Event Loops and Deferreds

June 11 2013
Dave King

Concurrency vs Parallelism

Parallelism: executing multiple things at once
Concurrency: serving many clients at once

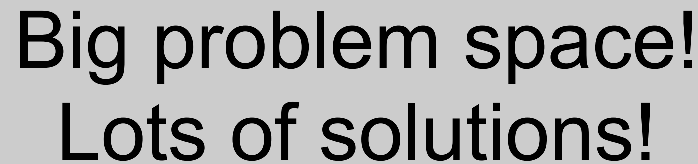
Today: *concurrency* (with an eye on program structure)

A Quick Note

Coroutines
Goroutines
Software Transactional Memory
Hewitt Actor Model
Erlang
pthread, mutex
java.nio
System Threads
Green Threads

A Quick Note

Coroutines
Goro
Software
Erlang
pthr
java.nio
System Threads
Green Threads



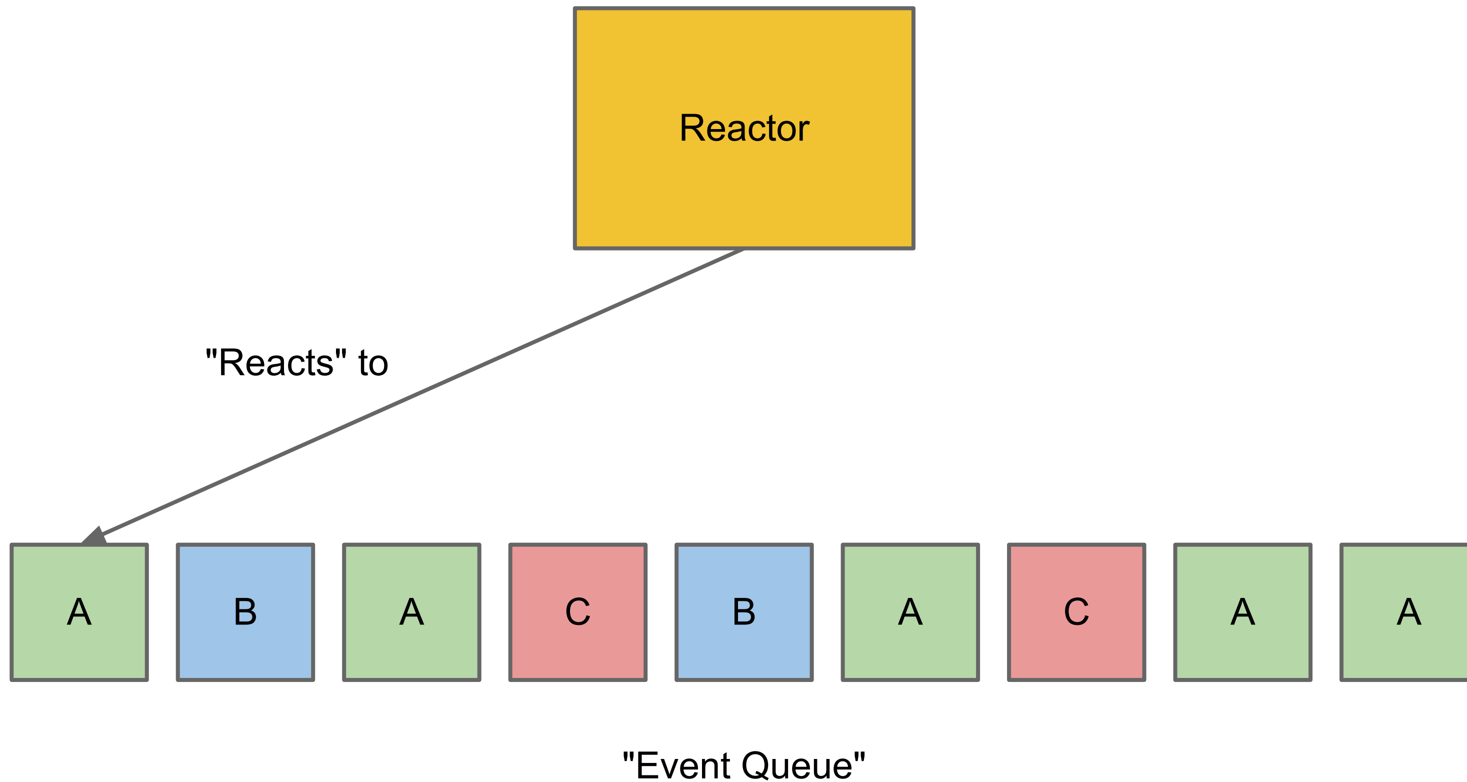
Concurrency: Event Loops

Windowing systems
JavaScript engines

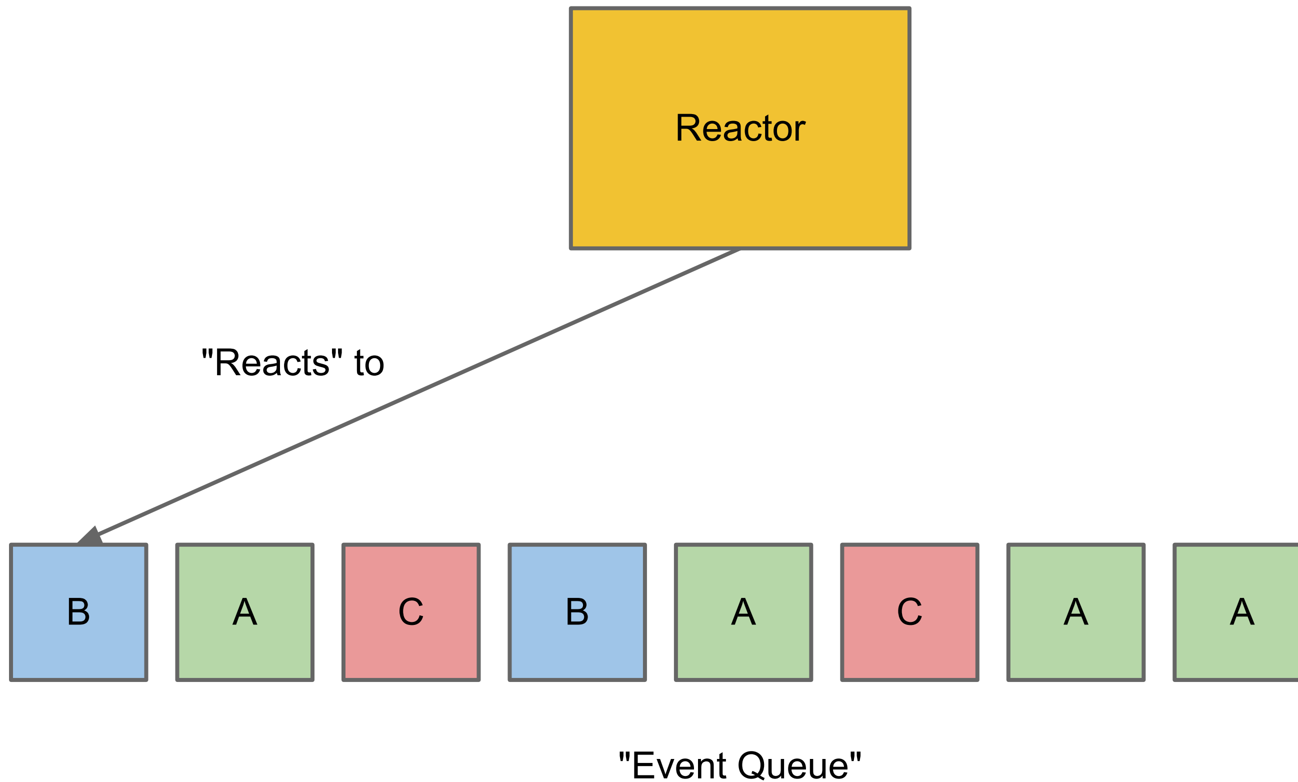
At its basic:

- Wait for something to happen
- Do the thing you wanted
- Repeat

Reactor Pattern



Reactor Pattern



A Really Basic Reactor

```
events = []
handlers = []

class Event(object):
  pass

def run_reactor():
  while true:
    if events:
      event = events.pop()
      for h in handlers:
        h()
```

Basic Reactor Example

```
def beep():
  print "beep"

handlers.push(beep)
events.push(Event())
run_reactor()
> "beep"


events.push(Event())
events.push(Event())
run_reactor()
> "beep"
> "beep"
```

Multiple Event Types

```
class Event(object):

  def __init__(self, event_type):
    self.event_type = event_type

handlers = {}
def run_reactor():
  while true:
    if events:
      event = events.pop()
      for h in handlers[event_type]:
        h()
```

Events with Types and Payloads

```
class Event(object):

  def __init__(self, event_type, payload):
    self.event_type = event_type
    self.payload = payload

handlers = {}
def run_reactor():
  while true:
    if events:
      event = events.pop()
      for h in handlers[event_type]:
        h(event.payload)
```

Event Architectures

Multiple event types that can pass payloads between

"When A happens, do B (work) and fire events C, D, and E"

Easier to break down into component parts (theoretically)

Don't Block the Event Loop

Reactor only "reacts" as fast as events can be processed

Event Loop is not for long-running computations!

Long-running computations depends on domain: usually CPU-intensive work or I/O

Don't Block the Event Loop

```
def handleEvent_bad(request):
  account_id = int(request.get('accountId'))
  # Bad!  Executing SQL will block the event loop
  account = Account.objects.get(id=account_id)
  return render_index(account)


def handleEvent_good(request):
  account_id = int(request.get('accountId'))
  def execute_db_query():
    account = Account.objects.get(id=account_id)
    return render_index(account)

  # offload long-running computation out of the
  # event loop
  runThread(execute_db_query)
```

CPS Programming (Callbacks)

```
doOneThing(a, function () {
  var b, c;

  b = "something";
  c = "something else";
  doAnotherThing(a, b, c, function () {
    // more stuff!
  });
});
```

Callbacks :sparkles:

```
// http://ianbishop.github.io/blog/2013/01/13/escape-from-callback-hell/

$.getJSON(url, {id:trackID, api_key:apiKey}, function(data) {
    var analysisURL = data.response.track.audio_summary.analysis_url;
    track = data.response.track;

    // This call is proxied through the yahoo query engine.
    // This is temporary, but works.
    $.getJSON("http://query.yahooapis.com/v1/public/yql",
      { q: "select * from json where url=\"" + analysisURL + "\"", format:
"json"},
      function(data) {
        if (data.query.results != null) {
          track.analysis = data.query.results.json;
          remixer.remixTrack(track, trackURL, callback);
        }
        else {
          console.log('error', 'No analysis data returned:  sorry!');
        }
    });
 });
```

Callbacks

Good for small programs
"Procedural" mindset
Can hide abstractions

What Problems Do Callbacks Solve?

Sequencing operations

Data flows between operations that must yield

Let's introduce an abstraction

Deferreds

A deferred is a computation that will finish later

Attach callbacks to it:

- "things to do on success"
- "things to do on failure"

When a deferred "fires", it executes its callbacks

Basic Deferreds

```
from twisted.internet.defer import Deferred

def one(value):
    return value * 2

def two(value):
    return value * 3

def done(value):
    print "All done!  Value is {0}".format(value)

d = Deferred()
d.addCallback(one)
d.addCallback(two)
d.addCallback(done)

d.callback(1)
```

Basic Deferreds

```
from twisted.internet.defer import Deferred

def one(value):
    return value * 2

def two(value):
    return value * 3

def done(value):
    print "All done!  Value is {0}".format(value)

d = Deferred()
d.addCallback(one)
d.addCallback(two)
d.addCallback(done)

d.callback(1)
```

Basic Deferreds

```
from twisted.internet.defer import Deferred

def one(value):
    return value * 2

def two(value):
    return value * 3

def done(value):
    print "All done!  Value is {0}".format(value)

d = Deferred()
d.addCallback(one)
d.addCallback(two)
d.addCallback(done)

d.callback(1)
```

Basic Deferreds

```
from twisted.internet.defer import Deferred

def one(value):
    return value * 2

def two(value):
    return value * 3

def done(value):
    print "All done!  Value is {0}".format(value)

d = Deferred()
d.addCallback(one)
d.addCallback(two)
d.addCallback(done)

d.callback(1)
```

Basic Deferreds

```
from twisted.internet.defer import Deferred

def one(value):
    return value * 2

def two(value):
    return value * 3

def done(value):
    print "All done!  Value is {0}".format(value)

d = Deferred()
d.addCallback(one)
d.addCallback(two)
d.addCallback(done)

d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

More Advanced Deferreds

```
from twisted.internet.defer import Deferred

def finished(value):
    print "All done!  Value is {0}".format(value)

def factorial(acc, value):
    if value == 1:
        return acc

    d = Deferred()
    d.addCallback(factorial, value - 1)
    d.callback(acc * value)
    return d

d = Deferred()
d.addCallback(factorial, 5)
d.addCallback(finished)
d.callback(1)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Callbacks can become Deferred

```
from twisted.internet.defer import Deferred

def exec_with_deferred(fn):
  d = Deferred()

  def on_finished(value):
    d.callback(value)

  fn(on_finished)
  return d

def func(on_finished):
  # May be a long-running task
  on_finished(5)

def done(value):
  print "All done!  Value is {0}".format(value)

d = exec_with_deferred(func)
d.addCallback(done)
```

Briefly Back to Event Loops

```
def get_session_info_from_cookie(self, request):
  return Session.objects.get(request.getCookie('sessionId'))

def do_upstream_request(self, session, request):
  upstream_url = "http://url_base/v1/{0}".format(sesion.accountId)
  # library call into Twisted -- returns a deferred
  d = make_upstream_request(upstream_url, 'GET')
  return d

def upstream_request_success(self, upstream_request, request):
  # indicate success in our logs
  # add to stats counter
  request.setStatusCode(upstream_request.statusCode)
  request.finish()

def upstream_request_failure(self, request):
  Log.error("Upstream request failure! :(", get_info(request))
  # add to stats counter
  request.setStatusCode(500)
  request.finish()

def respond(self, request):
  d = threads.deferToThread(self.get_session_info_from_cookie)
  d.addCallback(self.do_upstream_request, request)
  d.addCallback(self.upstream_request_success, request)
  d.addErrback(self.upstream_request_success, request)
  return d
```

Final Words

All programs are a composition of data flows

Deferreds make this data flow explicit

In an event loop, deferreds can be used to pass data between long-running operations