Go Circuit: Distributing the Go Language and Runtime

Petar Maymounkov
p@gocircuit.org

Problem: DEV–OPS isolation

App complexity vs manual involvement

Distribute cloud apps
How to describe complex deploy topologies

Job control
Visibility into process ecosystem.
Abstract "map" of available hosts.

Elastic apps
Admin manual included?
Yes. Not elastic then.

Provision negotiation

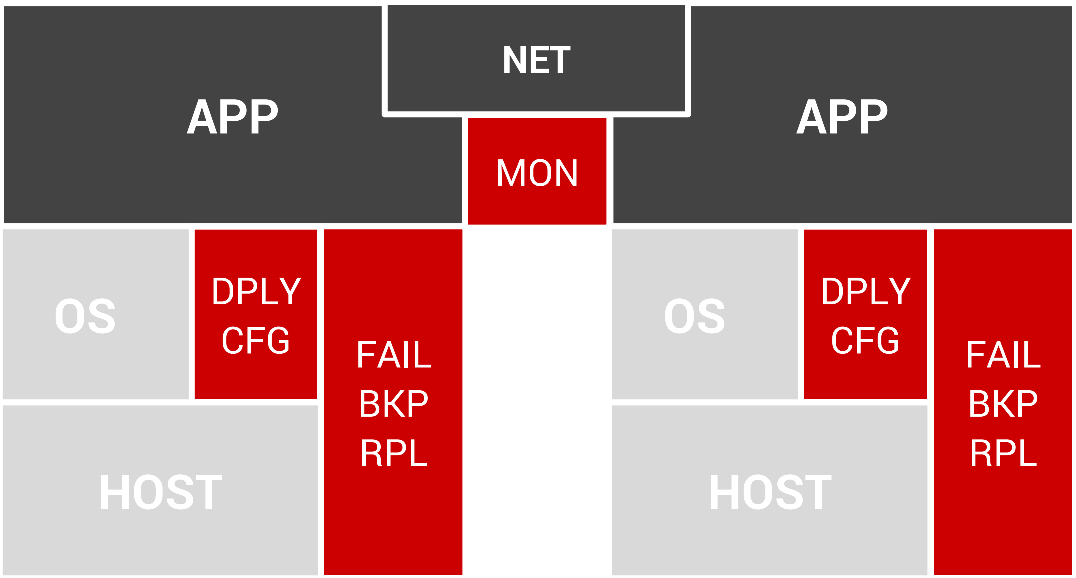
Cognitive load grows
Every app requires attention

Universal code format
Write once, run on any cluster
Provides for cluster-specific operations

Adopt incrementally

Co-habitable, minimal
Don't solve any app problems
Just abstract underlying capabilities

Too much humans in the stack. SaaS?



Key operations (like re-shard, replicate, etc.) **should** be in the app if you want **consistency** and **automation**

- Generic apps cannot provision hosts or self-deploy remotely
- Frameworks usually suffer:
  - Heavy-weight, cross-integration
  - External technologies (Catch 22)

True division of responsibilities

Networking
Job execution
Hardware negotiation & provision

Solution concept

Overview

Universal distribution format = Go source

- Open source executables
- Control app exposure to libraries
- Easy source-to-source transform for instrumentation
- Tiny source footprint for hard apps

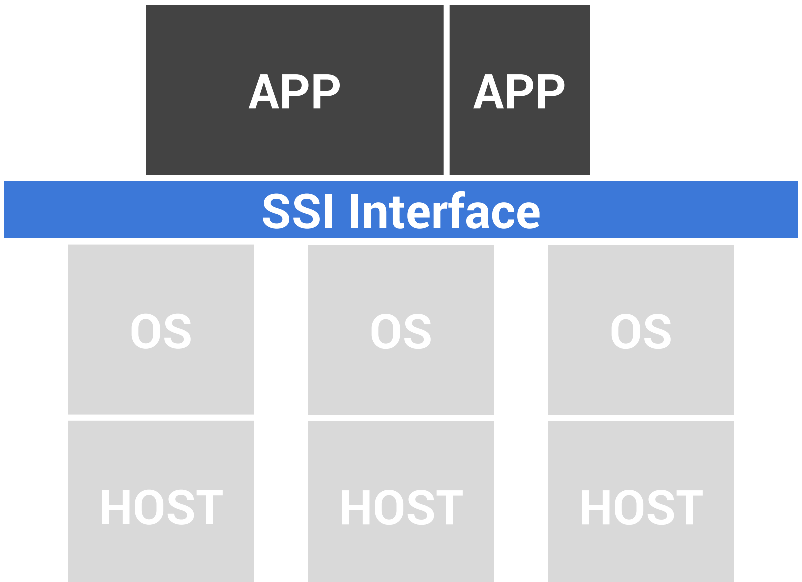
OPS vs APP isolation

- OPS need not understand how app works (black box)
- OPS controls networking, execution, provisioning, failure response, etc.

Elementary to deploy

- Compile entire solution into one “smart” binary
- Deploy sloppily
- App self-organizes at runtime

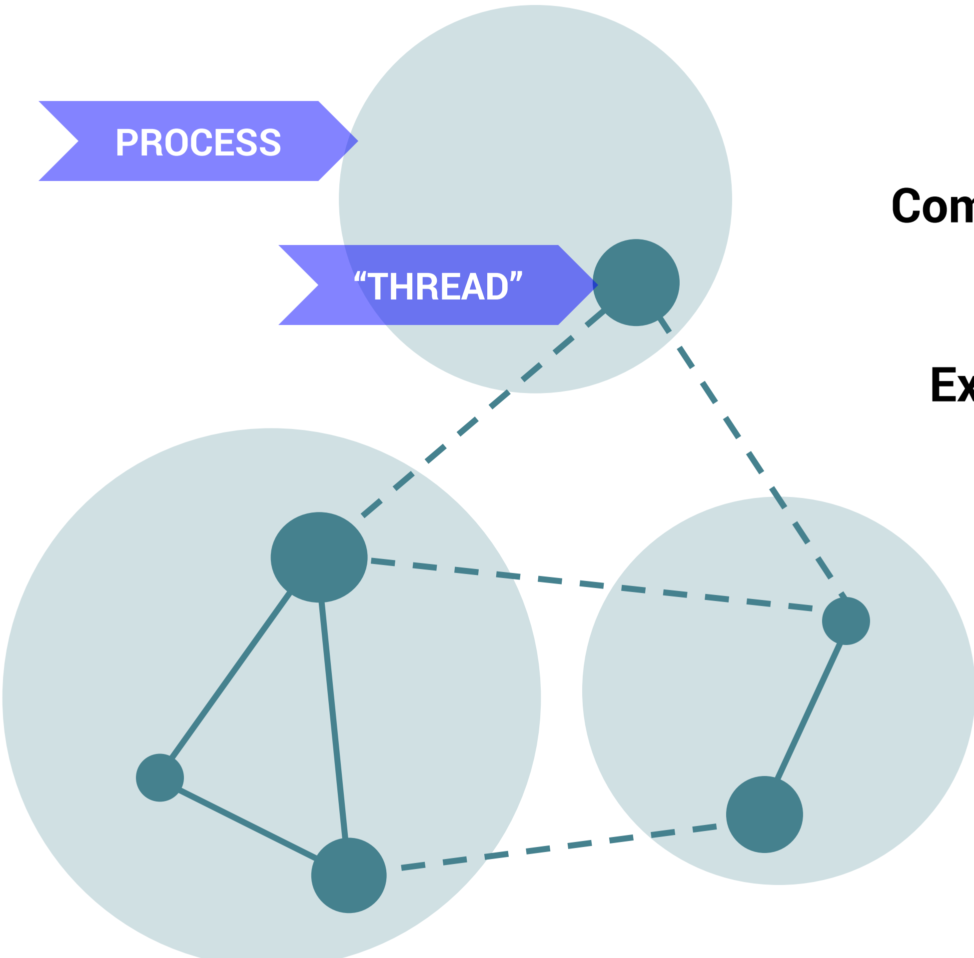
Minimal Single System Image (MSSI)



Goal: Abstract away and standardize capabilities/resources and behaviors (failures) of system.
Not: Conceal failures, add scheduling, provide reliability, etc.
The circuit is just a tool for building reliable systems efficiently and portably.

	Abstraction	Stack Level
Linux	POSIX	Binary
OpenSSI, etc. **monolithic**	POSIX	OS
Plan9 **modular**	File Protocol, 9P	Network
Erlang/OTP **monolithic**	CSP Functional Language	Language
Go Circuit **modular** **minimal**	CSP Imperative Language	Language

Concurrency inside, concurrency outside



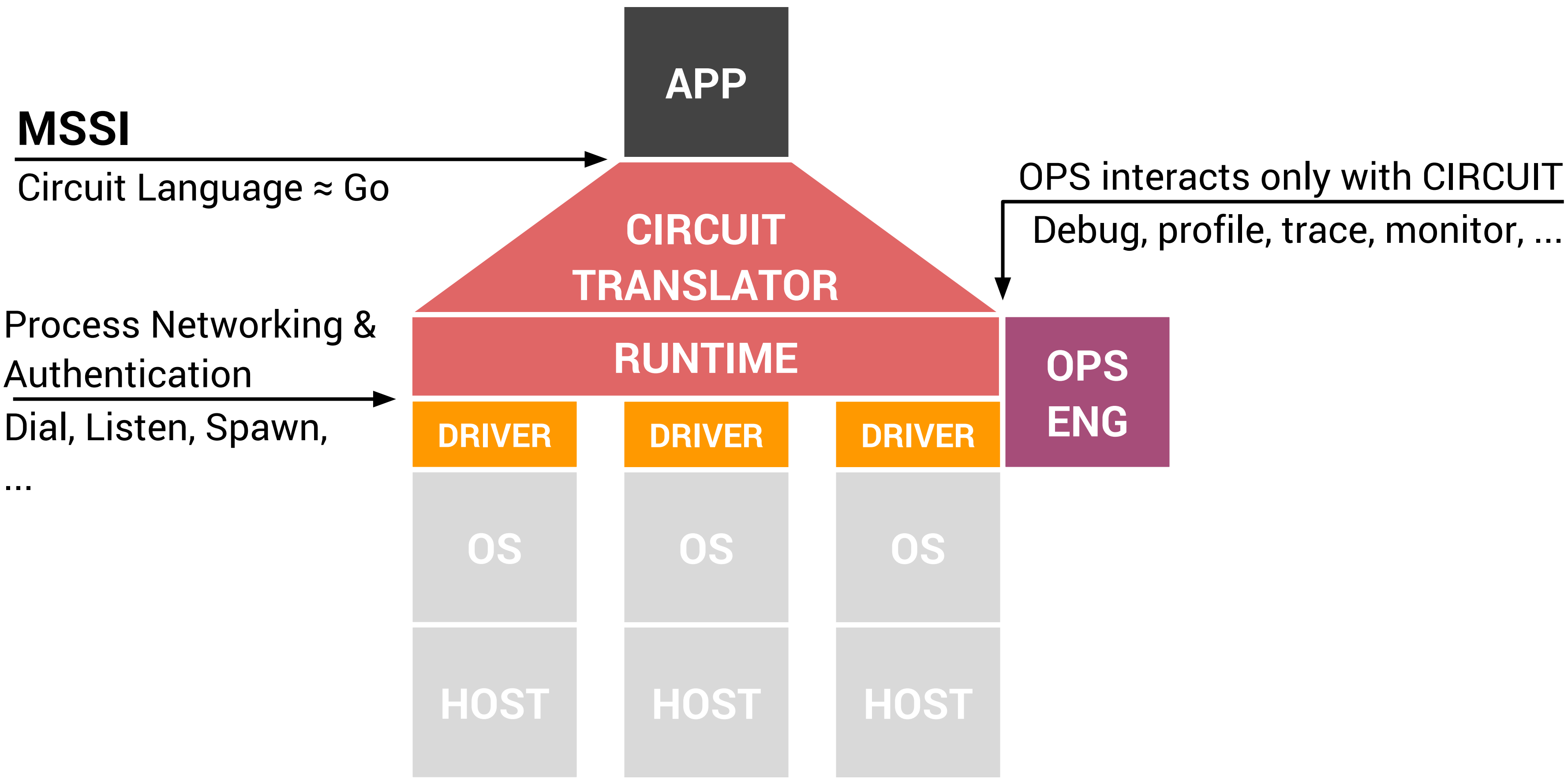
		Non-CSP	CSP
Comm	Protocols (DEV stack)	Sync Libraries	Lang
Exec	Framework (OPS stack)	Thread Async Libraries	Lang

Go Language

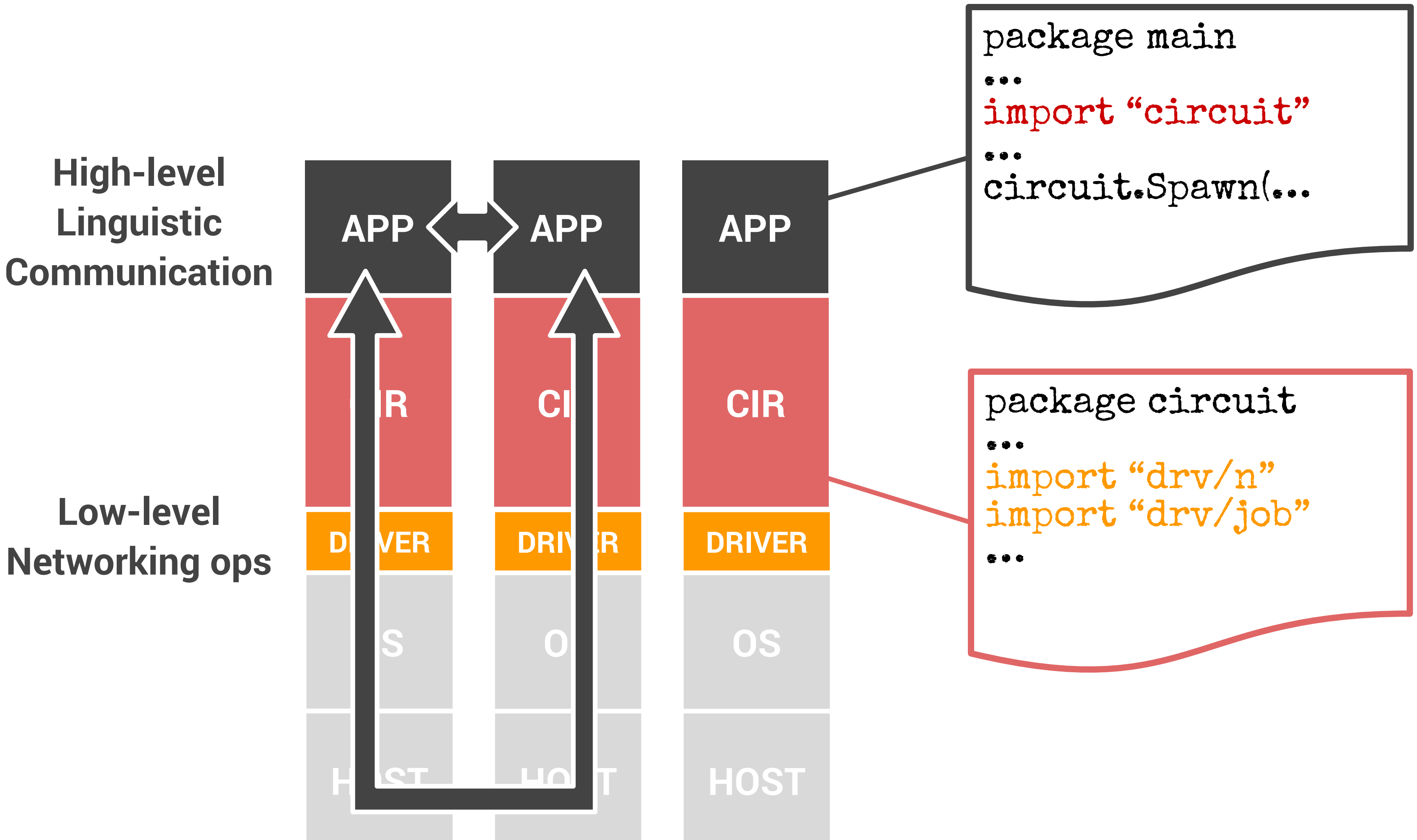
*** No cognitive switch**
*** Imperative**
*** Meaningful stack traces**
*** Debug, trace UI unchanged**

Concurrent Sequentially-communicating Processes (CSP)

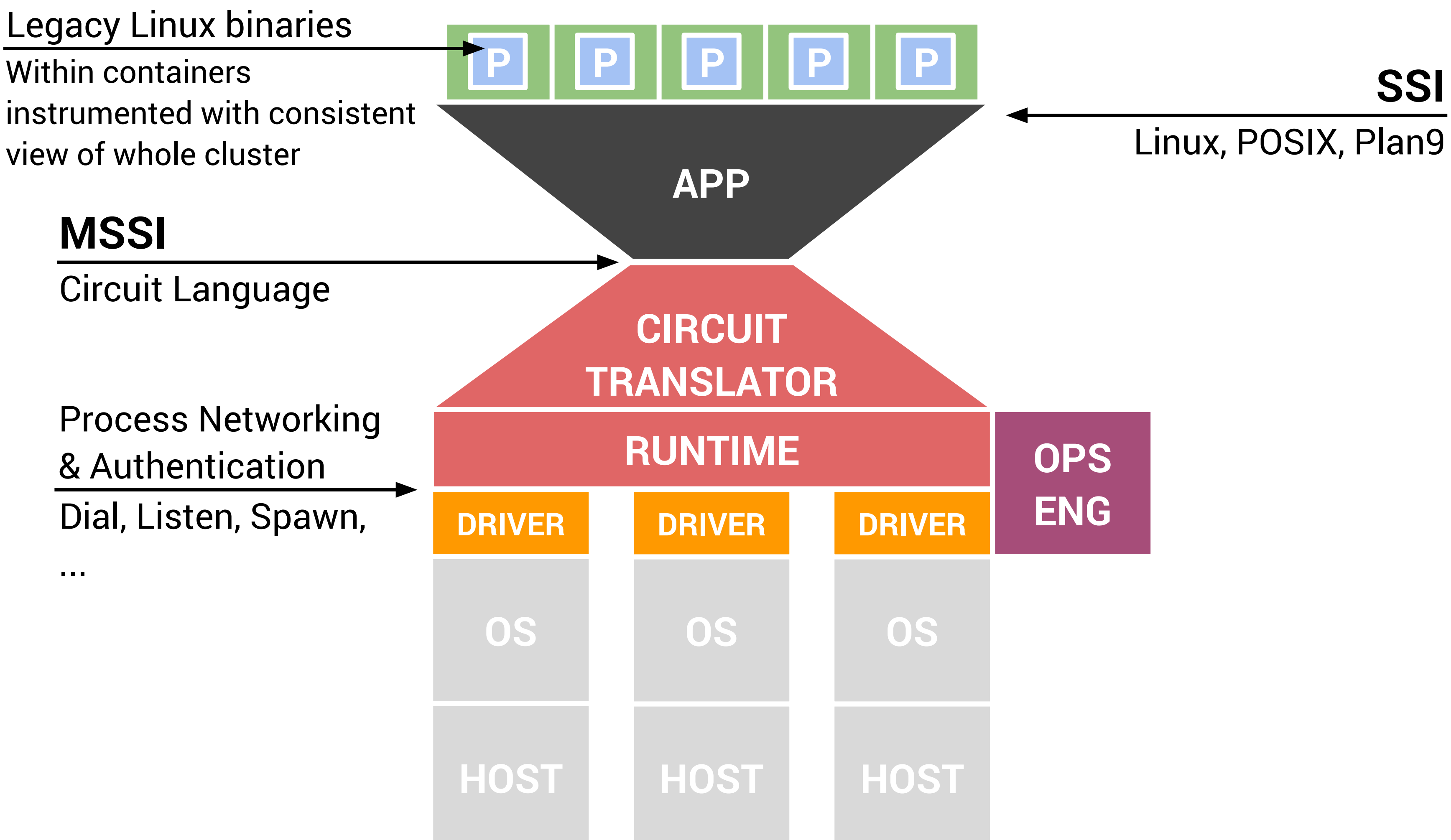
Solution stack



Solution translates language thru net



Example: Circuit + Docker = SSI Linux



Circuit Programming

Circuitizing a Go app and build

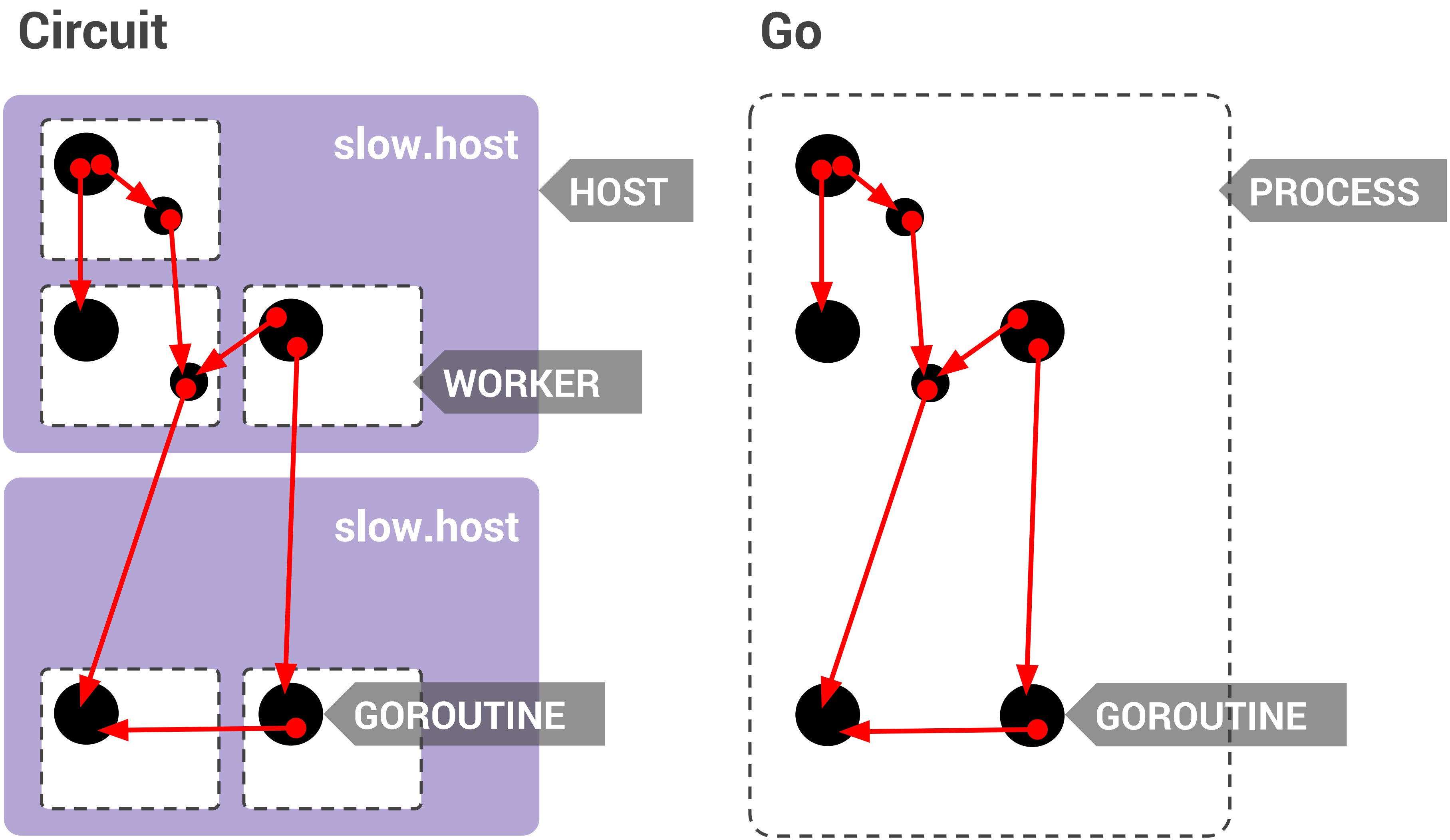
Add circuit runtime to any legacy Go program

```
package main

import _ "circuit/load"

func main() { ... }
```

Build

```
~/gopath/src/app/cmd$ go build
```

Go + spawn = distributed Go



Spawn

Run a function in this process

```
Compute(arg)
```

Run a function in a new process on a negotiated host

```
Spawn(
    “locus”,
    []string{“/db/shard”, “/critical”},
    SpawnCompute{},
    arg,
)
```

Declare spawn-able functions

```
type SpawnCompute struct{}

func (SpawnCompute) Compute(arg Arg) { ... }
```

Anchor file system (side note)

Global file system of live workers, arranged by the user

```
$ 4ls /...
/
/critical
/critical/R1a4124aa276f4a4e
/db
/db/shard
/db/shard/R1a4124aa276f4a4e
/db/shard/R8efd95a09bcde325
```

More on this later.

Spawn semantics

Returns

RETURN VALUES | WORKER ADDRESS | ERROR

```
func Spawn( ... ) ([]interface{}, n.Addr, error)
```

Worker dies as soon as function exits ...

```
func (SpawnCompute) Compute(arg Arg) (r1, r2 ReturnType) {
    ...
    return
}
```

... unless, we defer its death to the end of another goroutine.

```
func (SpawnCompute) Compute(arg Arg) {
    ...
    ExitAfter(func() { ... return })
    ...
    return
}
```

Values vs cross-interfaces

Communicate by value or cross-interface

```
func Compute(int, Struct, *T, []byte, map[Key]Value, interface{}, X) ...
```

```
type T struct {
    Cross X
    Value float64
}
```

Create a cross-interface to a local receiver

```
x := Ref(r)
```

Use a cross-interface: call its methods

```
reply = x.Call("MethodName", arg1, arg2) // Returns []interface{}
```

Cross-worker garbage collection

Cross-interfaces

```
func (SpawnCompute)
    Compute(X) (X, error)
{
    ...
    return Ref(v), nil
}
```

```
Spawn(
    “host”,
    nil,
    SpawnCompute{},
    Ref(v),
)
```

Permanent cross-interfaces

```
func (SpawnCompute)
    Compute(XPerm) (XPerm, error)
{
    ...
    return PermRef(v), nil
}
```

```
Spawn(
    “host”,
    nil,
    SpawnCompute{},
    PermRef(v),
)
```

Cross-call semantics

Cross-calls are not RPC

They return out-of-order, just like Go function calls do

```
var x X // Cross-interface
...
go func() {
    ch <- x.Call("Find", "needle")
}()
...
go func() {
    ch <- x.Call("Find", "haystack")
}()
...
<-ch // Wait for whoever returns first
...
```

Error propagation

Retain natural method signature

```
func (s *Server) ServeHTML(url string) (string, error) { ... }
```

Decouple application and system errors at call site

```
var x X // x is a cross-interface to a remote *Server instance
...
defer recover() // Catch system errors as panics
...
result := x.Call("ServerHTML", url) // Get app errors in return
...
```

Worker services

Expose a cross-interface to a worker object

```
Listen("ingest", r)
```

Recall that spawn returns a worker address

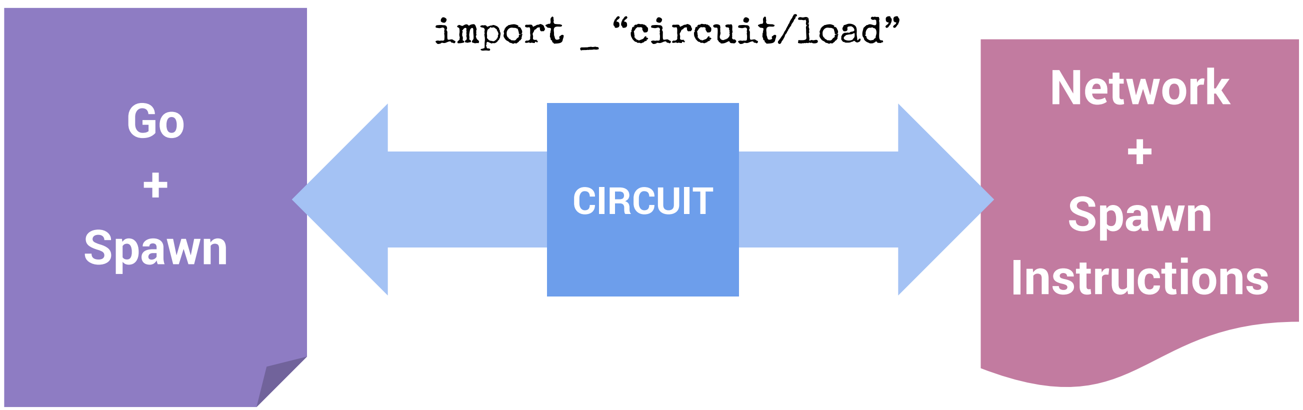
```
result, addr, err := Spawn(...)
```

Dial a worker and obtain an exposed cross-interface

```
x := Dial(addr, "ingest")
```

Circuit Architecture

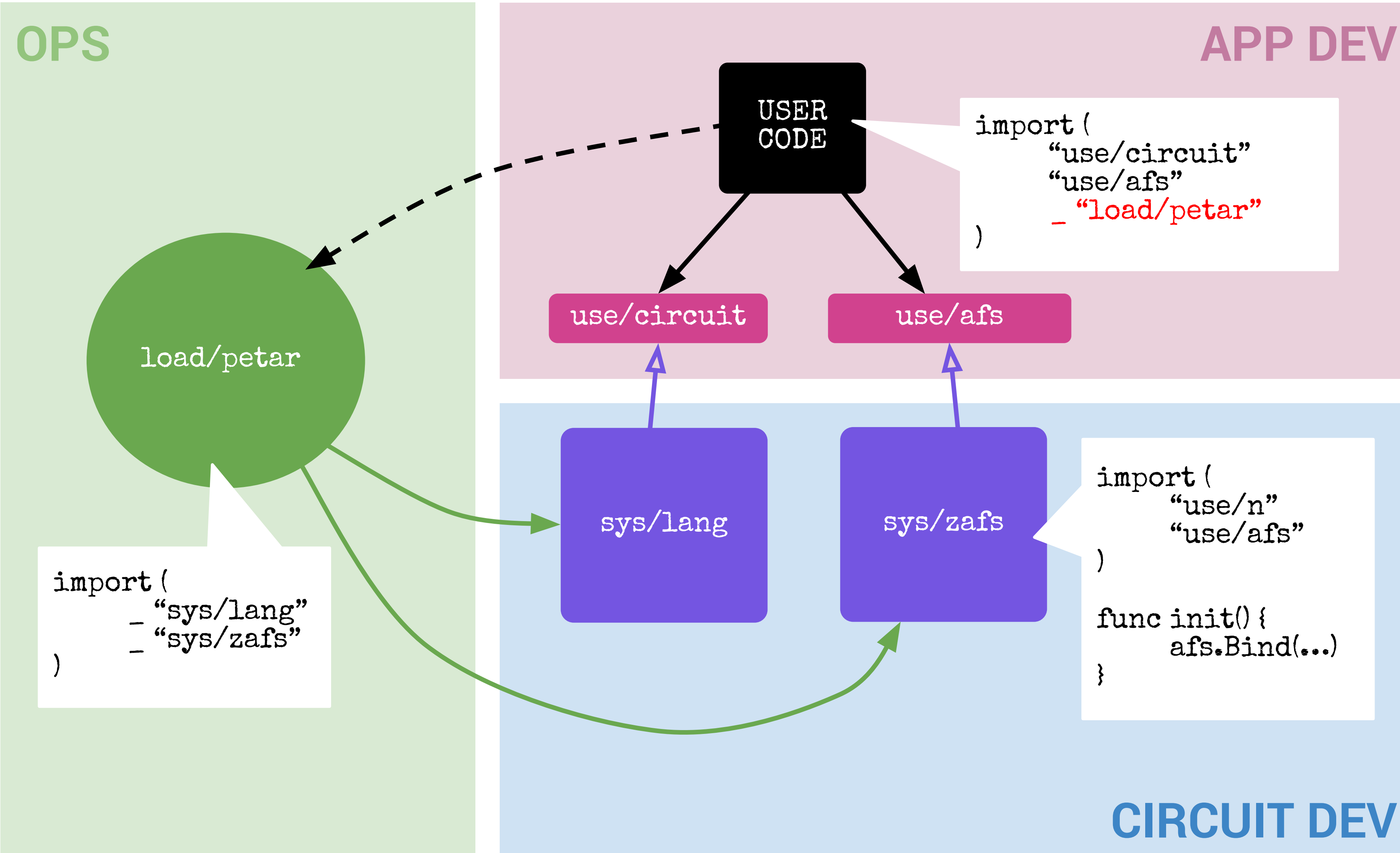
The circuit translates




```
type S struct {
      A float64
      B []X
}

func (R) F(s *S, x X) (X, error) {
      ...
      return Ref(v), nil
}

...

defer recover()
result := x.Call("F", s1, y)
```

- Reflect on in/out types of F
- Recurse within composite types to find all cross-interface fields
- Read input arguments
- Check type safety
- Dial remote worker address
- Encode and send cross-call request
- Read response or error
- etc.

Modules decouple. Universal code.



Circuit Facilities

Anchor FS + cli tools = debug, profile, …

Global file system of live workers, arranged by the user

```
$ 4ls /...
/
/critical
/critical/R1a4124aa276f4a4e
/db
/db/shard
/db/shard/R1a4124aa276f4a4e
/db/shard/R8efd95a09bcde325
```

Kill

```
$ 4kill /critical/...
```

Stack traces

```
$ 4ls /db/shard/... | xargs 4stk
```

Profile for 30 sec

```
$ 4cpu /critical/... 30
```

Monitor worker vitals (mem, cpu, …)

```
$ 4top /critical/...
```

The end.

Disclaimer. Talk describes soon-to-be-released R2.

Skipped Teleport Transport. See **github.com/petar/GoTeleport**

Twitter **@gocircuit**

gocircuit.org

Appendix

Historical perspective on CSP systems