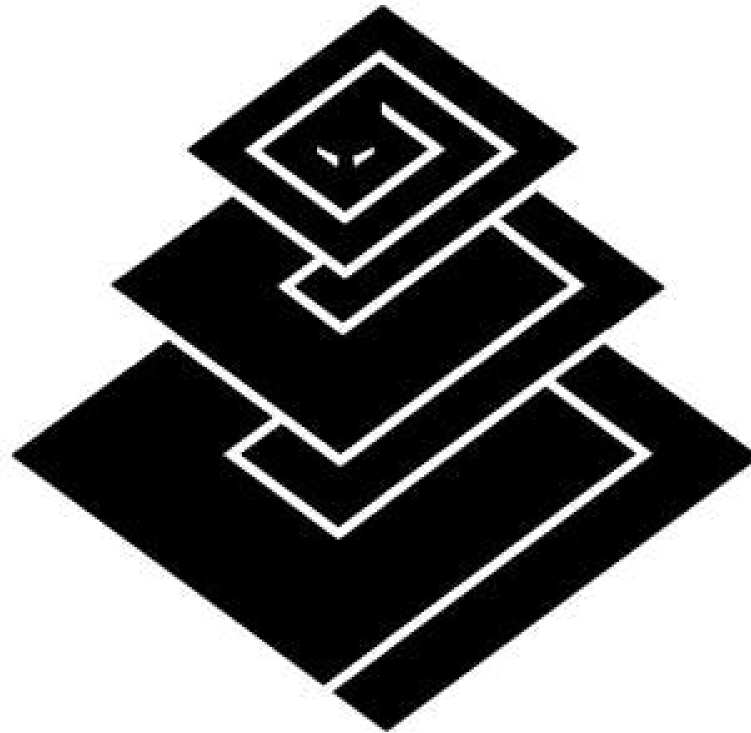
Twisted Matrix

Just the Basics (take 2)...



Waiting Code in Parallel

- Synchronous set of code with arbitrary delays
  - Most code paths spending most of their time waiting
- Multiple such operations needed
  - Other ops must execute while a given op is waiting
- Need to preserve (a subset of) state in order to service a sequence of operations
- Any operation may fail
  - Must handle errors

Threads (Brute Force)

- Each parallel process gets a thread
- Arguably the most “natural” approach
  - Even exception handling works as normal
  - Works with blocking calls and naïve C libraries
  - Consumes lots of resources
- Each blocking operation must support timeouts if required
- State is stored in the function's execution frame

“Green Threads”

- Overcome resource limitations of true threads
- Micro-threading and co-routines are the most common examples
- User code pretty much as with threads
- Requires low-level hackery (micro-threads) or very well behaved operations (co-routines)
- State stored in green-thread's execution frame

Asynchronous I/O Loops

- Non-blocking “polling” or “selecting” loop
  - Becomes the mainloop of the application
  - Tends to conflict when mixing multiple libraries
  - Resource friendly (single threaded, normally)
  - Tend to be limited in focus
- Easy to code for old hands (classical approach), but hard to code **well**.
- Agnostic with respect to storage of state between calls (most strategies work fine)

Seperation of Parallelisation and State

- Threading (and similar) systems use **implicit** storage of state
  - Data is stored in thread or green-thread context
  - Coder doesn't need to think about it
- With asynchronous operation, where you don't have the low-level suspend/resume support you need to **explicitly** store state
  - There's all sorts of ways to do this (coming up)

State Tables

- Graph-based modelling of program state
  - Extremely efficient
  - Can be implemented using objects with minimal support machinery
- Doesn't look much like functional code
- More formal than is needed for most situations

Continuations a.k.a. Call/CC

"Go do this op, suspending the current function;
when op is done, continue the function
(call the continuation I'm passing you)"

- Low-level; effectively must be a language feature
  - Allows raising errors/returning values naturally
- State stored in the continuation
  - Deferred ops must be continuation-aware
  - I/O loop must call registered continuations
- Too abstract for most users

Storing State in a Continuation

```
def x():
    value = a()
    try:
        cc = continuation.here()
        b(cc=cc)
        value2 = cc.suspend()
    except NetError:
        raise Something(value)
    else:
        value3 = c( value2 )
        try:
            cc = continuation.here()
            d(value3,cc=cc)
            value4 = cc.suspend()
        except NetError:
            raise SomeOther( value3 )
    return value4 * value2
```

Callback Passing

"Go do this, call these functions when finished"

- Constrained version of continuation passing
  - Common pattern is to have only success or failure
- Every operation needs to be callback aware
  - Requires rigorous standardisation
- And it's still pretty tedious to work with
  - Every function needs to guard against invalid callbacks, failures during callbacks, etceteras
  - So tedious the sample code doesn't do it ;)

Callback State with Nested Scopes

```
from __future__ import nested_scopes
def x( successCallback=None, failureCallback=None):
    def afterb( result ):
        value3=c( result )
        def afterd( result ):
            successCallback( result * value2 )
        def ondfail( err ):
            failureCallback( SomeOther( value3 ) )
        d( value3, successCallback=afterd,
           failureCallback=ondfail )
    def onbfail( err ):
        failureCallback( Something( value ) )
        value = a()
    b( successCallback=afterb,
failureCallback=ondfail )
```

Callback State with Objects

```
class x( object ):
    def __init__( self, onSuccess=None,
onFailure=None ):
        self.value = a()
        self.onSuccess = onSuccess
        self.onFailure = onFailure
        b(onSuccess=self.afterb,onFailure=self.ondfail)
    def afterb( self, result ):
        self.value2 = result
        self.value3=c( self.value2 )
        d
(self.value3,onSuccess=self.afterd,onFailure=self.ondfai
l)
    def onbfail( self, err ):
        self.failureCallback( Something( self.value ) )
    def afterd( self, result ):
        self.successCallback( result * self.value2 )
    def ondfail( self, err ):
        self.failureCallback( SomeOther( self.value3 ) )
```

In Summary

- Need a way to parallelise the code paths
  - Minimal resources, no systemic extensions
- Need a way to store state to continue processing
  - Where to continue, and in what context

Wasn't this talk supposed to be on Twisted?

We're getting there...

Twisted Matrix: Key Concepts

- Deferred objects
  - Rationalisation of callback-passing model
- Reactors
  - Rationalisation of I/O loop model
- Protocols, transports, factories, services
  - Commonly required base mechanisms
- Interfaces, thread pools, co-routines, etceteras

Deferred Objects

"I promise to return or fail"

- Promise of a return or error value
  - Returned from deferrable functions
  - Caller registers callbacks/errbacks
  - Deferred function only needs to know it's deferred
- Rationalisation of callback passing
  - Allows for chains of callbacks
  - Simplifies writing deferred code
- Accomodates most state-storage schemes

Using Defers

```
d = proxy.get( oids, *self._arguments,
**self._namedArguments )
d.addCallback( self.integrateSingleResult,
proxy=proxy )
d.addErrback( self.handleSingleError, oids=oids,
proxy=proxy )
self.partialDefers.append( d )
```

- Client code registers callbacks
- Can pass state to callbacks with arguments
- Reduces complexity of the deferred code and makes whole system more flexible

Multiple (Interlocking) Callbacks

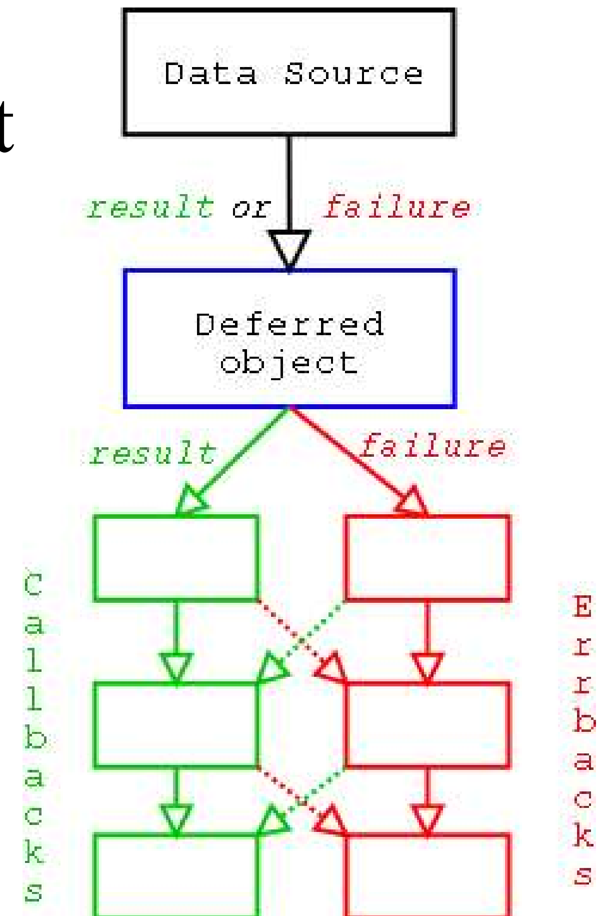
- Allow for setting up multiple actions on return
  - Allows composition of effects from small functions
  - Client uses same mechanism as provider

```
df = defer.Deferred()
df.addCallback( self.areWeDone, roots=roots,
request=request )
df.addCallback( self.proxy.getResponseResults )
df.addCallback( self.integrateNewRecord, rootOIDs =
roots[:] )

sendMessage(message, df)

return df
```

Deferred Error Handling

- Deferred callback chains are parallel
- Errbacks passed a "failure" object with a "value" which is the exception raised

```
def fail( result ):
    print 'failing'
    raise ValueError( result )
def recover( error ):
    print 'recovering'
    return error.value.args[0]
finalDF.addCallback( fail)
finalDF.addErrback( recover)
```



Sample in Deferreds

```
def x( ):
    finalDF = defer.Deferred()
    def afterb( result ):
        value2=result
        value3=c( result )
        def afterd( result ):
            finalDF.callback( result * value2 )
        def ondfail( err ):
            finalDF.errback(SomeOther( value3 ))
        df = d( value3 )
        df.addCallbacks( afterd, ondfail )
    def onbfail( err ):
        finalDF.errback( Something( value ) )
    value = a()
    df = b()
    df.addCallbacks( afterb, onbfail )
    return finalDF
```

Deferreds in Summary

- Simplify callback-based parallelisation
- Allow for any number of state-passing approaches
- Have bridges for e.g. threaded and micro-threaded operations
- Reduce complexity in writing deferred operations

Reactors

"You don't call Twisted, Twisted calls you."

- Rationalisation of Asynchronous I/O Loop
  - Manager of entire operating environments
  - "Event loop" for the application
  - Includes basic scheduling API
  - Manages ports (and other pollables)
- Requires integration work for each system that has it's own mainloop
- There can only be one (but it has plugins)

Reactor API Highlights

- IReactorCore
  - callWhenRunning(self, callable, *args, **kw)
  - run(self)
  - iterate(self, delay)
  - stop(self)
- IReactorTime
  - callLater(self, delay, callable, *args, **kw)

Reactor APIs (cont)

- IReactorTCP/IReactorUDP/IReactorSSL
  - listenUDP(self, port, protocol, interface, maxPacketSize)
  - connectTCP(self, host, port, factory, timeout, bindAddress)
  - listenTCP(self, port, factory, backlog, interface)
  - connectSSL(self, host, port, factory, contextFactory, timeout, bindAddress)
  - listenSSL(self, port, factory, contextFactory, backlog, interface)

Protocols

- Created by factories registered in listen*/connect* for each connection
- Handle network events by defining methods
  - Does **not** poll the ports directly!
  - LineReceiver mix-in gives line-oriented events
- Can use factories or services to share information with other protocol instances

Writing Protocols

(Shamelessly stolen from the Twisted tutorial)

```
from twisted.internet import protocol, reactor
from twisted.protocols import basic

class FingerProtocol(basic.LineReceiver):
    def lineReceived(self, user):
        self.transport.write(self.factory.getUser
(user)+"\r\n")
        self.transport.loseConnection()
class FingerFactory(protocol.ServerFactory):
    protocol = FingerProtocol
    def __init__(self, **kwargs):
        self.users = kwargs
    def getUser(self, user):
        return self.users.get(user, "No such user")

reactor.listenTCP(1079, FingerFactory(moshez='Happy
and well'))
reactor.run()
```

Other Interesting Features

- Pre-built mechanisms (protocols, content-types)
- Interface mechanism (moving to Zope's)
- Development model and effect of architecture on popularity