

# Twisted

“An event-driven networking engine  
written in Python”

Ying Li ([ying.li@rackspace.com](mailto:ying.li@rackspace.com))

Saturday, January 26, 13

I'm here to give everyone a brief introduction to Twisted, and the first question to ask is: what is Twisted? Right on the front page of [www.twistedmatrix.com](http://www.twistedmatrix.com), it says that Twisted is “an event-driven network engine written in Python”.

# Twisted

“An event-driven networking engine  
written in Python”

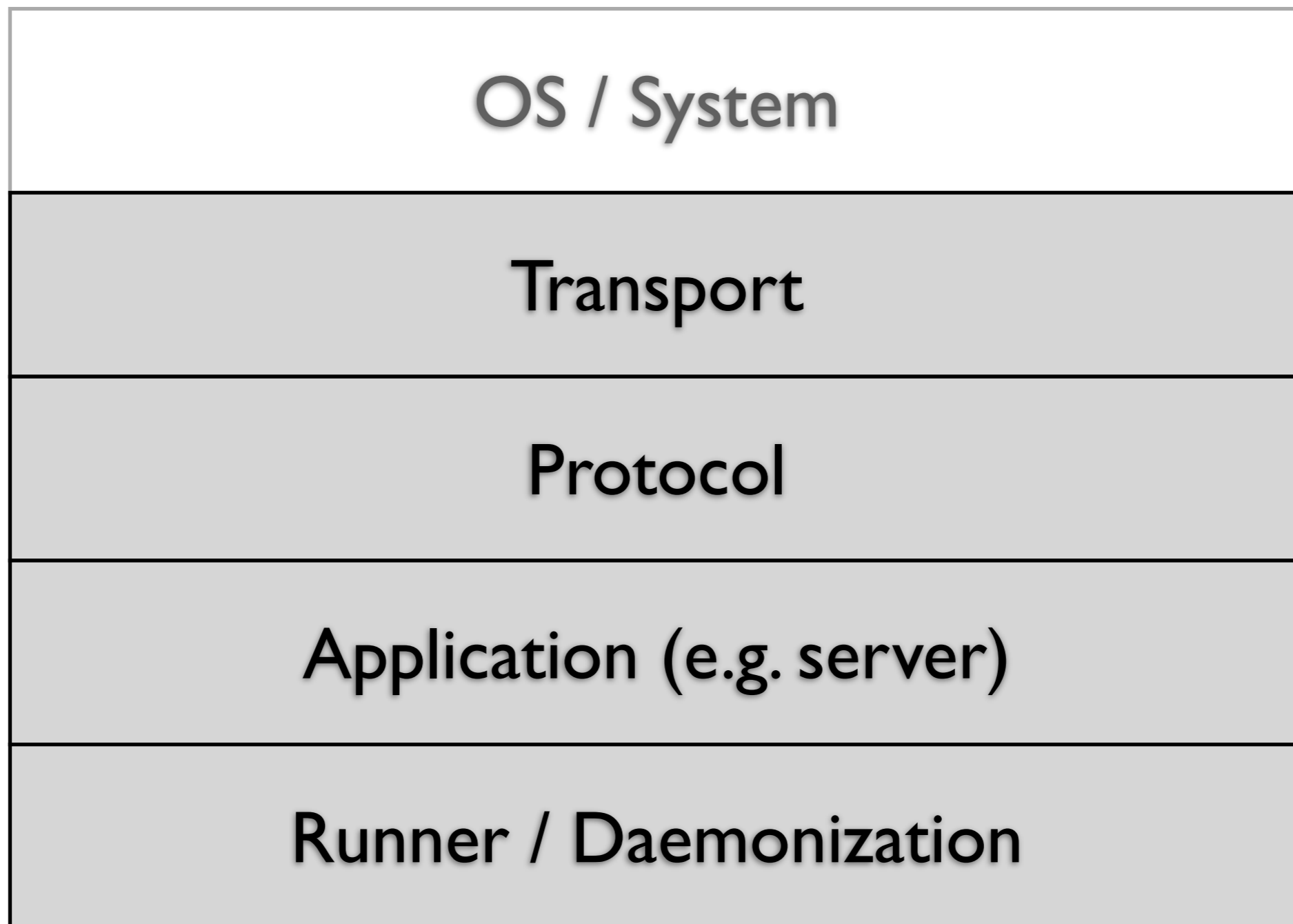
Saturday, January 26, 13

I'll get to event-driven later, but why does Twisted call itself a networking engine? What exactly does it provide?



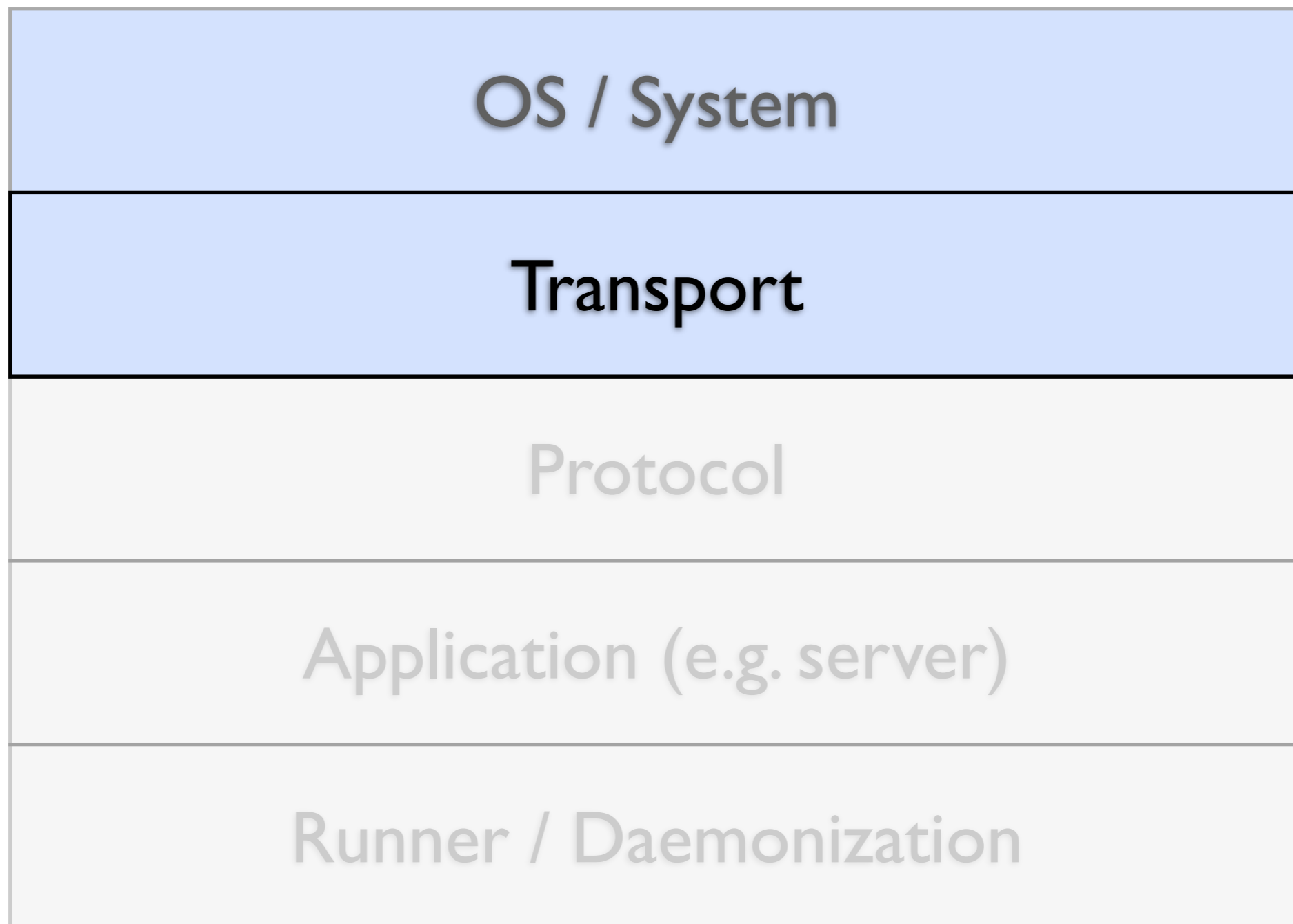
Saturday, January 26, 13

This is best explained at just going through the general anatomy of a Twisted network application.



Saturday, January 26, 13

I will describe four parts: transports, protocols, the application, and the running or daemonization of your application.



Saturday, January 26, 13

A transport is anything that can deliver bytes to and from a protocol (we'll get to that later). At the lowest level, a transport represents the connection between two endpoints communicating over a network. It describes connection details (like being stream- or datagram-oriented, flow control, etc.) and deals with the underlying system calls.

# Transport

- manages socket or pipe

Saturday, January 26, 13

It handles opening a socket or pipe in non-blocking mode and handles closing the connection to said socket or pipe. It also handles interpreting errors specific to a socket or pipe.

# Transport

- manages socket or pipe
- writes data

# Transport

- manages socket or pipe
- writes data
- reads data
- notifies protocol

Saturday, January 26, 13

And whenever there is data available, it reads the data from the socket and then passes on the said data to a protocol (via a callback – more later). Note that it does not handle the data itself. Likewise, it notifies the protocol via callbacks when connection events (connection being made, connection being lost) happen, but does nothing itself except for low level cleanup.



# Transport

- TCP
- UDP
- AF\_UNIX
- Pipes

Saturday, January 26, 13

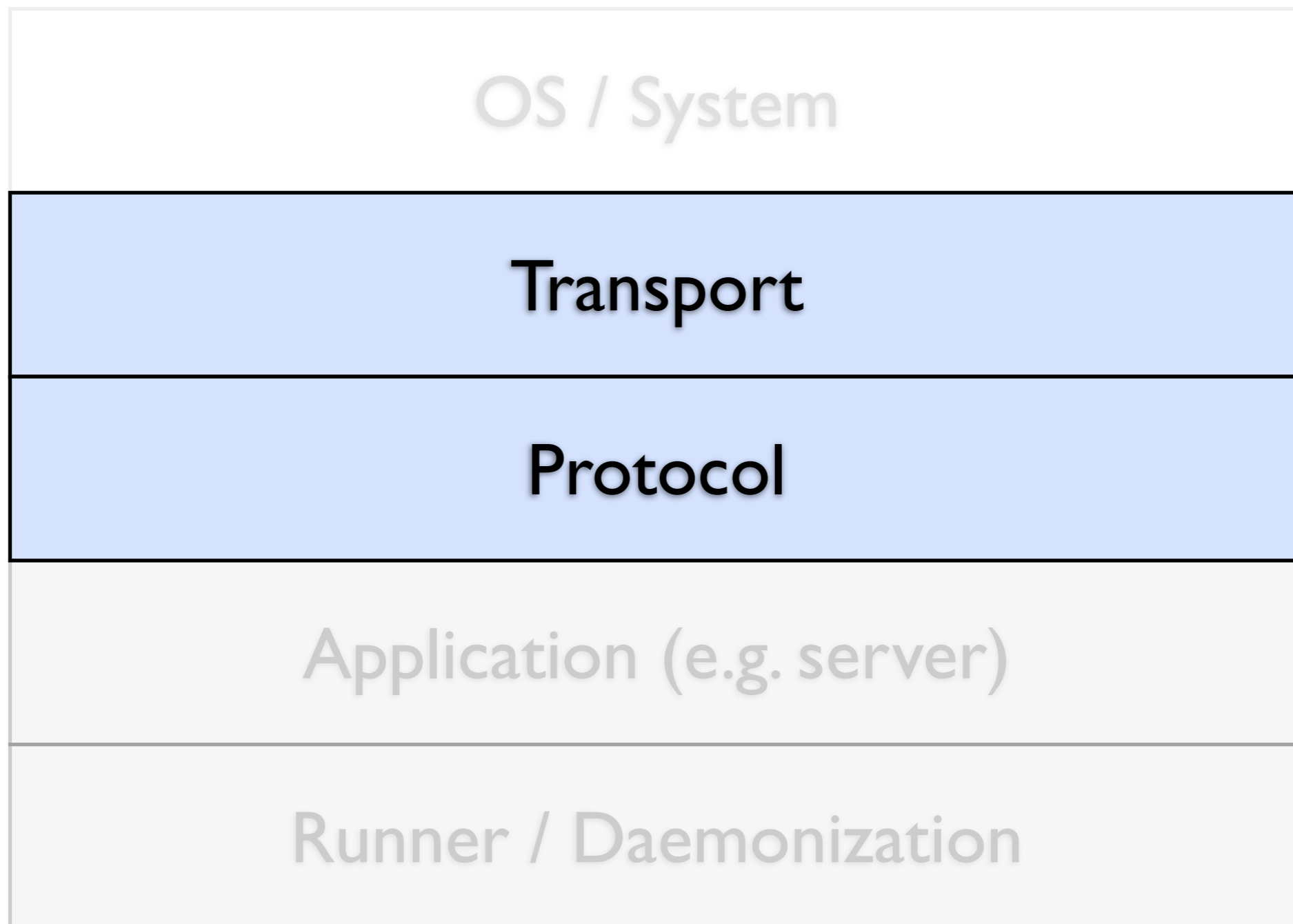
Twisted provides several implementations low level transports, including TCP-based transports, UDP-based transports, transports Unix sockets, and transports for inter-process communication (reading from and writing to pipes).

# Transport

- TCP
- UDP
- AF\_UNIX
- Pipes
- TLS
- memory

Saturday, January 26, 13

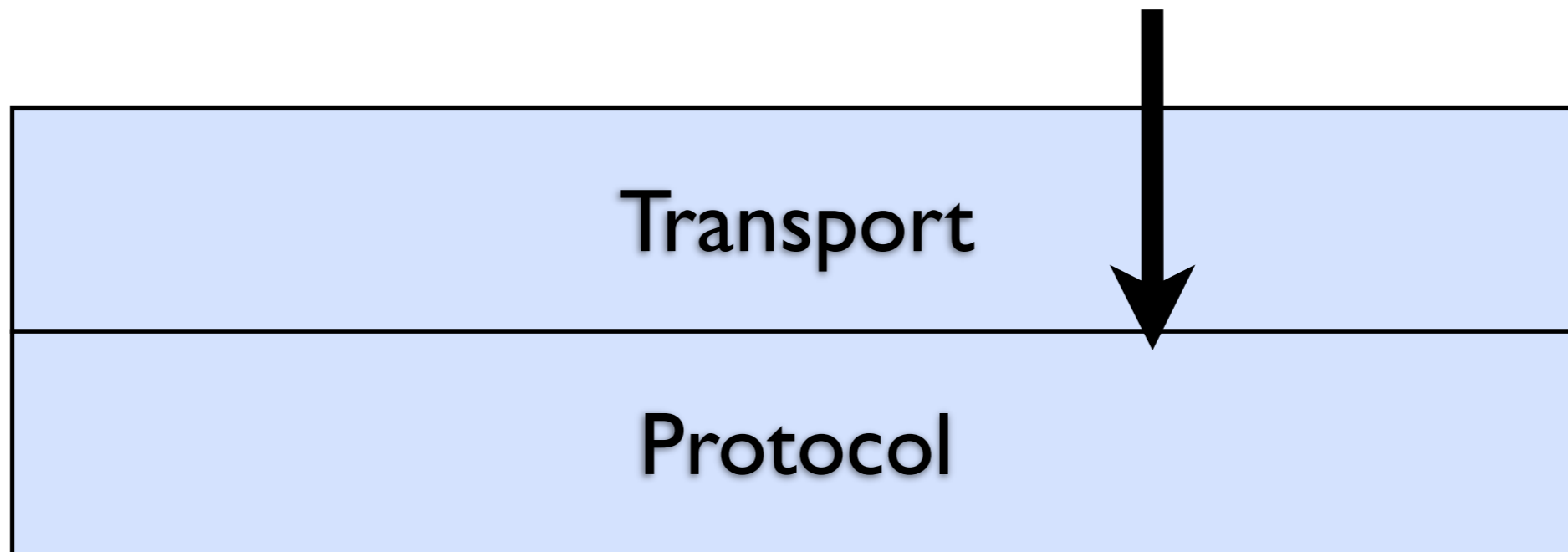
It also provides TLS as a transport (more on that later) and an in-memory transport for testing.



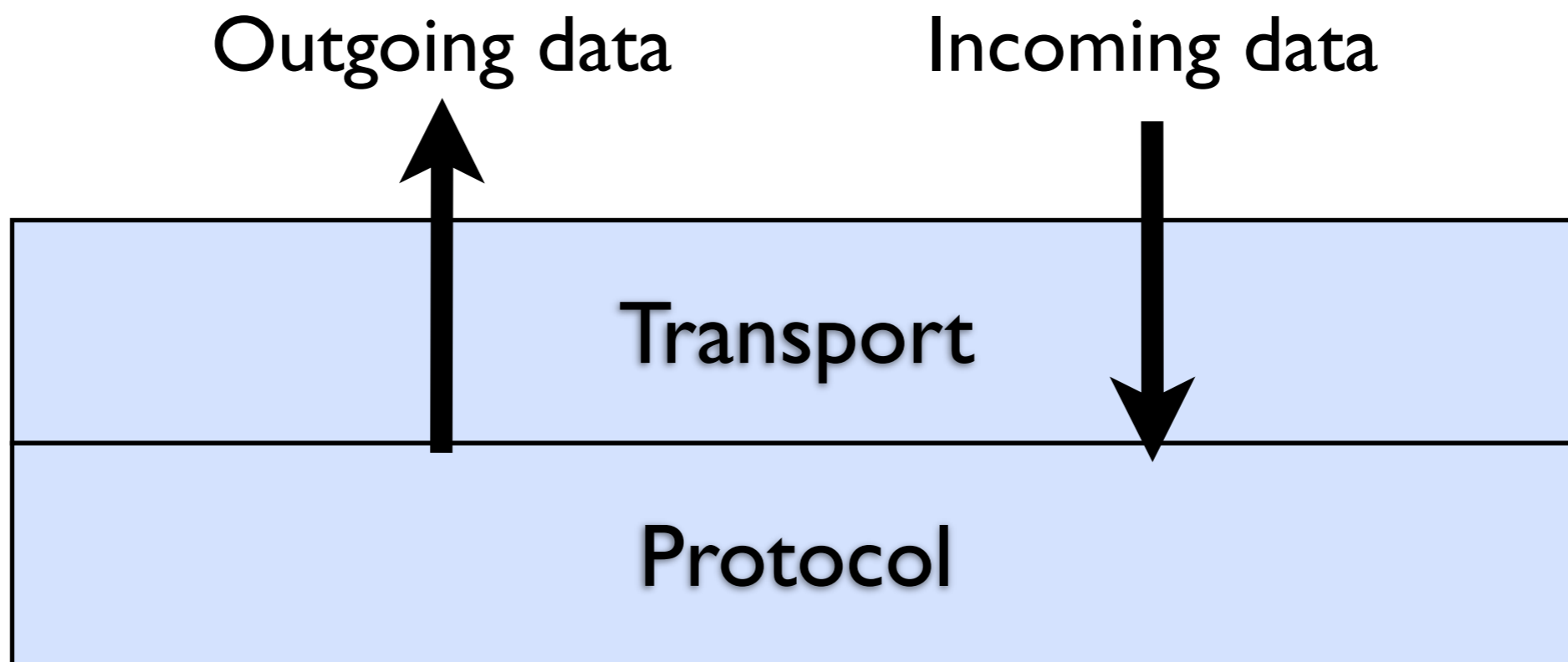
Saturday, January 26, 13

Whereas transports handle shuffling bytes around, protocols describe how to process those bytes (and other network events).

Incoming data



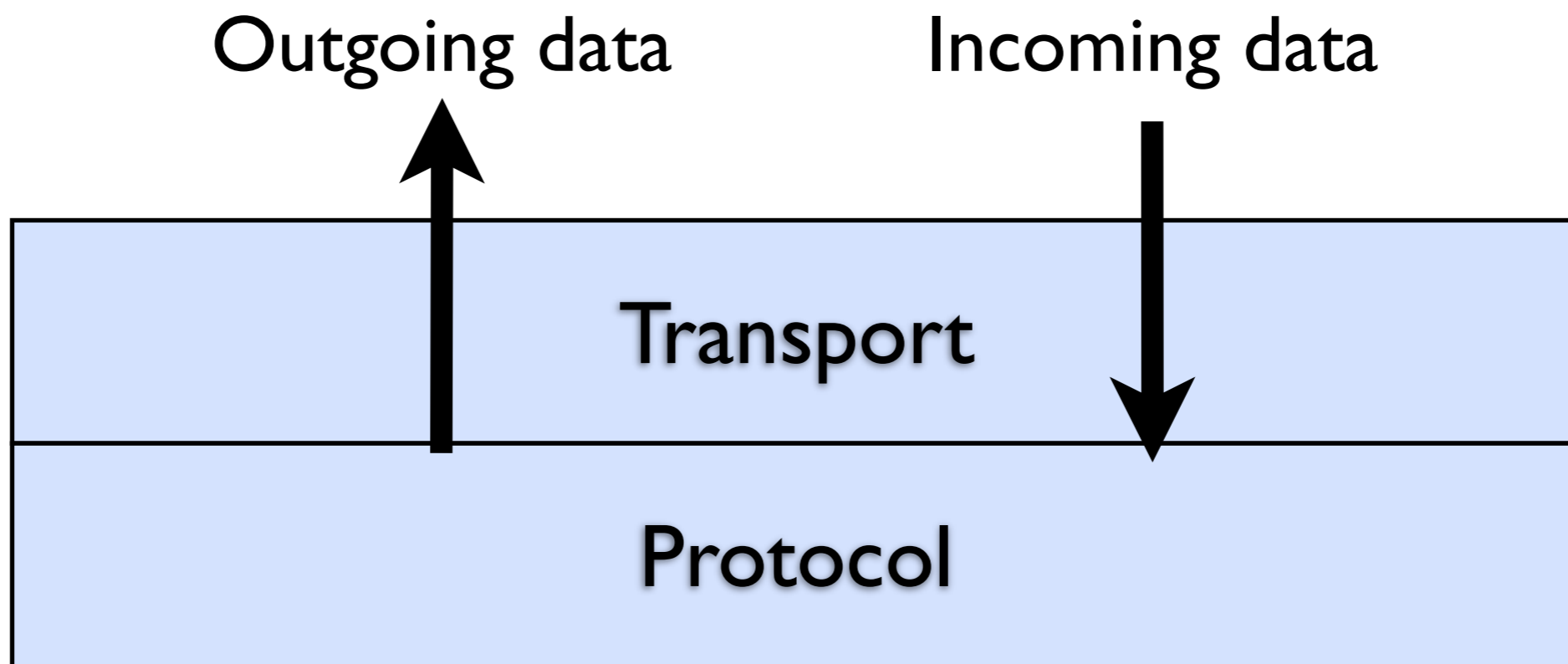
- handle data received



- handle data received
- write data to transport

Saturday, January 26, 13

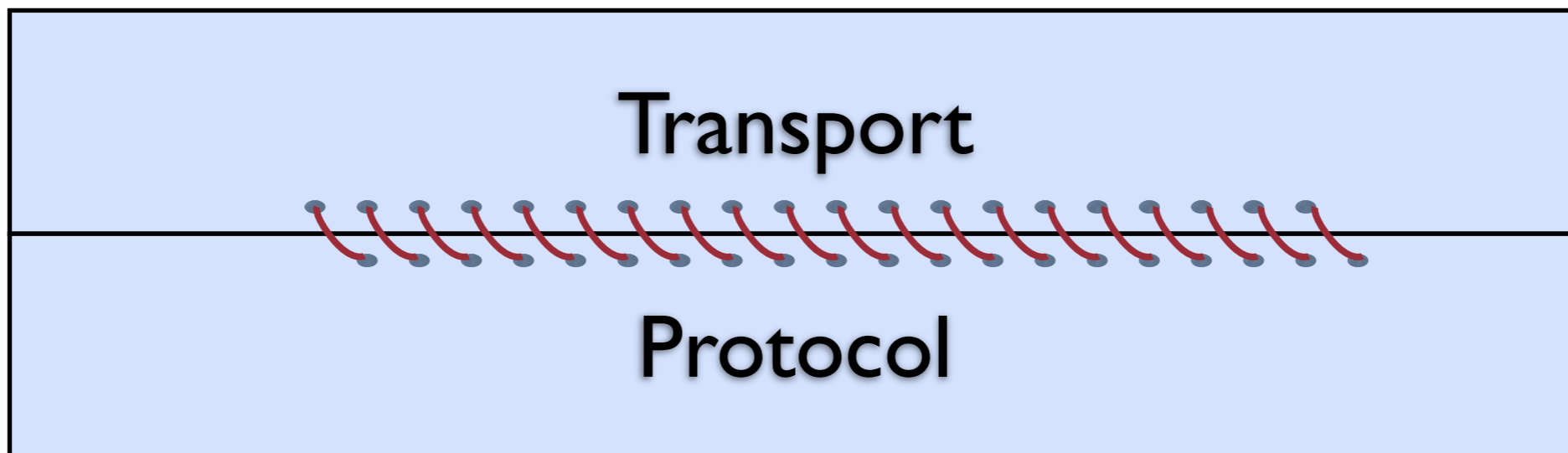
The protocol decides when to write data to its transport, which then in turn (eventually) writes it to a socket or pipe.



- handle data received
- write data to transport
- handle connections

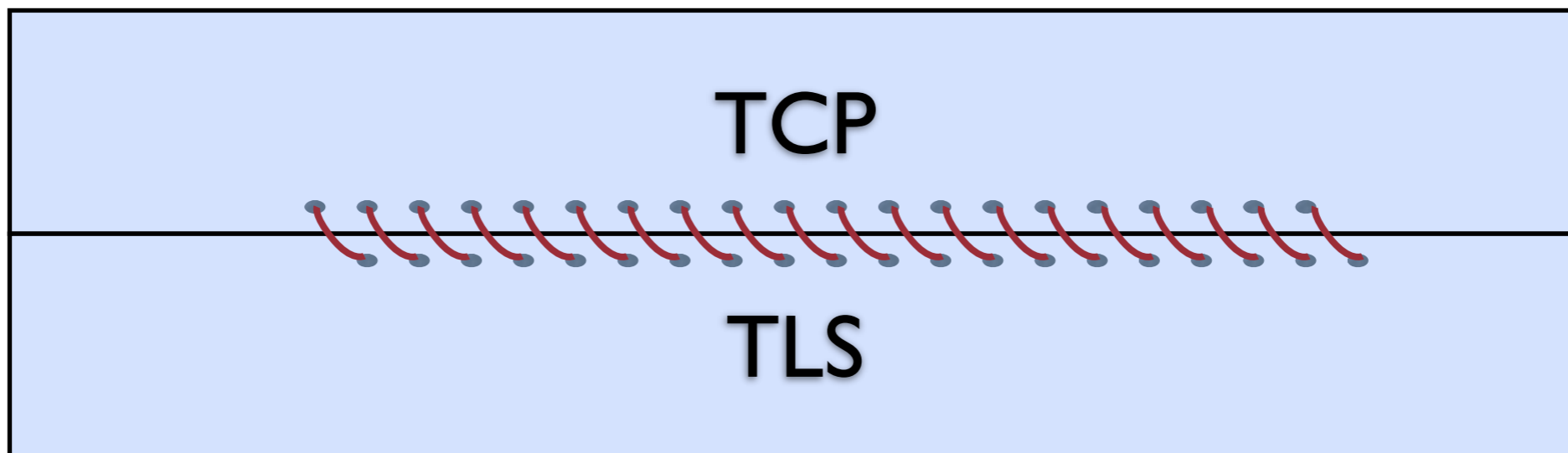
Saturday, January 26, 13

The protocol can also choose to do something when a connection has been made or a connection has been lost.



Saturday, January 26, 13

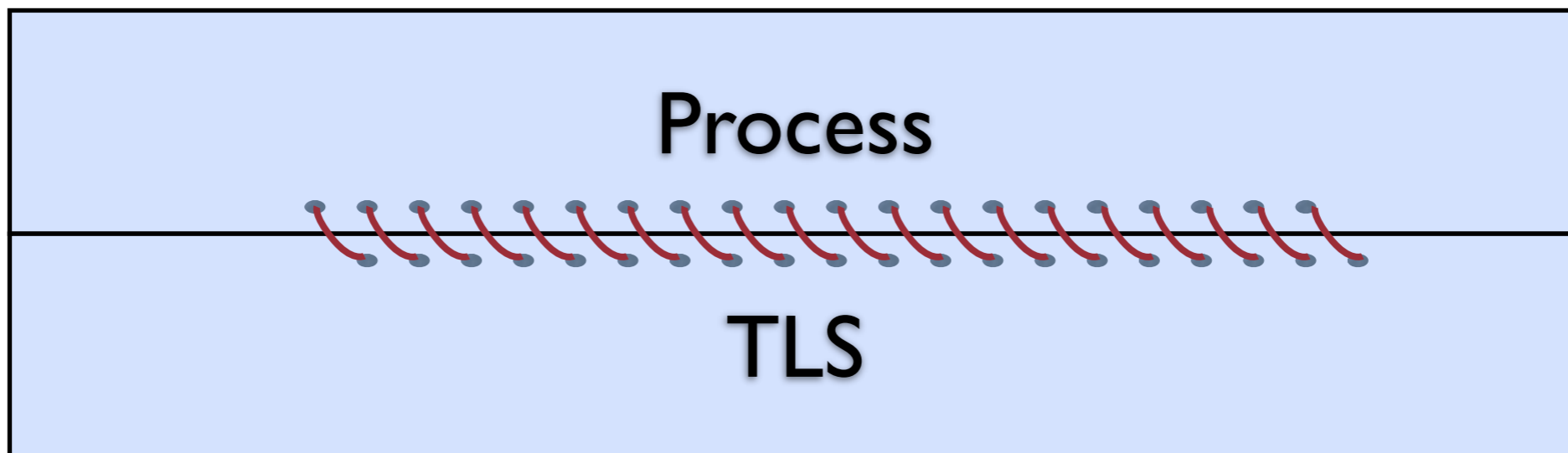
Protocols and transports are separate but tightly bound. What I mean by that is that you cannot have a protocol without a transport or a transport without a protocol, but a protocol may be bound to any type of transport.



Saturday, January 26, 13

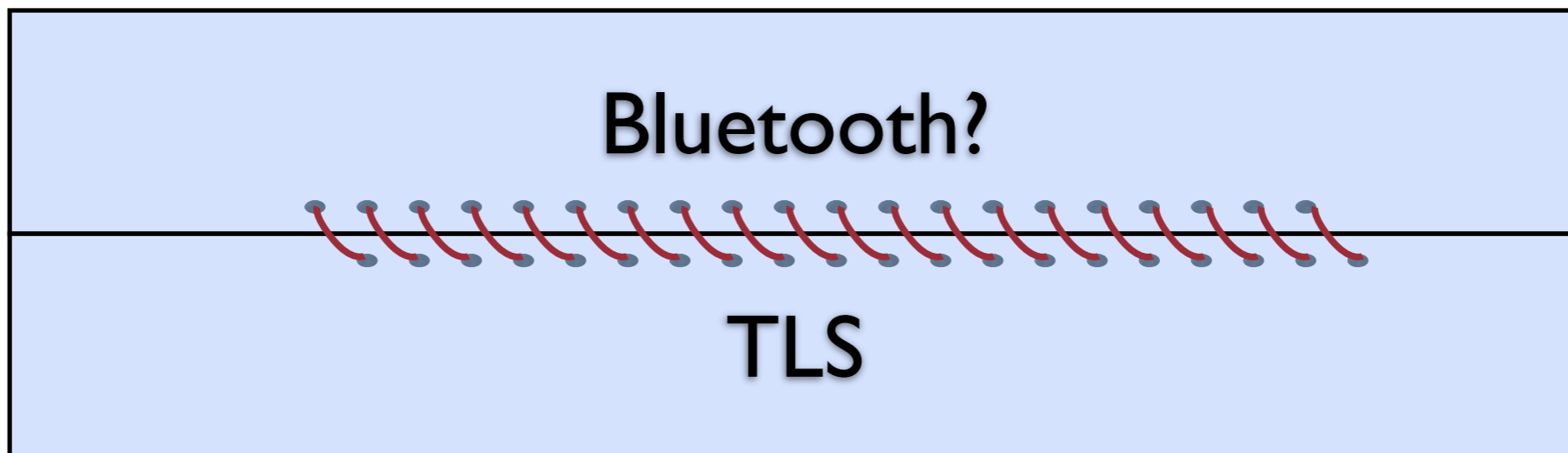
For example, the TLS protocol usually communicates over TCP, and has a TCP transport.





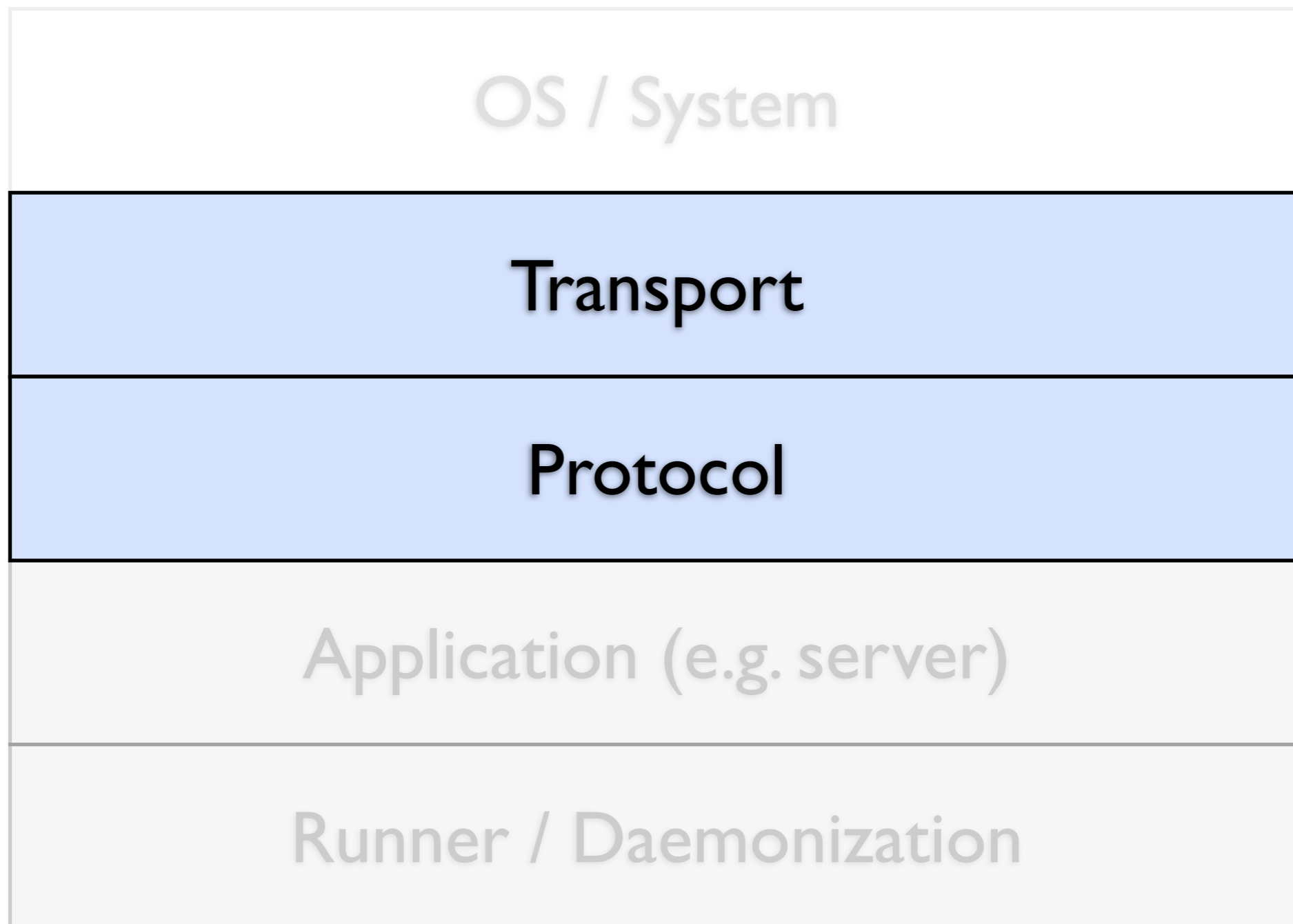
Saturday, January 26, 13

But it can also use a subprocess transport (a pipe) instead.



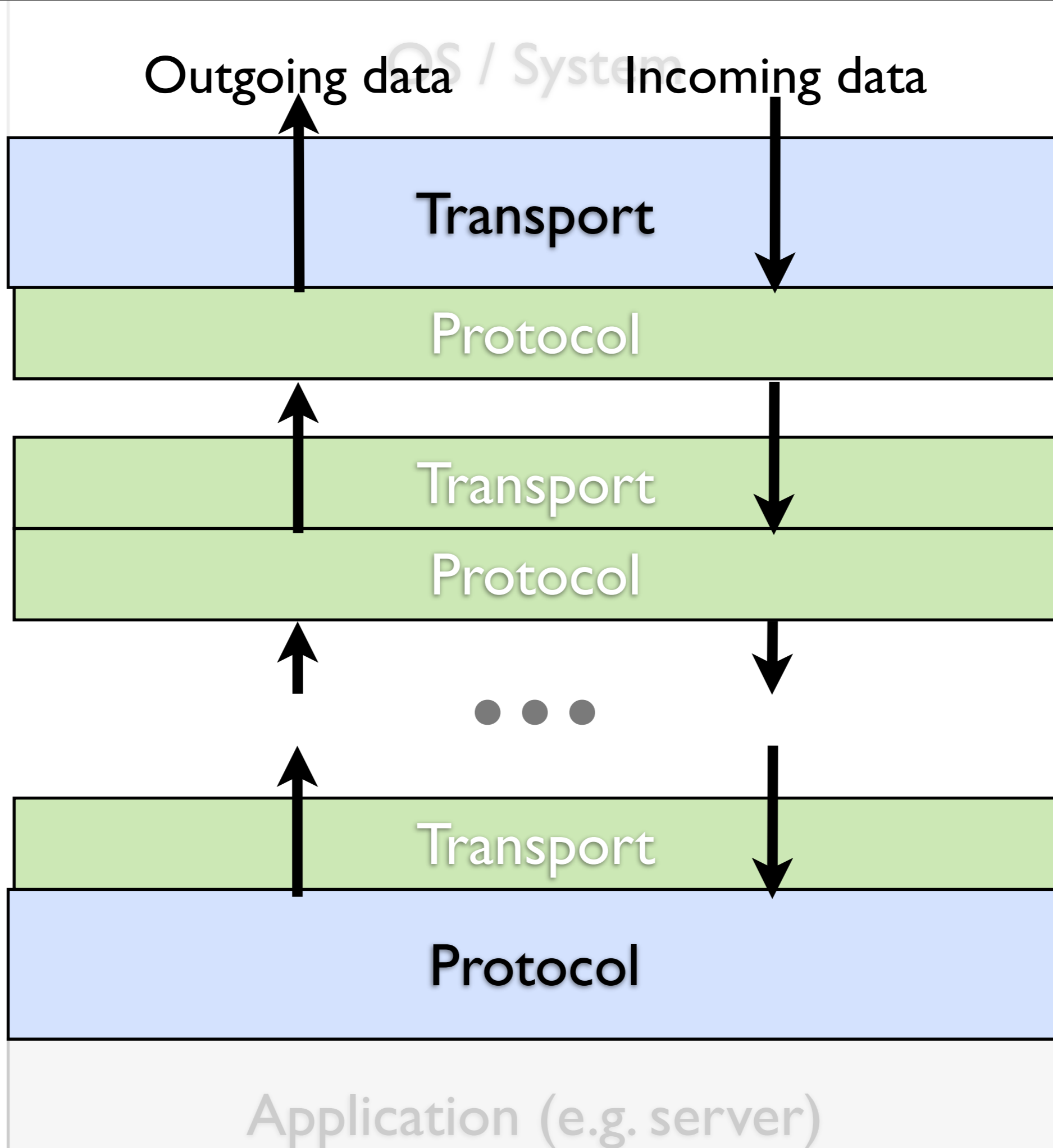
Saturday, January 26, 13

Or as yet unimplemented transports.



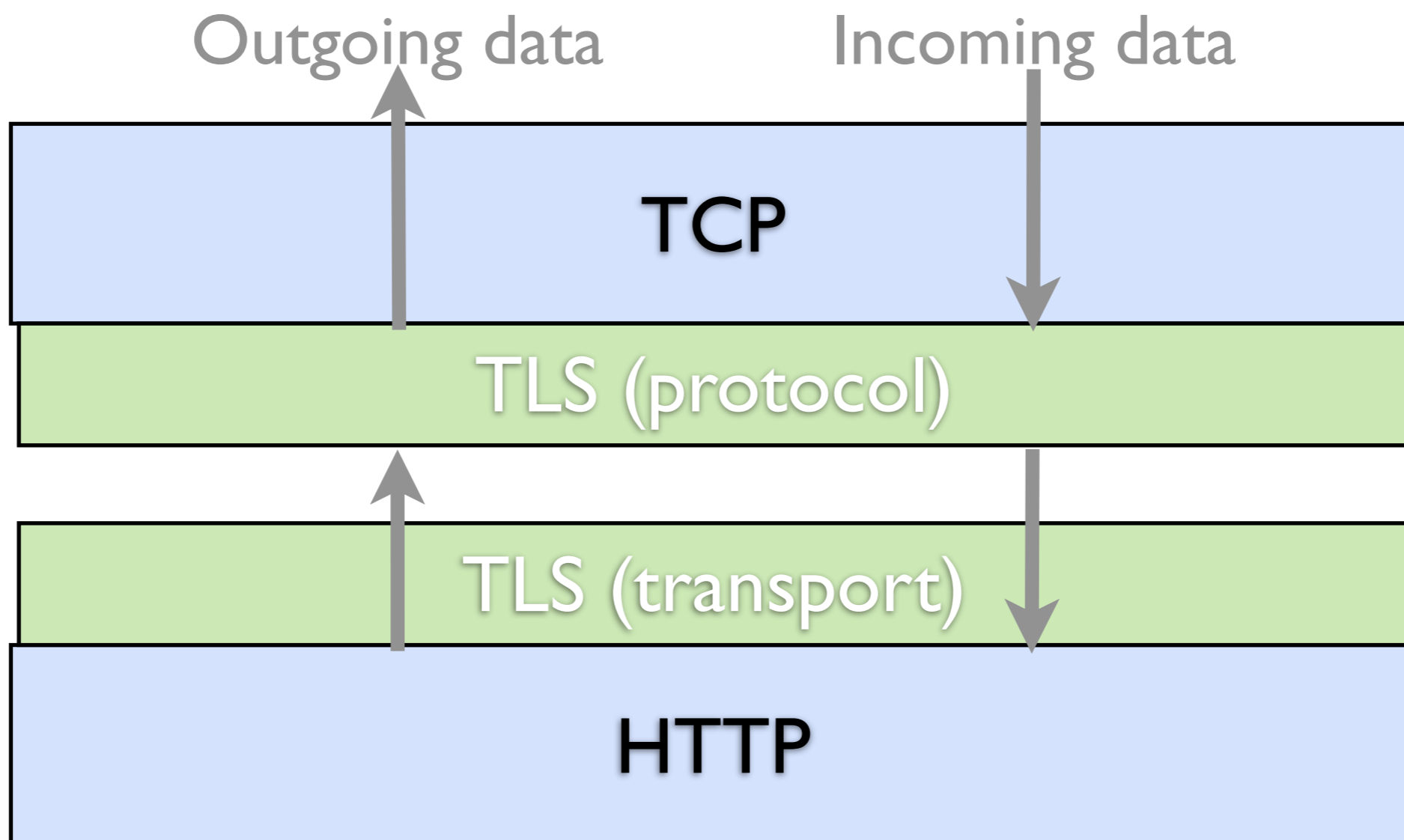
Saturday, January 26, 13

And actually this diagram is a simplification.



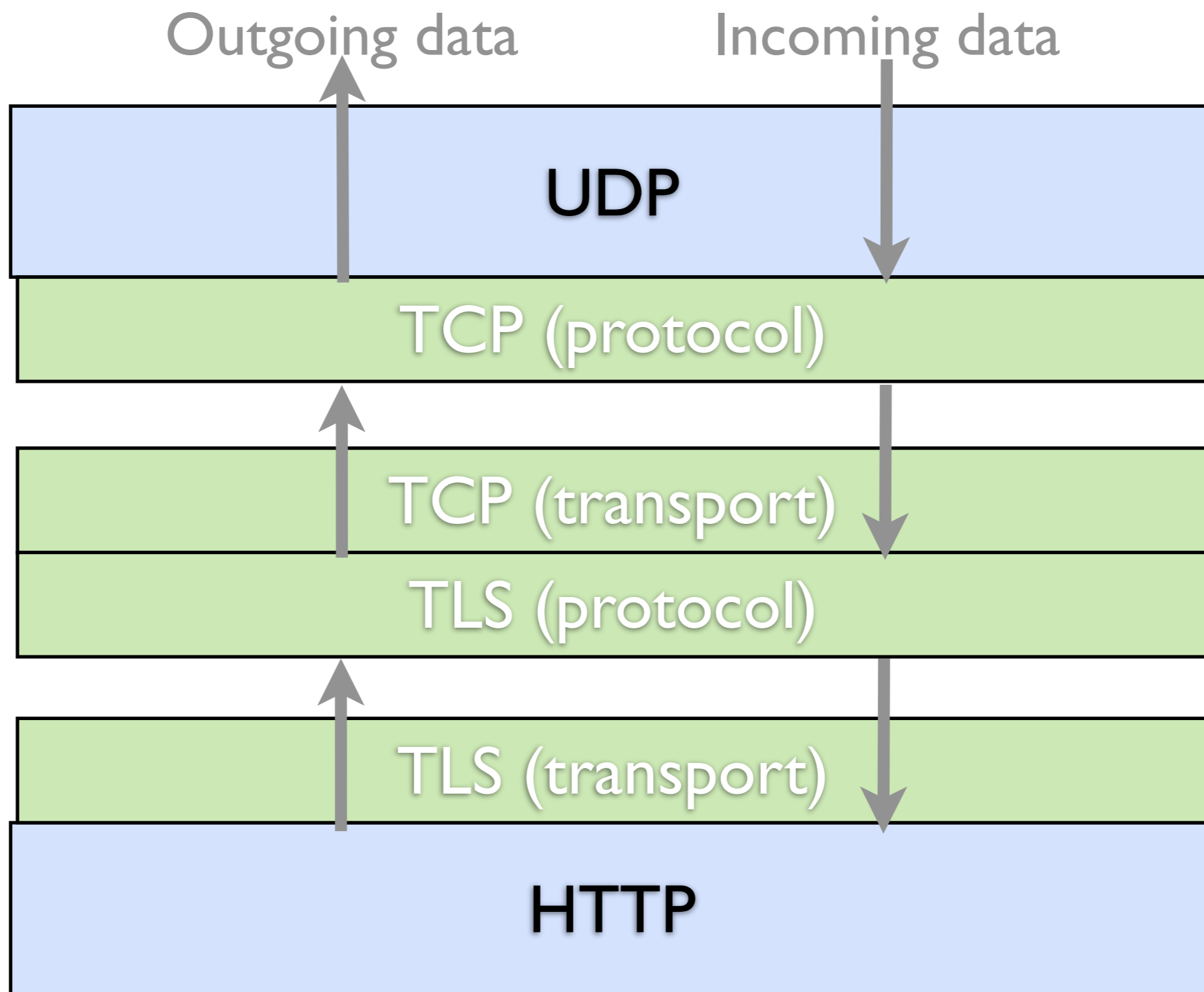
Saturday, January 26, 13

Transport and protocol pairs can be stacked. For example, TLS is a protocol (when bound with a low-level transport like TCP)...



Saturday, January 26, 13

but it is also a transport for a higher level protocol like HTTP.



Saturday, January 26, 13

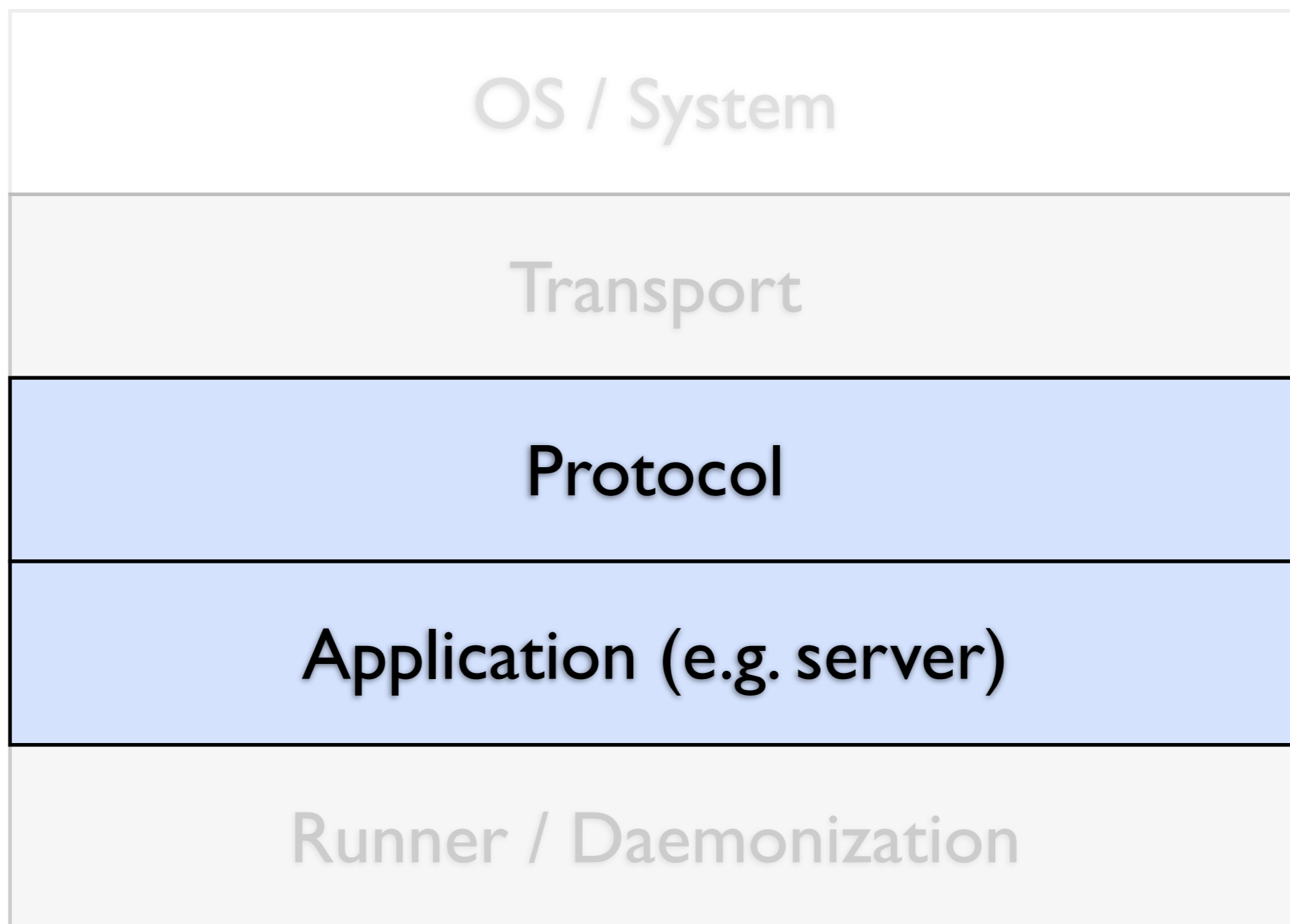
There is a peer-to-peer library, Vertex, which implements TCP over UDP as a Twisted protocol and transport, so theoretically you could even have HTTP over TLS over TCP over UDP.

## Protocol

- TLS
- HTTP
- IRC
- XMPP
- OSCAR
- IMAP
- SMTP
- POP
- DNS
- SSH
- SOCKS
- TELNET
- SIP (voip)
- NMEA (gps)
- ...more

Saturday, January 26, 13

Twisted provides implementations of a lot of protocols out of the box, including but not limited to these protocols on this slide.



Saturday, January 26, 13

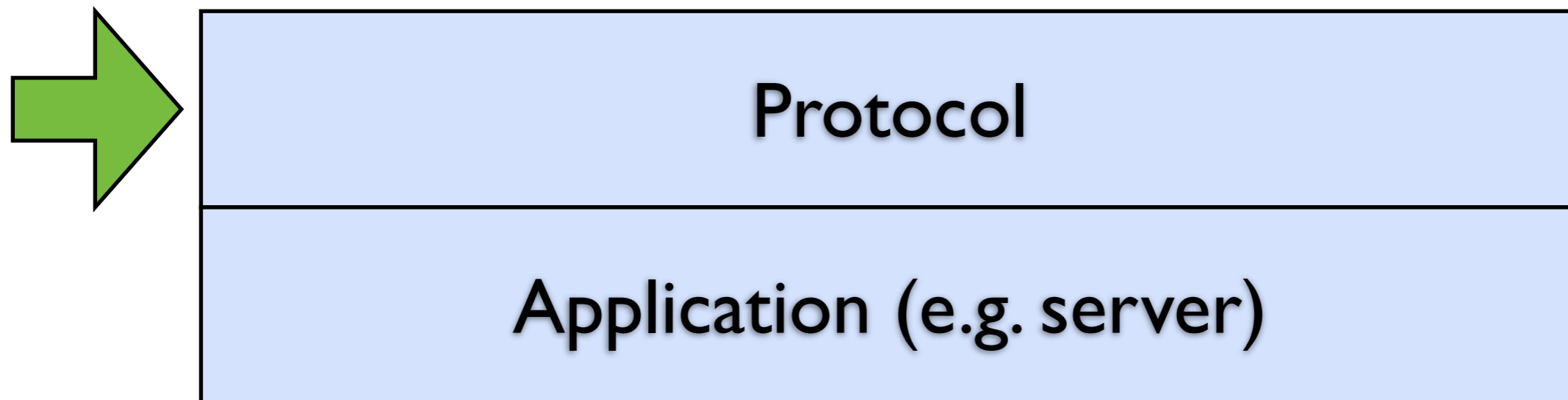
Above the protocol is the code that will build a protocol, and hook it up to a listening or connecting socket. For lack of a better word, I will call this the “application” in the English (not Twisted) sense of the word.



wait... that's it?

Saturday, January 26, 13

That doesn't seem like much, does it? What gives?



Saturday, January 26, 13

Well, most of the heavy lifting in the application is done by the Twisted protocol, not the application. After all, it decides how to handle data that comes in. It decides what data to write. It decides how to handle connection events. It may farm out this logic to other code, which is actually probably what most of the application will consist of.

## Application / Server

- IRC
- SMTP
- SSH
- SFTP
- HTTP
- NNTP
- POP
- Generic
- ...more

Saturday, January 26, 13

Twisted provides a lot of servers out of the box including, but not limited to, the servers on this slide. Do they look familiar? Perhaps like the slide of the protocols Twisted provides?

## Application / Clients

- IRC
- SMTP
- SSH
- SFTP
- HTTP
- NNTP
- POP
- ...more

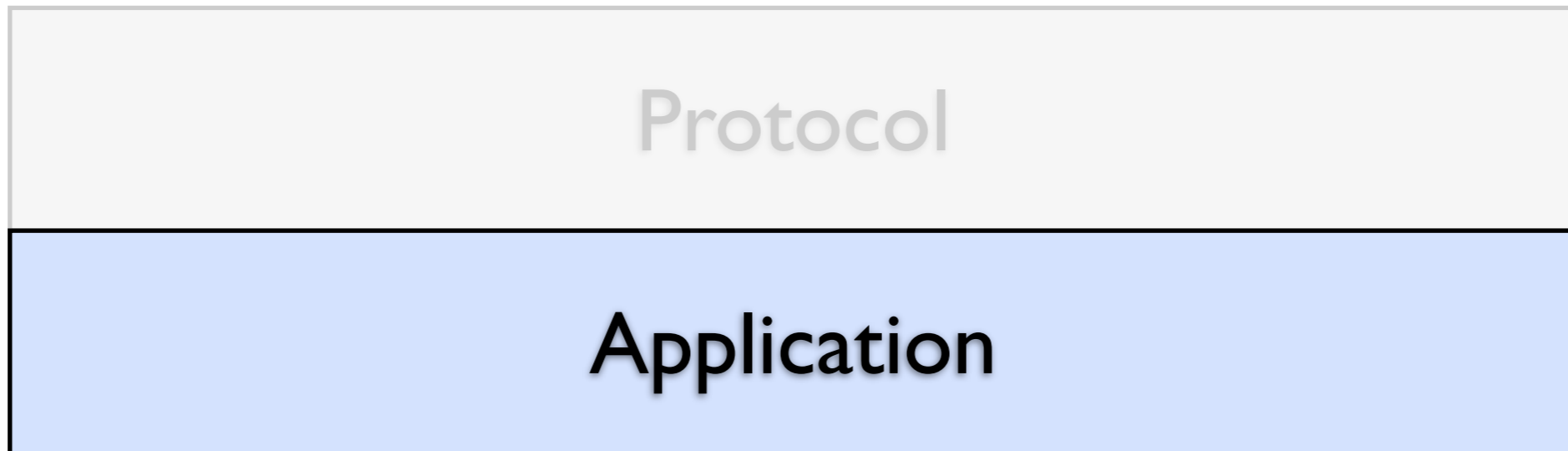
Saturday, January 26, 13

Twisted also provides a lot of clients out of the box. Notice that this is basically the same list as the servers. And they both look like the list of the Twisted protocols I mentioned before, because they build the said protocols.

**server and/or client**

Saturday, January 26, 13

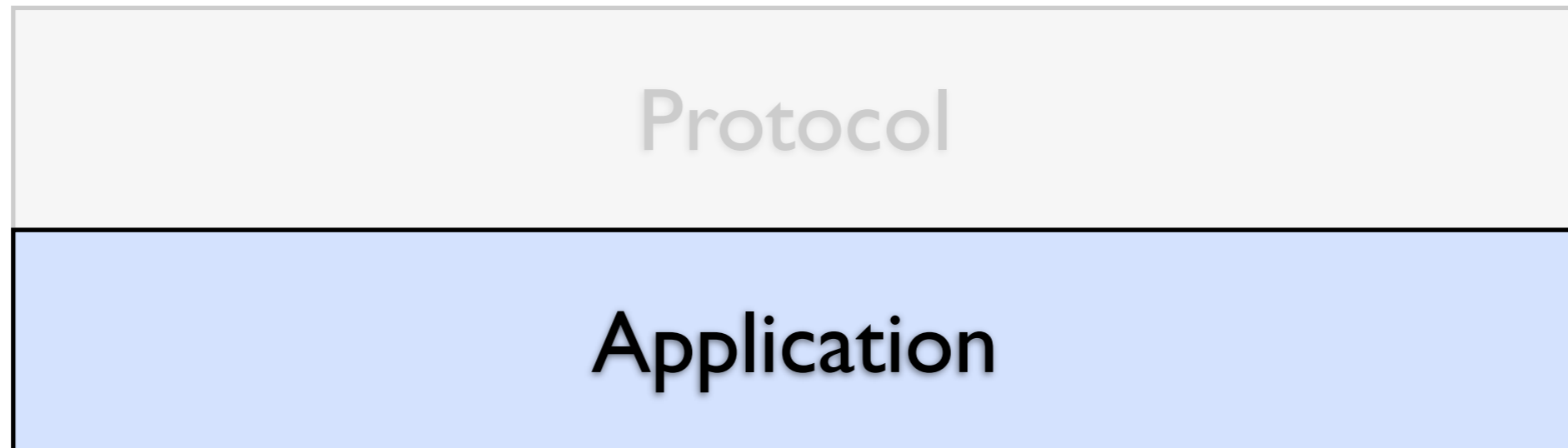
Notice that I mention both servers and clients when I talk about Twisted applications.



- builds protocol
- hooks up protocol

Saturday, January 26, 13

Remember that the application builds a protocol, and hooks it up. That means the application could be just three lines of code that instantiates a Twisted protocol (like the echo protocol), and that tells it to **connect** on a particular port (thus making it a client) or that tells it to **listen** on a certain port (thus making it a server).



- builds multiple protocols
- database
- authentication
- hooks up protocol

Saturday, January 26, 13

A application can also be something more complex. For example, a mail application has to connect multiple protocols (for example, SMTP, IMAP, and POP) to a single back end – like a database. The protocols also have to consult with the same back end for authentication. The application needs to set all this up.

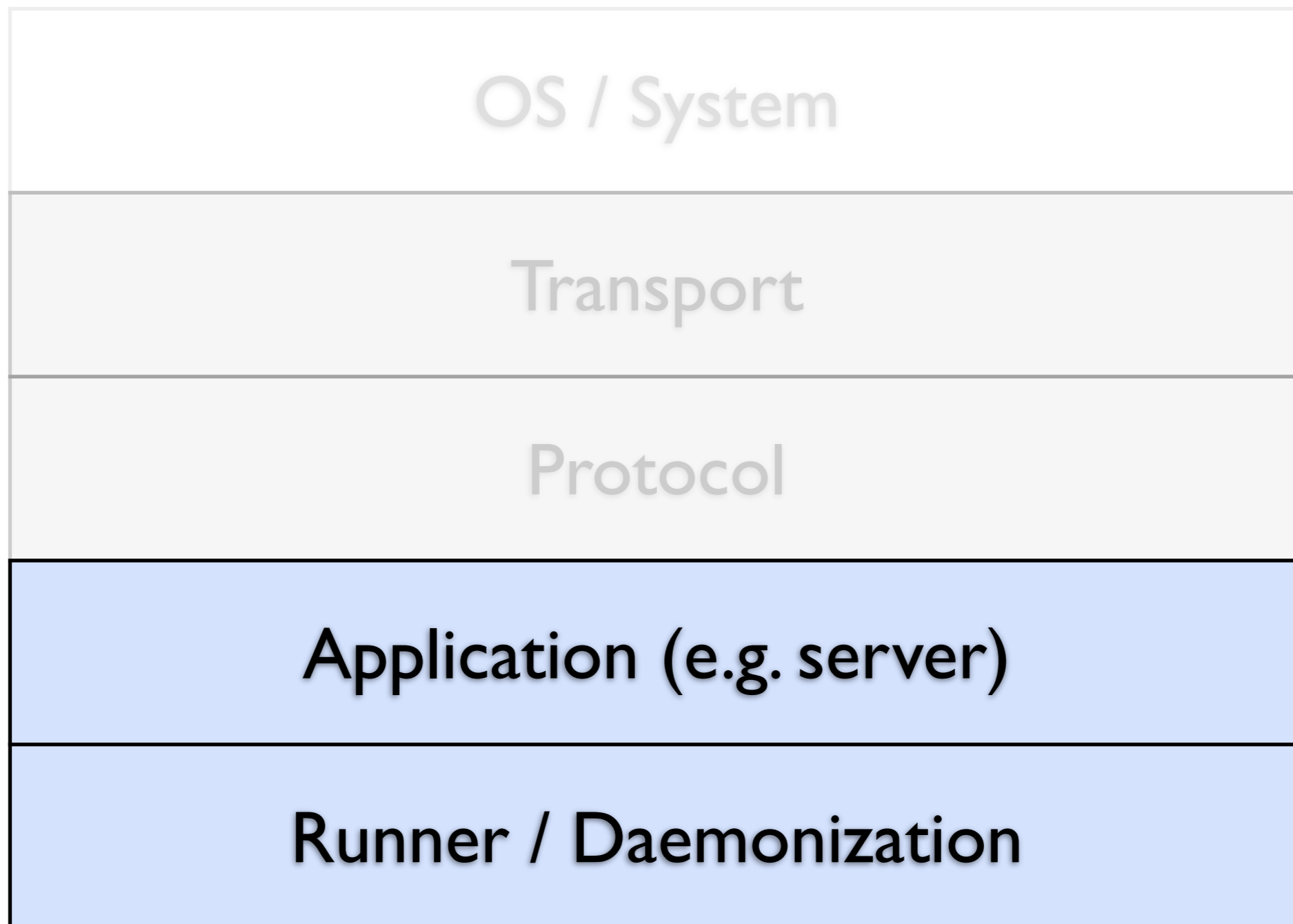
# Authentication

- unix user/password
- ssh key
- user/password list file
- in-memory user/password

Saturday, January 26, 13

Oh, and since we're talking about authentication, Twisted also provides the cred module, which can let you add credentials checking to your application. You can add unix username/password authentication, ssh key authentication, or user/password authentication for your application against an in-memory or in-file list of usernames and passwords.

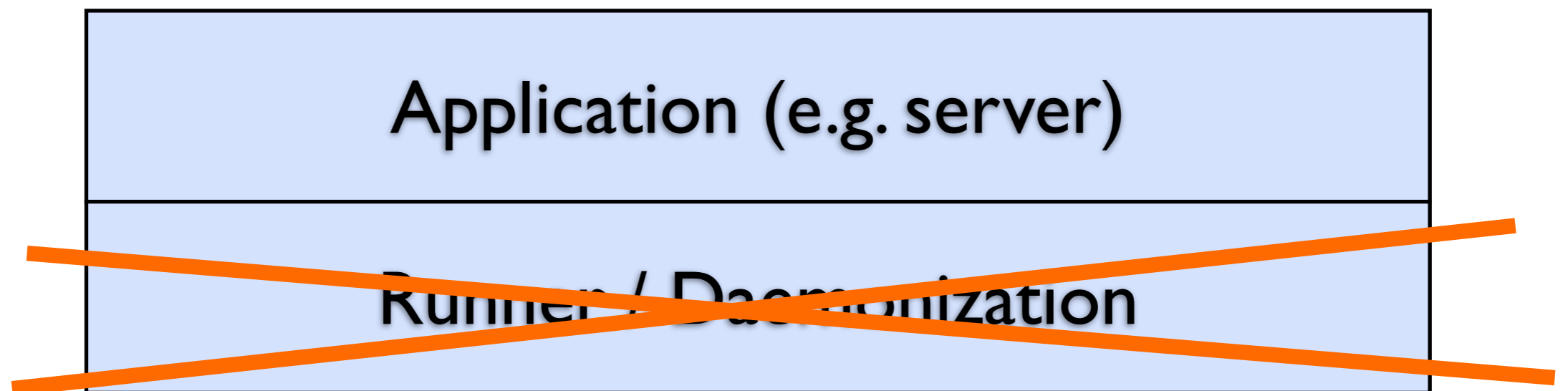




Saturday, January 26, 13

So the application has logic for both building a protocol and hooking it up. How do we kick off (or run) a application?

# python myapplication.py



Saturday, January 26, 13

If we wrote our little application as 3 lines of code in a script that instantiates a protocol and then tells it to connect to a port, we can just run that script.

```
twistd [options] <myplugin>
```

Saturday, January 26, 13

If we want to actually deploy our code rather than run a python script, we can write a Twisted plugin and use the twistd utility.

# twistd

- daemonization
- logging
- privilege dropping
- chroot
- non-default reactor
- profiling

Saturday, January 26, 13

twistd enables us to daemonize the application, write time-stamped log files to a particular location, drop privileges once the application has started, run the application in a chroot, use a non-default reactor, and/or profile our application. Which seem like pretty useful things to be able to do.

```
twistd [options] <myplugin>  
      [my_plugin_options]
```

Saturday, January 26, 13

You may even provide command line options specific to your plugin to the twistd utility.

twistd

```
twistd web --port 80 --path /srv/web
```

Saturday, January 26, 13

Twisted actually comes with several useful plugins built in. Twisted web for example runs a web server on a specified port. It can serve static files from a particular directory (in this slide `/srv/web`), or a script passed to it on the command line.

**twistd**

**twistd web --port 8080 --path /srv/web**

**twistd telnet --port 4040**

Saturday, January 26, 13

twistd telnet runs a telnet server on the specified particular port

## twistd

- twistd web
- twistd telnet
- twistd dns
- twistd ftp
- twistd mail
- twistd conch (ssh)
- ...more (see `twistd --help`)

Saturday, January 26, 13

And lots more. To see all the plugins available, after you've installed twisted, type `twistd --help`



# Twisted

“An event-driven networking engine written in Python”

Saturday, January 26, 13

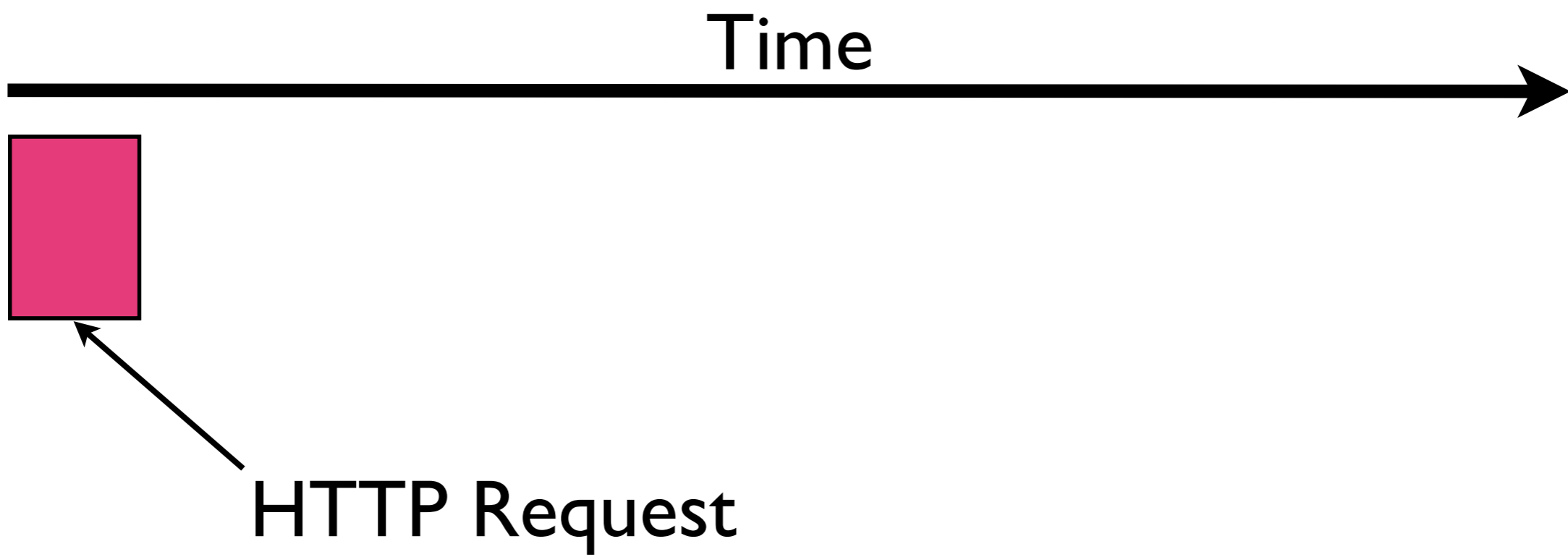
So that is what an application in Twisted looks like from a high level... I've been mentioning callbacks a lot, and previously I said that Twisted was event-driven. What does that mean? This is best described by giving a (contrived) example.

# Concurrency Models

(A story of 3 HTTP requests)

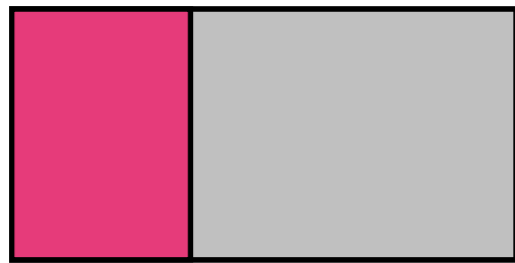
Saturday, January 26, 13

Let's say we are writing an application that performs three simple tasks, each of which does some blocking I/O. Say, we want to make HTTP requests to 3 different endpoints and interpolate the results in some way (like counting the total number of words in all 3 responses). So the easiest way to do this is to write a single threaded application which makes the 3 requests sequentially, then interpolates.

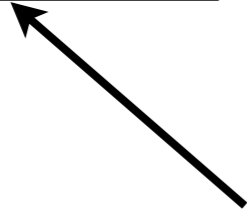


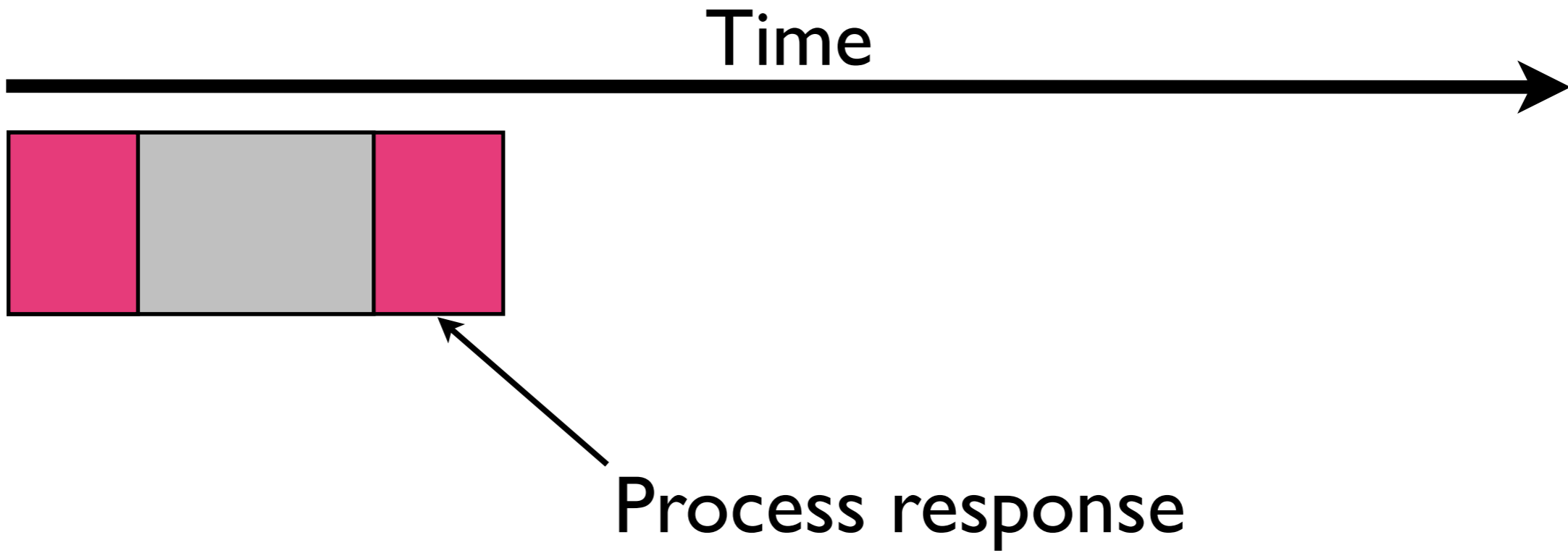
Saturday, January 26, 13

This is not to scale or anything, just a visual representation. That red square represents the time it takes to form an HTTP request and write it to a socket.



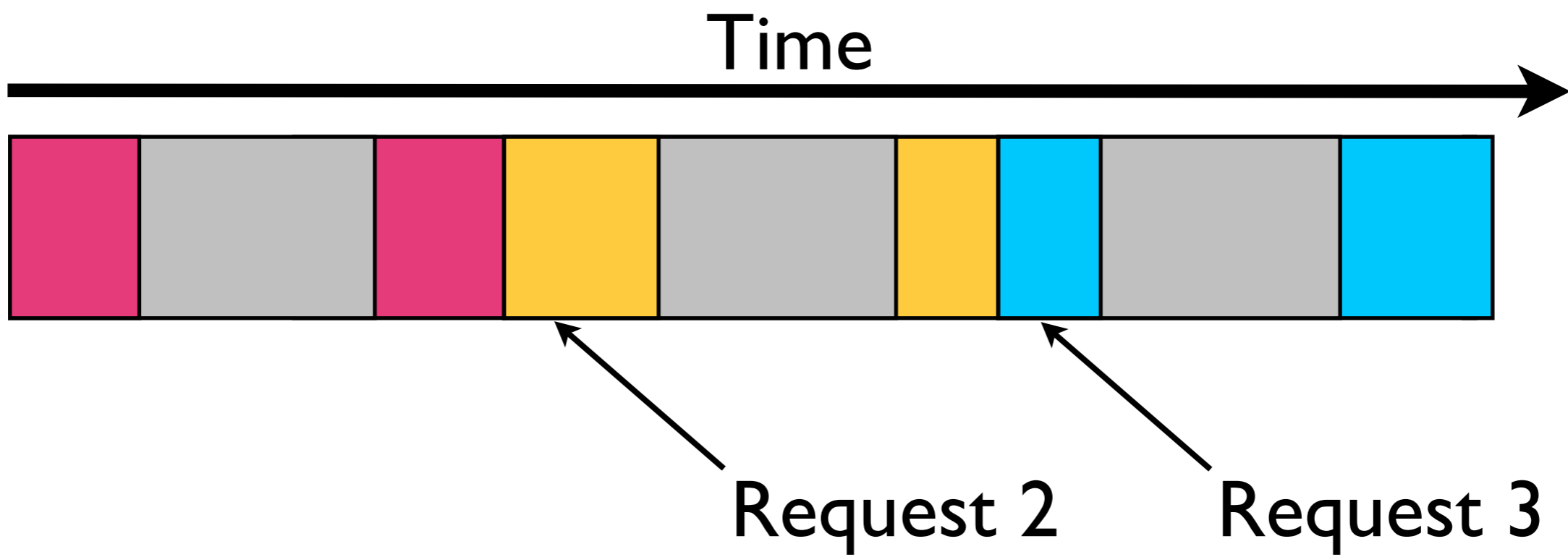
Await response





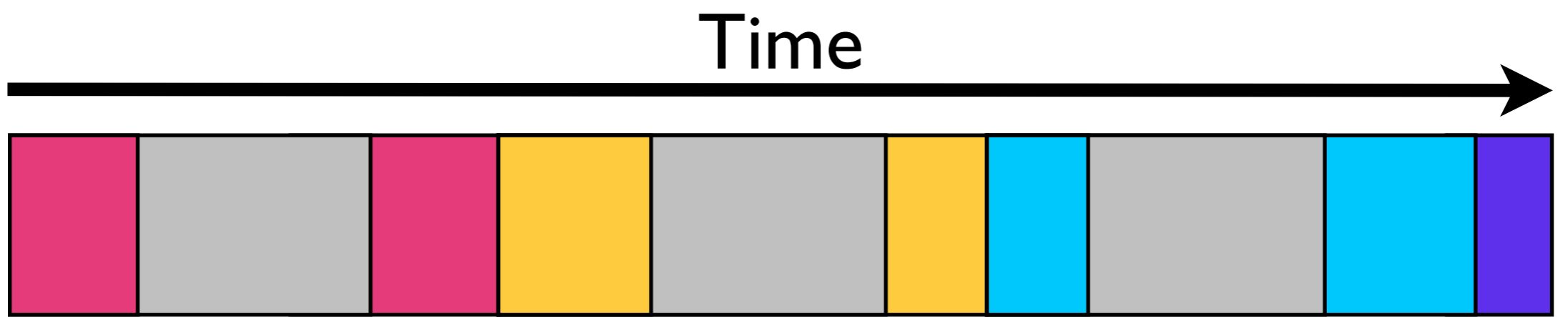
Saturday, January 26, 13

We get the response back and we process it in some way...



Saturday, January 26, 13

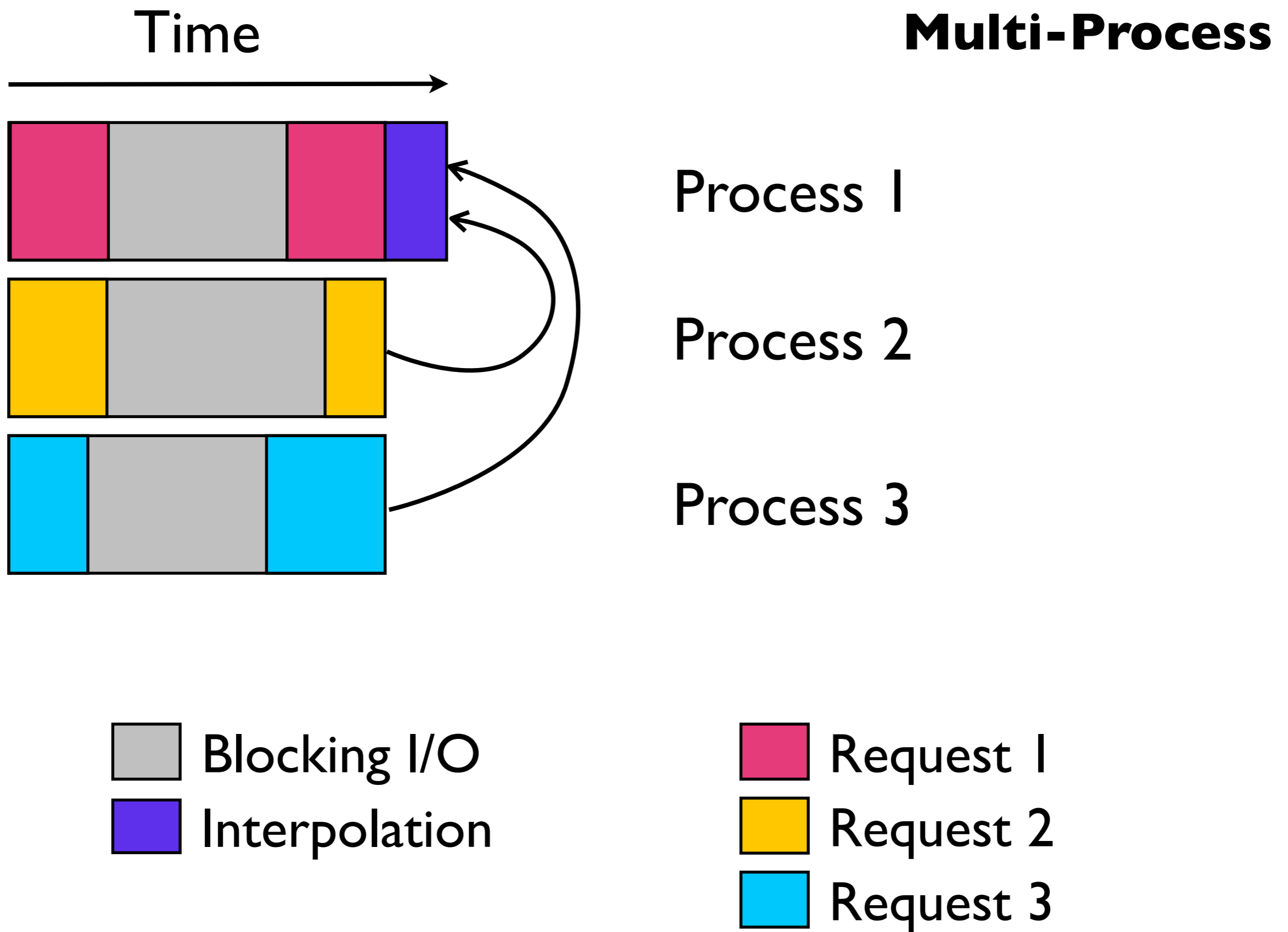
And then we repeat this 2 more times with the other two endpoints.



Interpolate  
Responses

Saturday, January 26, 13

After we have all 3 responses, we can interpolate them (or otherwise manipulate them together).  
This single-threaded application is very easy to understand and debug, but is unnecessarily slow.



Saturday, January 26, 13

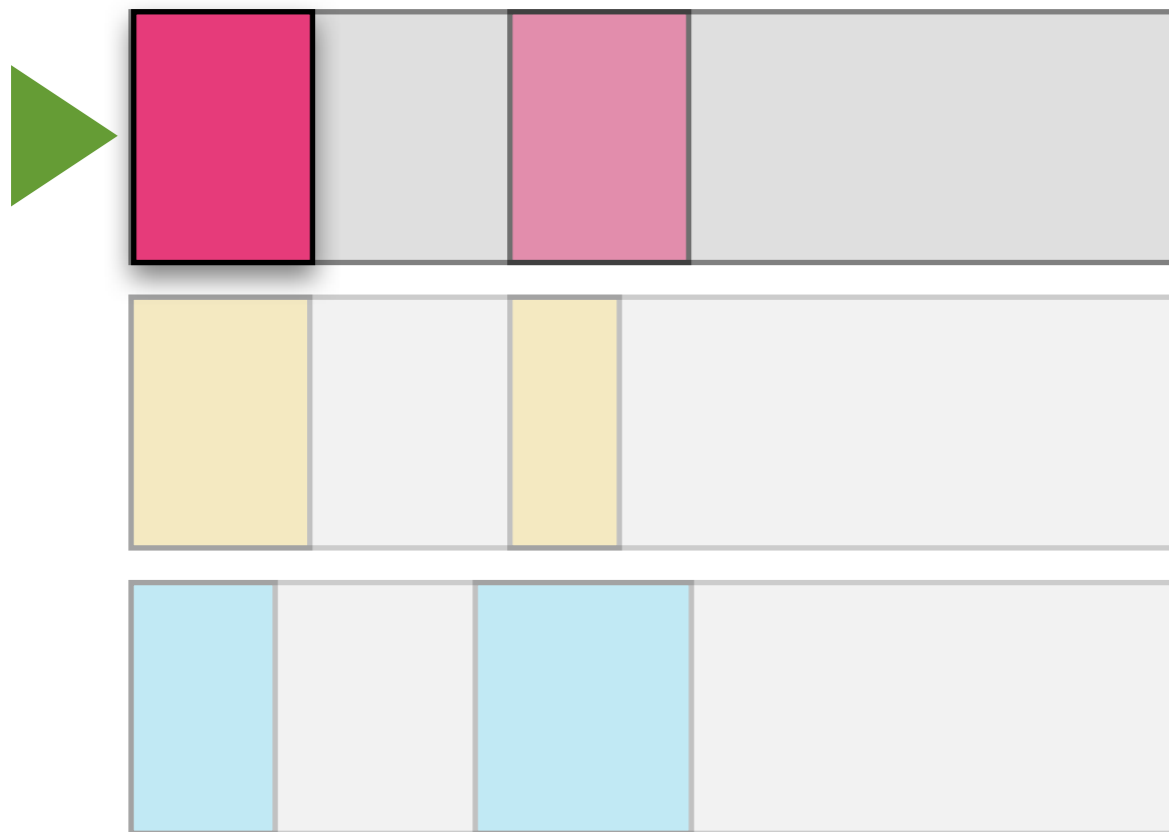
We can instead make each request in a separate OS process. That way, the 3 requests can be made in parallel and no single request task has to wait for any of the others. However, there is a memory and scheduling overhead to every additional process. And, there is a data serialization overhead for interprocess communication, which would be needed to interpolate the results. (In this example, from processes 2 and 3 back to process 1).

Also, this diagram would only apply if each process ran on a separate CPU. If they all ran on the same CPU, the OS scheduler would have to switch between processes, much like what happens if multiple threads are run on one CPU.



Time  
→

# Multi-Threaded



Thread 1

Thread 2

Thread 3

Blocking I/O  
Interpolation

Request 1  
Request 2  
Request 3

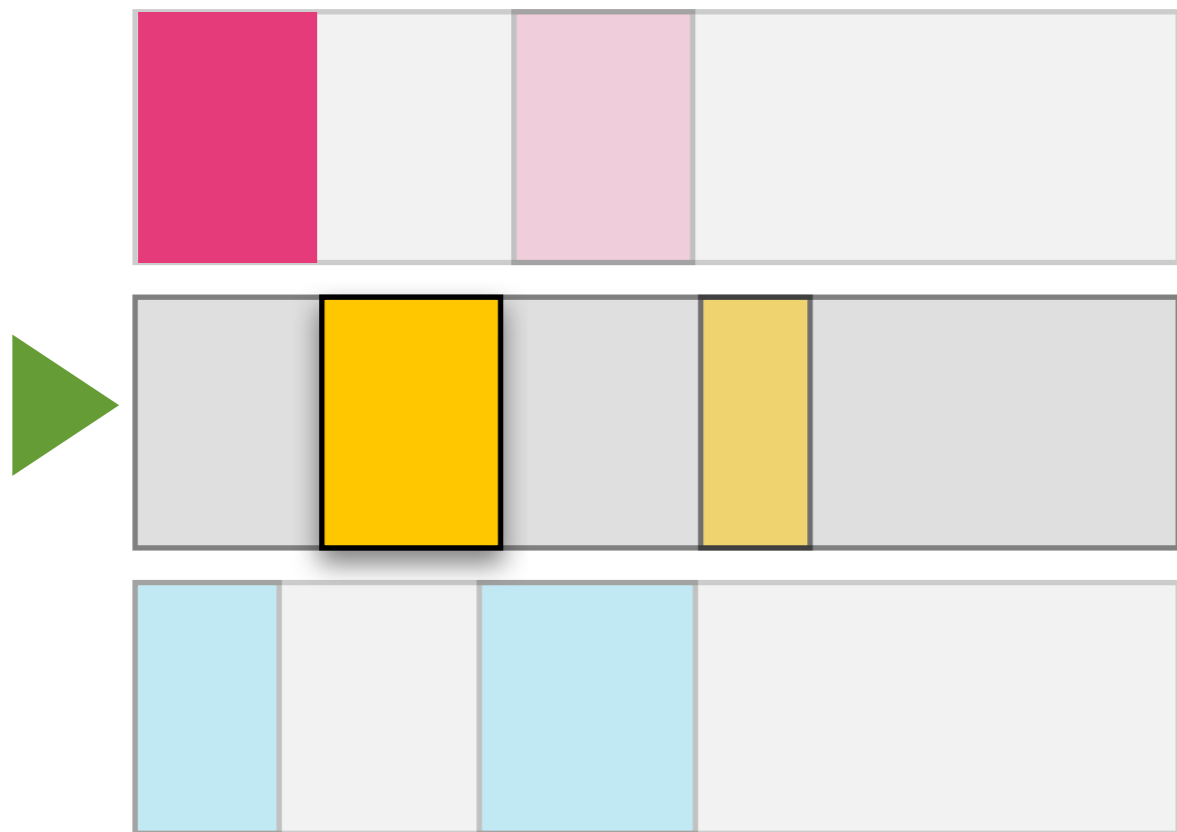
Saturday, January 26, 13

With multiple threads on one CPU, the three requests are made as if they will be run completely independently. But only one thread can run at once. So the scheduler picks one thread to run...

Time



# Multi-Threaded






Thread 1

Thread 2

Thread 3

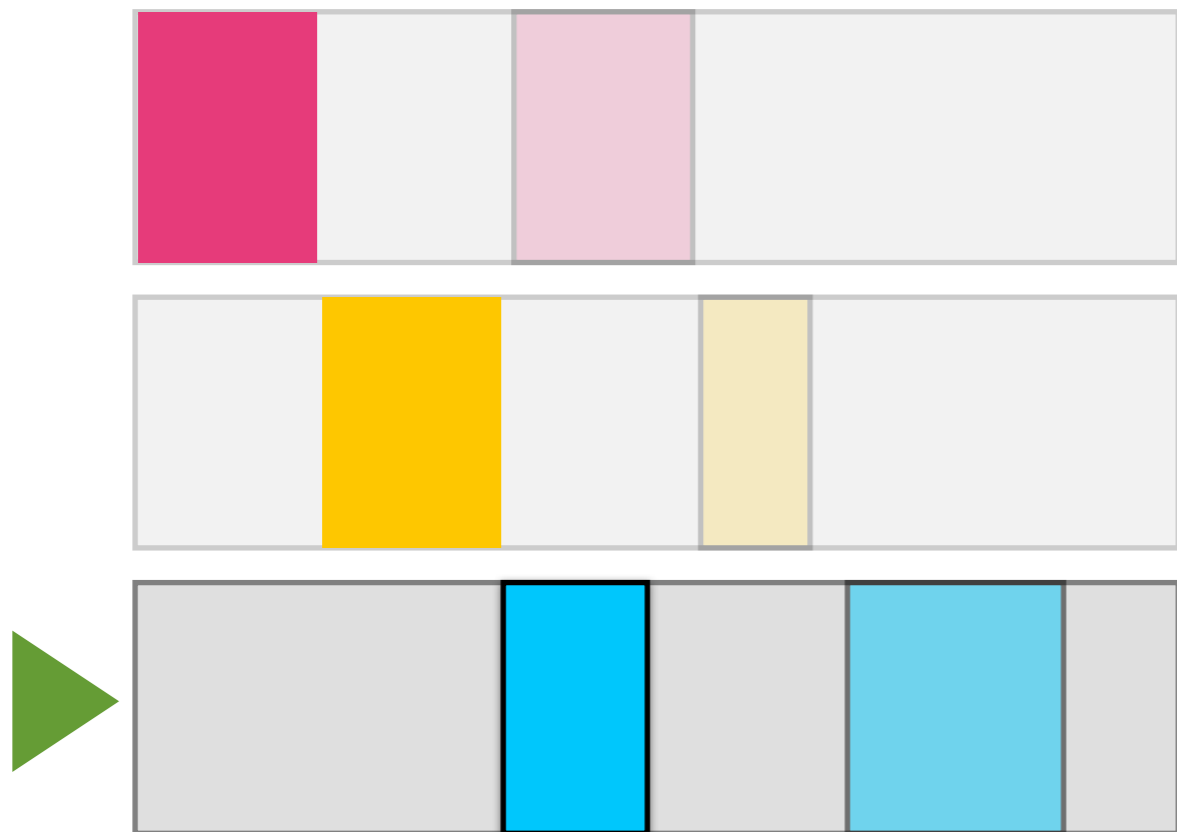
 Blocking I/O  
 Interpolation

 Request 1  
 Request 2  
 Request 3

Time



# Multi-Threaded






Thread 1

Thread 2

Thread 3

-  Blocking I/O
-  Interpolation

-  Request 1
-  Request 2
-  Request 3

Time



# Multi-Threaded



Thread 1






Thread 2



Thread 3

 Blocking I/O  
 Interpolation

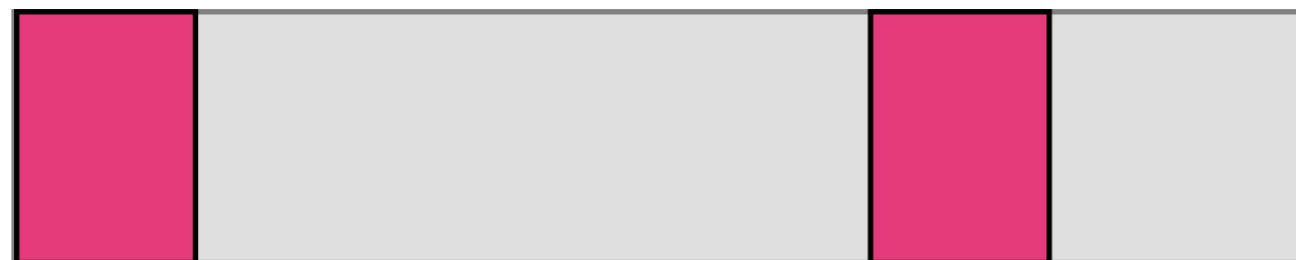
 Request 1  
 Request 2  
 Request 3

Saturday, January 26, 13

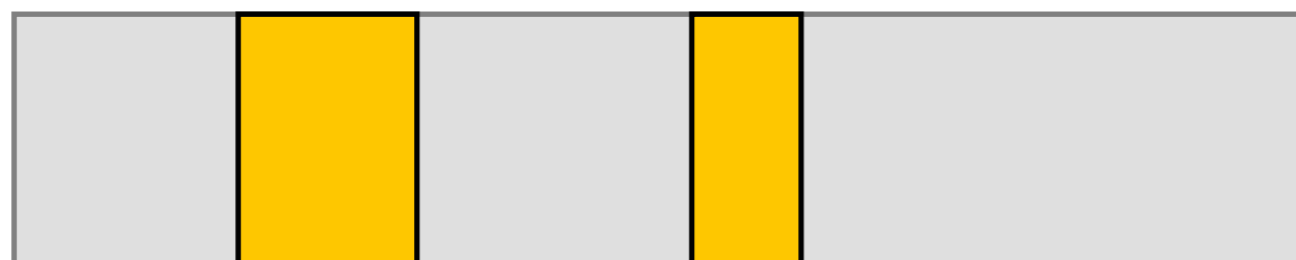
So the threads do not quite run in parallel. Even if multiple CPUs were available, we cannot guarantee that each thread runs on a different CPU.

Time

# Multi-Threaded



Thread 1






Thread 2



Thread 3

 Blocking I/O  
 Interpolation

 Request 1  
 Request 2  
 Request 3

Saturday, January 26, 13

And if these are OS threads, there is no guarantee that the scheduler will pick one of our threads to run when one thread blocks. So our threads may have to wait some for some other thread we do not control to run.

# Threads

- Locking
- Re-entrancy
- Debugging

Saturday, January 26, 13

But even if we were using green threads, rather than OS threads, we'd still have to worry about locking read and writes from shared data, re-entrancy of our functions. Debugging is also made harder by the fact that thread-safety bugs tend to appear under heavy load, and can be difficult to reproduce due to their non-deterministic nature.

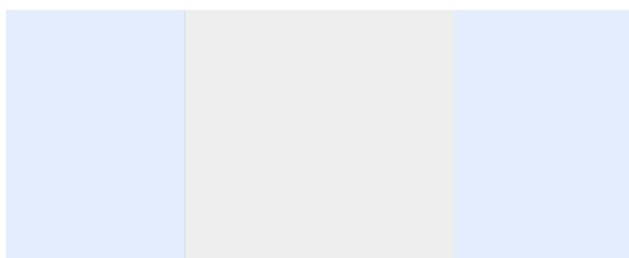
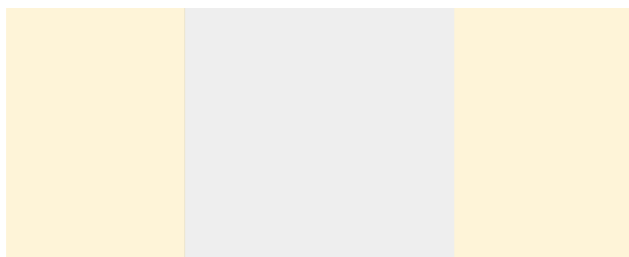
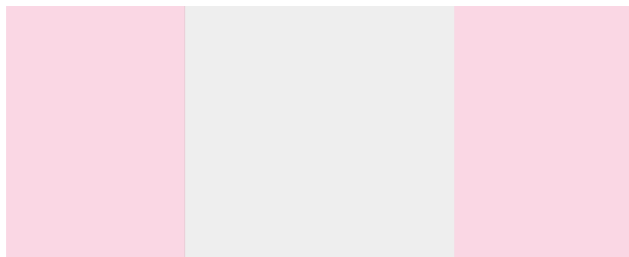
# Event-Driven

(not to scale)

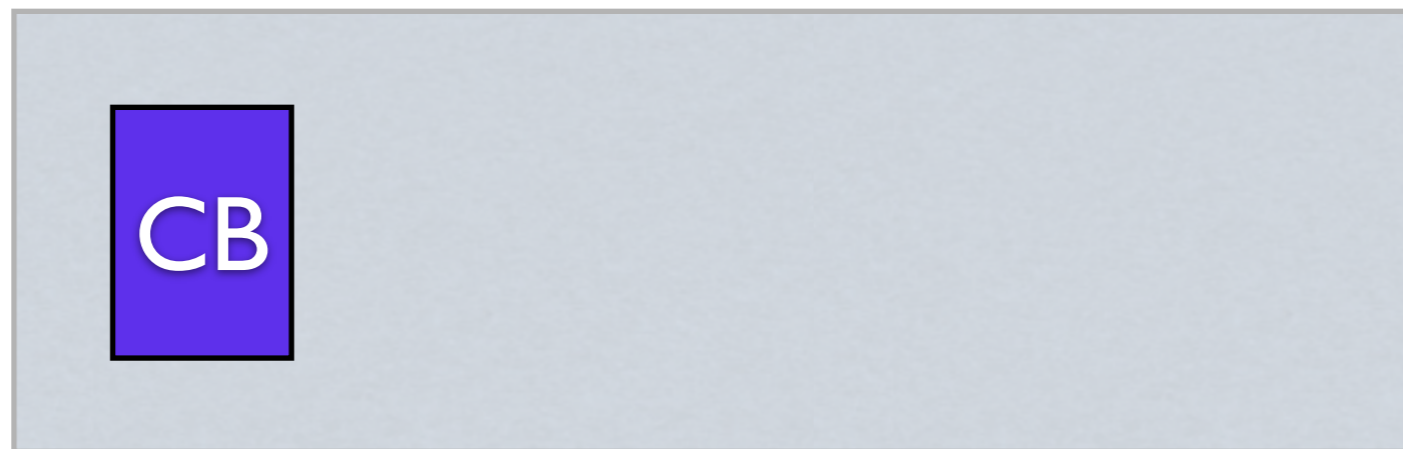
Saturday, January 26, 13

Now let's describe how to do this in an event driven way. In this model, there is a loop, called an event loop which waits for events and dispatches them to handlers.

Time



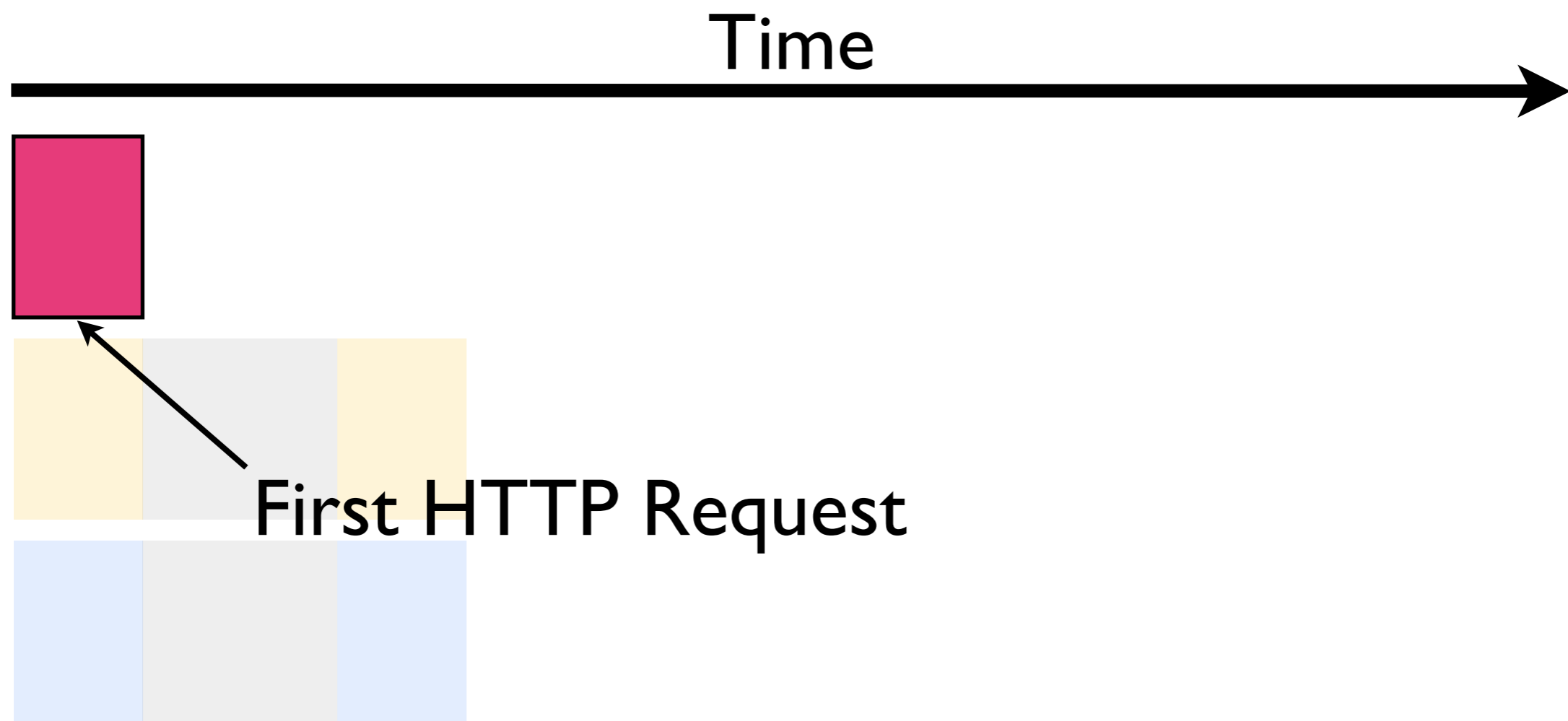
Callbacks:



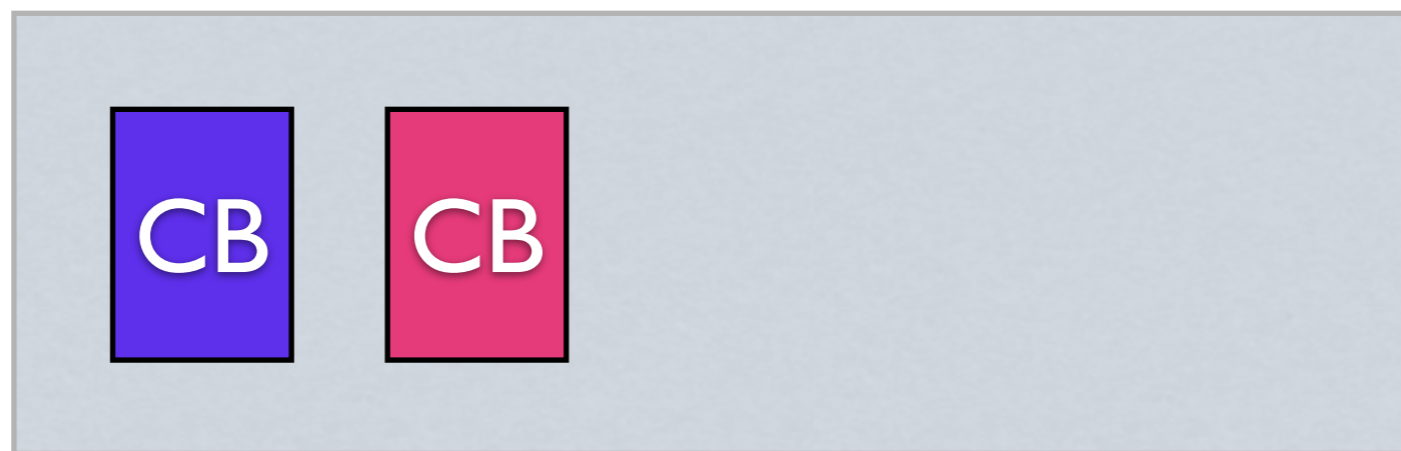
Saturday, January 26, 13

In this model the event loop (and everything else) is run in a single thread. First, we make a callback (which is just some code that will be run later) that interpolates responses. We register this callback to run once the responses for 3 requests have been received and processed.



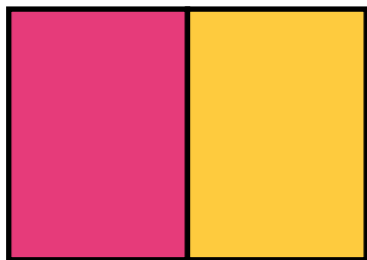


Callbacks:

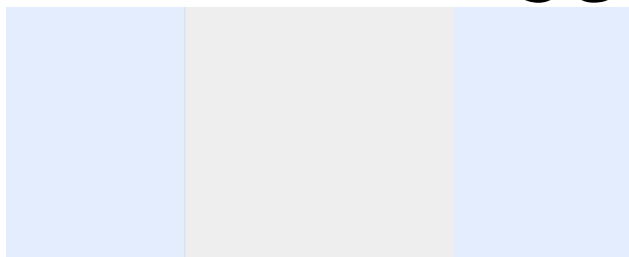
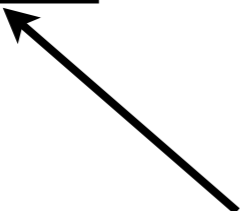


Saturday, January 26, 13

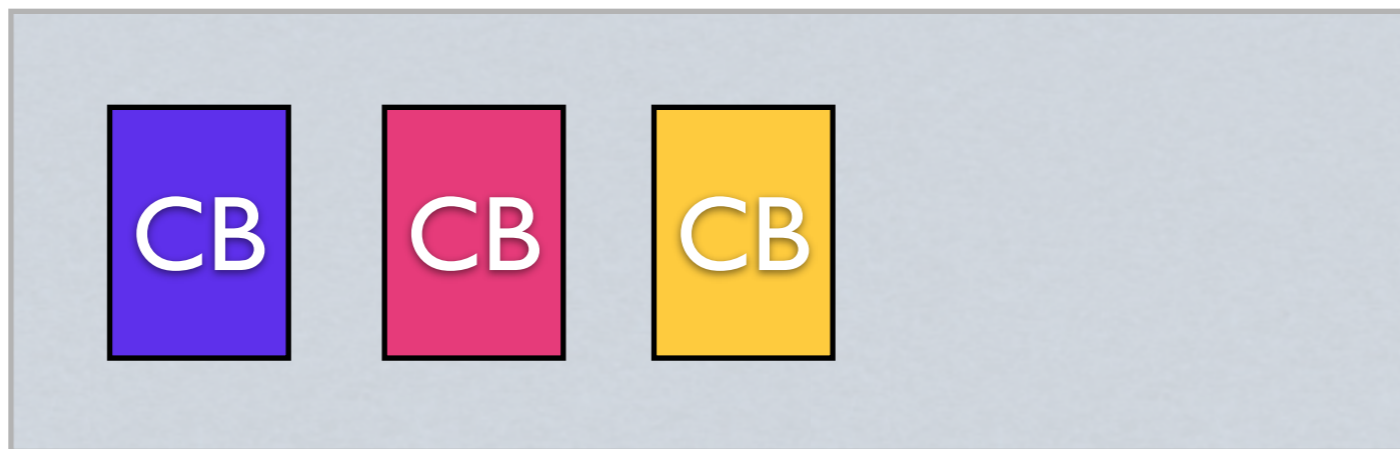
Now we can start making the requests. We make the first request, and register a callback with which to handle the result. This callback automatically get called when the response comes in, so rather than having to wait for the response, we can yield to other code that is ready to run...



Second HTTP Request



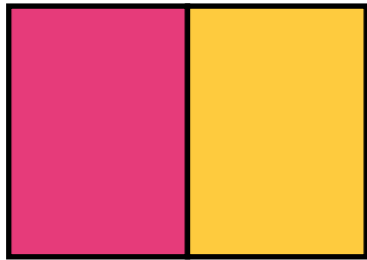
Callbacks:



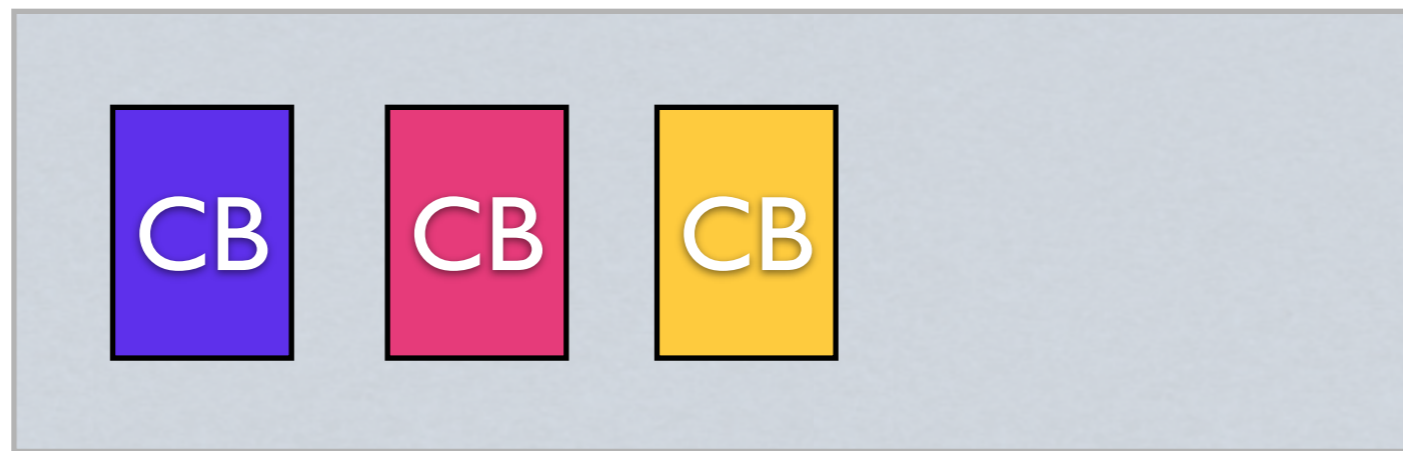
Saturday, January 26, 13

Which is the code that makes the second request. This second request also registers a callback to handle its response, and then yields.

Time

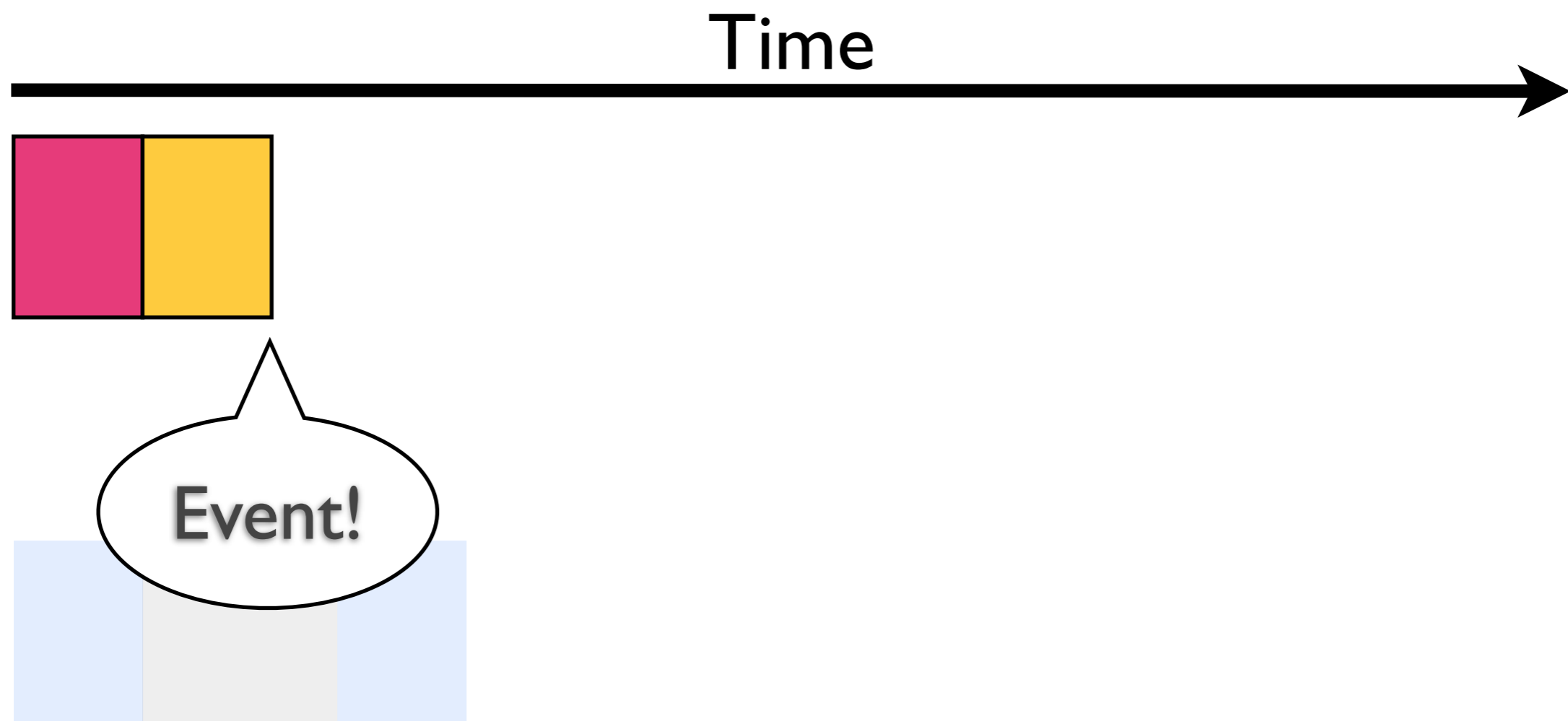


Callbacks:

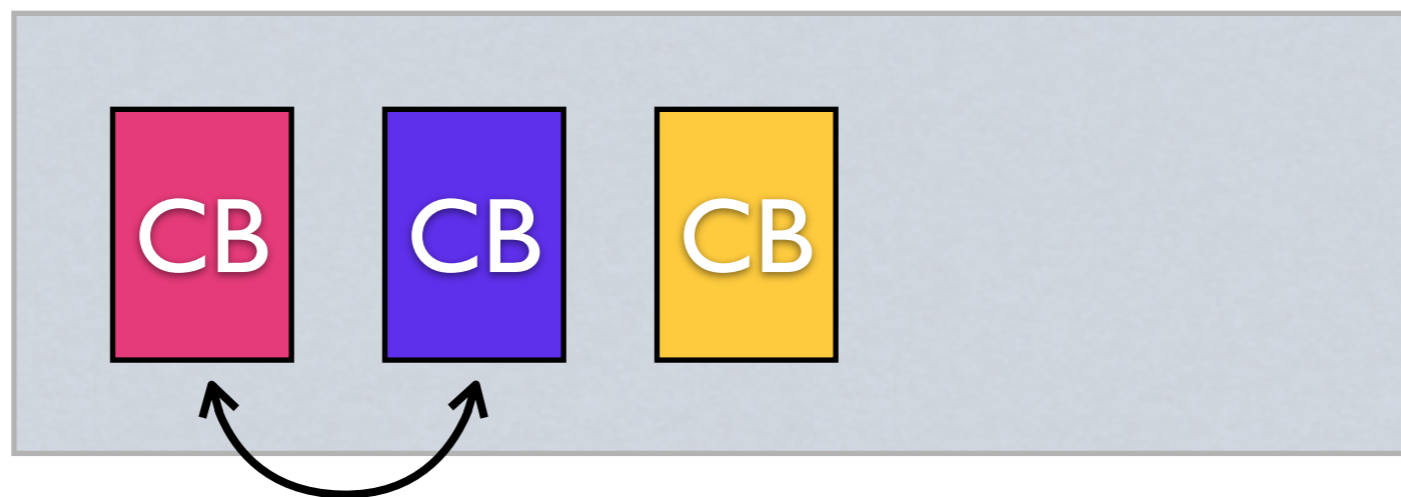


Saturday, January 26, 13

Oh hey! The response from the first request came in just as we finished the second request.



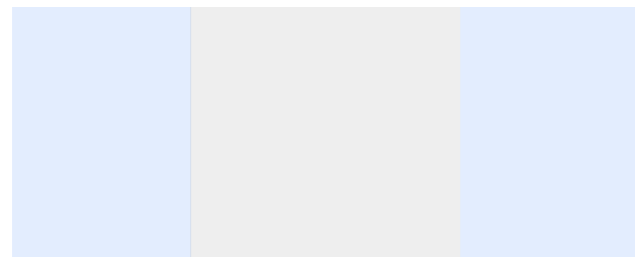
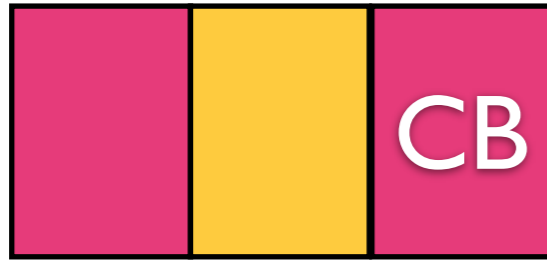
Callbacks:



Saturday, January 26, 13

This event triggers the first request's callback. Since nothing else is running right now (the second request has finished being made), the first callback can run.

Time



Callbacks:



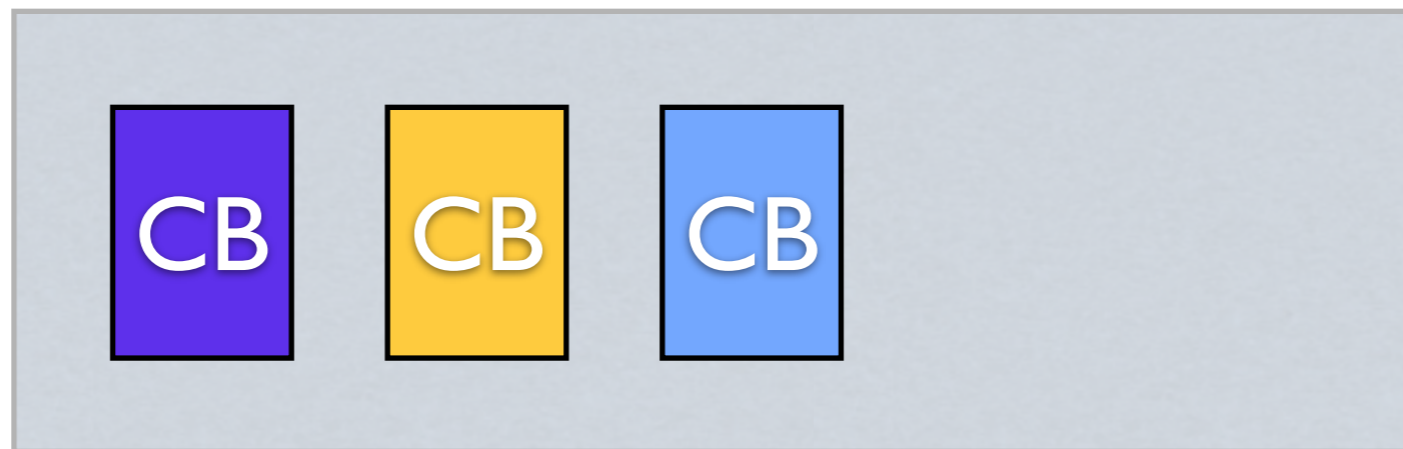
Saturday, January 26, 13

Ok, now, the first callback has finished running. What we originally going to do next? Oh right, make the third request. Let's do that.

Time



Callbacks:



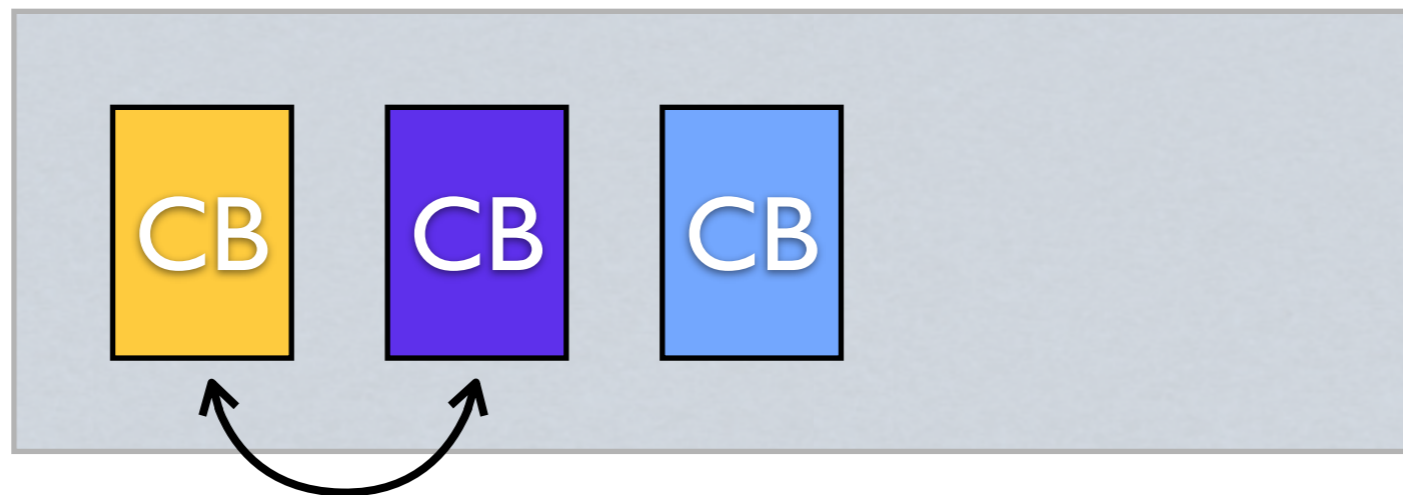
Saturday, January 26, 13

While we were making the third request, the response from the second request came in. The event loop, being a polite algorithm, will not rudely interrupt the third request while it is running...

Time

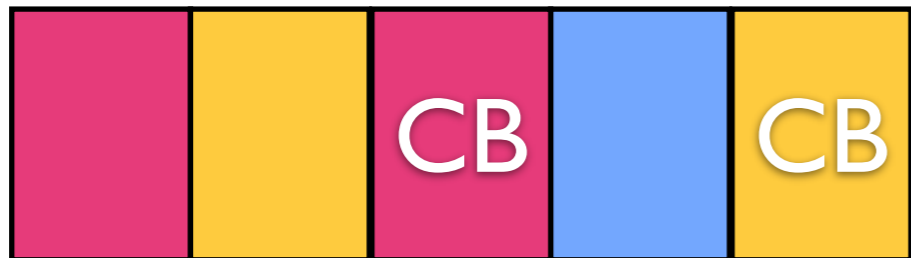


Callbacks:

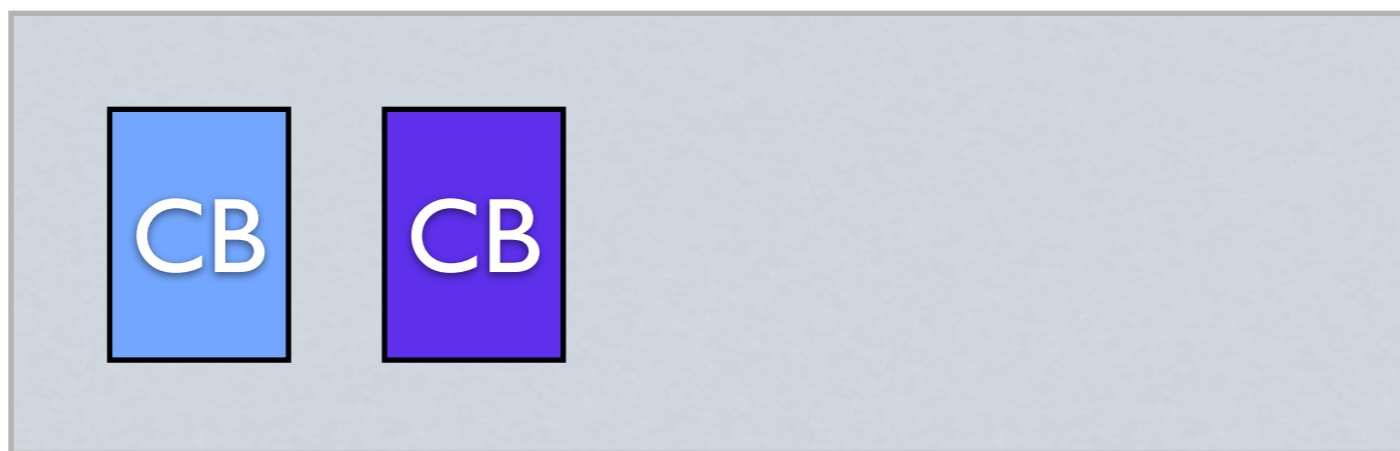


Saturday, January 26, 13

So the second request's callback is queued to run next after the third request yields.



Callbacks:

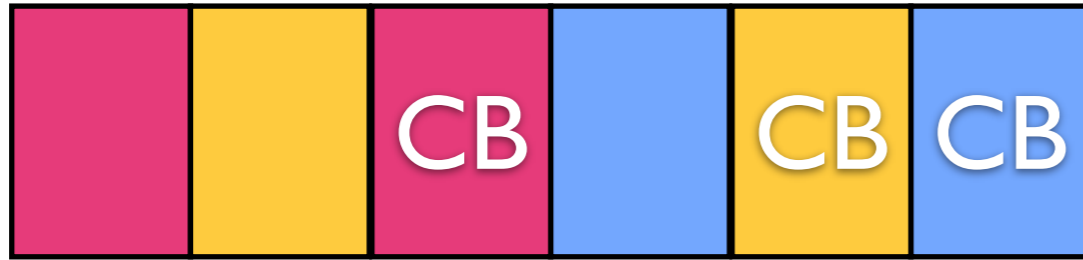


Saturday, January 26, 13

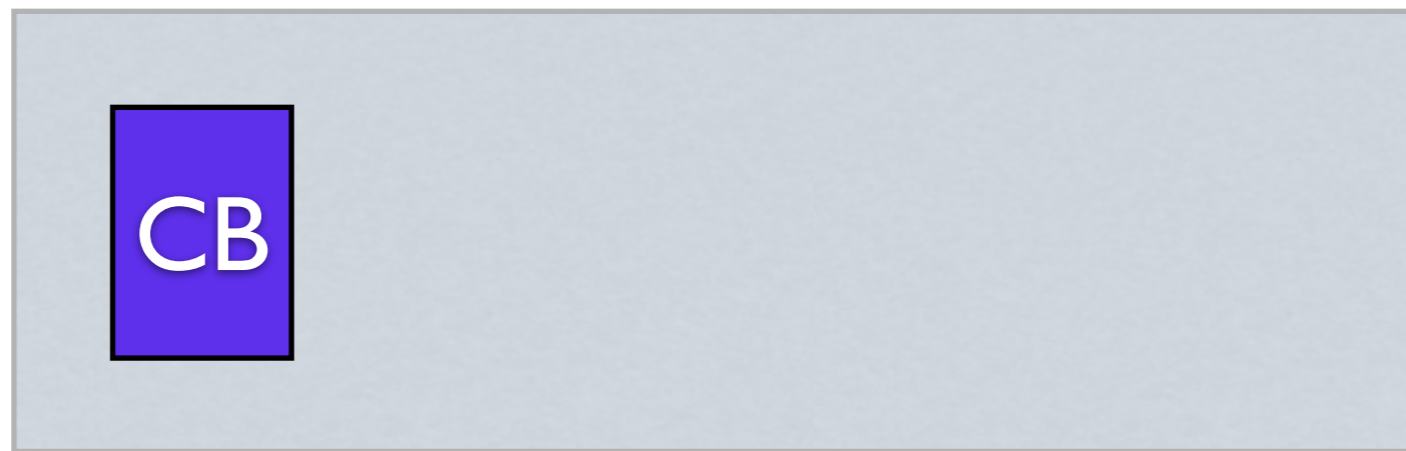
While we are finishing processing the second request's response, the third response comes in.



Time

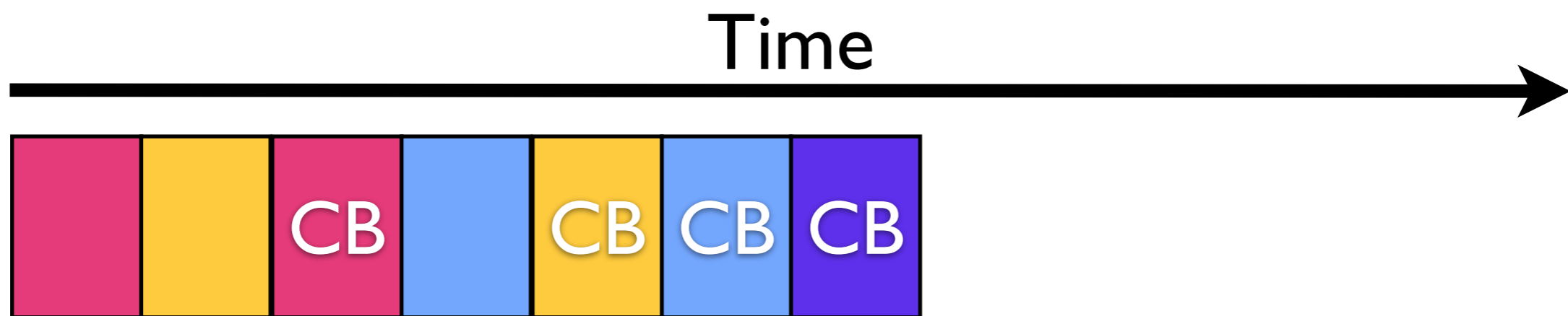


Callbacks:

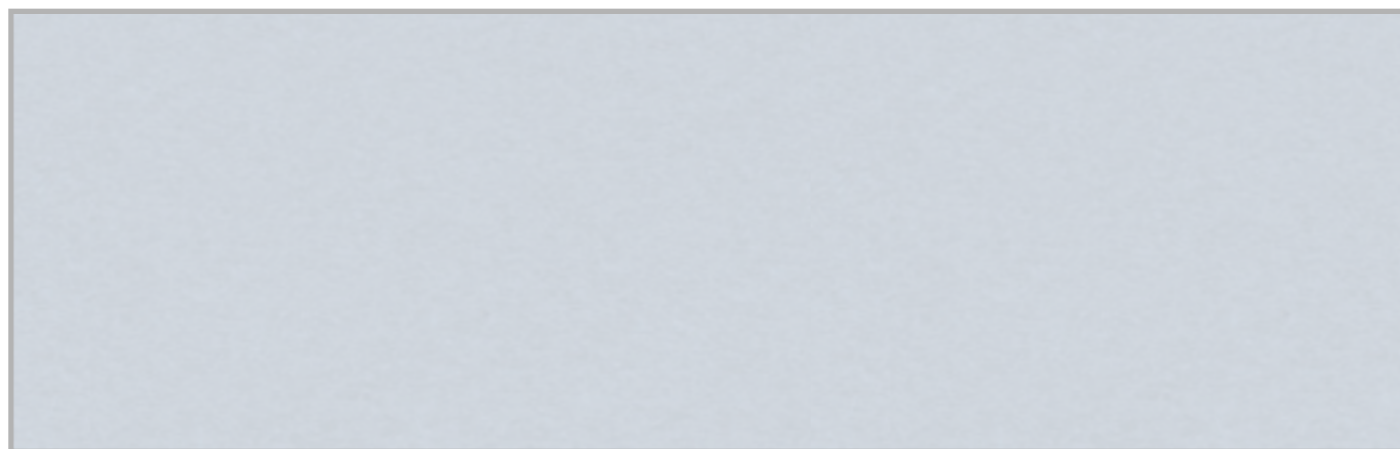


Saturday, January 26, 13

We then run the 3rd request's callback once the 2nd's is done.

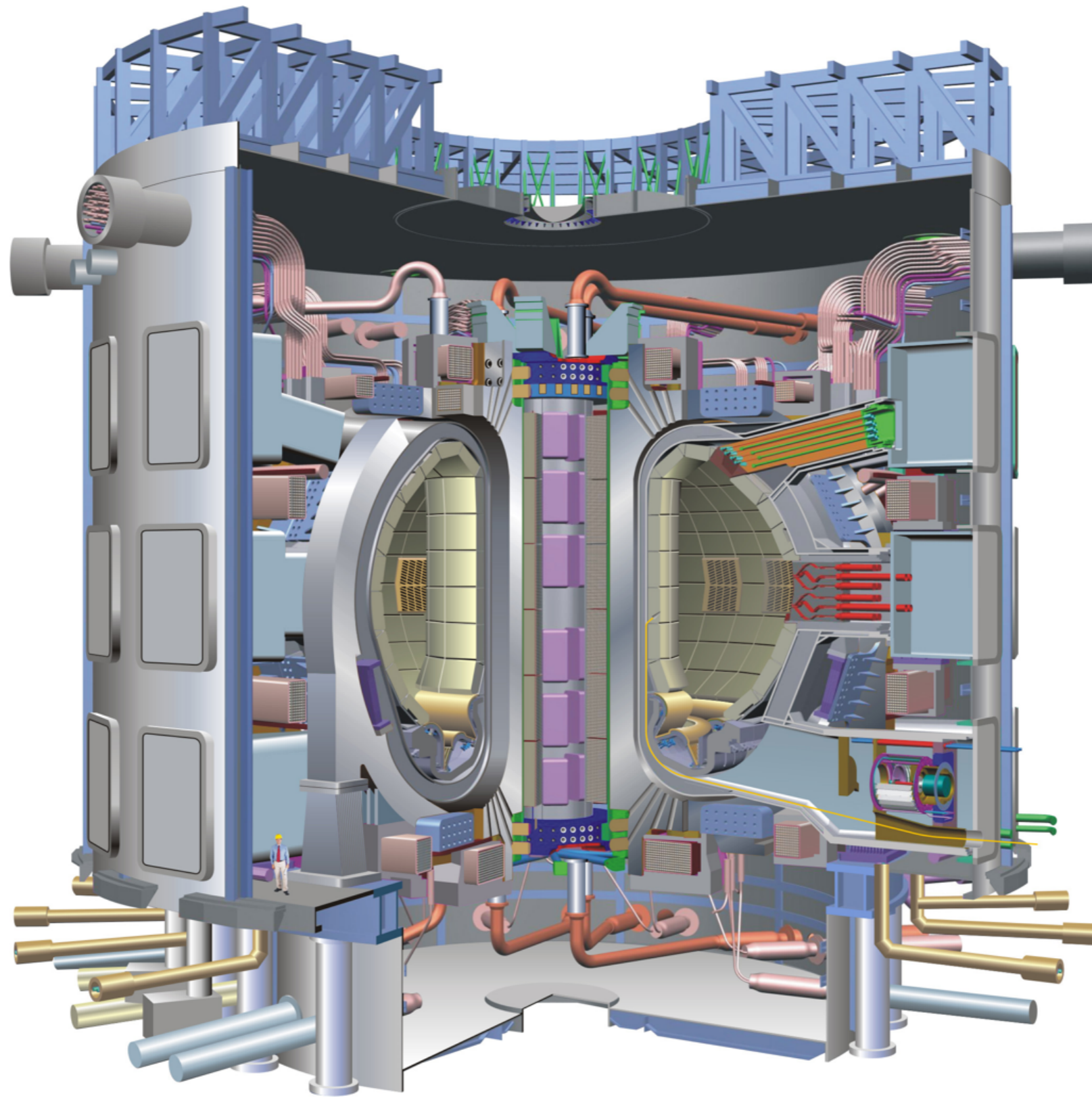


Callbacks:



Saturday, January 26, 13

Now that we have finished processing all 3 responses, the final interpolation callback can be run. Notice the lack of waiting for I/O. So this obviously a contrived example (and diagrams), but in an application with enough data and events to keep the event loop busy, the event loop will probably always have some code to run.



Saturday, January 26, 13

Twisted's event loop is the reactor, so called because it reacts to things.

**Events** → **Handlers**

Saturday, January 26, 13

It demultiplexes events and dispatches them to their respective callback handlers.

# Events → Handlers

- file descriptors
- timed events

Saturday, January 26, 13

One kind of event is any time a file descriptor, which has been registered with the reactor, is ready for I/O (for instance, has data ready to be read). Another type of event is a timed event – for example, if five seconds have passed since we told the reactor “run this callback in 5 seconds”.

- select
- poll
- epoll
- kqueue
- CoreFoundation
- IOCP
- ...more

Saturday, January 26, 13

The reactor supports a number of underlying multiplexing APIs, which tell it when file descriptors are ready for I/O. It abstracts away all these APIs and provides a common interface and errors.

# Deferreds

Saturday, January 26, 13

Twisted also provides an abstraction, called a Deferred, to keep track of callbacks and control the asynchronous processing of information.

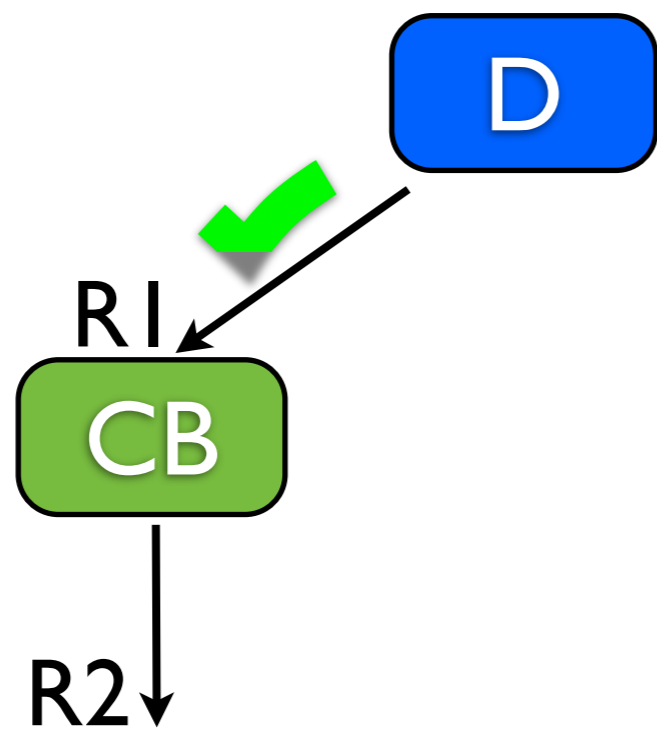
# Deferreds

(promises of future results)

Saturday, January 26, 13

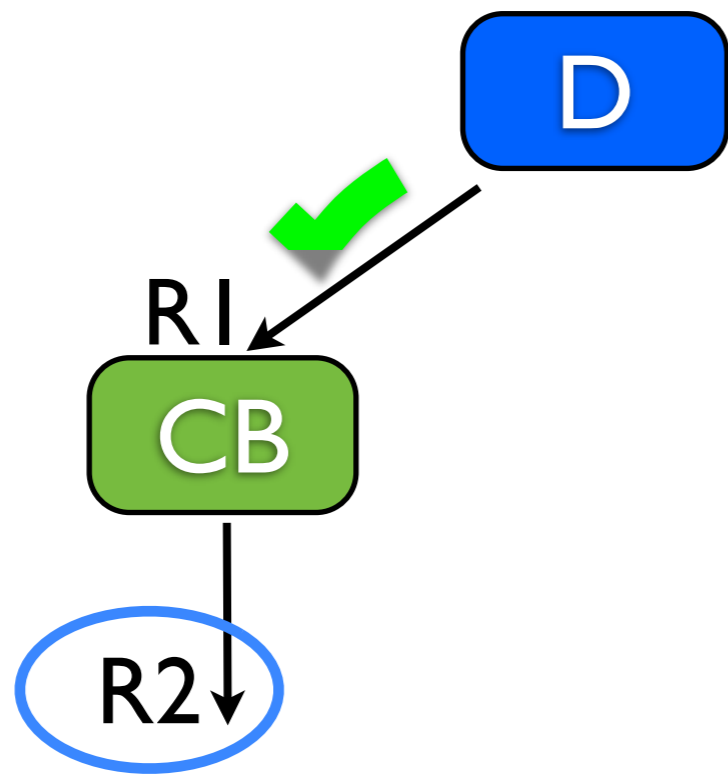
A function that returns a Deferred is returning a promise of some kind of result at some point in the future. I say “some kind of result” because that result could be an failure.





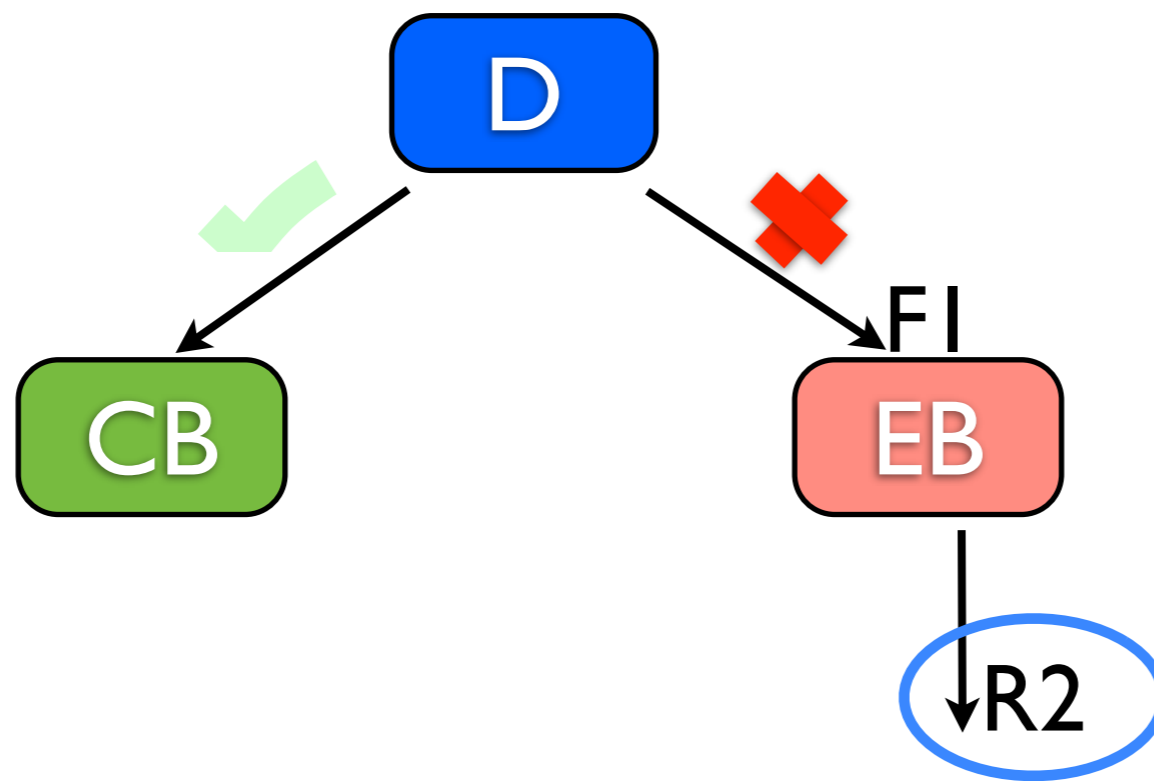
Saturday, January 26, 13

You can register a callback to a Deferred to handle a successful result. When the Deferred succeeds, this callback would take a result, R1, and return a new result, R2.



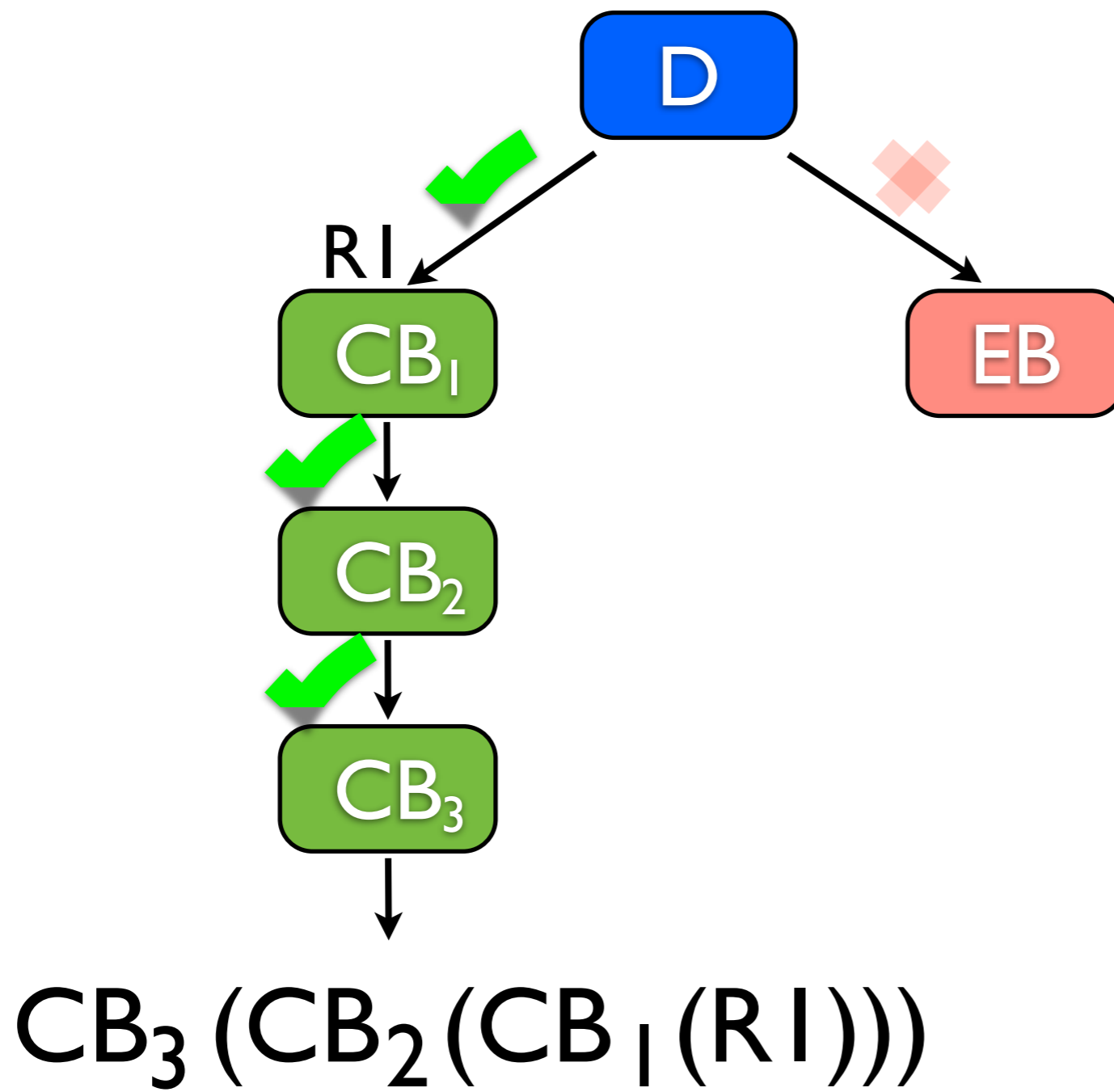
Saturday, January 26, 13

And now the promised future result of the Deferred will be R2 instead of R1.



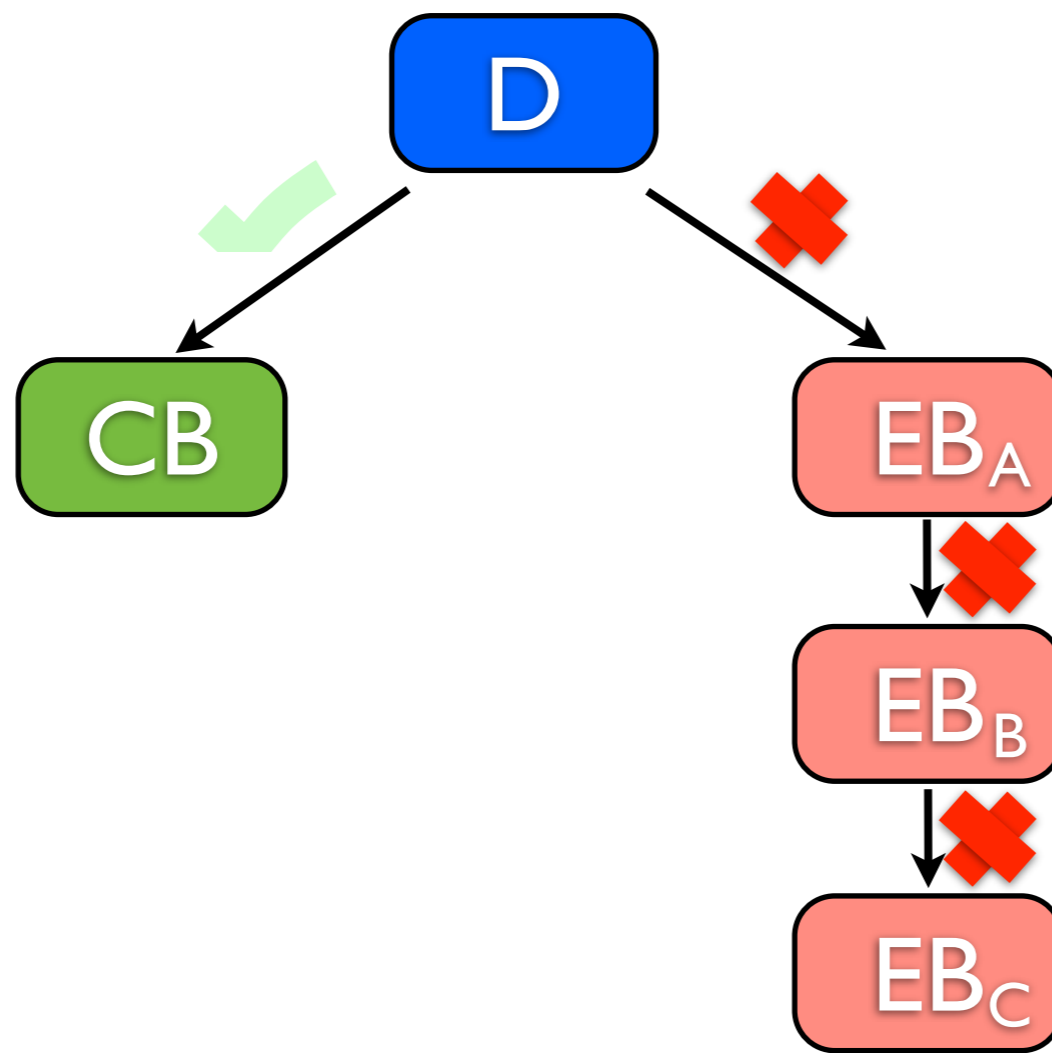
Saturday, January 26, 13

You can also register an errback to handle a failure result. When the Deferred fails, this errback would take a failure, F1, and return a result, R2, which could either be some value or a failure. The promised future value of the Deferred would now be the R2 returned by the errback.



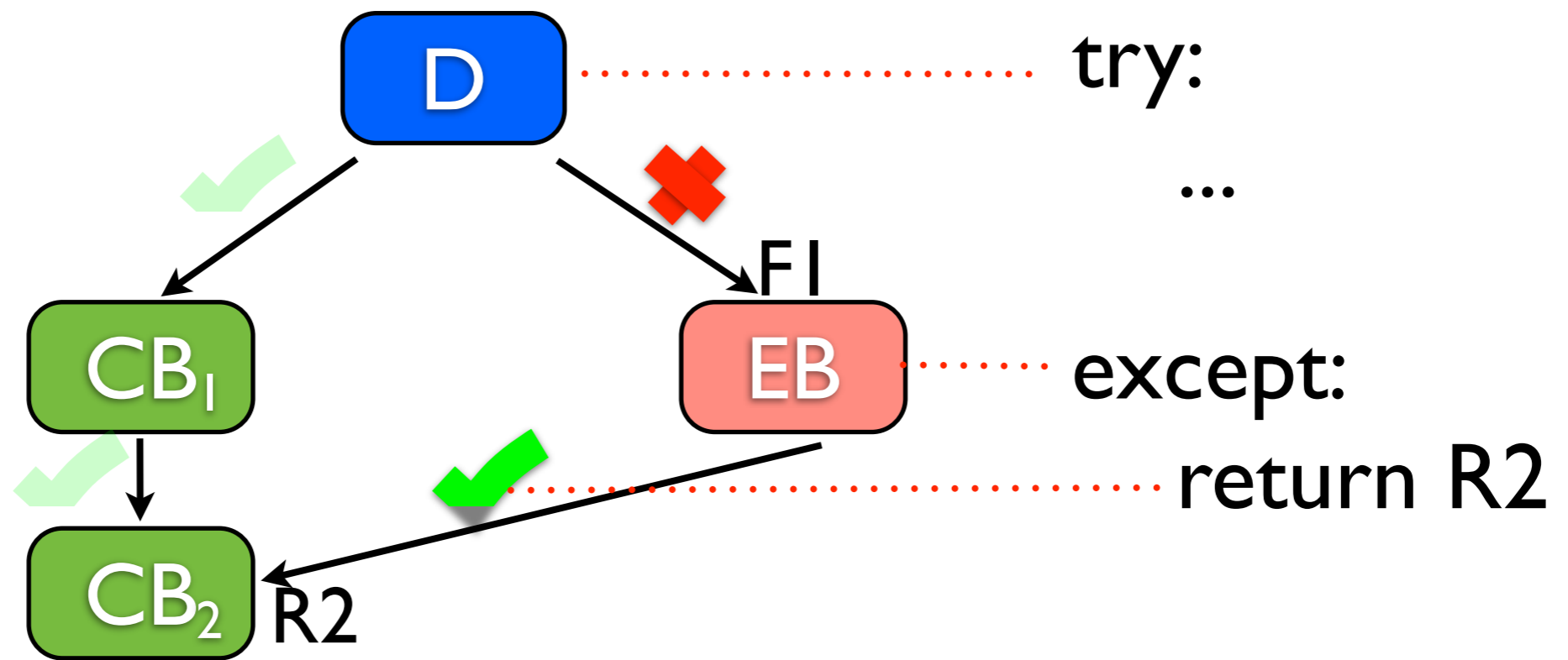
Saturday, January 26, 13

Callbacks can be chained one after another. This is the equivalent of composing all the callbacks – applying one function to the results of another. Well, only if the Deferred succeeds, and each successive callback succeeds.



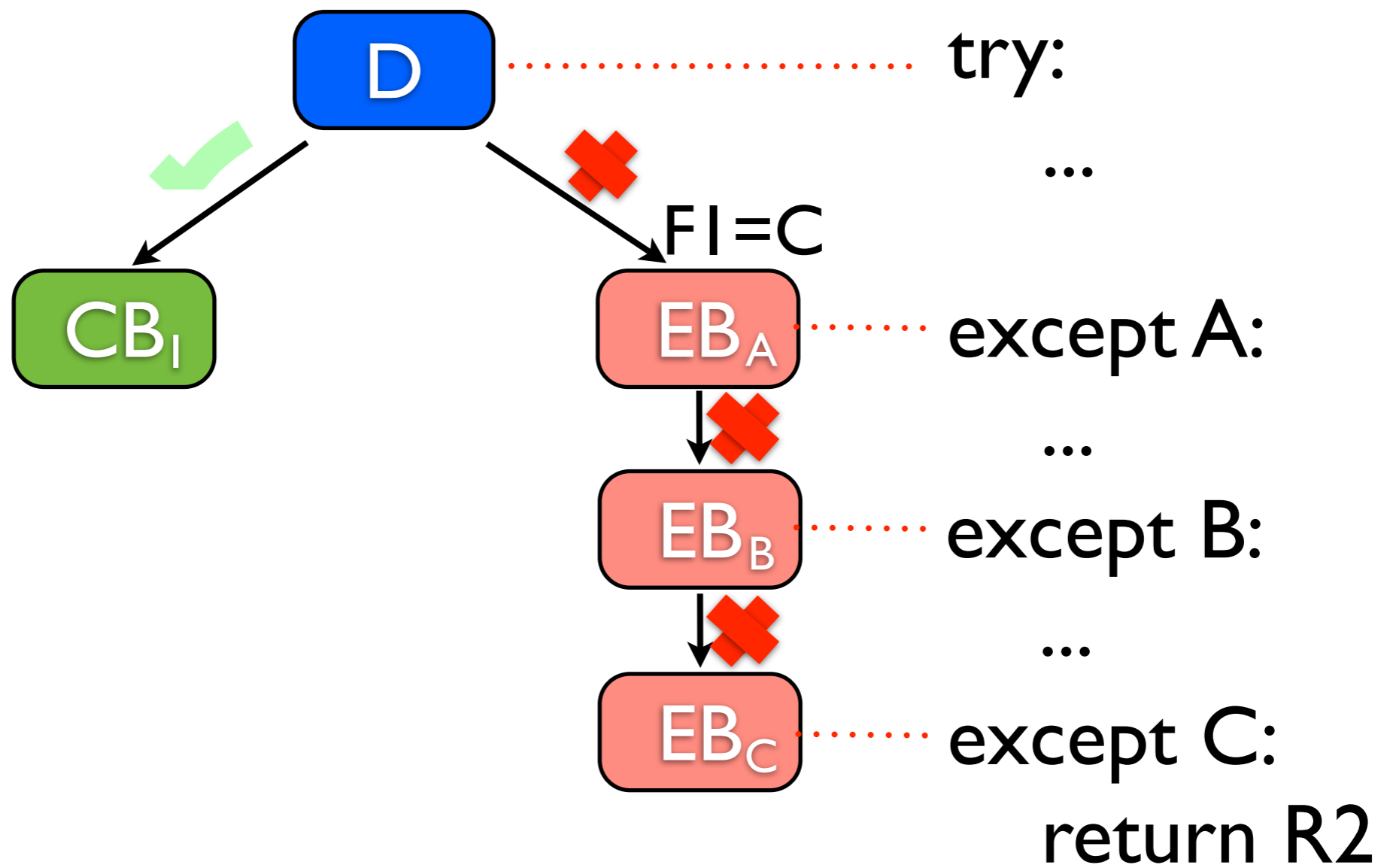
Saturday, January 26, 13

Errbacks can also be chained. Remember when I said previously that an errback can return either some value or a failure?



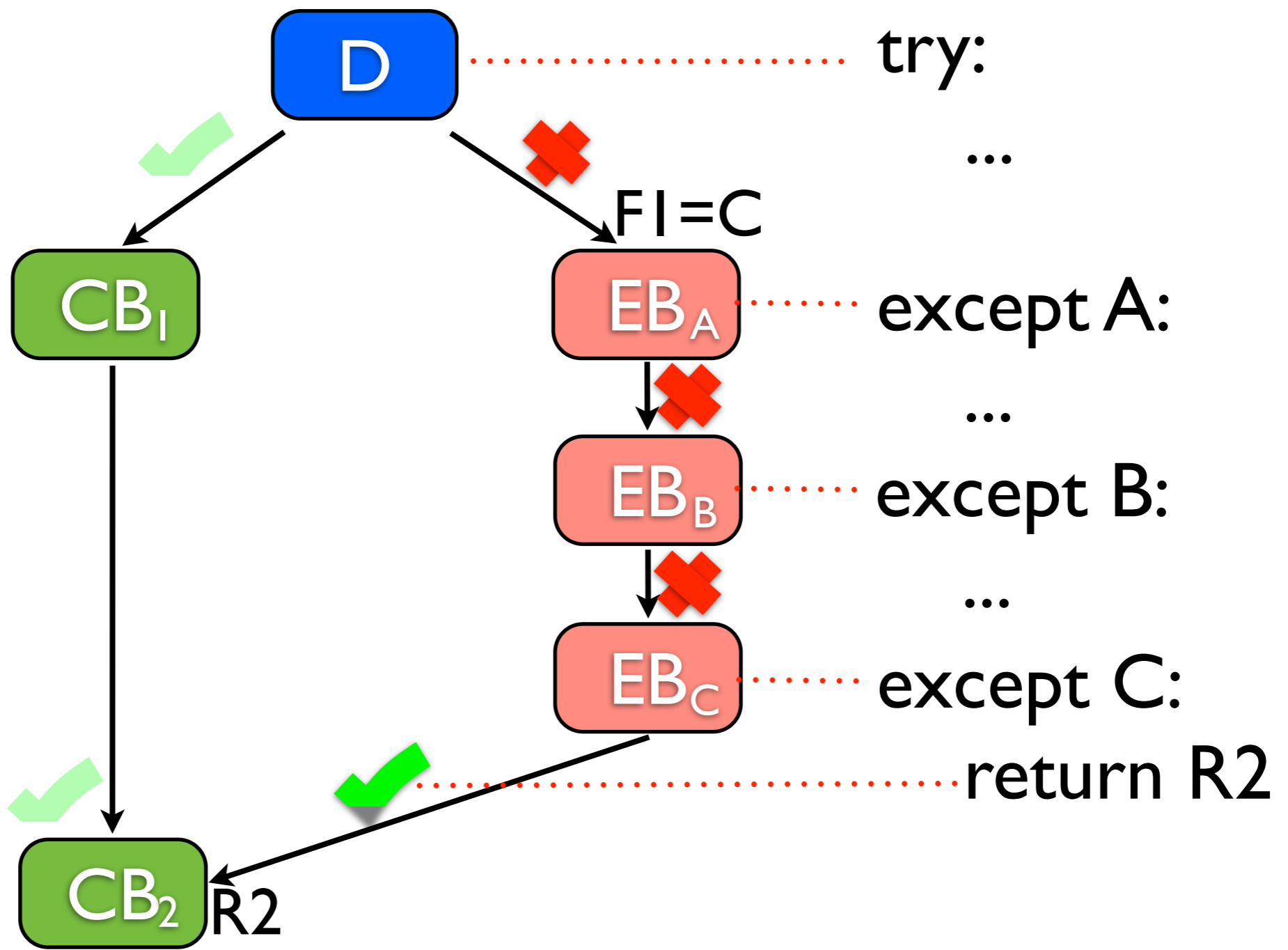
Saturday, January 26, 13

An errback behaves like the “except” in a try/except block – it can just handle the failure and allow the code to continue executing as normal (by returning a value, or nothing). If it does then control, and the result, is then passed to the next callback in the callback chain.

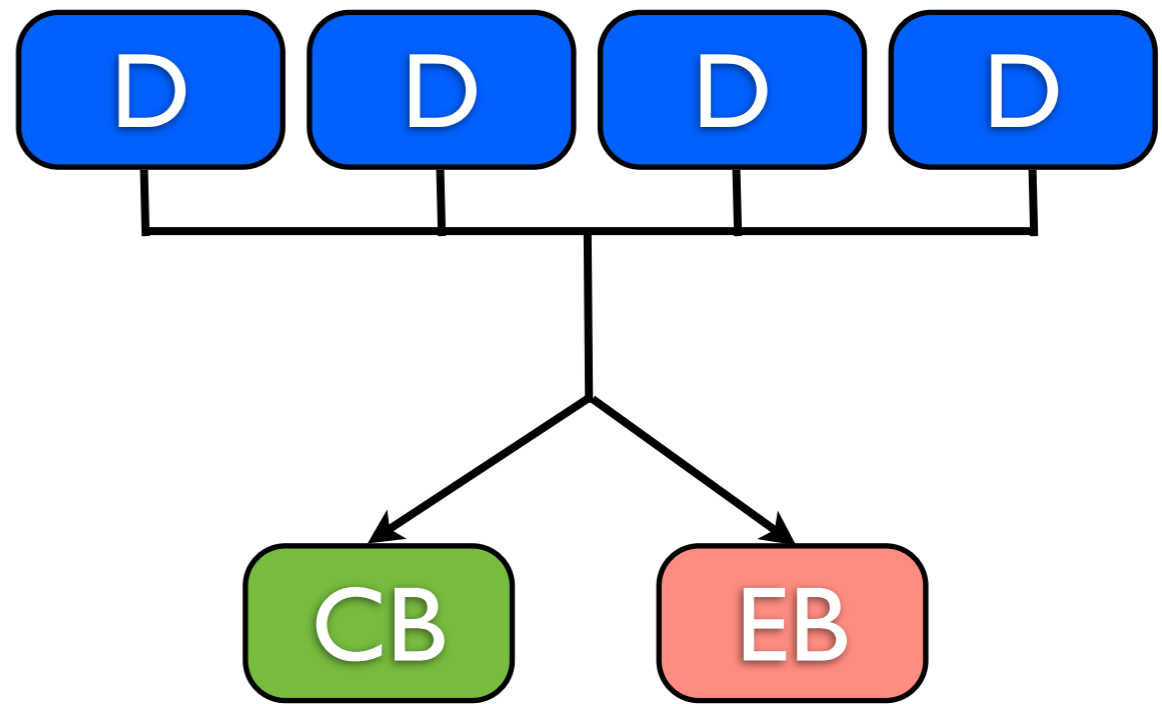
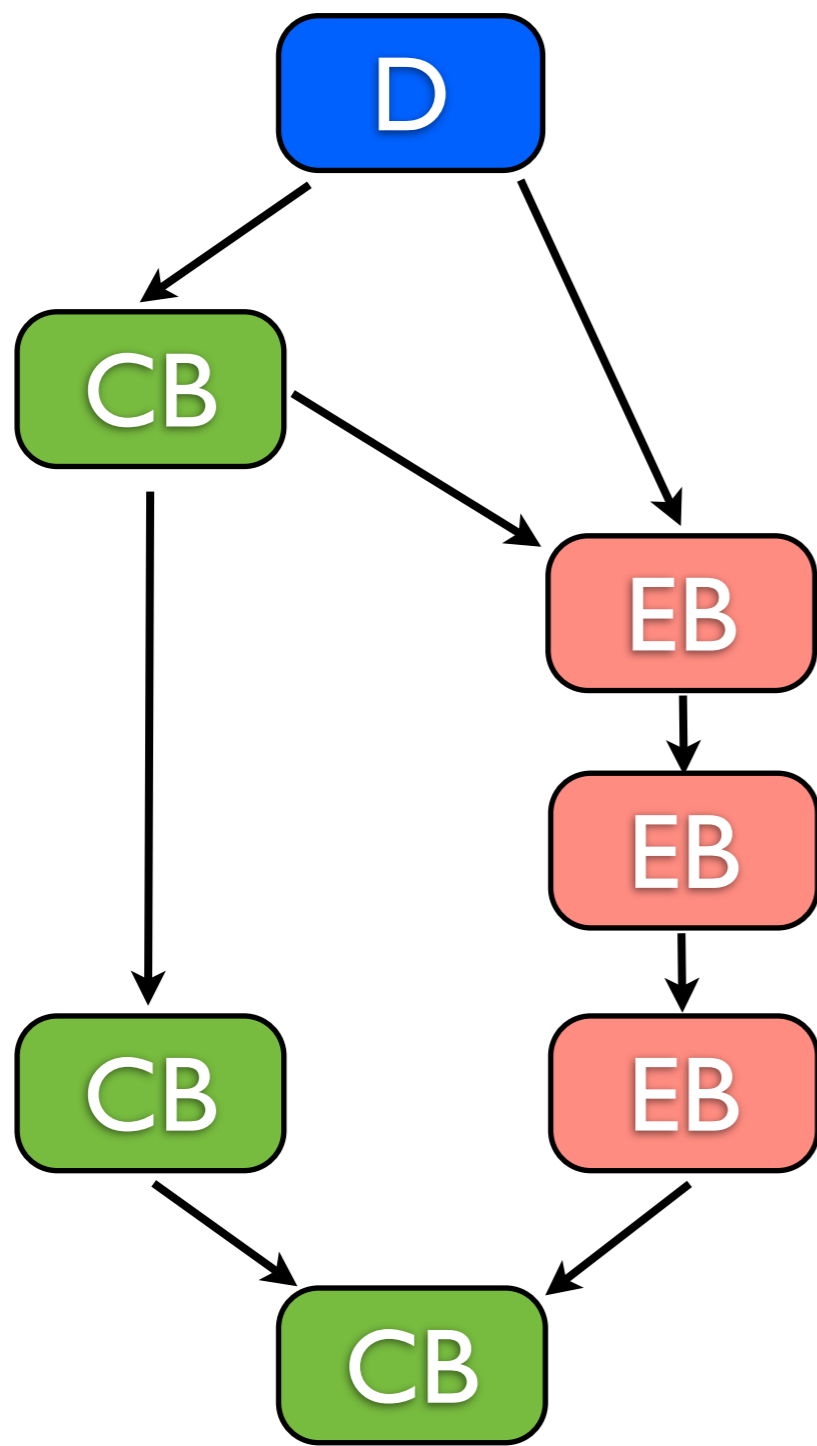


Saturday, January 26, 13

But it can also only except one particular type of failure. For instance, the first errback may only handle failures of type A, and so if the failure F1 is of type C, it would get propagated down to the third errback, which can handle it.







Saturday, January 26, 13

Deferreds and other Deferred-derivative abstractions allow you to build complex process chains that would be more difficult to create with other types of callback abstractions. And a Deferred keeps track of all of its callbacks and errbacks, so the only the Deferred need be passed from function to function.

# courtesy of <http://callbackhell.com>

```
    fs.readdir(source, function(err, files) {
if (err) {
    console.log('Error finding files: ' + err)
} else {
    files.forEach(function(filename, fileIndex) {
        console.log(filename)
        gm(source + filename).size(function(err, values) {
            if (err) {
                console.log('Error identifying file size: ' + err)
            } else {
                console.log(filename + ' : ' + values)
                aspect = (values.width / values.height)
                widths.forEach(function(width, widthIndex) {
                    height = Math.round(width / aspect)
                    console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                    this.resize(width, height).write(destination + 'w' + width + '_' + filename, function(err) {
                        if (err) console.log('Error writing file: ' + err)
                    })
                }).bind(this)
            }
        })
    })
}
})
})
})
```

Saturday, January 26, 13

Without any callback abstractions (or good discipline), you can easily end up with code that looks like this. This is not to say that Javascript does not have any flow control abstractions. For instance, `async.waterfall`, or...

- jQuery Deferreds
- MochiKit Deferreds
- Dojo Deferreds
- Google Closure Deferreds

Saturday, January 26, 13

Deferreds! This abstraction has been borrowed by other event-driven libraries such as jQuery and MochiKit. Dojo and Google Closure later adapted MochiKit's Deferreds. So Deferreds are now also a popular Javascript flow control abstraction.

# Twisted

- easier to reason about concurrency
- lots of components
- useful abstractions

Saturday, January 26, 13

In conclusion, being event-driven means that concurrency Twisted is easier to reason about. And Twisted provides a lot of pre-built components that let you build anywhere from a canned webserver to a custom, complex networking application. And its abstractions, including its callback abstraction, means that your code can be modular, neat, and relatively easy to maintain.