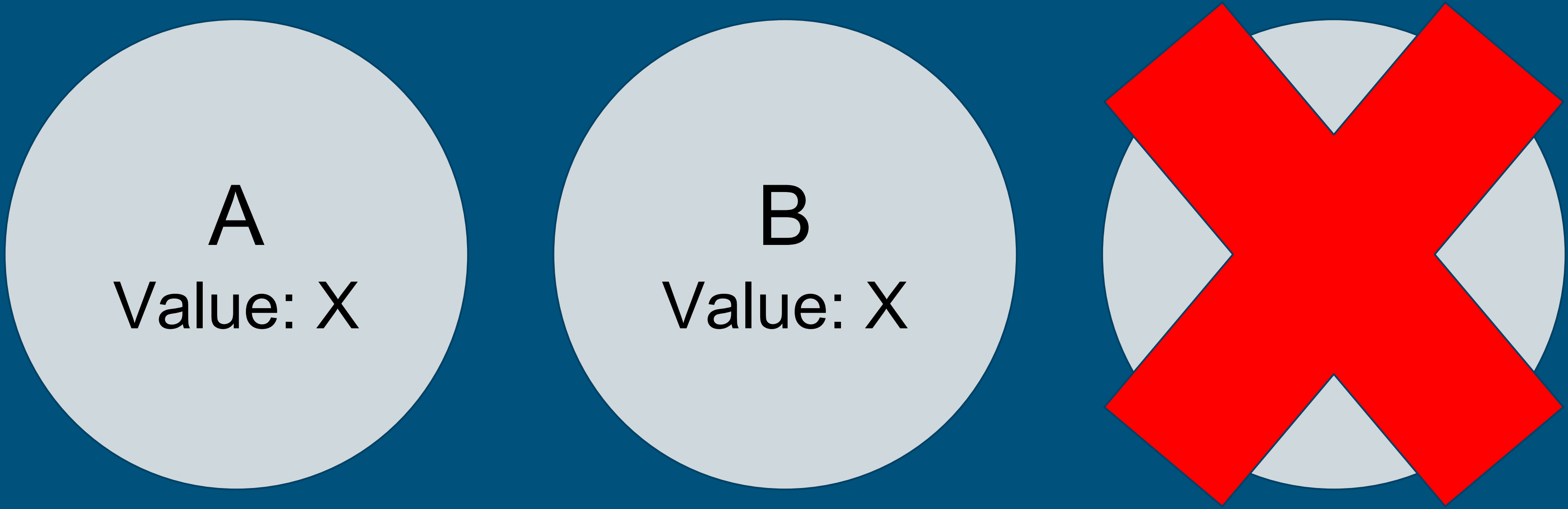
Value: X

B

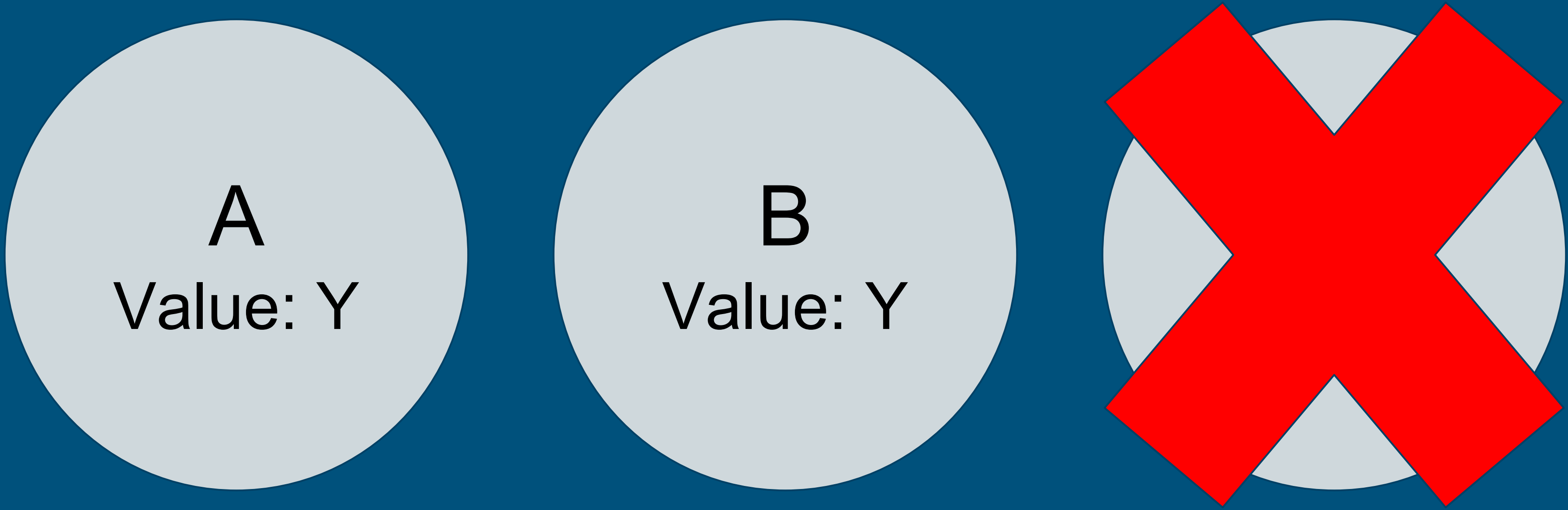
Value: X

C

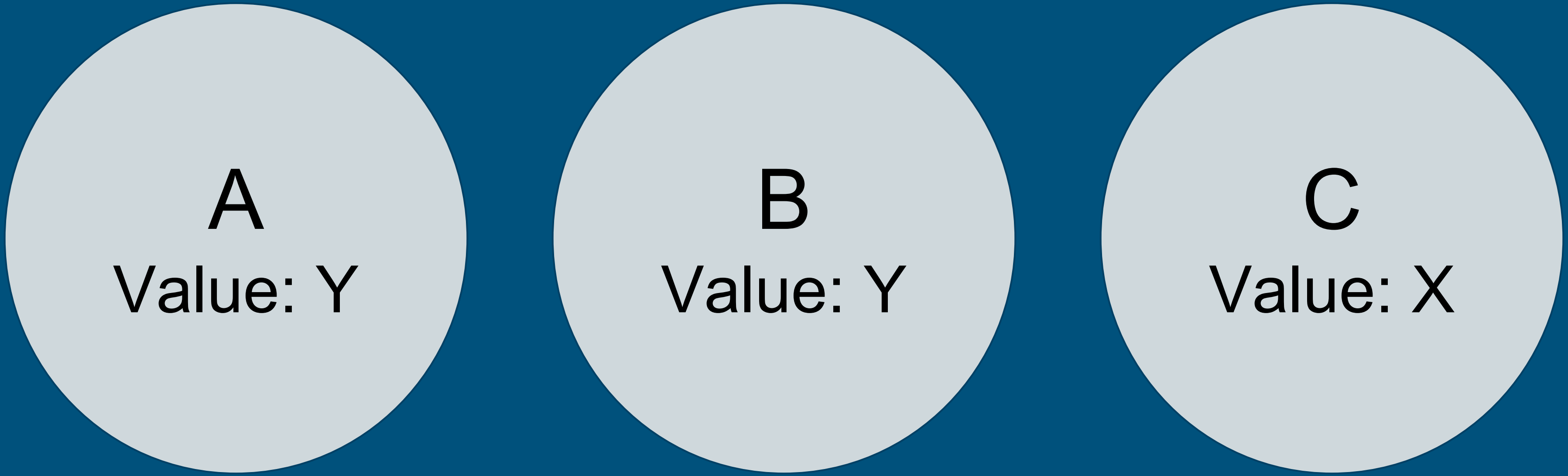
Value: X



Cluster undergoes a network partition!
C can’t talk to A or B
A or B can’t talk to C



Client updates the value to Y
C doesn’t find out because messages are dropped



A
Value: Y

B
Value: Y

C
Value: X

C comes back but thinks the value is still X

Consensus requires coordination

Raft = consensus algorithm for managing replicated log

Elected leader is put in charge of managing the log

Three States!

- Leader
- Follower
- Candidate

One leader per term

Leader sends heartbeat messages

What happens if a follower doesn’t get a heartbeat? Election time!

In the game of Raft leadership elections, you win or you lose.

1. Write goes to leader
2. Leader appends command to log
3. Tells other servers via RPC to append it to their logs (followers will say no if they’re behind)
4. Once majority append, leader commits
5. Leader tells nodes in subsequent messages of the last committed entry
6. Nodes commit

Solves problems in our bad system

Consistency and partition-tolerance are achieved through requiring a majority of nodes to act

Further Raft Reading

- The Raft Paper
- The Secret Lives of Data (Raft Visualization)

Controller = loop that watches cluster state and makes changes to ensure we keep the desired state

Replica Set Controller makes sure there’s a given number of pods running at any time

Deployment controller manages the whole deployment process of your app

Scheduler watches for unscheduled pods and assigns them to a given node

The Scheduling Algorithm

1. Filter out nodes that aren’t desired or not a great fit
2. Rank the remaining nodes
3. Pick the top ranked node

Step 1: Filter Against Predicates

- HostName
- MatchNodeSelector
- InterPodAffinityMatches
- PodToleratesNodeTaints

More Predicates!

- PodFitsHostPort
- PodFitsResources
- CheckNodeMemoryPressure
- CheckNodeDiskPressure
- CheckNodeCondition

Ranking applies a series of weighted priority functions that return a score (higher score is more desirable)

Functions are run against each node, added up, and the node with the highest score is the winner!

Some Ranking Functions

- LeastRequestedPriority
- BalancedResourceAllocation
- SelectorSpreadPriority
- ImageLocalityPriority
- NodeAffinityPriority
- TaintTolerationPriority

What happens when we submit a deployment to Kubernetes?

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: hello-kubecon-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: hello-kubecon
    spec:
      containers:
      - name: hello-kubecon
        image: boingram/hellohttp:latest
        ports:
        - containerPort: 8080
```

```yaml
kind: Service
apiVersion: v1
metadata:
  name: hello-kubecon-service
spec:
  selector:
    app: hello-kubecon
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

How do we submit our deployment? Kubectl!

What We Expect

1. We create deployment
2. Deployment creates a replica set
3. Replica set creates three pods
4. Our scheduler schedules those three pods
5. Kubelet will run scheduled pods

What actually happens…

```
involvedObject:
  kind: Deployment
  name: hello-kubecon-deployment

message: Scaled up replica set
hello-kubecon-deployment-2009686459 to 3

reason: ScalingReplicaSet
source:
  component: deployment-controller
```

```
involvedObject:
  kind: ReplicaSet
  name: hello-kubecon-deployment-2009686459

message: 'Created pod:
hello-kubecon-deployment-2009686459-nwc7k'

reason: SuccessfulCreate
source:
  component: replicaset-controller
```

```
involvedObject:
  kind: Pod
  name: hello-kubecon-deployment-2009686459-nwc7k

message: Successfully assigned
hello-kubecon-deployment-2009686459-nwc7k to
gke-cluster-1-default-pool-ed78e24c-33jg

reason: Scheduled
source:
  component: default-scheduler
```

```
involvedObject:
  kind: ReplicaSet
  name: hello-kubecon-deployment-2009686459

message: 'Created pod:
hello-kubecon-deployment-2009686459-03hfh'

reason: SuccessfulCreate
source:
  component: replicaset-controller
```

```
involvedObject:
  kind: Pod
  name: hello-kubecon-deployment-2009686459-03hfh

message: Successfully assigned
hello-kubecon-deployment-2009686459-03hfh to
gke-cluster-1-default-pool-ed78e24c-33jg

reason: Scheduled
source:
  component: default-scheduler
```

```
involvedObject:
  kind: ReplicaSet
  name: hello-kubecon-deployment-2009686459

message: 'Created pod:
hello-kubecon-deployment-2009686459-05kv9'

reason: SuccessfulCreate
source:
  component: replicaset-controller
```

```
involvedObject:
  kind: Pod
  name: hello-kubecon-deployment-2009686459-05kv9

message: Successfully assigned
hello-kubecon-deployment-2009686459-05kv9 to
gke-cluster-1-default-pool-ed78e24c-33jg

reason: Scheduled
source:
  component: default-scheduler
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-05kv9

message: pulling image "boingram/hellohttp:latest"

reason: Pulling
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-03hfh

message: pulling image "boingram/hellohttp:latest"

reason: Pulling
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-nwc7k

message: pulling image "boingram/hellohttp:latest"

reason: Pulling
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-03hfh

message: Successfully pulled image
"boingram/hellohttp:latest"

reason: Pulled
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-03hfh

message: Created container

reason: Created
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-03hfh

message: Started container

reason: Started
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-05kv9

message: Successfully pulled image
"boingram/hellohttp:latest"

reason: Pulled
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-05kv9

message: Created container

reason: Created
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

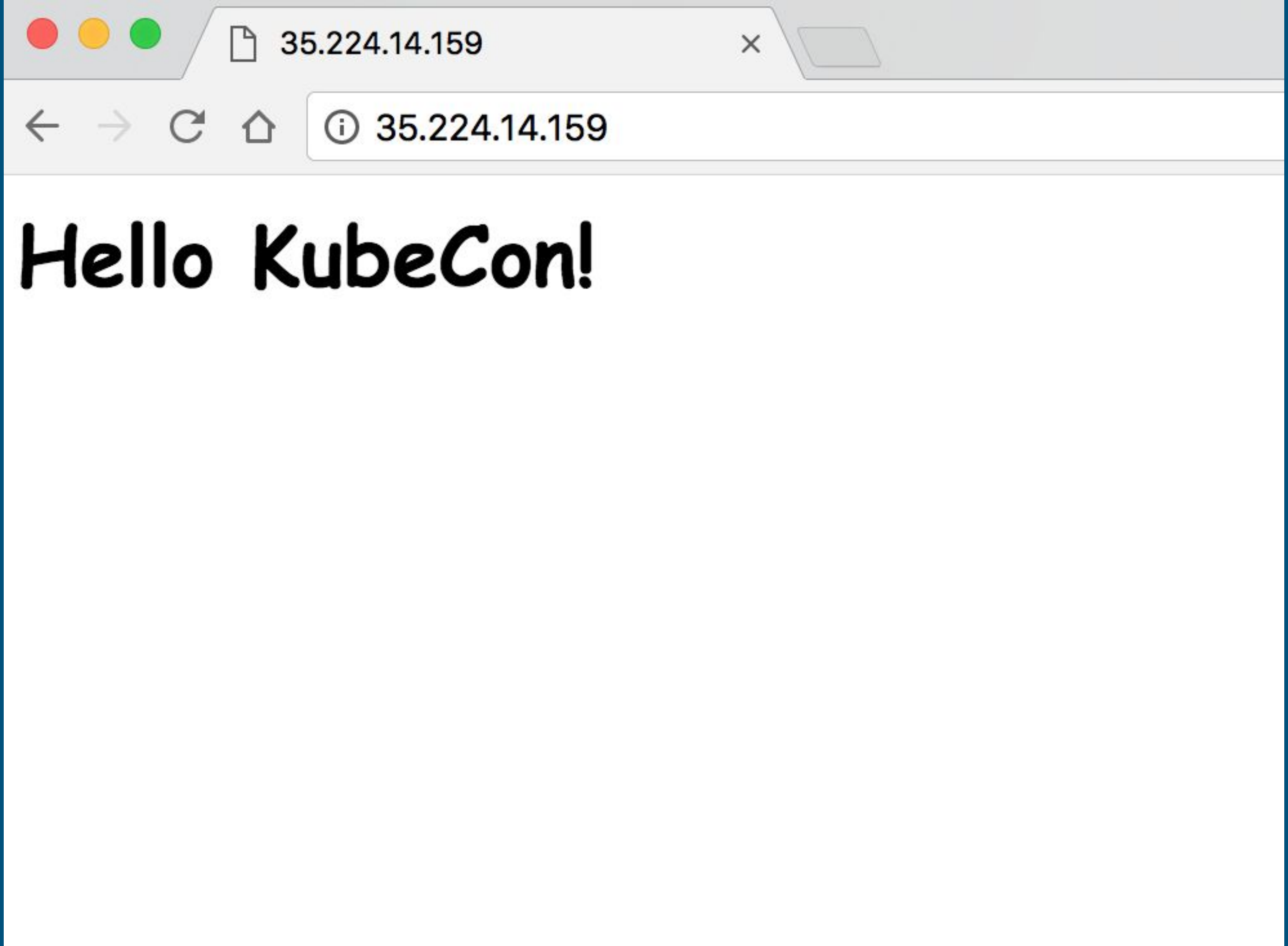
```yaml
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-05kv9

message: Started container

reason: Started
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-nwc7k

message: Successfully pulled image
"boingram/hellohttp:latest"

reason: Pulled
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-nwc7k

message: Created container

reason: Created
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

```
involvedObject:
  fieldPath: spec.containers{hello-kubecon}
  kind: Pod
  name: hello-kubecon-deployment-2009686459-nwc7k

message: Started container

reason: Started
source:
  component: kubelet
  host: gke-cluster-1-default-pool-ed78e24c-33jg
```

We did it!



Hello KubeCon!

Things We’ve Done!

- Look at Kubernetes components
- Shown how it handles distributed state
- Dove into how we reconcile state and schedule pods
- Traced a deployment through the system