TAMING COMPLEXITY IN JAVASCRIPT WITH MACHINA.JS

JIM COWART / @IFANDELSE

WHY ARE WE HERE?

- Workflow in JavaScript can be fun! How?

WHY ARE WE HERE?

- Workflow in JavaScript can be fun! How?
- What are Finite State Machines?

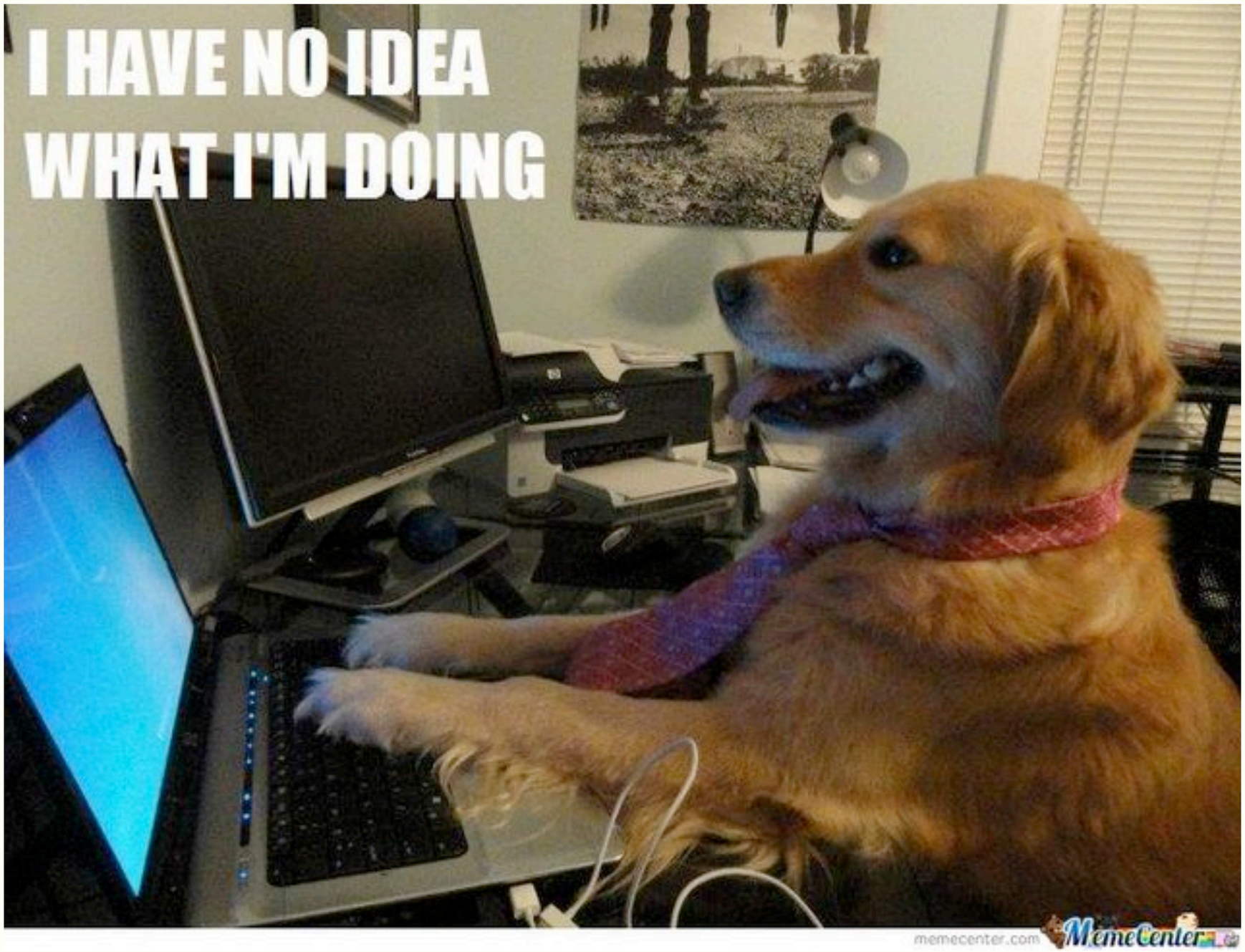
WHY ARE WE HERE?

- Workflow in JavaScript can be fun! How?
- What are Finite State Machines?
- What FSM behaviors are provided by machina.js?

WHY ARE WE HERE?

- Workflow in JavaScript can be fun! How?
- What are Finite State Machines?
- What FSM behaviors are provided by machina.js?
- How can we use this in the real world?

WHO AM I?



WHO AM I?

- Developer Advocate @ Telerik
- @ifandelse
- OSS Author & Contributor
  (http://github.com/ifandelse)
- Amateur Pattern Geek

PRESENTATION ~~RULES~~?

WHAT IS THE PROBLEM?

- How do you:
  - Manage online/offline state in your app?
  - Handle complex UI Workflow?
  - How do you structure order-dependent initialization?

WHAT IS THE PROBLEM?

- How do you:
  - **Manage online/offline state in your app?**
  - Handle complex UI Workflow?
  - How do you structure order-dependent initialization?

CONNECTIVITY DETECTION OPTIONS

- jQuery ajaxError event

CONNECTIVITY DETECTION OPTIONS

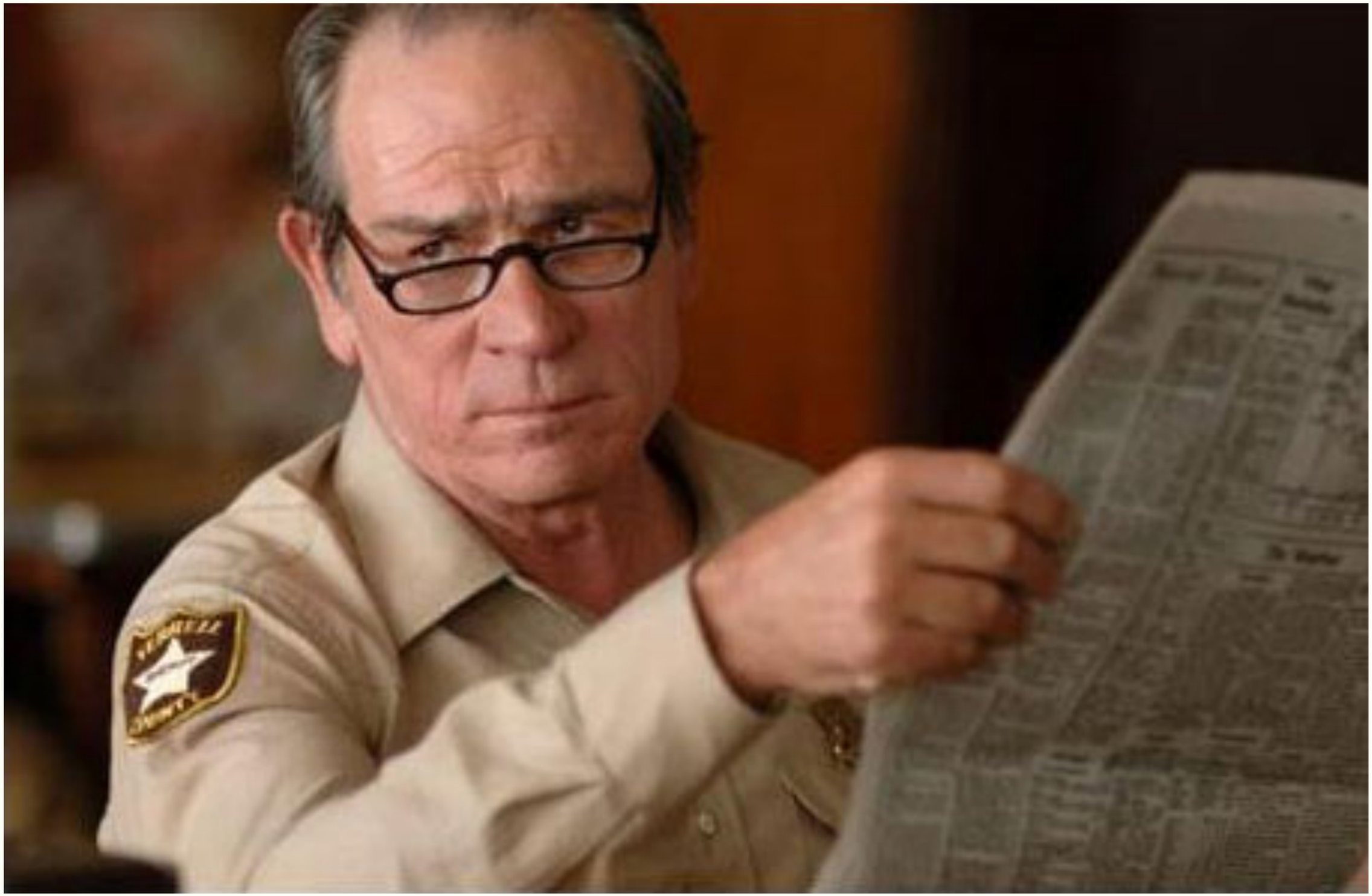
- jQuery ajaxError event
- navigator.onLine

A QUICK ASIDE ABOUT NAVIGATOR.ONLINE

“This attribute is inherently unreliable. A computer can be connected to a network without having Internet access.”

Hugs and Kisses,

- the W3C



Thursday, August 29, 13

CONNECTIVITY DETECTION OPTIONS

- jQuery ajaxError event
- navigator.onLine
- addEventListener(“online”, handler);*

CONNECTIVITY DETECTION OPTIONS

- jQuery ajaxError event
- navigator.onLine
- addEventListener(“online”, handler);*
- navigator.network.connection.type (PhoneGap)

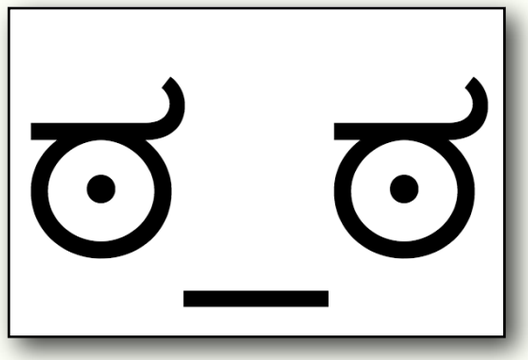
CONNECTIVITY DETECTION OPTIONS

- jQuery ajaxError event
- navigator.onLine
- addEventListener(“online”, handler);*
- navigator.network.connection.type (PhoneGap)
- window.applicationCache error

CONNECTIVITY DETECTION

```
1   if (navigator.onLine) {
2       save(customer);
3   } else {
4       queueUpForLater(customer);
5   }
```

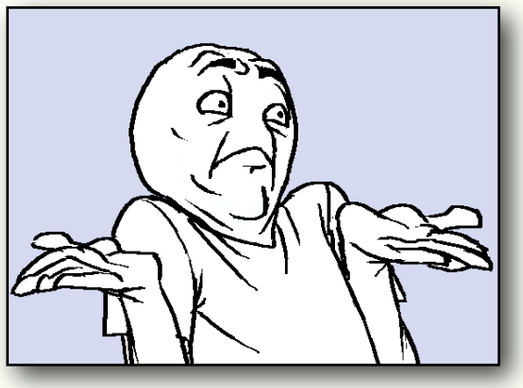
CONNECTIVITY DETECTION



Who wants to have this all over the application?

```
1   if (navigator.onLine) {
2     save(customer);
3   } else {
4     queueUpForLater(customer);
5   }
```

```
1  // assuming we have an app object
2  window.addEventListener("offline",function(){
3     app.setStatus("offline");
4  });
5  window.addEventListener("online",function(){
6     app.setStatus("online");
7  });
8  window.applicationCache.addEventListener(
9     "error",
10    function() {
11       app.setStatus("offline");
12    }
13 );
```



Is this better?

Than before, yes. Overall, NO.

```
1   // assuming we have an app object
2   window.addEventListener("offline",function(){
3     app.setStatus("offline");
4   });
5   window.addEventListener("online",function(){
6     app.setStatus("online");
7   });
8   window.applicationCache.addEventListener(
9     "error",
10    function() {
11      app.setStatus("offline");
12    }
13  );
```

Sure...

Single Source of Application State

>

Peppered Spaghetti Branching

But What About...

- The Commuter Problem

But What About...

- The Commuter Problem
- False Negatives & Positives

But What About...

- The Commuter Problem
- False Negatives & Positives
- Deliberate choice to go offline

But What About...

- The Commuter Problem
- False Negatives & Positives
- Deliberate choice to go offline
- Testability

We have lots of different abstractions for *_similar_* input

AN ABSTRACTION THAT....

Reacts differently to the same input depending on state

MAKE THIS EASY...

While We Are	And This Happens	Let's Do This
Online	http request	send request to server
	window.offline	set app to offline
Offline	http request	queue request up
	window.online	set app to online

FINITE STATE

MACHINE

WHAT IS A FINITE STATE MACHINE?

?

WHAT IS A FINITE STATE MACHINE?

- A computational abstraction that:

WHAT IS A FINITE STATE MACHINE?

- A computational abstraction that:
  - Has a finite number of states in which it can exist

WHAT IS A FINITE STATE MACHINE?

- A computational abstraction that:
  - Has a finite number of states in which it can exist
  - Can only be in one state at any time

WHAT IS A FINITE STATE MACHINE?

- A computational abstraction that:
  - Has a finite number of states in which it can exist
  - Can only be in one state at any time
  - Accepts input

WHAT IS A FINITE STATE MACHINE?

- A computational abstraction that:
  - Has a finite number of states in which it can exist
  - Can only be in one state at any time
  - Accepts input
  - Can produce output determined by state &/ or input

WHAT IS A FINITE STATE MACHINE?

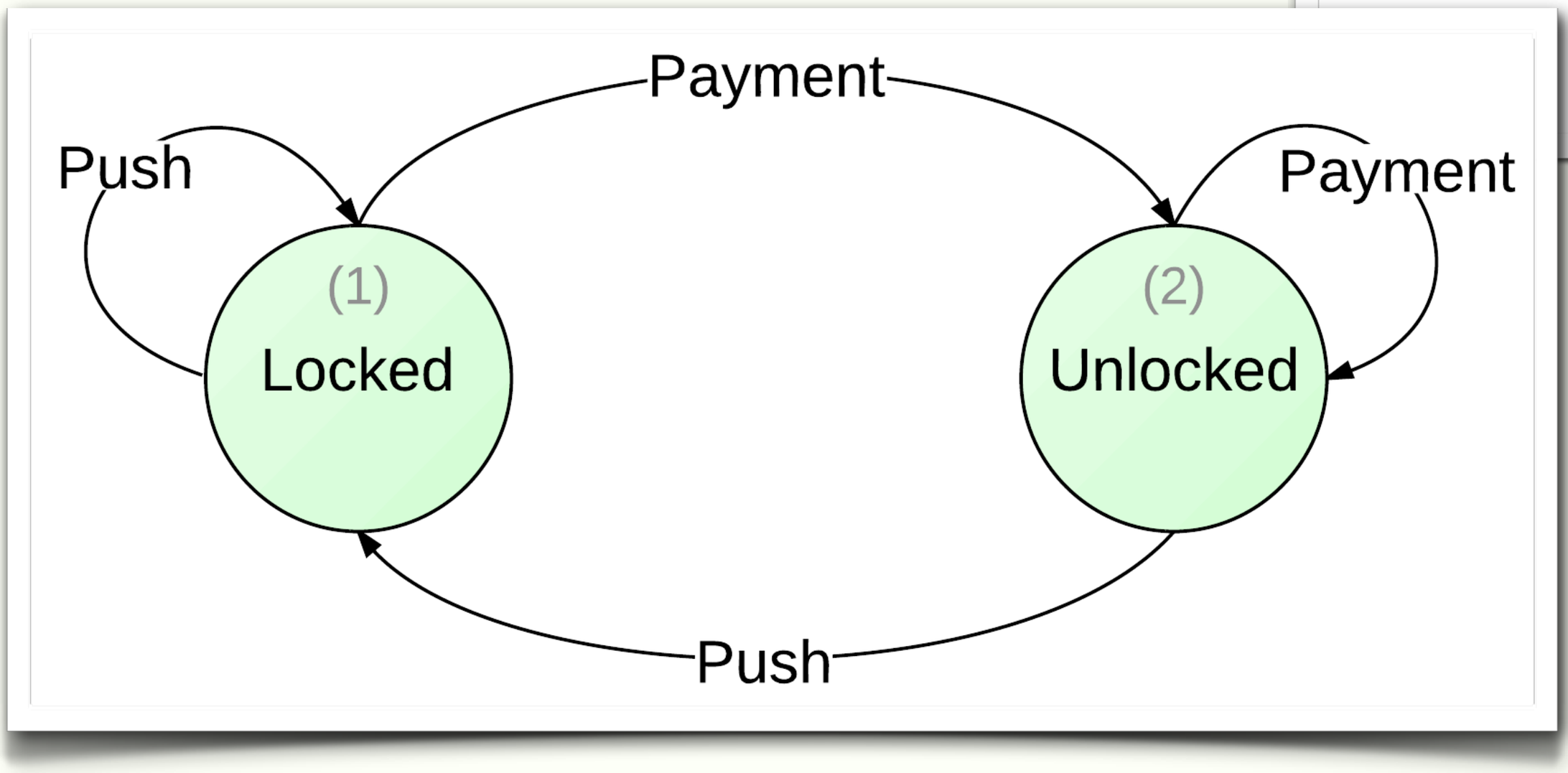
- A computational abstraction that:
  - Has a finite number of states in which it can exist
  - Can only be in one state at any time
  - Accepts input
  - Can produce output determined by state &/ or input
  - Can transition from one state to another*

THAT WAS A LOT OF TEXT ON ONE SCREEN

REAL WORLD EXAMPLE*

State	Input	Next State	Output
Locked	payment	Unlocked	turnstile released
	push	Locked	None
Unlocked	payment	Unlocked	None
	push	Locked	locks turnstile

DIAGRAMS FTW?



STILL AWAKE?

GOOD!

HERE'S SOME FINE

PRINT ON STATE

MACHINES

TWO BASIC TYPES

- Acceptor

TWO BASIC TYPES

- Acceptor
- Transducer
- Moore machine - output depends on state (entry actions)
- Mealy machine - output depends on state and input

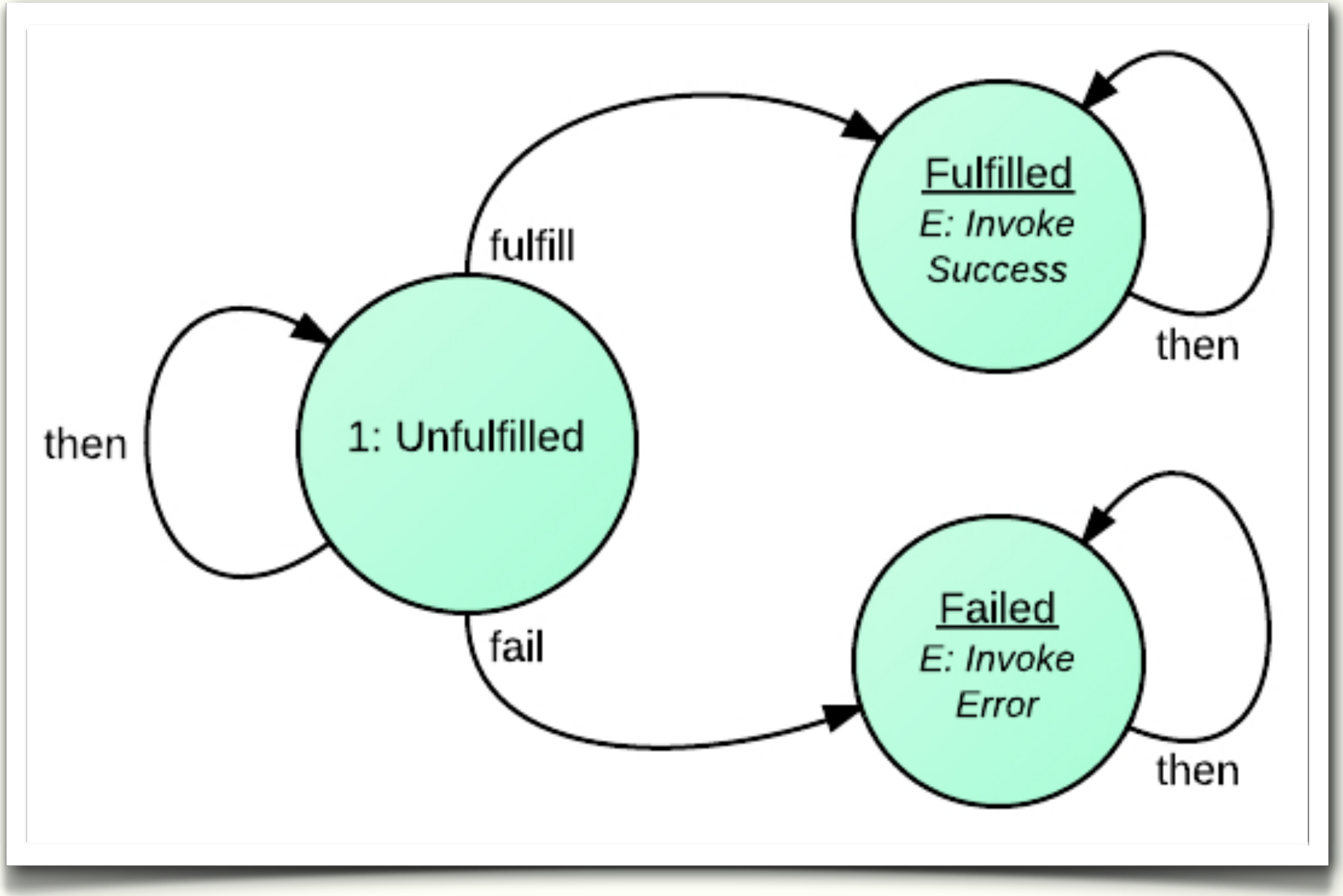
- Deterministic - only one transition possible for each state

- Non-deterministic - zero or more transitions possible from each state

DETERMINISM

- Deterministic - only one transition possible for each state
- Non-deterministic - zero or more transitions possible from each state

PROMISES & FSMS



ENTER MACHINA.JS

ENTER MACHINA.JS

- STATES ARE ORGANIZED INTO A ‘STATES’ OBJECT ON THE FSM

ENTER MACHINA.JS

- STATES ARE ORGANIZED INTO A 'STATES' OBJECT ON THE FSM
- EACH OBJECT PROPERTY IS A STATE

ENTER MACHINA.JS

- STATES ARE ORGANIZED INTO A 'STATES' OBJECT ON THE FSM
- EACH OBJECT PROPERTY IS A STATE
- EACH STATE OBJECT'S MEMBERS ARE FUNCTIONS* THAT RESPOND TO INPUT

ENTER MACHINA.JS

- STATES ARE ORGANIZED INTO A 'STATES' OBJECT ON THE FSM
- EACH OBJECT PROPERTY IS A STATE
- EACH STATE OBJECT'S MEMBERS ARE FUNCTIONS* THAT RESPOND TO INPUT
- CALLING "transition(stateName)" CHANGES STATE

ENTER MACHINA.JS

- FSM MAPS INPUT TO MATCHING HANDLER NAME:

  “handle(inputName, args*)”

ENTER MACHINA.JS

- FSM MAPS INPUT TO MATCHING HANDLER NAME:

  “handle(inputName, args*)”

- “_onEnter” & “_onExit” & “*”

ENTER MACHINA.JS

- FSM MAPS INPUT TO MATCHING HANDLER NAME:
  “handle(inputName, args*)”
- “_onEnter” & “_onExit” & “*”
- “deferUntilTransition([stateName])”

ENTER MACHINA.JS

- FSM MAPS INPUT TO MATCHING HANDLER NAME:
  “handle(inputName, args*)”
- “_onEnter” & “_onExit” & “*”
- “deferUntilTransition([stateName])”
- “deferUntilNextHandler()”

ENTER MACHINA.JS

- FSM MAPS INPUT TO MATCHING HANDLER NAME:
  "handle(inputName, args*)"
- "_onEnter" & "_onExit" & "*"
- "deferUntilTransition([stateName])"
- "deferUntilNextHandler()"
- BUILT-IN EVENT EMITTER


Sigh...so much API info... what if I need to *blink*?

OMG CODE!

Instances and Constructors

```
var fsm = new machina.Fsm({ ... });
```

```
var Fsm = machina.Fsm.extend({ ... });
```

OMG CODE!

Instances and Constructors

```
var fsm = new machina.Fsm({ ... });
```

What goes here?

```
var Fsm = machina.Fsm.extend({ ... });
```

```
var fsm = new machina.Fsm({
    initialState: “locked”,
    states: {
        locked: {
            payment: “unlocked”
        },
        unlocked: {
            push: “locked”
        }
    }
});
```

```
locked: {
  payment: “unlocked”
}
```

This is short for this:

```
locked: {
  payment: function() {
    this.transition(“unlocked”);
  }
}
```

```
var fsm = new machina.Fsm({
    initialState: “locked”,
    states: {
        locked: {
            payment: “unlocked”
        },
        unlocked: {
            push: “locked”
        }
    }
});
```

```
// you could do this
fsm.handle("push"); // sorry, not so much
fsm.handle("payment"); // transition-> unlocked
fsm.handle("payment"); // oops, wasted money
fsm.handle("push"); // yay, I get through
```

OVERSIMPLIFIED USAGE

```
// you could do this
fsm.handle(“push”); // sorry, not so much
fsm.handle(“payment”); // transition-> unlocked
fsm.handle(“payment”); // oops, wasted money
fsm.handle(“push”); // yay, I get through
```

OVERSIMPLIFIED USAGE

```
// but you'll probably prefer to do this
// i.e. - top level methods wrapping handle()
fsm.push(); // sorry, not so much

fsm.pay(); // transition-> unlocked

fsm.pay(); // oops, wasted money

fsm.push(); // yay, I get through
```

HOW DO WE APPLY THIS?

HOW DOES IT HELP MANAGE CONNECTIVITY STATE?

CONNECTIVITY STATES

- Online
- Offline (the user said so!)
- Disconnected (Oops, no connection)
- Probing (detecting if we're online)

ONLINE STATE INPUT & TRANSITIONS

State	Input	Next State	Output
Online	window.offline	Probing	Emit Transition Event
	appCache.error	Probing	Emit Transition Event
	request.timeout	Probing	Emit Transition Event
	go.offline	Offline	Emit Transition Event

OFFLINE STATE INPUT & TRANSITIONS

State	Input	Next State	Output
Offline	go.online	Probing	Emit Transition Event

DISCONNECTED STATE INPUT & TRANSITIONS

State	Input	Next State	Output
Disconnected	go.online	Probing	Emit Transition Event
	go.offline	Offline	Emit Transition Event
	window.online	Probing	Emit Transition Event
	appCache. downloading	Probing	Emit Transition Event

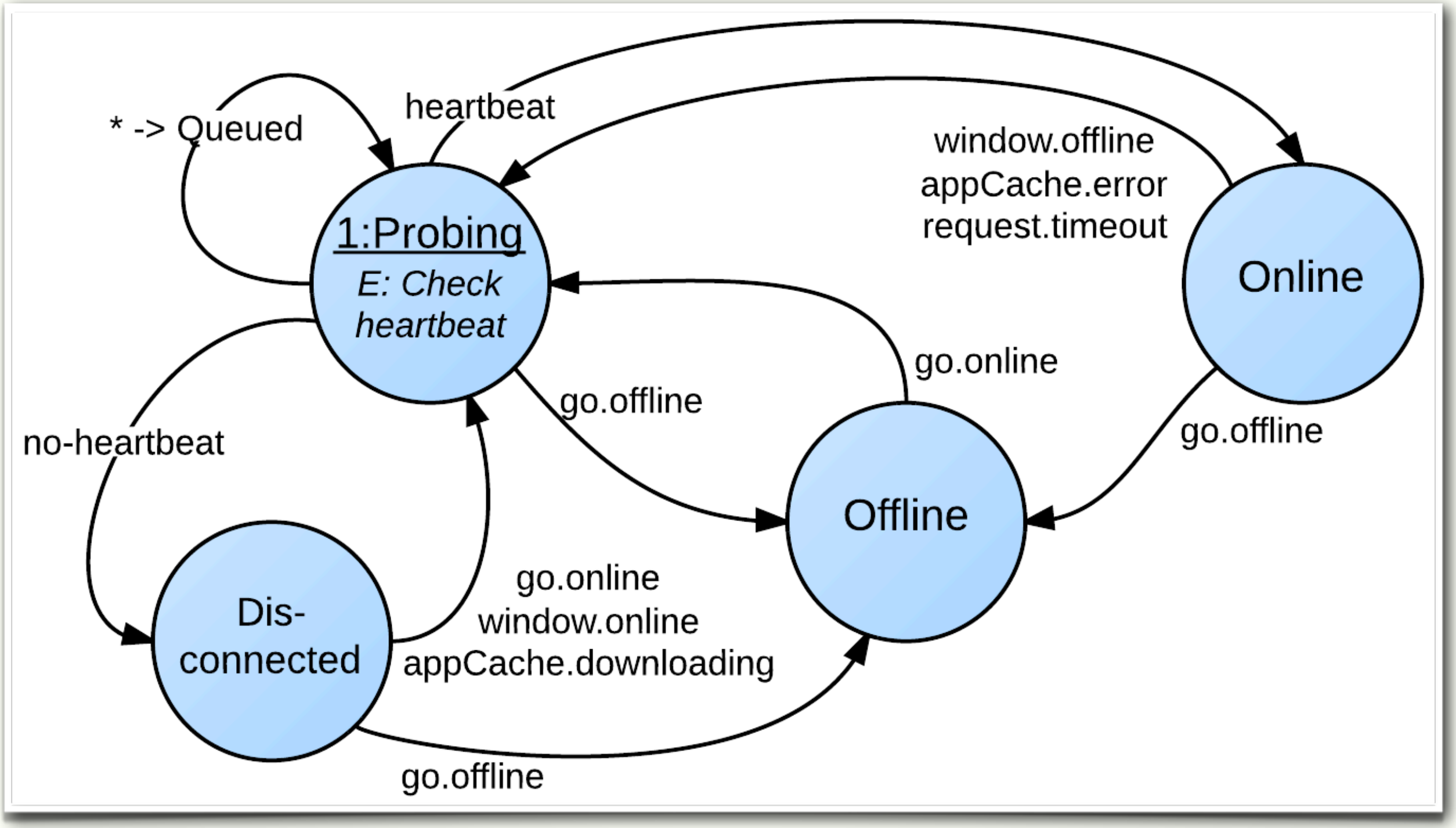
PROBING STATE INPUT & TRANSITIONS

State	Input	Next State	Output
Probing	heartbeat	Online	Emit Transition Event
	no-heartbeat	Disconnected	Emit Transition Event
	go.offline	Offline	Emit Transition Event

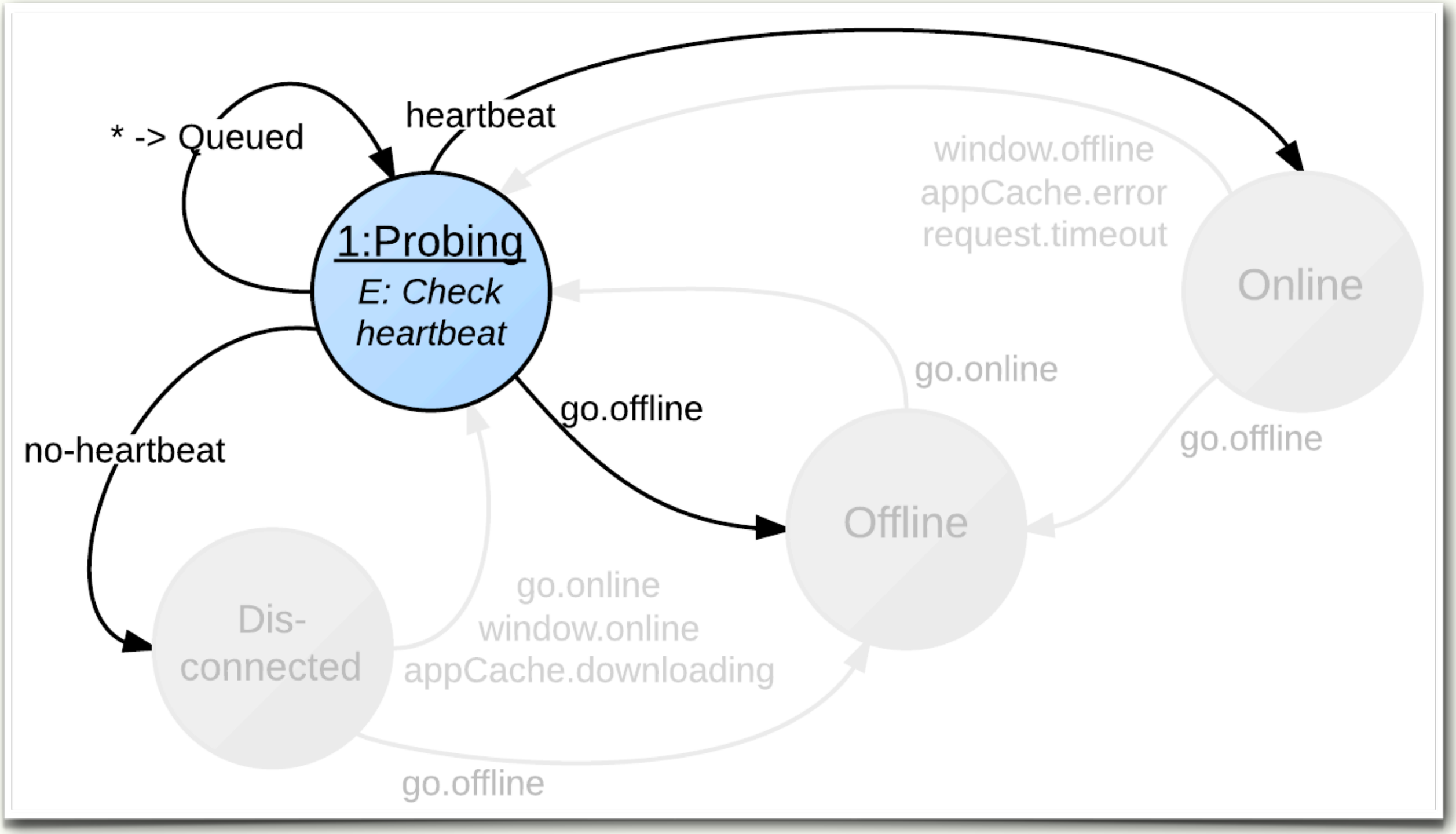
State	Input	Next State	Output
Online	window.offline	Probing	Emit Transition Event
	appCache.error	Probing	Emit Transition Event
	request.timeout	Probing	Emit Transition Event
	go.offline	Offline	Emit Transition Event
Offline	go.online	Probing	Emit Transition Event
Disconnected	go.online	Probing	Emit Transition Event
	go.offline	Offline	Emit Transition Event
	window.online	Probing	Emit Transition Event
	appCache. downloading	Probing	Emit Transition Event
Probing	heartbeat	Online	Emit Transition Event
	no-heartbeat	Disconnected	Emit Transition Event
	go.offline	Offline	Emit Transition Event


MY
EYES
DIYLOL.COM

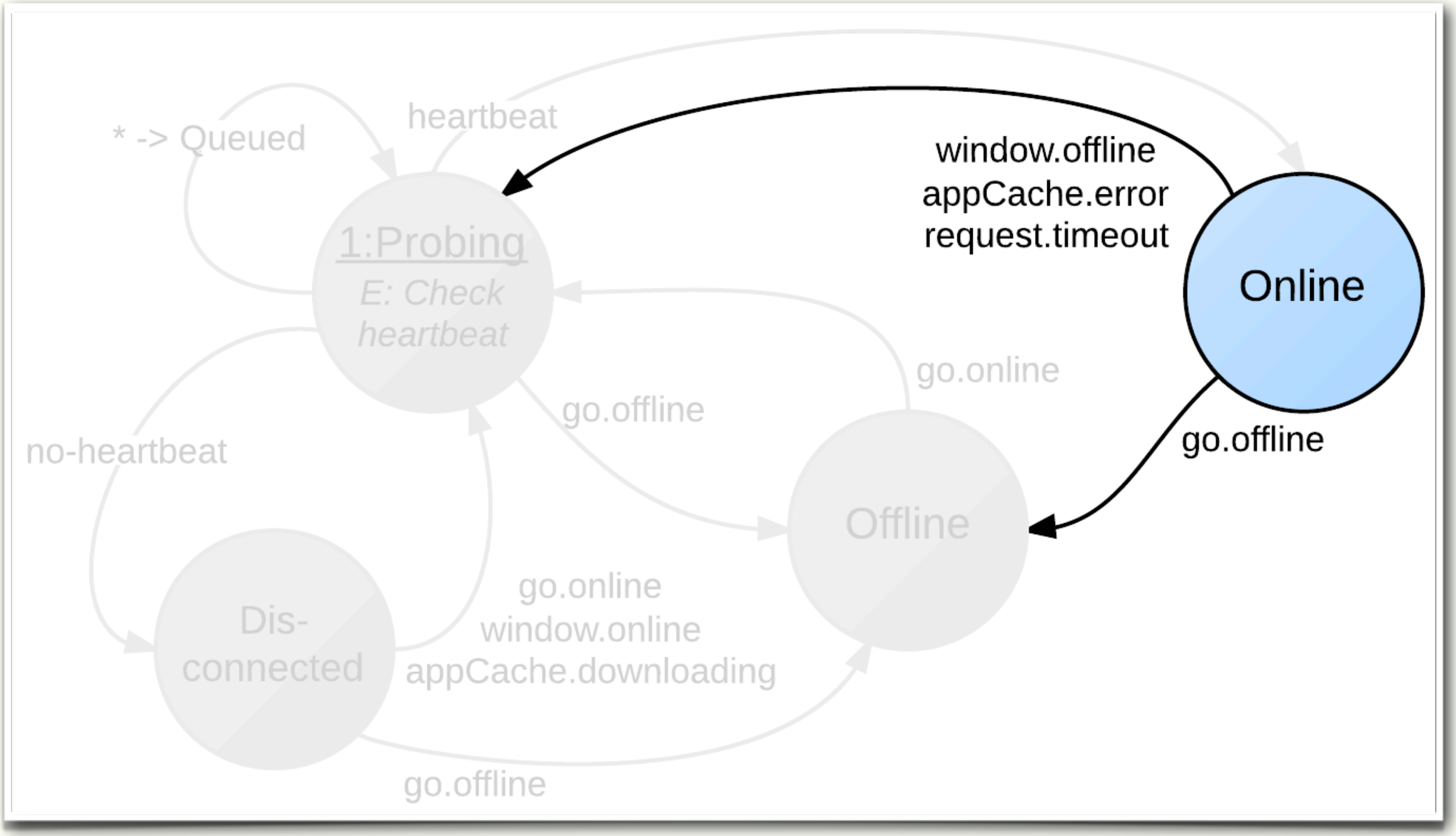
CONNECTIVITY STATE MACHINE



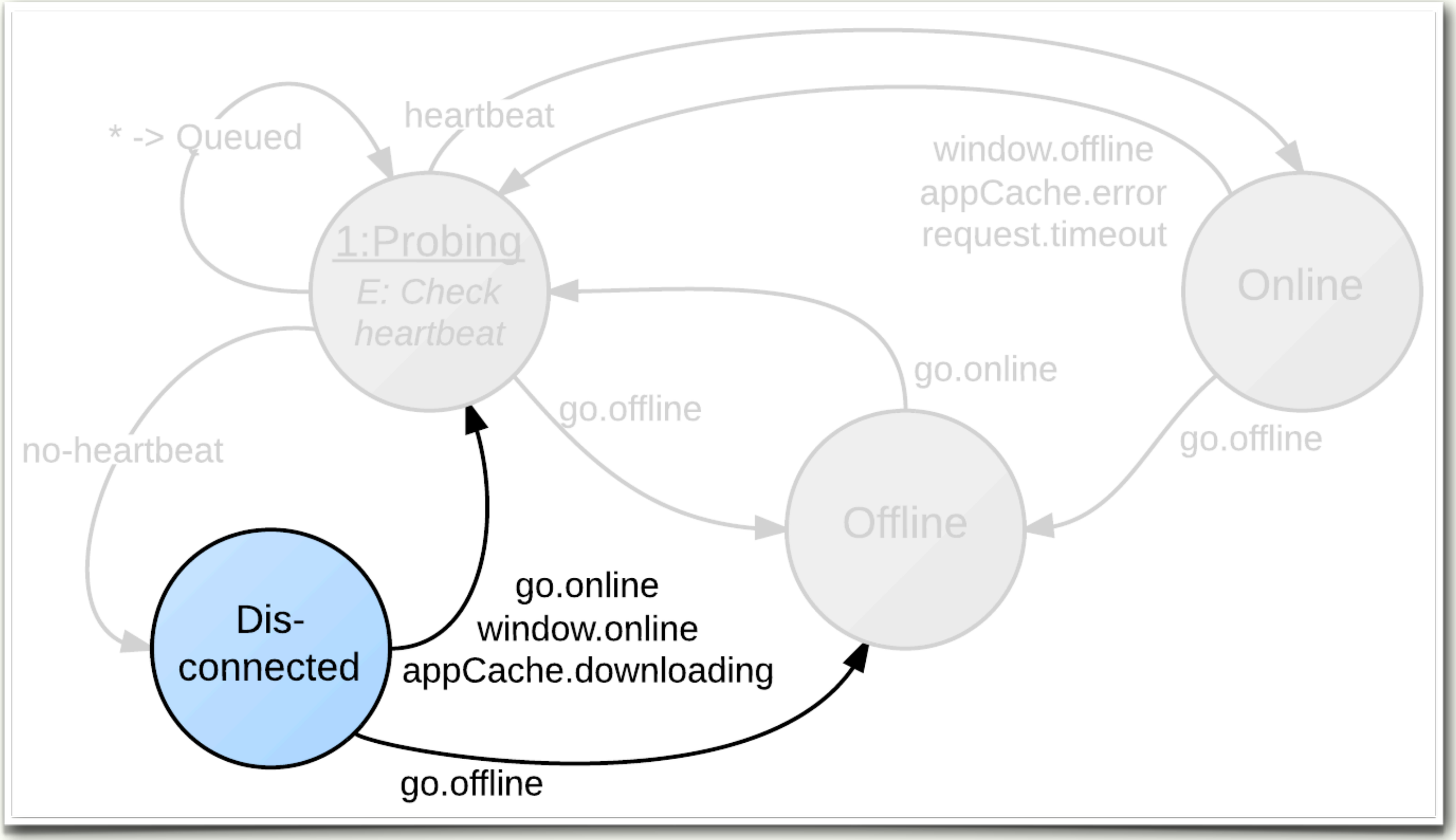
CONNECTIVITY STATE MACHINE



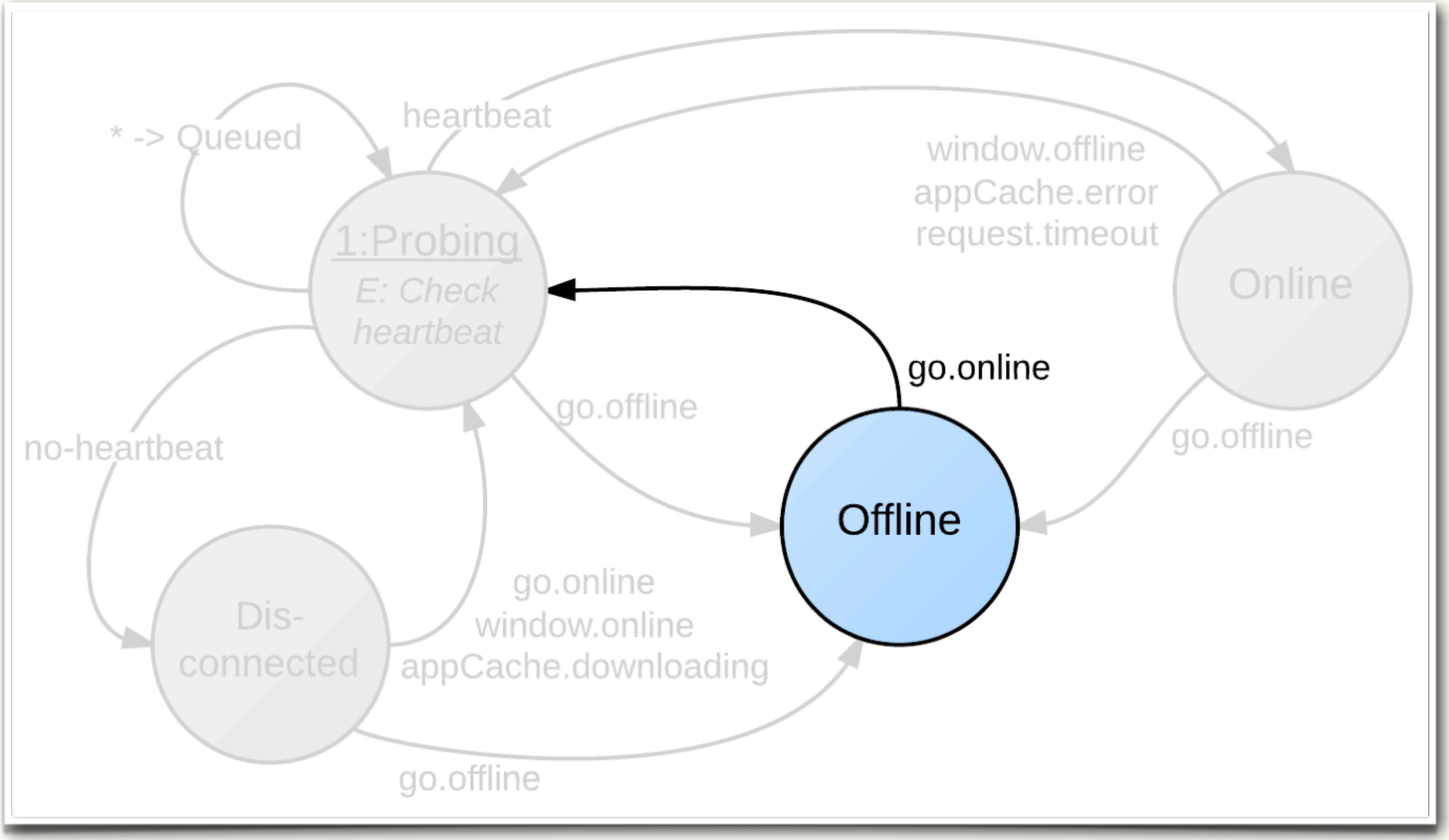
CONNECTIVITY STATE MACHINE



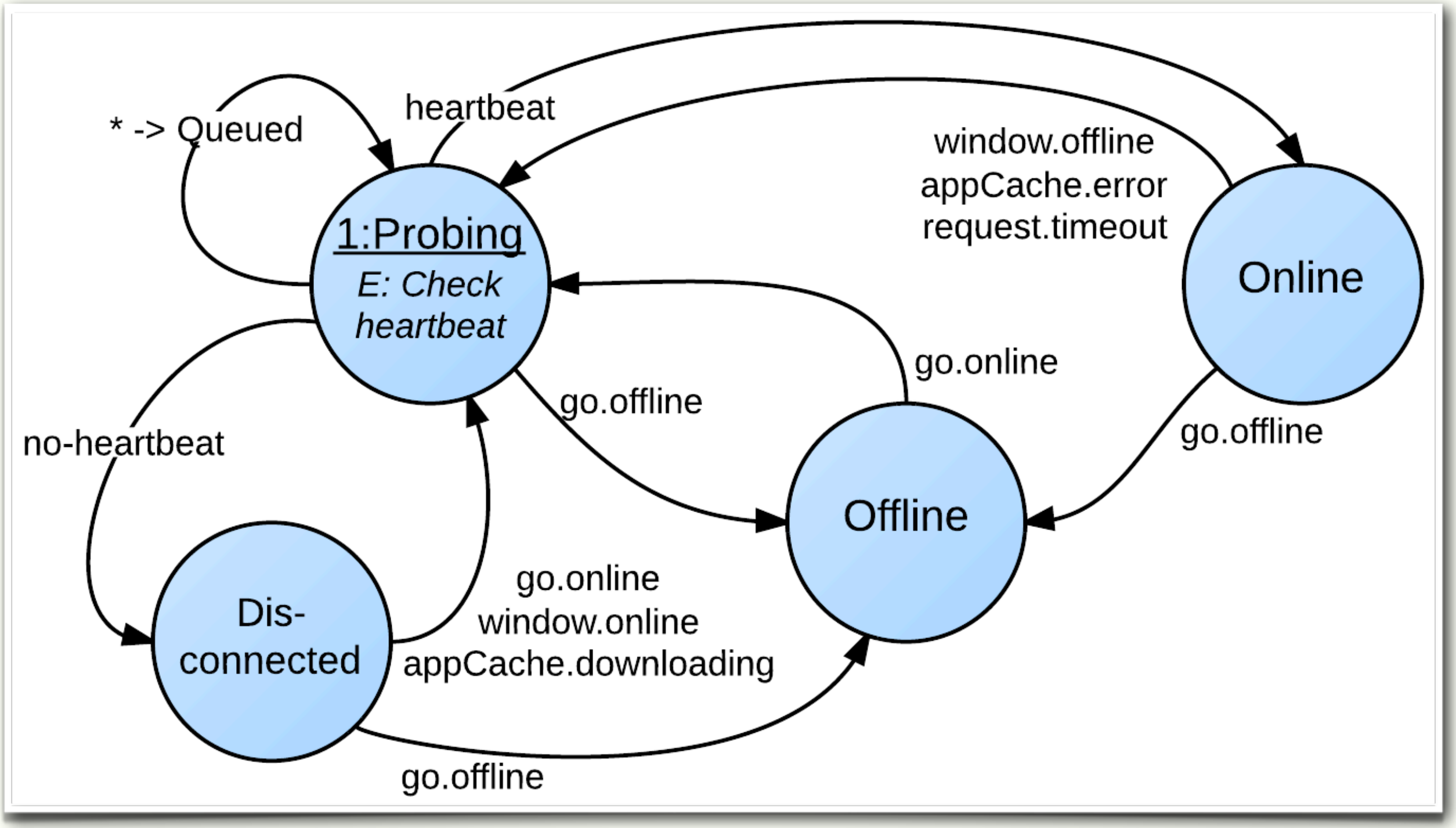
CONNECTIVITY STATE MACHINE



CONNECTIVITY STATE MACHINE



CONNECTIVITY STATE MACHINE



CODE

WAIT...WHAT HAPPENED TO THE HTTP BEHAVIOR?

SIBLING STATE MACHINES

Connectivity FSM

Communications FSM

State: Queuing

SIBLING STATE MACHINES

Connectivity FSM

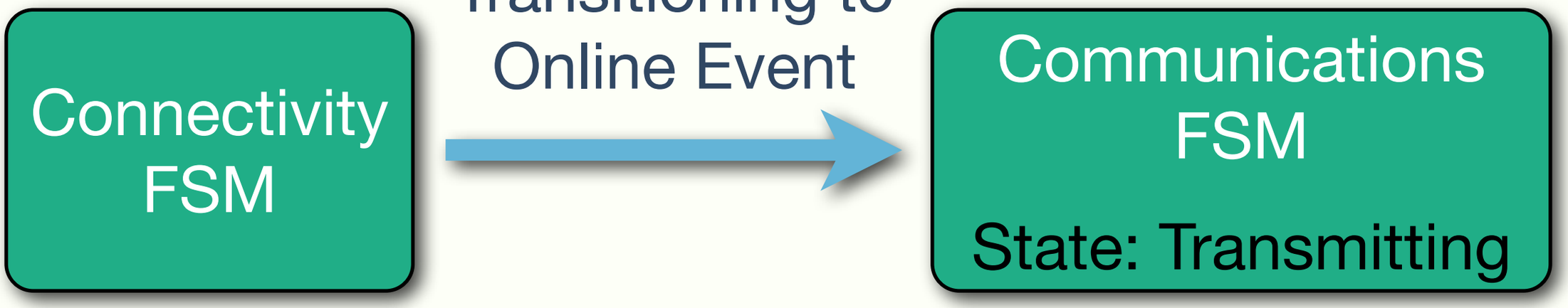
I emit events

Communications FSM

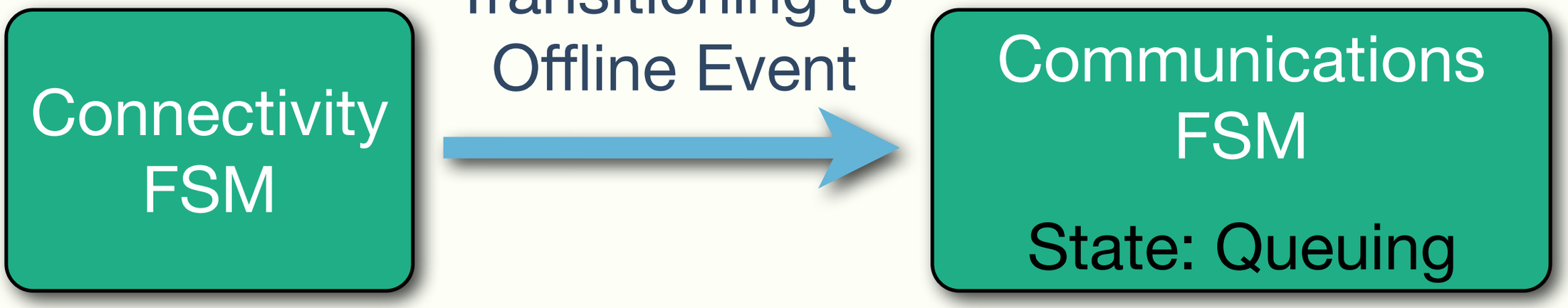
State: Queuing

Sweet. I'll listen...

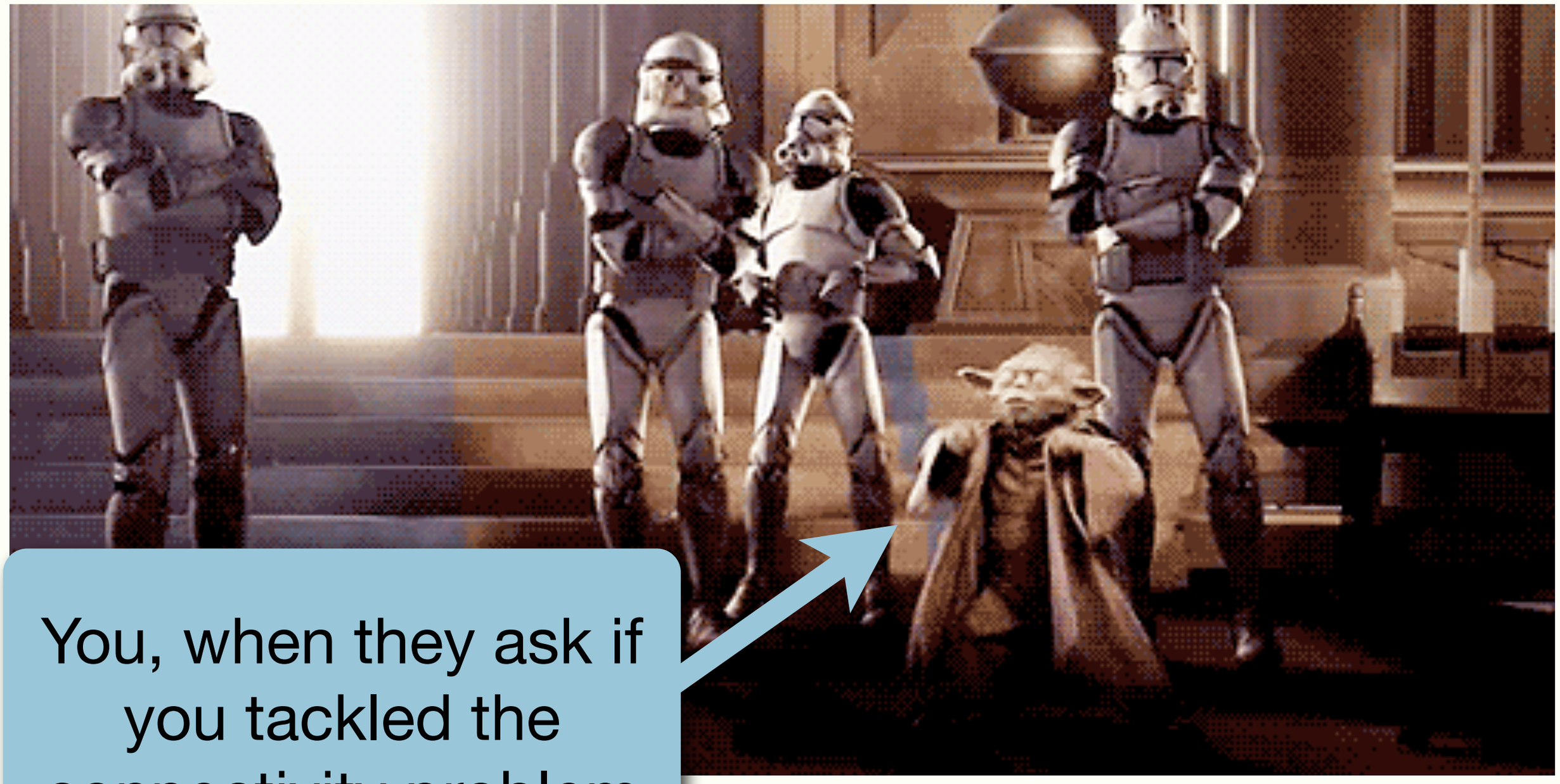
SIBLING STATE MACHINES



SIBLING STATE MACHINES



FSMS WORKING TOGETHER: POWERFUL WAY TO MANAGE & ISOLATE COMPLEXITY



You, when they ask if you tackled the connectivity problem in an extensible way...

WHAT IS THE PROBLEM?

- How do you:
  - ~~Manage online/offline state in your app?~~
  - **Handle complex UI Workflow?**
  - ~~How do you structure order-dependent initialization?~~

UI WORKFLOW

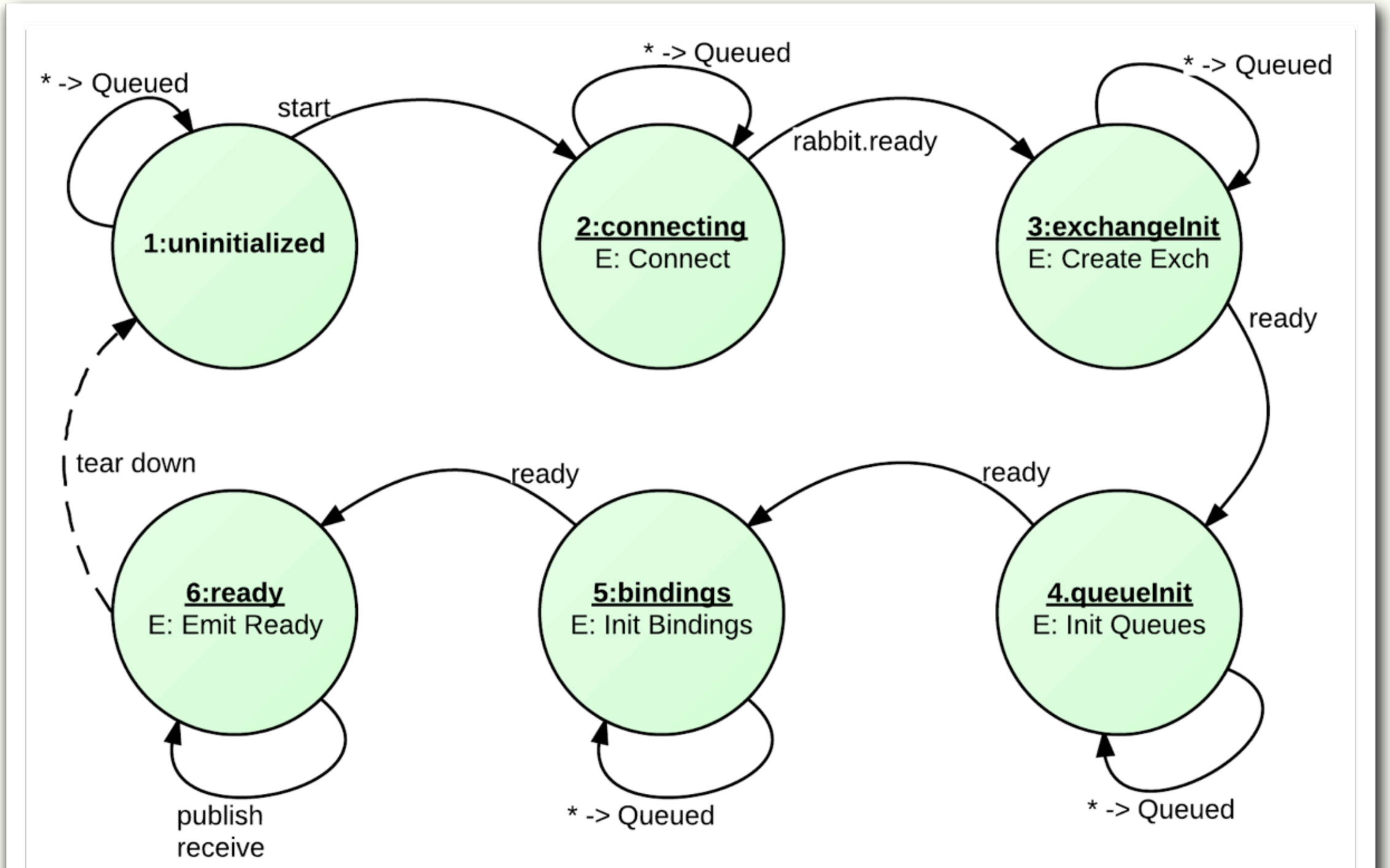
PSEUDO

CODE

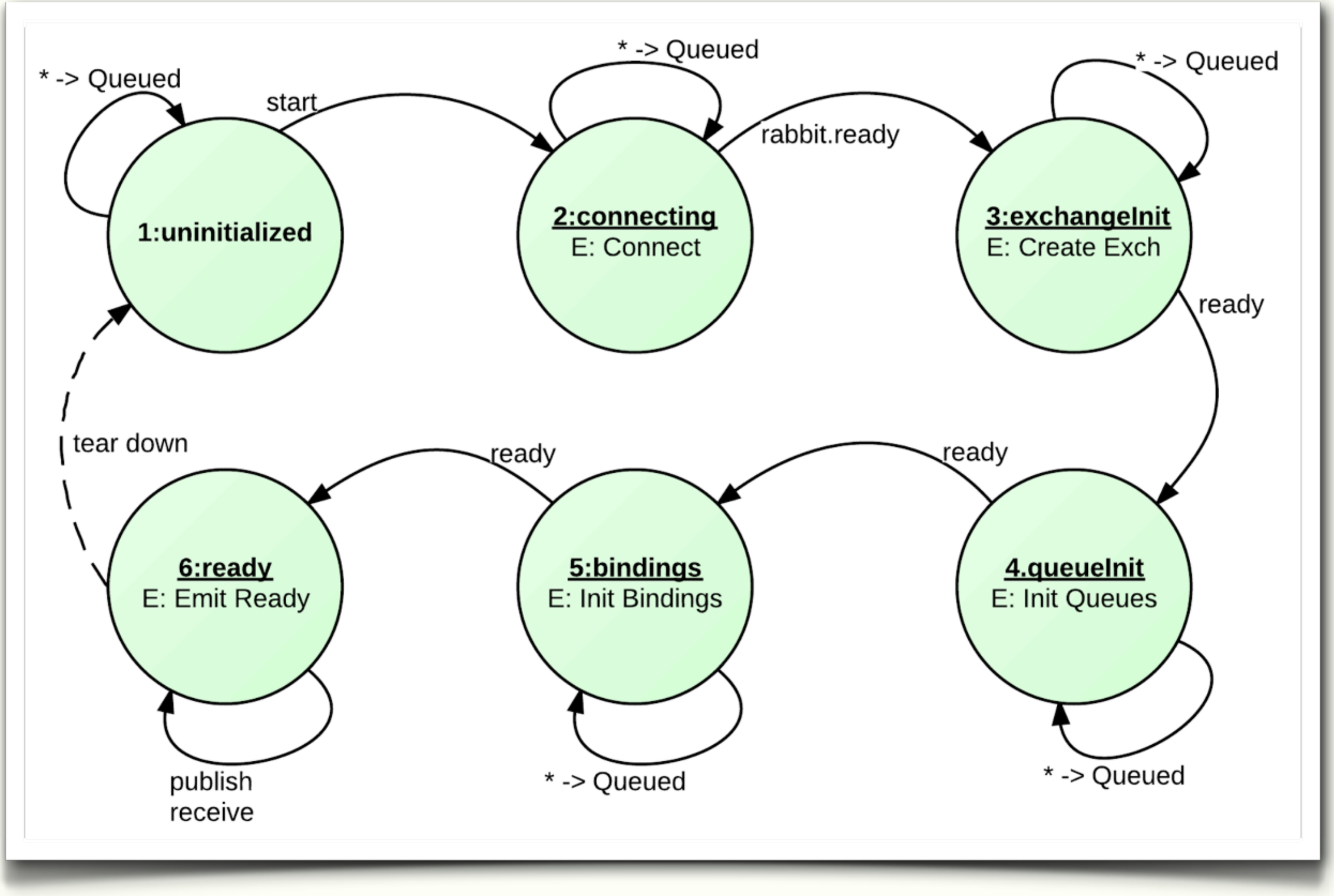
Taken from https://github.com/ifandelse/machina.js/issues/4

WHAT IS THE PROBLEM?

- How do you:
  - ~~Manage online/offline state in your app?~~
  - ~~Handle complex UI Workflow?~~
  - **How do you structure order-dependent initialization?**


* -> Queued
start
* -> Queued
rabbit.ready
* -> Queued
1:uninitialized
2:connecting
E: Connect
3:exchangeInit
E: Create Exch
ready
tear down
ready
ready
6:ready
E: Emit Ready
5:bindings
E: Init Bindings
4.queueInit
E: Init Queues
publish
receive
* -> Queued
* -> Queued

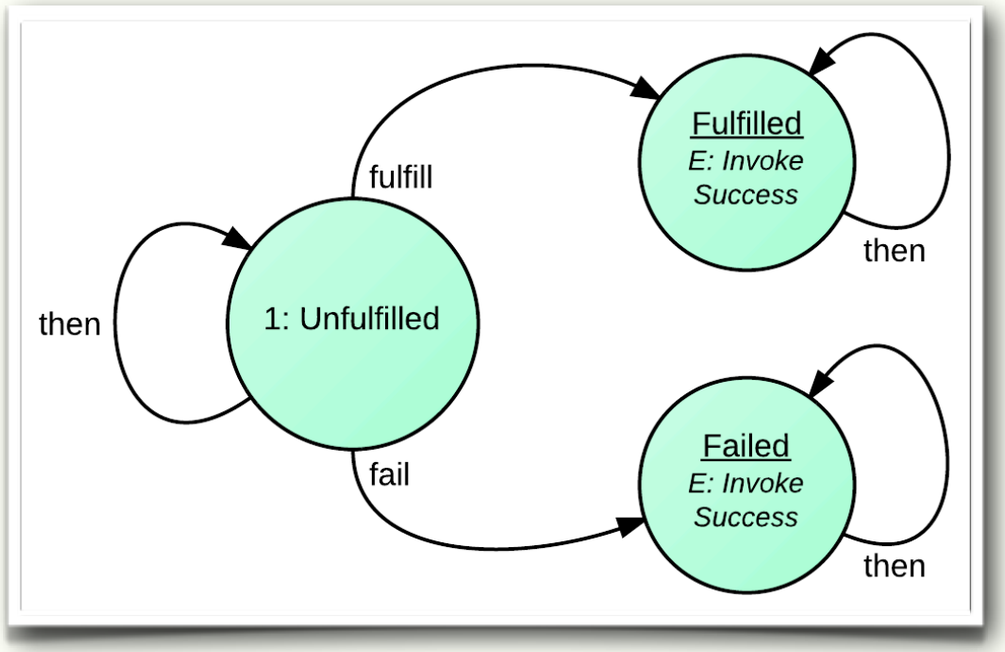
INITIALIZATION STATE MACHINE



CODE

See: https://github.com/a2labs/amqp-bootstrapper

PROMISES
& FSMS



CODE

(IF WE HAVE TIME)

HTTPS://GITHUB.COM/A2LABS/MACHINA.PROMISE

PROS & CONS

- PROS
  - EXTREMELY VERSATILE
  - LENDS WELL TO GOOD SEPARATION OF CONCERNS
  - GREAT FOR LONG-RUNNING ASYNC WORKFLOWS
  - EXPRESSIVE

PROS & CONS

- CONS
  - MODELING COMPLEX/HIERARCHICAL FSMS IS "HARD"
  - LESS FAMILIAR PATTERN (FOR MANY)

FURTHER RESOURCES

- Finite State Machine - Wikipedia
- Taking Control With machina (Doug Neiner)
- Learn You Some Erlang - Finite State Machines
- machina.js on FreshBrewedCode
- machina.js on github
- “state” - cool FSM project by Nick Fargo
- Harvey Mudd CS paper on FSMs

Q & A