MMO Game Servers in Twisted Python

Dan Maas / Battlehouse.com

Swiss Python Summit 2017

5-year-old company with a small engineering team

We make games with Python

BATTLEHOUSE™

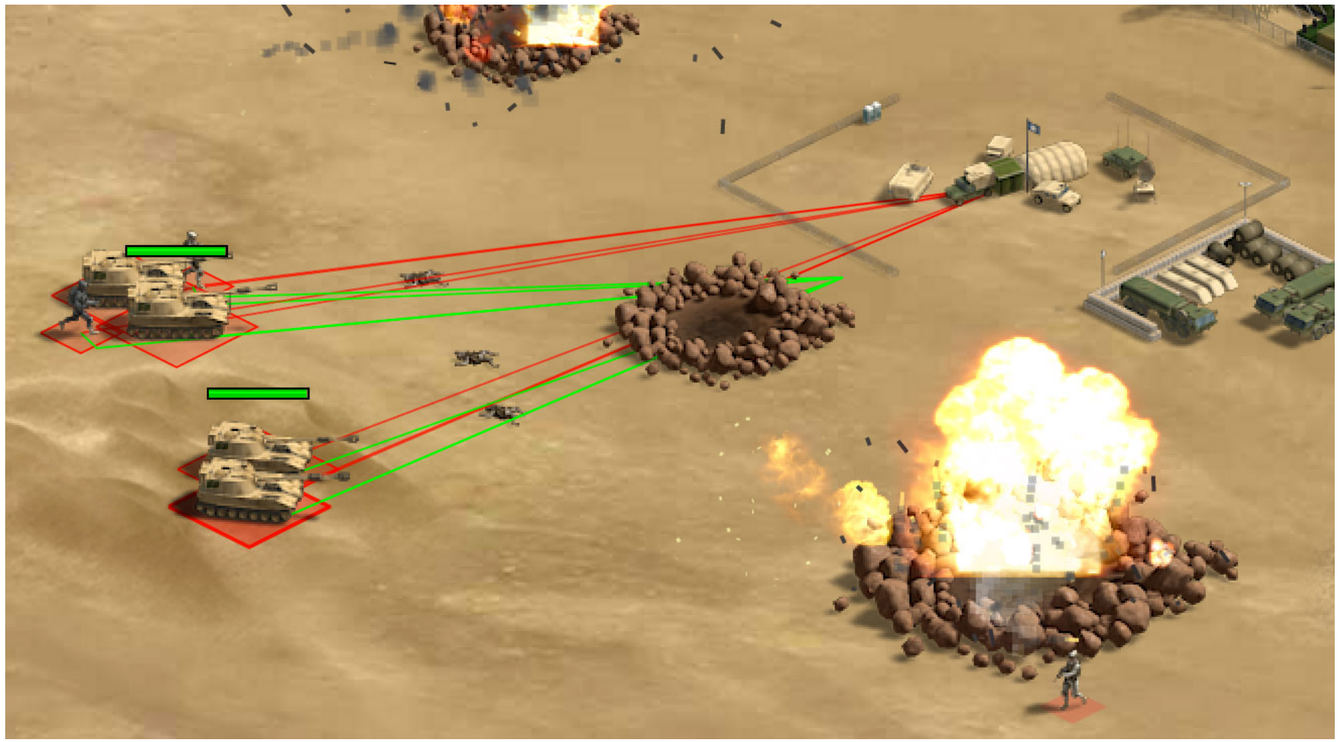
One engine, 7 game titles, 4 platforms

Thunder Run trailer

- 4,000,000+ fans
- 20,000+ daily players
- 1,000+ concurrent players

We make "Builder RTS" games

RTS combat engine



We make "Builder RTS" games

Base management / upgrade system

We make "Builder RTS" games

Available on Facebook and Battlehouse.com

Today's Topics

1. System architecture
2. How to write an asynchronous server
    - with Twisted Python
3. Tips for creating "production quality" services

System Architecture

Game = Engine + Game Data + Art

Game = Engine + Game Data + Art

- Engine: Server, Client, Analytics
- Game Data: Units, buildings, items
- Art: Images, sounds

Engine

- Client / Server "web app"
- Server: Python
  - with Twisted as main networking library
- Client: JavaScript/HTML5 Canvas
  - with Google Closure Compiler

Also…

- Analytics system
  - MongoDB->ETL->SQL & map/reduce
- Gamedata build pipeline
  - Makefile-driven Python scripts
- Art build pipeline

The Server

- Client sends requests (by HTTP or WebSocket) to run game actions
  - “Upgrade this building"
  - "Produce this unit"
  - "Buy this thing in the Store"
- Check requirements; if OK, then mutate player state; send reply

Server Design Requirements

- High scale
  - 20,000+ daily players, 2,000+ concurrent players
- **Low latency**
  - ~200ms hard limit to all requests

Server Implementation

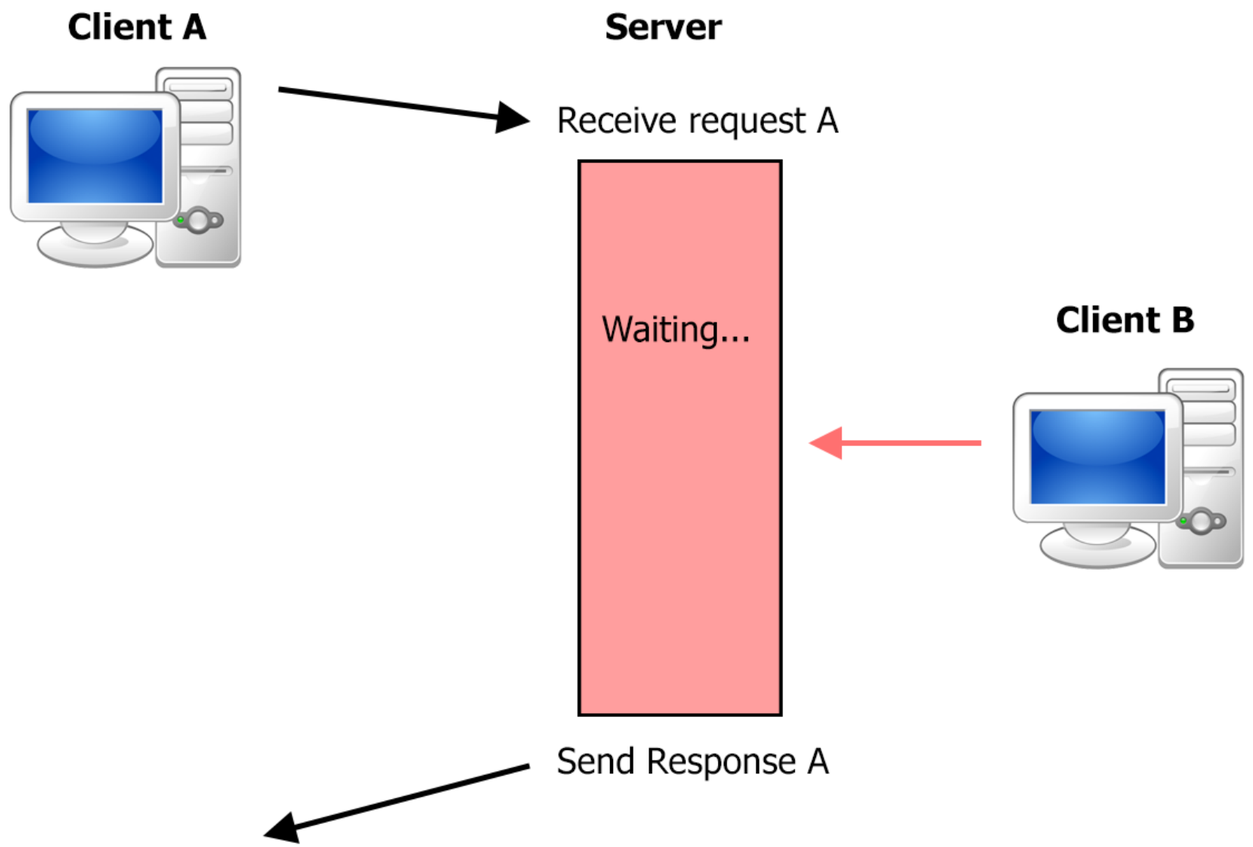
- Python
- Twisted Asynchronous HTTP server
- Cluster of processes (on Amazon EC2)
- Support ~100 online players per CPU core
    - Scale by adding more cores

What is Twisted?

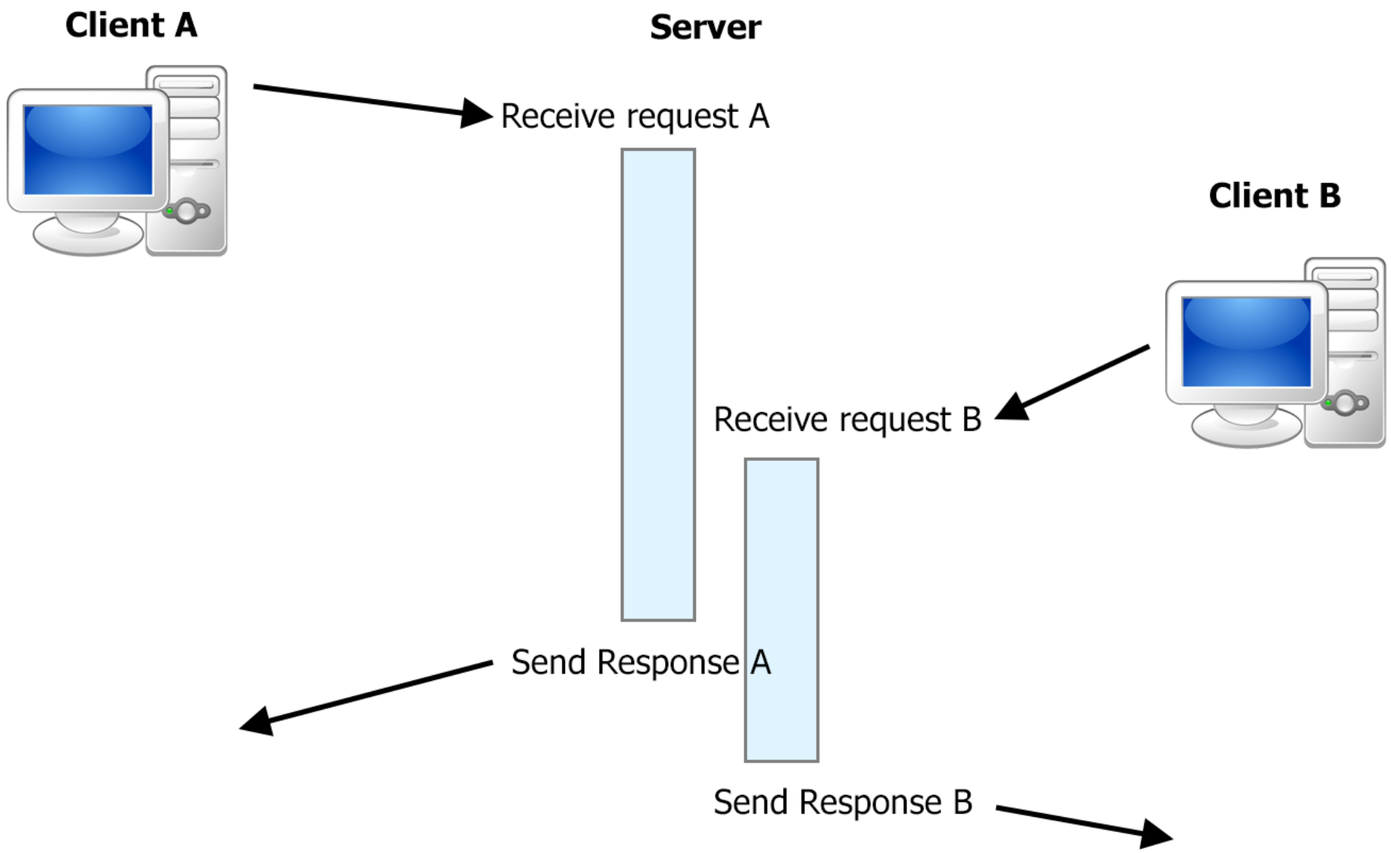
- Networking library
- Asynchronous event loop
- Supports *many* internet protocols
  - HTTP, SSH, FTP, SMTP, ...
- Consistent Python API
- Good OO design, easy to extend and customize

What is an asynchronous server?

Synchronous server



Asynchronous server



We use both synchronous and asynchronous code

"Fast" requests are synchronous

1ms - 100ms:

- Change state in memory, or make fast database query
- Examples: Add damage, Buy item

"Slow" requests must be asynchronous

100ms - 10 seconds:

- Reading/writing Amazon S3 on login/logout
- Querying Facebook API
- Slow database queries, like fetching sorted Top Scores

How to do this in Python?

Use Twisted!

First: a *synchronous* HTTP server with Twisted

```
from twisted.web import server, resource
from twisted.internet import reactor

class MyResource(resource.Resource):
    def render(self, request):
        name = request.args['name'][0]
        return "Hello %s!\n" % name

reactor.listenTCP(8080, server.Site(MyResource()))
reactor.run()
```

```
$ curl 'http://localhost:8080/?name=Dan'

"Hello Dan!"
```

Now, add a slow operation

```
from twisted.web import server, resource
from twisted.internet import reactor

class MyResource(resource.Resource):
    def render(self, request):
        name = request.args['name'][0]
        # LONG DATABASE QUERY here
        return "Hello %s!\n" % name

reactor.listenTCP(8080, server.Site(MyResource()))
reactor.run()
```

Let's make this *asynchronous*

```
from twisted.web import server, resource
from twisted.internet import reactor

class MyResource(resource.Resource):
    def render(self, request):
        name = request.args['name'][0]
        # ... start working ...
        return twisted.web.NOT_DONE_YET

# later, when work is done...
request.write("Hello")
request.finish()
```

General Pattern

1. "Before" code
    - Parse and validate parameters
2. Slow asynchronous operation
3. "After" code
    - Return result to client

How to connect the "before" and "after" parts of the code?

How to "glue" together asynchronous code?

- Key problem in writing low-latency servers
- Major evolution in the last few years

Complications

- Multi-step operations
- Error handling

Many options have been tried

1. OS threads

2. Explicit callbacks

3. Promises/futures

(Twisted uses a mixture of 2. and 3.)

Explicit callbacks

```
doThis.then(doThat(), onError=handleIt())
```

- Became popular due to Node/Javascript
- Easy to see what the code is doing
- But, gets ugly in complex cases
  - Chaining
  - Error handling

Promises/Futures

```
result = yield from ...
```

- Clean and clear syntax
- Easy to chain and handle errors
- Python (and many other languages) converging in this direction

Twisted's hybrid approach: `Deferred`

- Create a `Deferred` object to represent (the result of) an asynchronous operation
- Attach callbacks and error handlers
- "Fire" the `Deferred` when the result is ready

```
def render(self, request):
    name = request.args['name'][0]
    d = defer.Deferred()
    d.addBoth(self.complete_deferred_request, request)
    my_async_function(d)
    return twisted.web.NOT_DONE_YET

def complete_deferred_request(self, body, request):
    if body == twisted.web.server.NOT_DONE_YET:
        return body # still asynchronous
    request.write(body)
    request.finish()

def my_async_function(d):
    result = # ... do something
    d.callback(result)
```

Control flow becomes a mess

Don't write your code this way

@inlineCallbacks

Improvement!

```
@inlineCallbacks
def render(self, request):
    result = yield my_async_function()
    returnValue(result)
```

- Magically turns the part of the function after "yield" into a

callback

  - Clever use of Python generators

Chaining and Error Handling

```
@inlineCallbacks
def render(self, request):
    try:
        result_1 = yield my_async_function_1()
        result_2 = yield my_async_function_2(result_1)
        returnValue(result_2)
    except:
        log_exception()
        returnValue('error')
```

The Future: async/await

```
async def render(self, request):
    result_1 = await my_async_function_1()
    result_2 = await my_async_function_2(result_1)
    return result_2
```

- Python 3.5+ only
- Language itself now handles asynchronous code

Review

- Asynchrony is vital to reduce latency
- Asynchronous code becomes a mess if you are not careful
- Twisted and Python help clarify control flow and

error handling

Hints and Tips

Hints and Tips

- Things your production code will need:

  1. Tracking of in-flight requests

  2. Latency profiling

Monitor in-flight requests

- How will you know if async requests are piling up?
- Or failing en masse?
- Asynchronous frameworks are bad at telling you this

Montior in-flight requests

- Create your own "bookkeeper"

AsyncHTTP_Facebook

dropped	0
attempted	61988
ok	61292
errors	346
retries	350
num_on_wire	0
num_in_queue	0
num_waiting_for_retry	0

- Also: re-try failed requests, when appropriate

Hints and Tips

- Handle failure/cancel paths
  - What if user logs out during execution of a slow database query?
- Sometimes more complex than the main path (e.g. mutex issues)

Hints and Tips

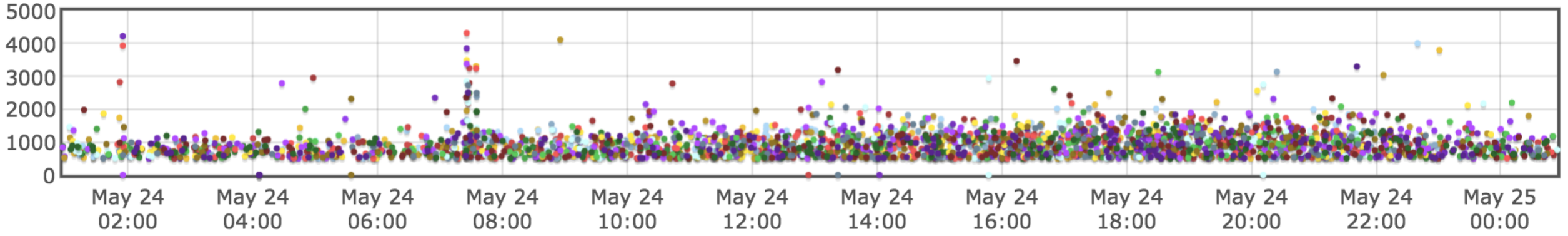
- Asynchronous code is hard to write! Don't panic :)

Latency Profiling

Latency Profiling

- Important metric is not CPU usage, but **request latency**
- Especially **maximum** request latency
- Traditional CPU profiling does not help here

Server Latency



Wrap every "entry point" with time measurements

```
def render_wrapper(self, request):
    start_time = time.time()
    result = self.do_render(request)
    end_time = time.time()
    unhalted_time = end_time - start_time
    record_latency(request, unhalted_time)
```

(conceptually - this can be cleaned up with decorators)

Collect data on each request:

- Average latency (performance hotspot)
- Maximum latency (latency hotspot)
  - Advanced version: track 95th and 99th percentile latencies

Server Latency

Approximate unhalted load: **4.3%**
Average request latency: **0.5 ms**

Sort by Max

Request	Average	Max	Total	Total %	#Calls
ALL	0.5 ms	2522.7 ms	9432.8 s	68.3%	19167367
CONTROLAPI(HTTP:reconfig)	2522.2 ms	2522.2 ms	2.5 s	0.0%	1
handle_client_hello	2.4 ms	934.4 ms	23.7 s	0.2%	9820
PLAYER_STATE_QUERY	0.1 ms	929.3 ms	2.1 s	0.0%	30645
PING_OBJECT	2.2 ms	808.2 ms	139.1 s	1.0%	62164
complete_client_hello	155.7 ms	762.0 ms	1510.9 s	10.9%	9701
VISIT_BASE2	15.1 ms	690.7 ms	89.7 s	0.6%	5921
complete_attack	15.6 ms	683.1 ms	718.2 s	5.2%	46081
QUARRY_COLLECT	43.6 ms	676.8 ms	555.1 s	4.0%	12736
player_table:deserialize	18.7 ms	530.9 ms	353.6 s	2.6%	18888

Watch total “unhalted” time

- What % of the time the CPU is waiting for the next request?
- Approaching 50% = danger!

Watch total “unhalted” time

- What % of the time the CPU is waiting for the next request?
- Approaching 50% = danger!

Have fun with Twisted!

Q&A

Bonus topic: Adding WebSockets

- Community patch to Twisted
  - http://twistedmatrix.com/trac/ticket/4173
- WebSocket messages call the same handlers as HTTP requests

– Hack: create a fake HTTP request for each WebSocket message

```
class WSFakeRequest(object):
    def __init__(self, proto):
        self.proto = proto
    def write(self, buf):
        if self.proto.connected:
            self.proto.transport.write(buf)

class MyWSProcotol(protocol.Protocol):
    def dataReceived(self, data):
        response = MyHTTPProcotol.render(WSFakeRequest(self))
        if response == twisted.web.server.NOT_DONE_YET: pass
        self.transport.write(response)
```

Websockets results

- Improved robustness vs. HTTP, but performance did not change
  - HTTP Keepalive is doing its job
- Beware protocol and browser bugs