DISTRIBUTED COMPUTING

Lecture 1 - Setting the Scene

Marco Massenzio
http://codetrips.com
http://linkedin.com/in/mmassenzio

ROADMAP

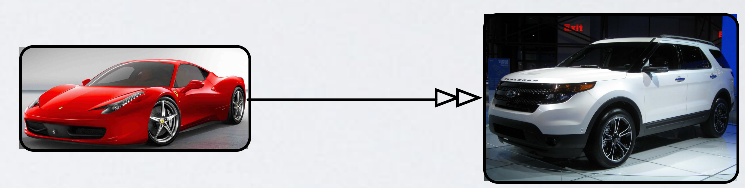
- Why should we care?
- Principles of Distributed Computing
- MapReduce - Hadoop & friends
- Zookeeper - distributed computing patterns
- NoSQL - storing & retrieving data at scale
- Availability & Scalability - you really have to pick one
- CAP theorem - reality sucks
- Spark - streaming data for "quasi-realtime" big data analysis

WHAT DO I NEED?

- Linux (Ubuntu) or Mac OSX laptop
- Oracle VirtualBox (free download)
- Python 3.4 (2.7 should work too)
- (some) Java (for MapReduce)
- (some) Familiarity with AWS (mostly EC2, S3, ELBs, Route53)
- Ability to download/install packages without much hand-holding
- (optional, recommended) Access to a good IDE (Eclipse or IDEA PyCharm/IntelliJ, free for students)
- (basic) Linux command-line (eg, ability to SSH into a remote instance, scp files across networks, mkdir, chown/chmod, very little else)
- Ability to use Google and StackOverflow

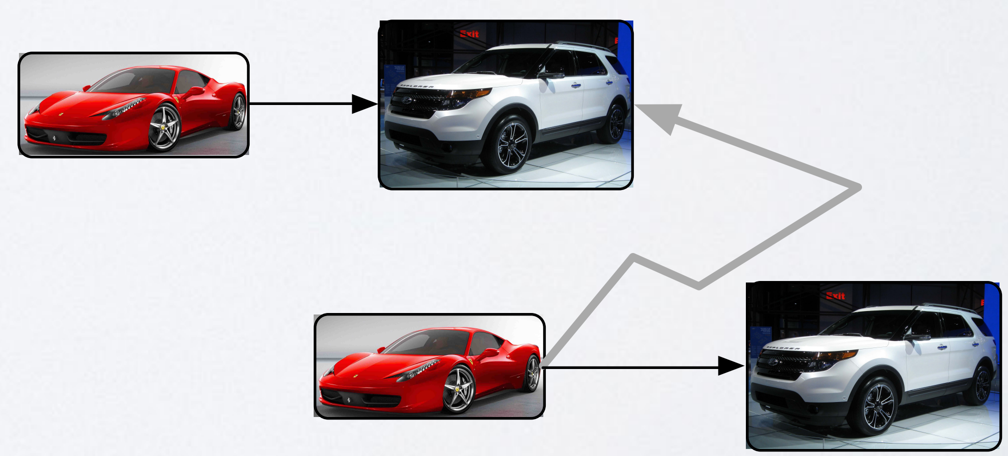
WHY DO WE NEED DISTRIBUTED COMPUTING?

- Scalability



- Availability

- Performance



For a more formal summary: http://www.slideshare.net/emcacademics/milind-hadoop-trainingbrazil

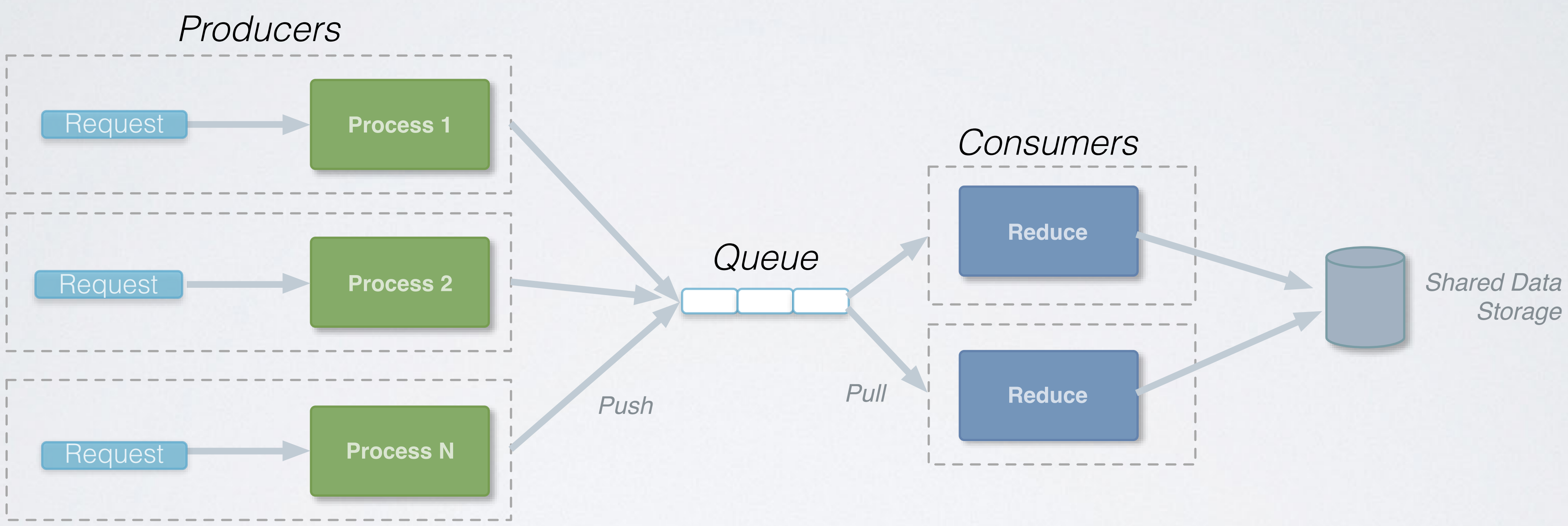
SCALABILITY

- Ability to serve a much higher rate of requests per seconds (typical metric: QPS) without significantly changing the architecture of the application
- Horizontal scalability: add more instances/processes of the same kind
- Vertical scalability: add more computing power and/or storage to each single instance
- They are not mutually incompatible; but require different architectural considerations (and usually are driven by very different requirements)
- **Beware**: adding more instances, one increases the probability that any one of them will fail (but increases the probability that at least one will not)

MULTI-CORE CPUS

- For "CPU-bound" processes, adding more cores may make sense (provided the code can take advantage of that!)
- For I/O-bound processes this does not really help, unless one can distribute the data too (disk access is currently the limiting factor)
- Increased complexity of multi-threading code has led to consider different approaches (event-driven asynchronous architectures; actor-based systems; no-shared-state approaches)
- Distributed computing is usually taken to mean processes and systems based on multiple instances (physical or, more commonly, virtual) - it may relate to multi-process systems too

CONSUMER/PRODUCER PATTERN



Demo multi-process sensors app

SINGLE-PROCESS APPROACH

```python
def get_n_samples(sensor, num):
    """ Samples the cheap sensor and returns ```num``` values

    :param sensor: the sensor to sample
    :type sensor: Sensor
    :param num: the number of samples to return, default 1,000
    :return: the list of samples
    :rtype: list of bool
    """
    count = num
    samples = []
    for x in sensor.get():
        samples.append(x)
        if count <= 0:
            break
        count -= 1
    return samples
```

```python
def should_run(sensors=3, samples=1000, faulty=1.0):
    """ Finds out if we had a radioactive leak

    We define a leak if more than half the sensors return an alarm, for more than three
    consecutive samplings.

    Naive implementation, samples the sensors and assumes they will all fit in memory.

    :return: whether the sensor is faulty
    :rtype: bool
    """
    num = sensors
    sensors = [Sensor(faulty_pct=faulty) for _ in range(0, num)]
    samples = [get_n_samples(s, num=samples) for s in sensors]
    tot_count = len(samples[0])
    for x in xrange(0, tot_count):
        count = 0
        for i in range(num):
            if samples[i][x]:
                count += 1
        if count > 0:
            logging.error("At sample %d, %s sensors were in the ALARM state", x, count)
            if count > num / 2:
                break
    # Just because I wanted to show the use of for/else - a very Pythonic pattern!
    else:
        return False
    return True
```

MULTI-PROCESSING

```python
def producer(q, delay=0.500):
    """ It will forever put the sensor's readings onto the queue

    :param q: the queue to push sensor data to
    :param delay: between readings
    :return: None
    """
    print("[{}] producer started".format(os.getpid()))
    sensor = Sensor(faulty_pct=30.0)
    for x in sensor.get():
        q.put(x)
        time.sleep(delay)


def consumer(q, threshold=5):
    """ Reads values from the queue and raises an alarm if more than ```threshold```
        consecutive values are True

    :param q: the queue to read from
    :return: never, unless the threshold is exceeded
    """
    print("[monitor] Started with threshold {}".format(threshold))
    count = 0
    while count < threshold:
        reading = q.get(block=True)
        if reading:
            count += 1
        else:
            # reset the counter
            count = 0
    print("[monitor] Threshold exceeded - exiting")
```

In Python, because of the GIL, multi-threading only makes sense for IO-bound processes

MULTI-PROCESSING (2)

Poor man's Process pool
(use multiprocessing.Pool)

Consumer Process

Queue

```
def main():
    pool = []
    q = mp.Queue()
    monitor = mp.Process(target=consumer, name="Monitor", args=(q, 2))
    monitor.start()
    for i in range(10):
        proc_name = 'Proc-{}'.format(i)
        p = mp.Process(target=producer, name=proc_name, args=(q,))
        p.start()
        pool.append(p)
    print("[main: {}] waiting for monitor to complete (when threshold is exceeded)".format(os.getpid()))
    monitor.join()
    for p in pool:
        p.terminate()
    print("[main: {}] finished".format(os.getpid()))
```

Producer Processes

Poor man's IPC
(use multiprocessing.Pipe)

In Python, because of the GIL, multi-threading only makes sense for IO-bound processes

MULTIPROCESSING FOR REAL IS ACTUALLY (A LOT) MORE COMPLICATED

Shared state

```python
def consumer(queue, idx, threshold=5, shared=None):
    """ Reads values from the queue and raises an alarm

    More than ```threshold``` consecutive values that are True will trigger an alarm.

    :param queue: the queue to read from
    :param threshold: The threshold at which we trigger the alarm, across ALL monitors
    :param shared: an optional shared ```Value`` for multiple Monitors
    :type shared: multiprocessing.Value
    :return: never, unless the threshold is exceeded
    """
    log("[monitor: {}] Started with threshold {}".format(os.getpid(), threshold))
    count = 0
    try:
        while shared.value < threshold:
            reading = queue.get(block=True)
            if reading:
                count += 1
                log('Alerting: {}'.format(count))
            else:
                # reset the counter
                count = 0
            if shared is not None:
                with shared.get_lock():
                    # NOTE: the double-check, as things may have changed between the test on the
                    # while and here; not doing this, causes some monitors to never terminate
                    if count == 0 and shared.value < threshold:
                        shared.value = 0
                    else:
                        shared.value += count
        log("[monitor-{}] Threshold exceeded - exiting".format(idx))
    except KeyboardInterrupt:
        # User pressed Ctrl-C, safe to ignore
        pass
```

MP Locking

Surprising facts

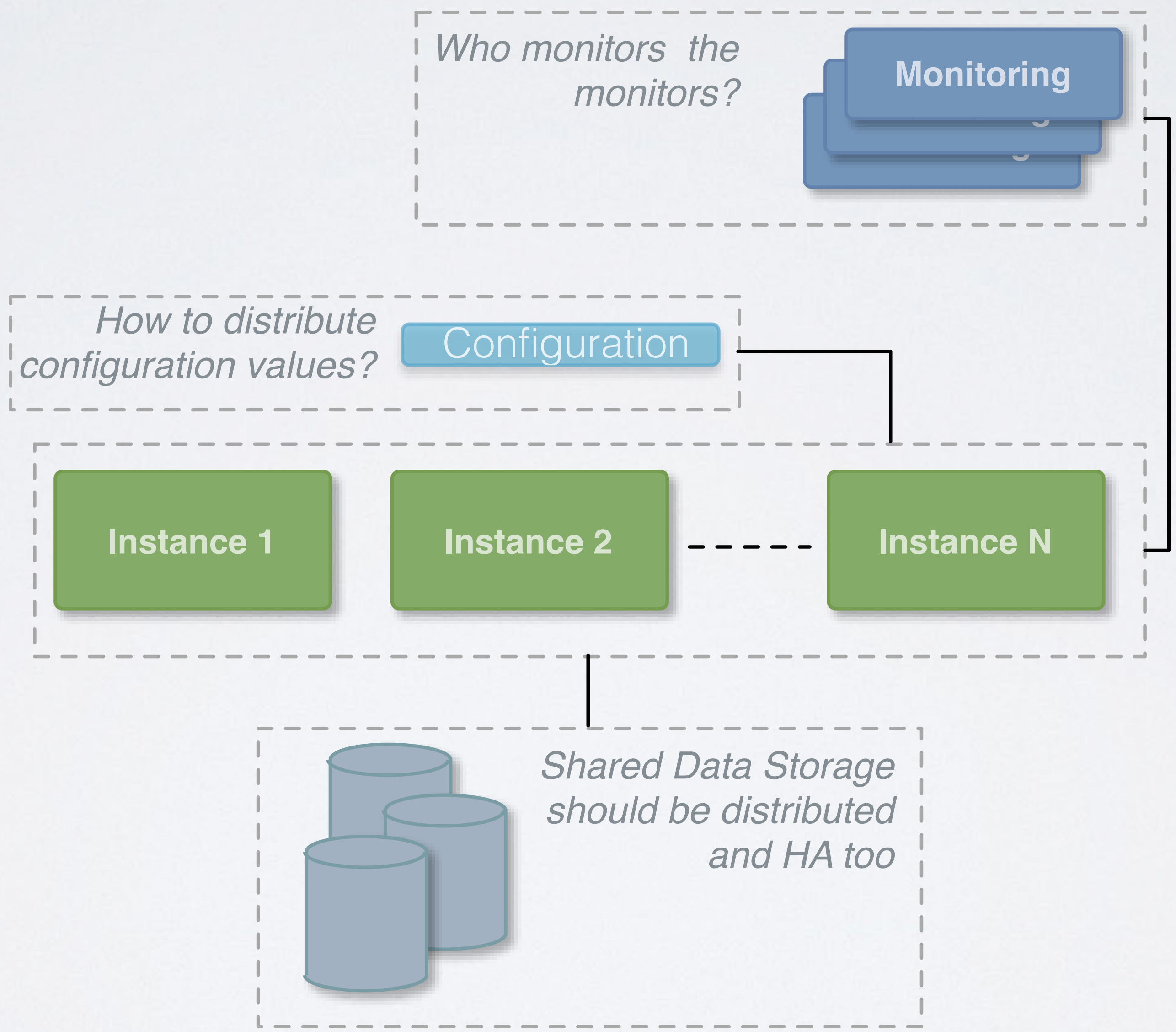
DISTRIBUTED COMPUTING ACROSS MULTIPLE INSTANCES

For "horizontal" scalability and increased availability

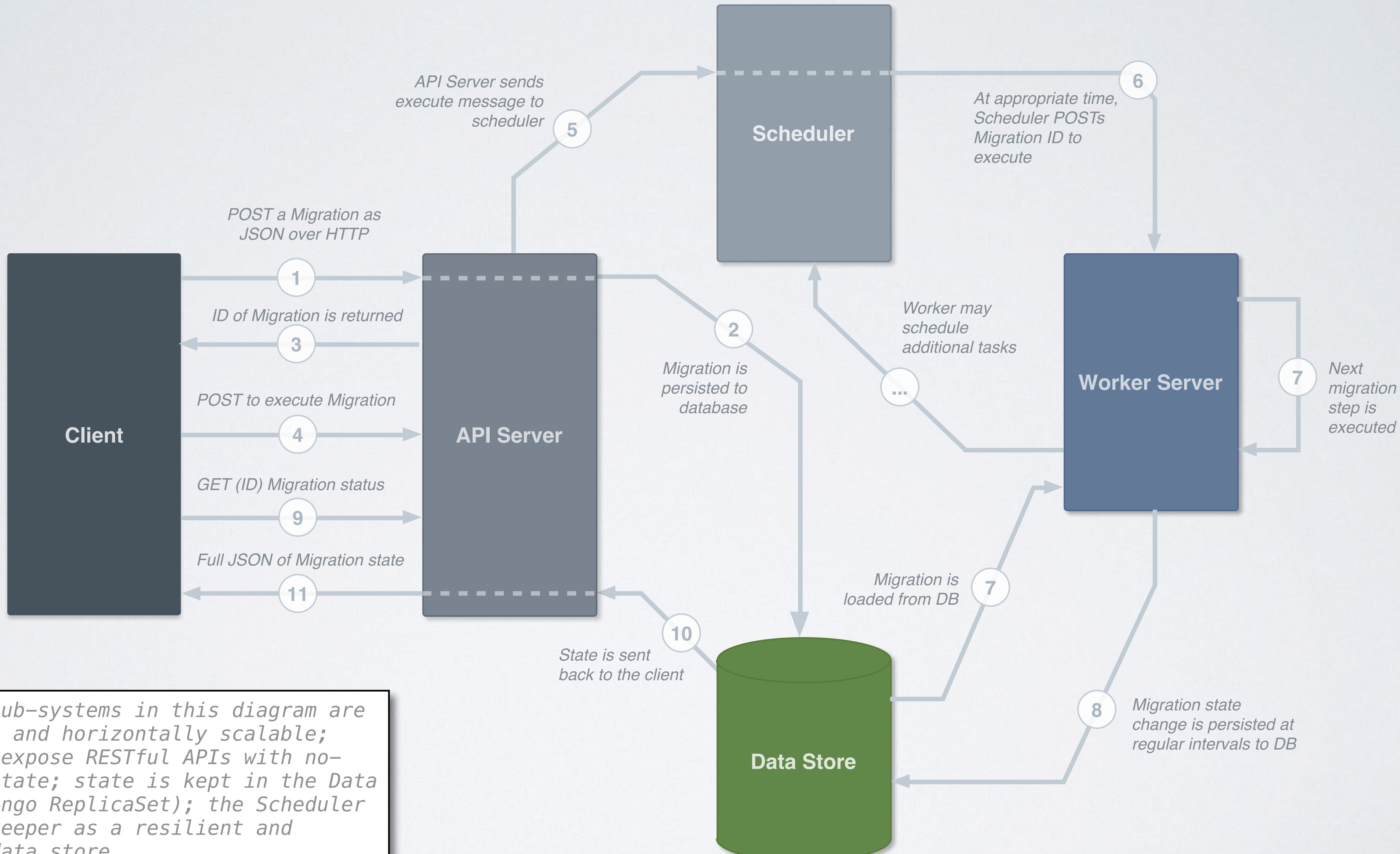
Challenges:

- Stateless architectures;
- Network protocol design (proprietary, or using well-established protocols: HTTP, ZMQ, TCP);
- Coordination & Monitoring of instances;
- Failure management & Restarting of instances
- Configuration management, resource discovery and fault diagnosis

DISTRIBUTED COMPUTING ACROSS MULTIPLE INSTANCES (2)



REAL-LIFE EXAMPLE: SEPARATION OF CONCERNS



All the sub-systems in this diagram are stateless and horizontally scalable; they all expose RESTful APIs with no-session state; state is kept in the Data Store (Mongo ReplicaSet); the Scheduler uses Zookeeper as a resilient and durable data store.
Not shown, there are also a Monitoring subsystem and an Analytical Engine.

SUMMARY

- Distributed computing is necessary to enable performance, availability and scalability of computing systems;
- Coordination, configuration and communication across distributed instances/processes become major concerns;
- The computation model must change too - in particular, it is no longer safe to assume 'data locality' (or even its availability);
- Simplicity and homogeneity are two valuable attributes worth pursuing when designing distributed systems (think functional models);
- Synchronization and shared memory using multi-threading primitives fly in the face of "simplicity" and make the system more brittle (and more complicated - way more complicated - to diagnose and debut) - and may negatively impact on performance too.

PROJECT

- Build various components over the span of the course
- Will require to interact with AWS APIs
- Build a "multiprocessing" computation
- Build a MapReduce (running against AWS EMR, using Python Streaming)
- Build a more complex MR, using a NoSQL DB (probably MongoDB) to store results
- Mostly meant to illustrate the issues to bear in mind when building a distributed system, rather than test your programming skills (but extra credit for clean, readable code!)
- All the sample code shown is on github: https://github.com/massenz/MSAN694
- Some of my rants: http://codetrips.com
- And some of my gists too: https://gist.github.com/massenz
- All of the slides will be posted on github too
  (and on the Course intranet, soon as I'm given access to it)