The simplicity of implementing a job scheduler with Circuit

DEVIEW 2015
Seoul, South Korea

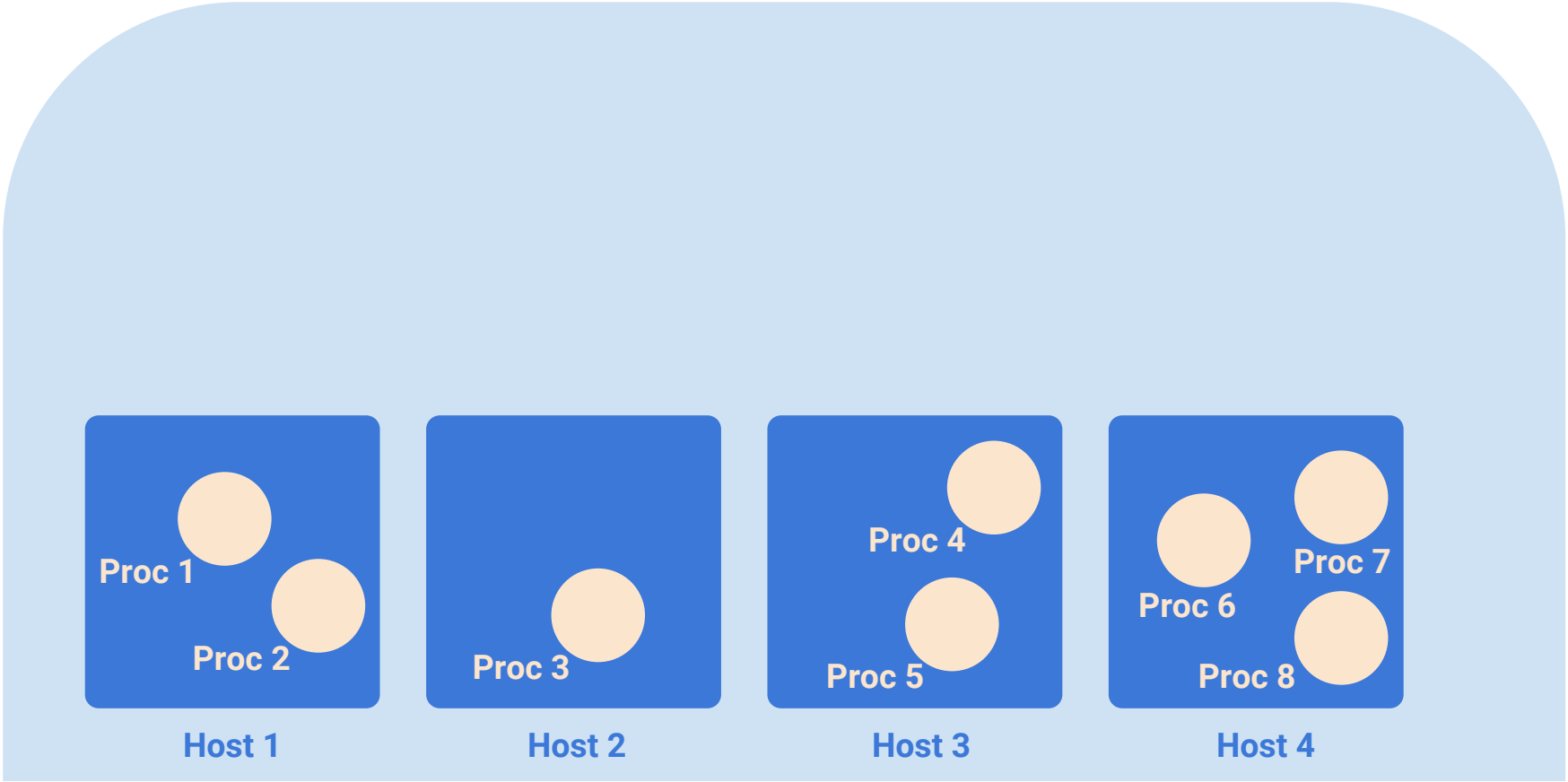
Petar Maymounkov

Circuit: Light-weight cluster OS

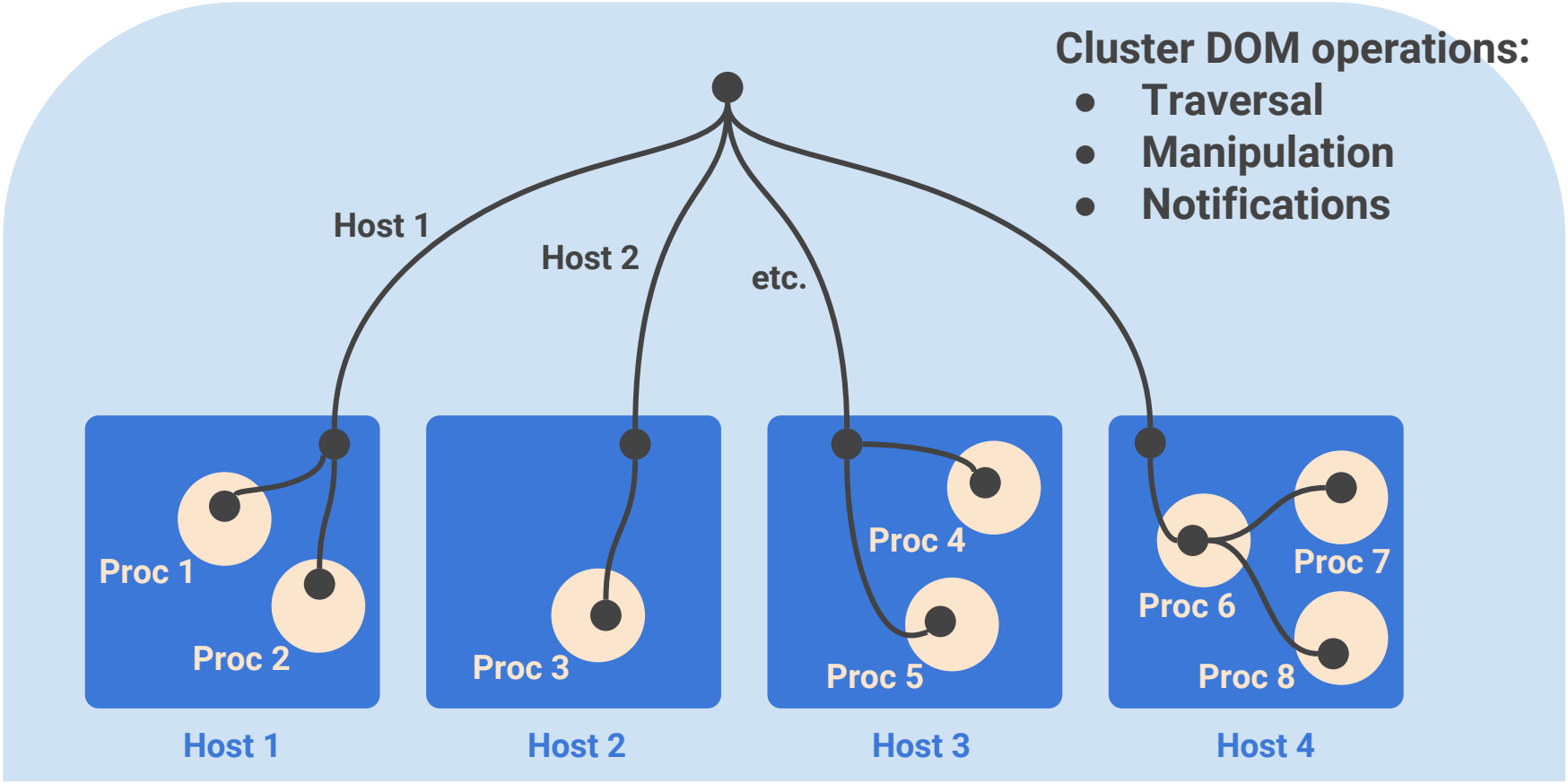
- **Real-time API to see and control:**
  - Hosts, processes, containers
- **System never fails**
  - API endpoint on every host
  - Robust master-less, peer-to-peer membership protocol

API overview

API: Model of cluster



API: Abstraction

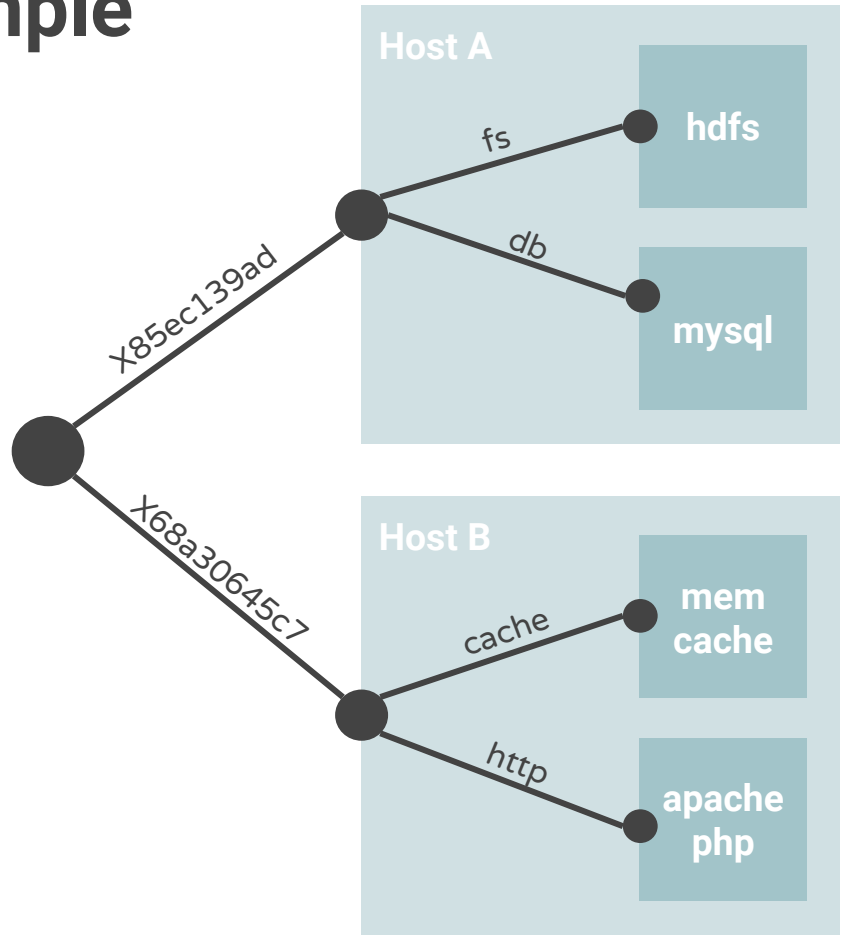


API: Entrypoints

- Command-line tool
- Go client package (more later)

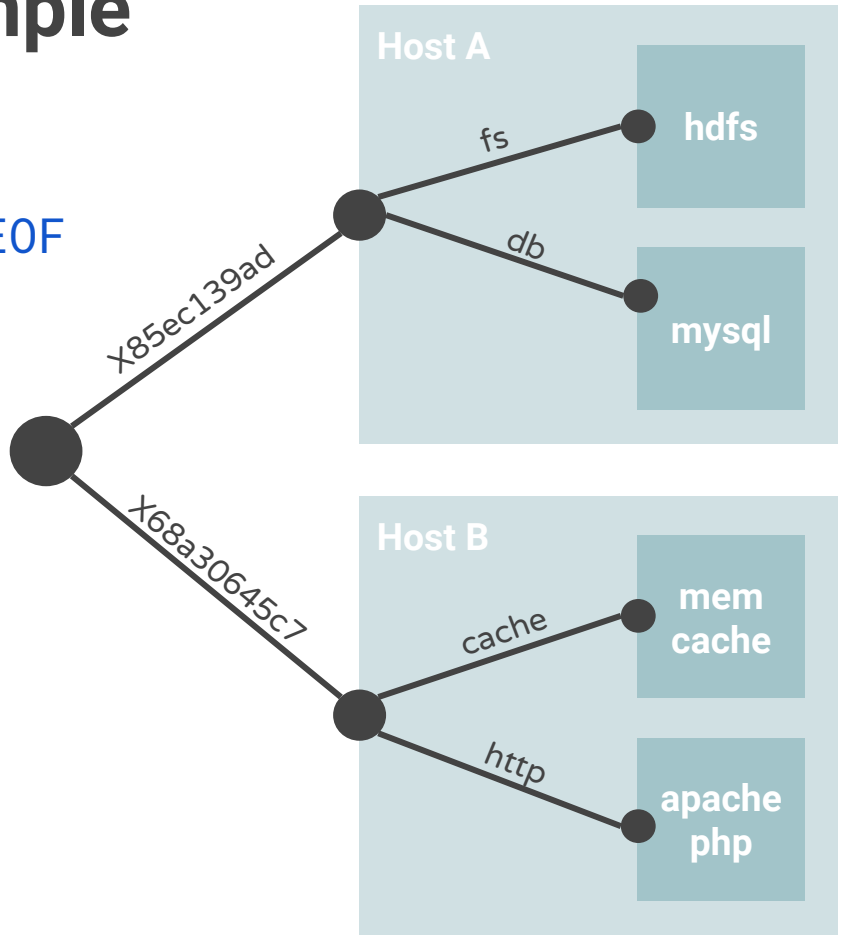
API: Command-line example

```
$ circuit ls -l /...
server    /X85ec139ad
server    /X85ec139ad/fs
proc      /X85ec139ad/db
server    /X68a30645c
proc      /X68a30645c/cache
docker    /X68a30645c/http
```



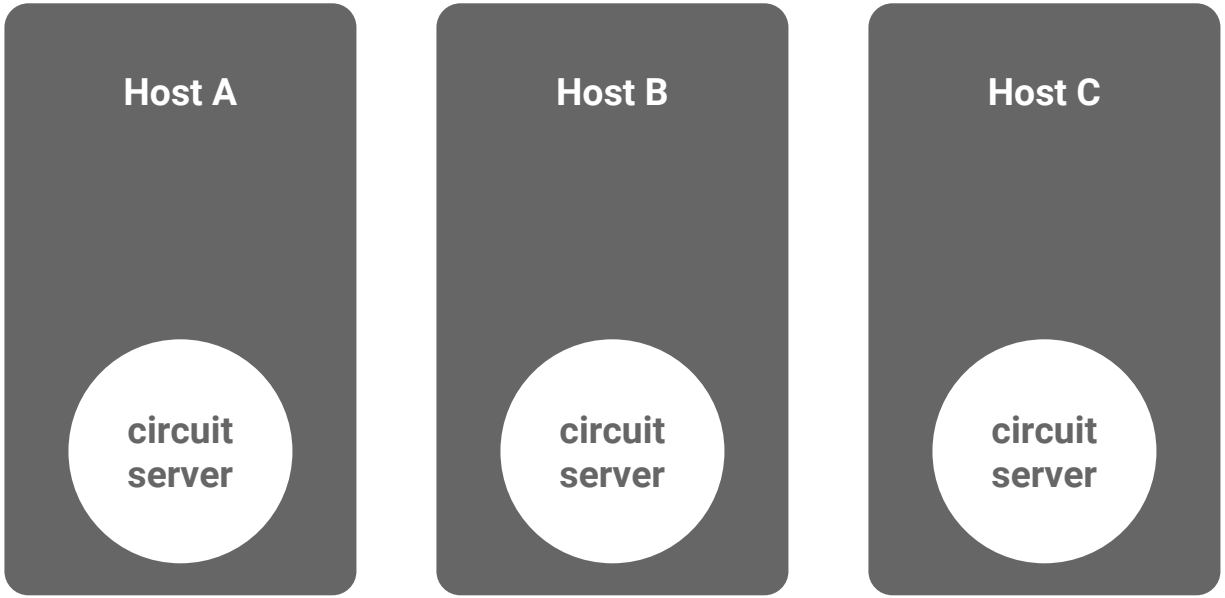
API: Command-line example

```
$ circuit mkproc /X85ec/test <<EOF
{“Path”:”/sbin/lscpu”}
EOF

$ circuit stdout /X85ec/test
Architecture: x86_64
etc.

$ circuit wait /X85ec/db

$ circuit stderr /X68a3
etc.
```



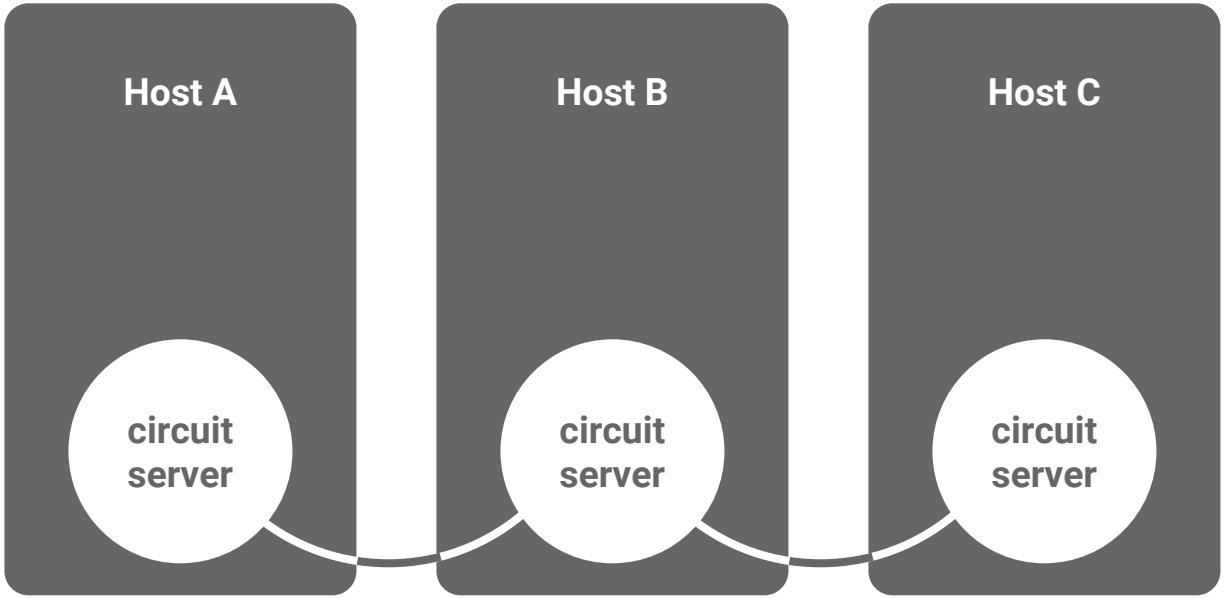
System architecture

System architecture: Boot individual hosts



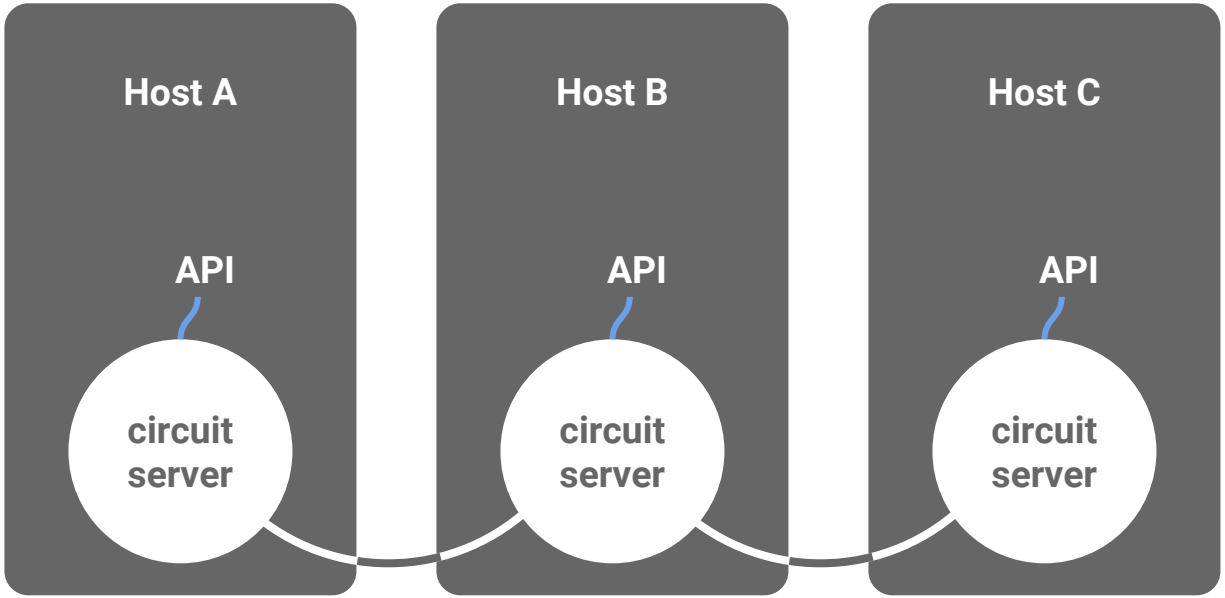

```
$ circuit start
```

System architecture: Discovery



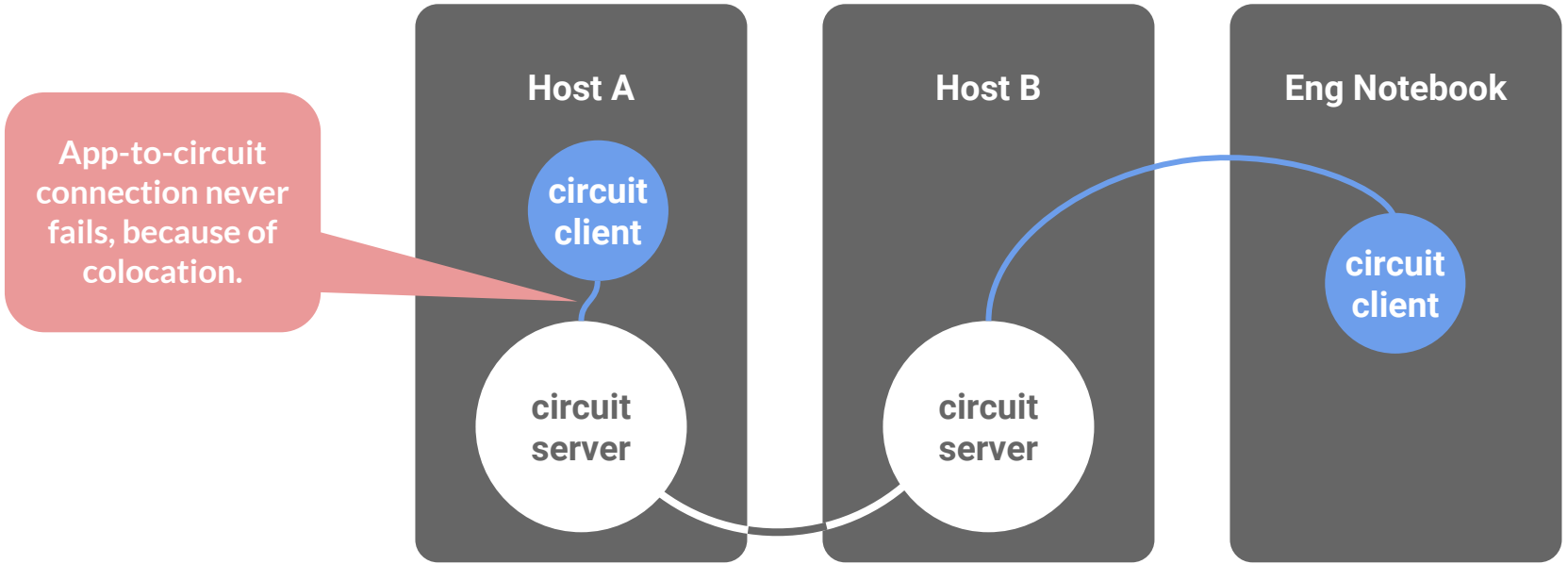

```
$ circuit start --discover=228.8.8.8:8822
```

System architecture: API endpoints




```
$ circuit start
circuit://127.0.0.1:7822/17880/Q413a079318a275ca
```

System architecture: Client connections




```
$ circuit start
circuit://127.0.0.1:7822/17880/Q413a079318a275ca
```

Go API overview

Go: Entrypoint

```
import . “github.com/gocircuit/circuit/client”

func Dial(
    circuitAddr string,
    crypto      []byte,
) *Client

func DialDiscover(
    udpMulticast string,
    crypto       []byte,
) *Client
```

Go: Error handling

- Physical errors are panics
- Application errors are returned values

```
func Dial(
    circuitAddr string,
    crypto      []byte,
) *Client
```

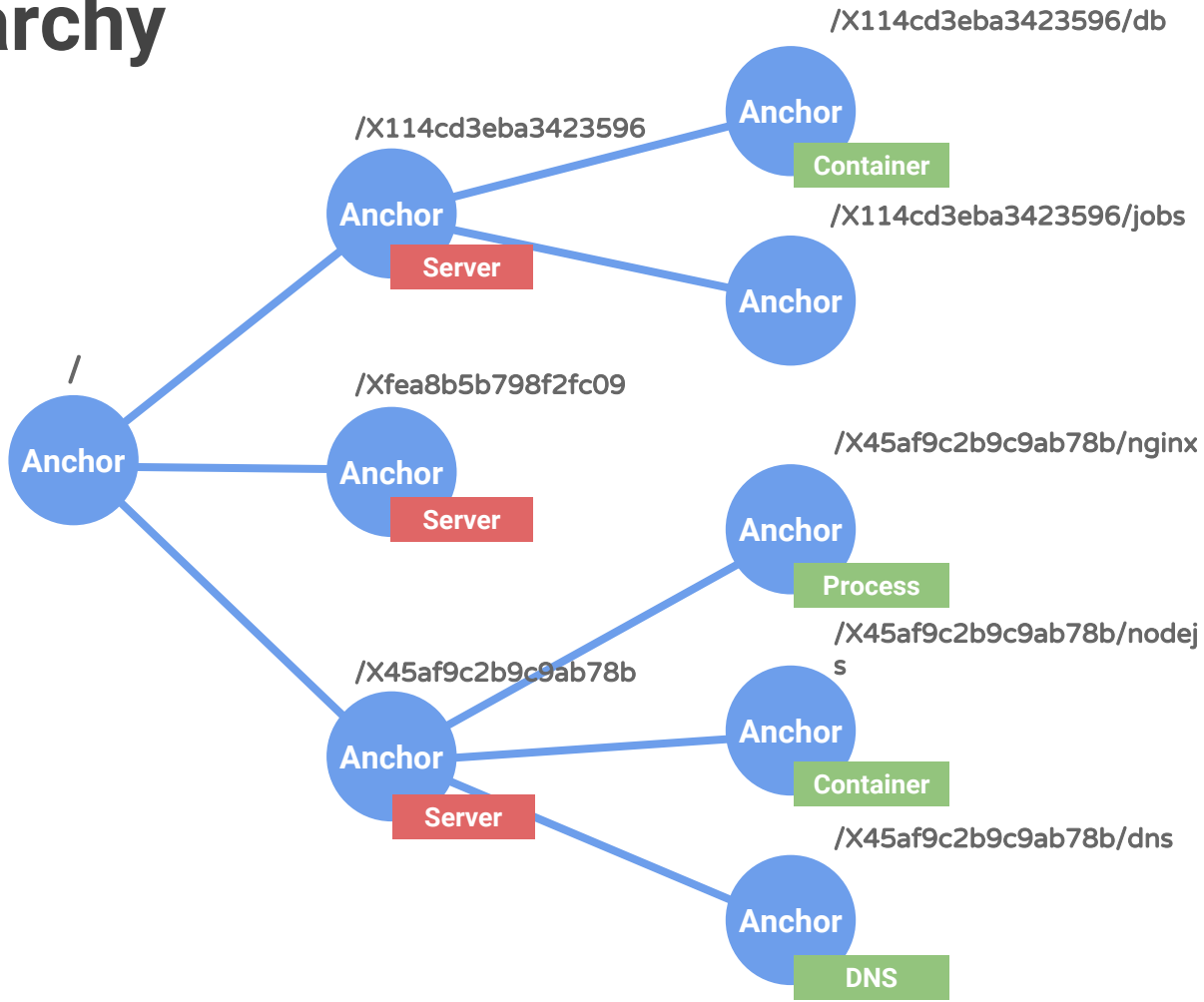
Go: Anchor hierarchy

An **anchor** is a node in a **unified namespace**

An **anchor** is like a **“directory”**. It can have children anchors

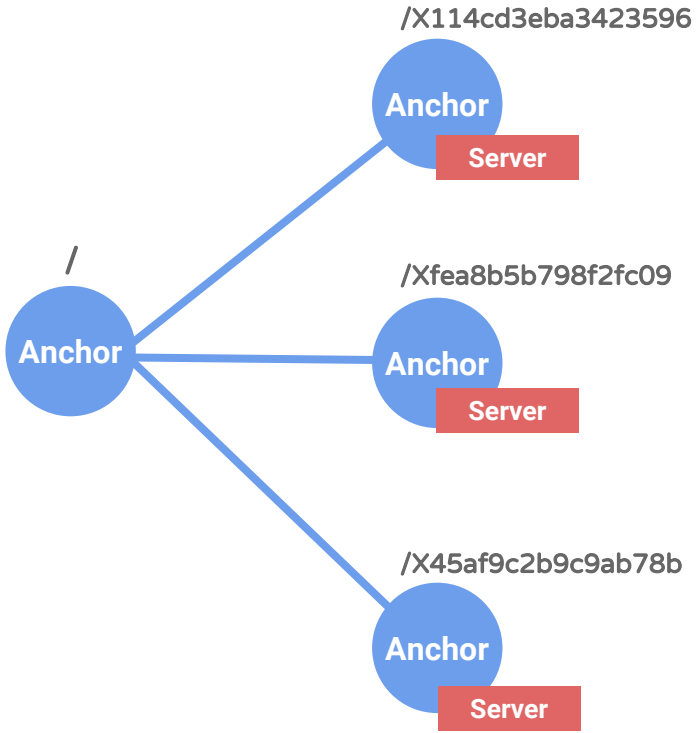
An **anchor** is like a **“variable”**. It can be empty or hold an object (server, process, container, etc.)

Client connection is the **root anchor**

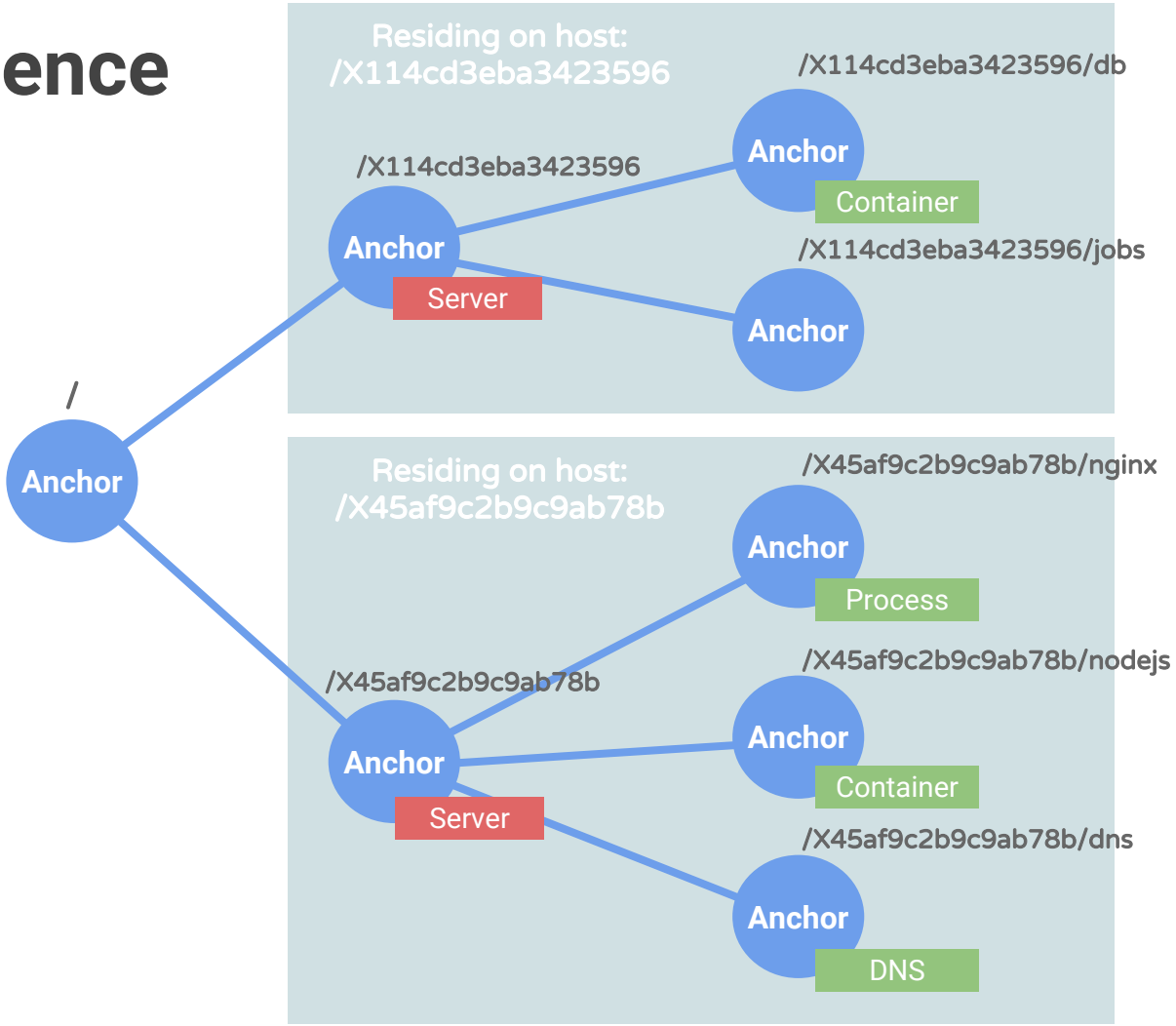


Go: Anchor API

```
type Anchor interface{
    Walk(path []string) Anchor
    View() map[string]Anchor
    MakeProc(Spec) (Proc, error)
    MakeDocker(Spec) (Container, error)
    Get() interface{}
    Scrub()
}

type Proc interface{
    Peek() (State, error)
    Wait() error
    …
}
```



Go: Anchor residence



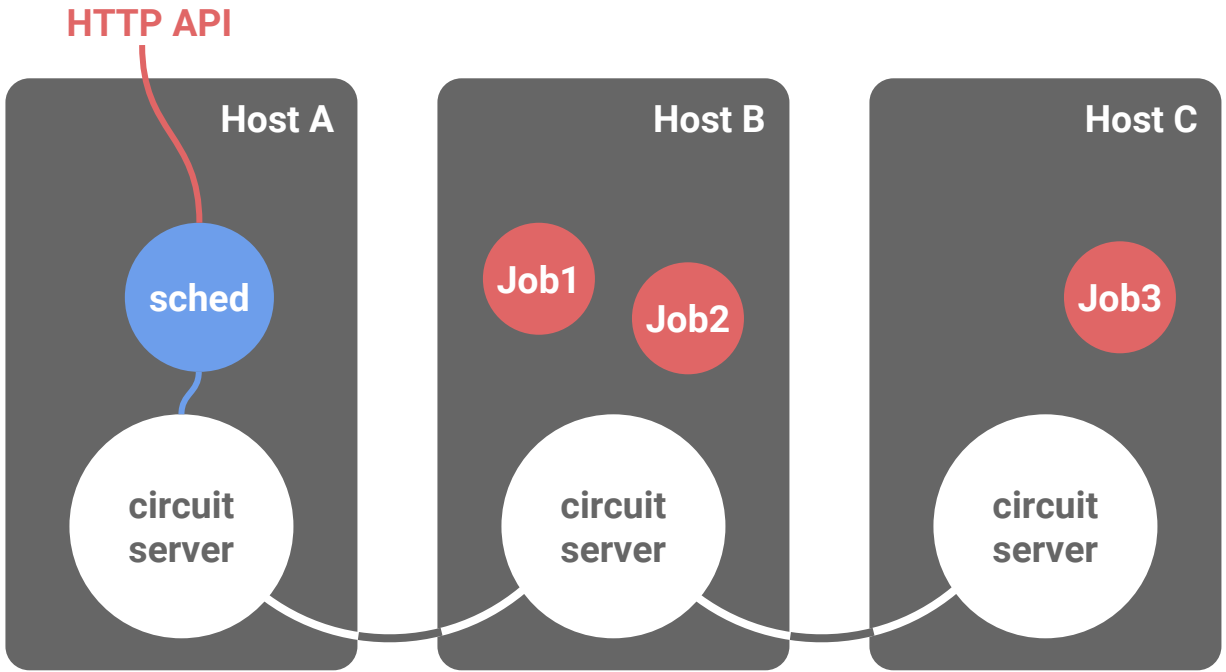
Implementing a job scheduler

Scheduler: Design spec

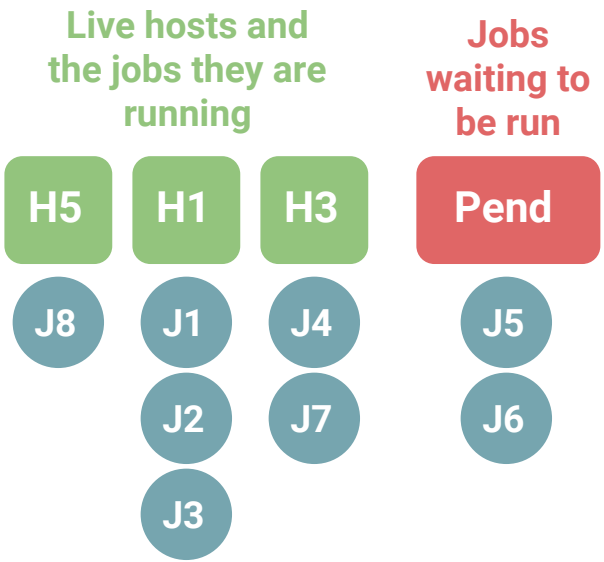
- User specifies:
  - Maximum number of jobs per host
  - Address of circuit server to connect to

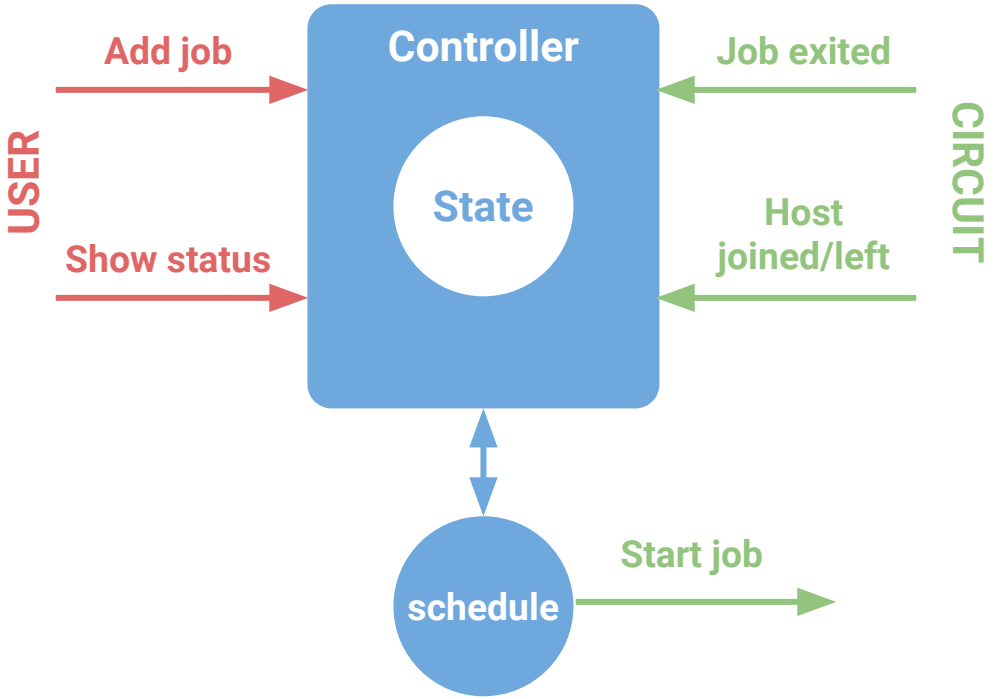
- HTTP API server:
  - Add job by name and command spec
  - Show status

Scheduler: Service architecture



Scheduler: State and logic



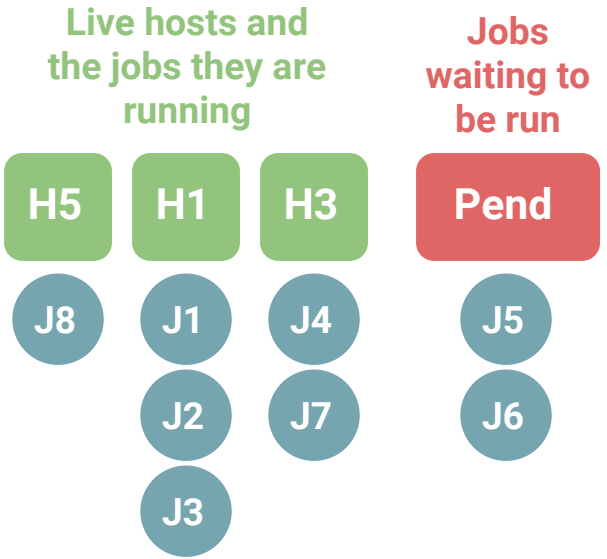


Scheduler: Main

```
import “github.com/gocircuit/circuit/client”

var flagAddr    = flag.String(“addr”, “”, “Circuit to connect to”)
var flagMaxJobs = flag.Int(“maxjob”, 2, “Max jobs per host”)

func main() {
    flag.Parse()
    defer func() {
        if r := recover(); r != nil {
            log.Fatalf("Could not connect to circuit: %v", r)
        }
    }()
    conn := client.Dial(*flagAddr, nil)
    controller := NewController(conn, *flagMaxJobs)
    … // Link controller methods to HTTP requests handlers.
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Scheduler: Controller state

```
type Controller struct {
    client         *client.Client
    maxJobsPerHost int
    sync.Mutex // Protects state fields.
    jobName        map[string]struct{}
    worker         map[string]*worker
    pending        []*job
}

type worker struct {
    name string
    job  []*job
}

type job struct {
    name string
    cmd  client.Cmd
}
```



Scheduler: Start controller

```
func NewController(conn *Client, maxjob int) *Controller {
    c := &Controller{…} // Initialize fields.
    // Subscribe to notifications of hosts joining and leaving.
    c.subscribeToJoin()
    c.subscribeToLeave()
    return c
}
```

Scheduler: Subscribe to host join/leave events

```
func (c *Controller) subscribeToJoin() {
    a := c.client.Walk([]string{c.client.ServerID(), "controller", "join"})
    a.Scrub()
    onJoin, err := a.MakeOnJoin()
    if err != nil {
        log.Fatalf("Another controller running on this circuit server.")
    }
    go func() {
        for {
            x, ok := onJoin.Consume()
            if !ok {
                log.Fatal("Circuit disappeared.")
            }
            c.workerJoined(x.(string)) // Update state.
        }
    }()
}
```

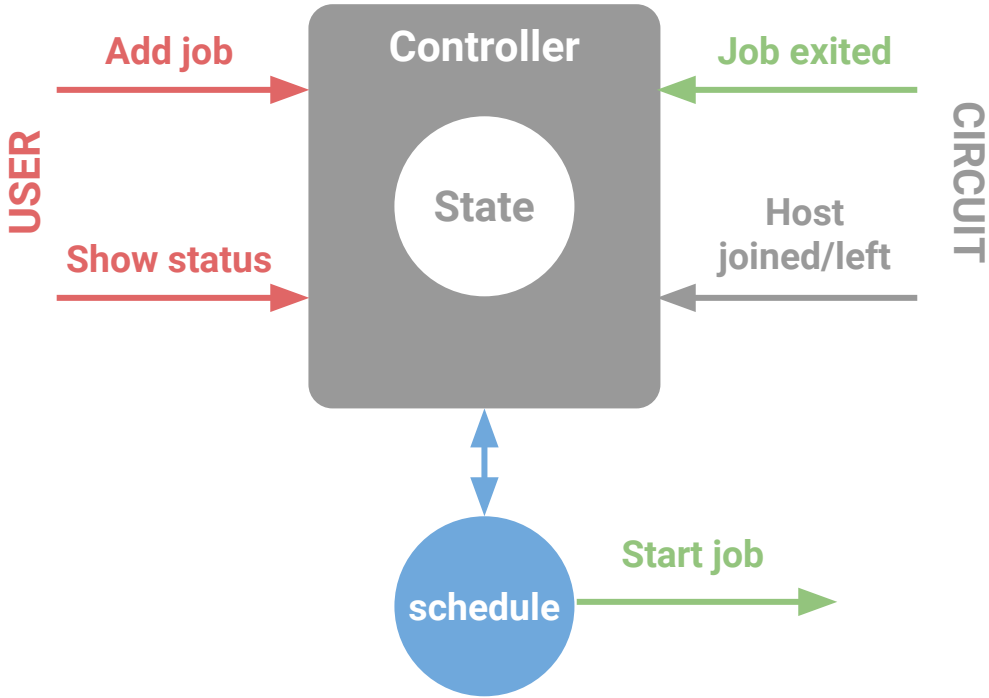
Place subscription on server the scheduler is connected to.

Pick a namespace for scheduler service.

Scheduler: Handle host join/leave

```
func (c *Controller) workerJoined(name string) {
    c.Lock()
    defer c.Unlock()
    … // Update state structure. Add worker to map with no jobs.
    go c.schedule()
}

func (c *Controller) workerLeft(name string) {
    c.Lock()
    defer c.Unlock()
    … // Update state structure. Remove worker from map.
    go c.schedule()
}
```

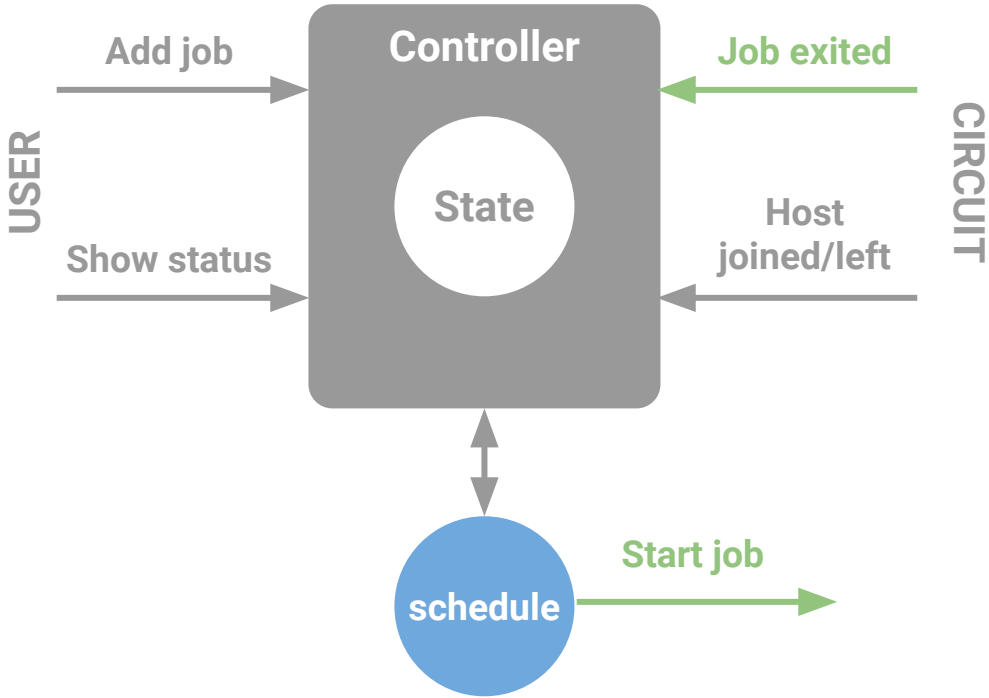
Scheduler: So far ...



Scheduler: User requests

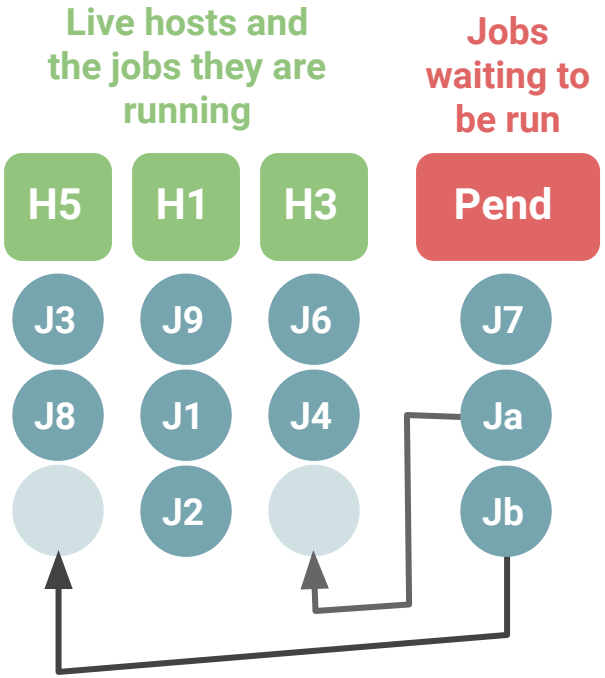
```
func (c *Controller) AddJob(name string, cmd Cmd) {
    c.Lock()
    defer c.Unlock()
    … // Update state structure. Add job to pending queue.
    go c.schedule()
}

func (c *Controller) Status() string {
    c.Lock()
    defer c.Unlock()
    … // Print out state to string.
}
```

Scheduler: So far ...



Scheduler: Controller state

```
func (c *Controller) schedule() {
    c.Lock()
    defer c.Unlock()
    // Compute job-to-worker matching
    var match []*match = …
    for _, m := range match {
        … // Mark job as running in worker
        go c.runJob(m.job, m.worker)
    }
}

type match struct {
    *job     // Job from pending
    *worker  // Worker below capacity
}
```



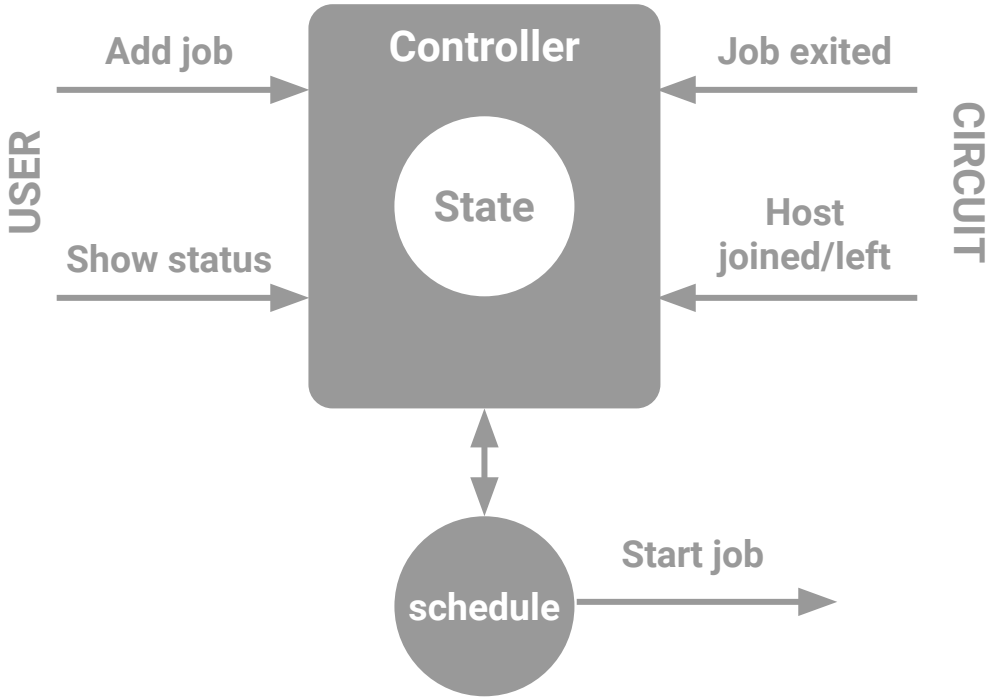
Scheduler: Run a job

```
func (c *Controller) runJob(job *job, worker *worker) {
    defer func() {
        if r := recover(); r != nil { // Worker died before job completed.
            … // Put job back on pending queue.
        }
    }()
    jobAnchor := c.client.Walk([]string{worker.name, "job", job.name})
    proc, err := jobAnchor.MakeProc(job.cmd)
    … // Handle error, another process already running.
    proc.Stdin().Close()
    go func() { // Drain Stdout. Do the same for Stderr.
        defer func() { recover() }() // In case of worker failure.
        io.Copy(drain{}, proc.Stdout())
    }()
    _, err = proc.Wait()
    … // Mark complete or put back on pending queue.
}
```

Place process on desired host.

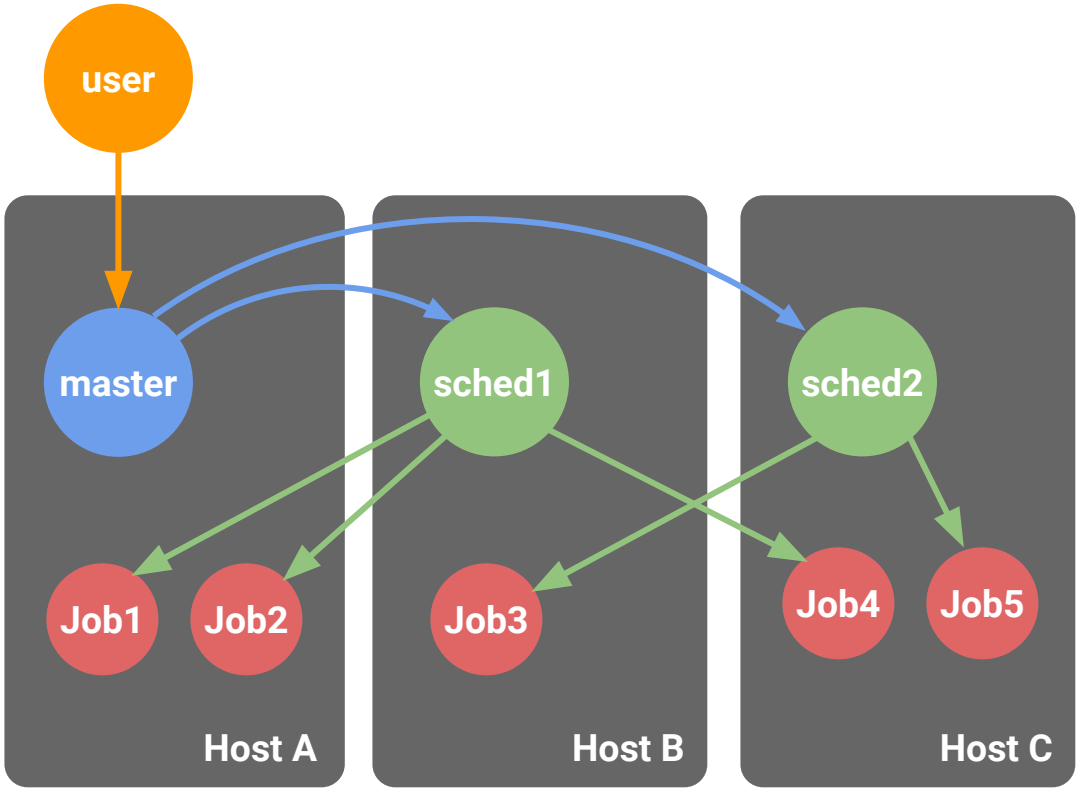
Pick namespace for jobs.

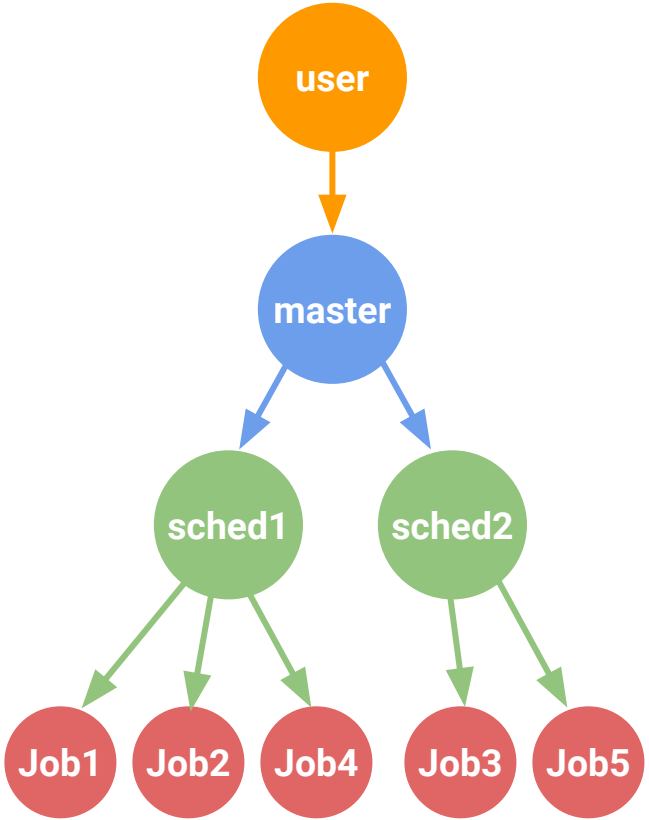
Scheduler: Demo.



Universal cluster binaries

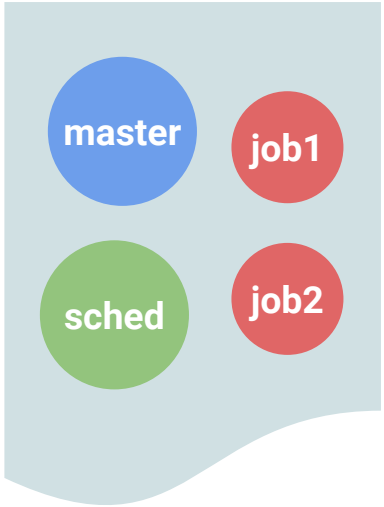
Recursive processes: Execution tree



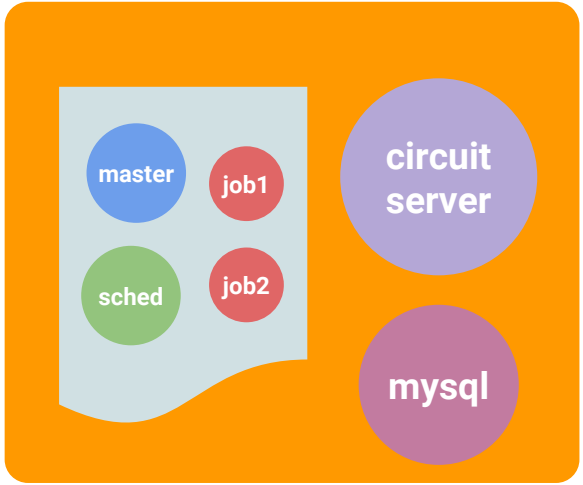


Universal distribution

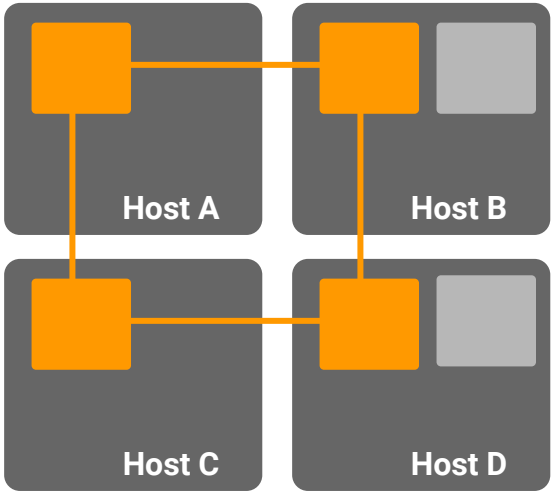
One Go binary that takes on different roles



Package binary & circuit in an executable container image



Customer runs container alongside other infrastructure with isolation



The vision forward

- Easy (only?) way to **share** any cloud system
  - “Ship with Circuit included” vs “Hadoop required”?
  - Circuit binary + Your binary = Arbitrary complex cloud
  - Like Erlang/OTP but language agnostic

Thank you.

Circuit website:

http://gocircuit.org

Source for Job Scheduler demo:

http://github.com/gocircuit/talks/deview2015

Twitter:

@gocircuit