Scalable and distributed applications in Python

Julien Danjou – 23th September 2017 – PyCon FR

Hello!

I am Julien Danjou

Principal Software Engineer at **Red Hat**

You can find me at **julien@danjou.info** and 🐦 **@juldanjou**

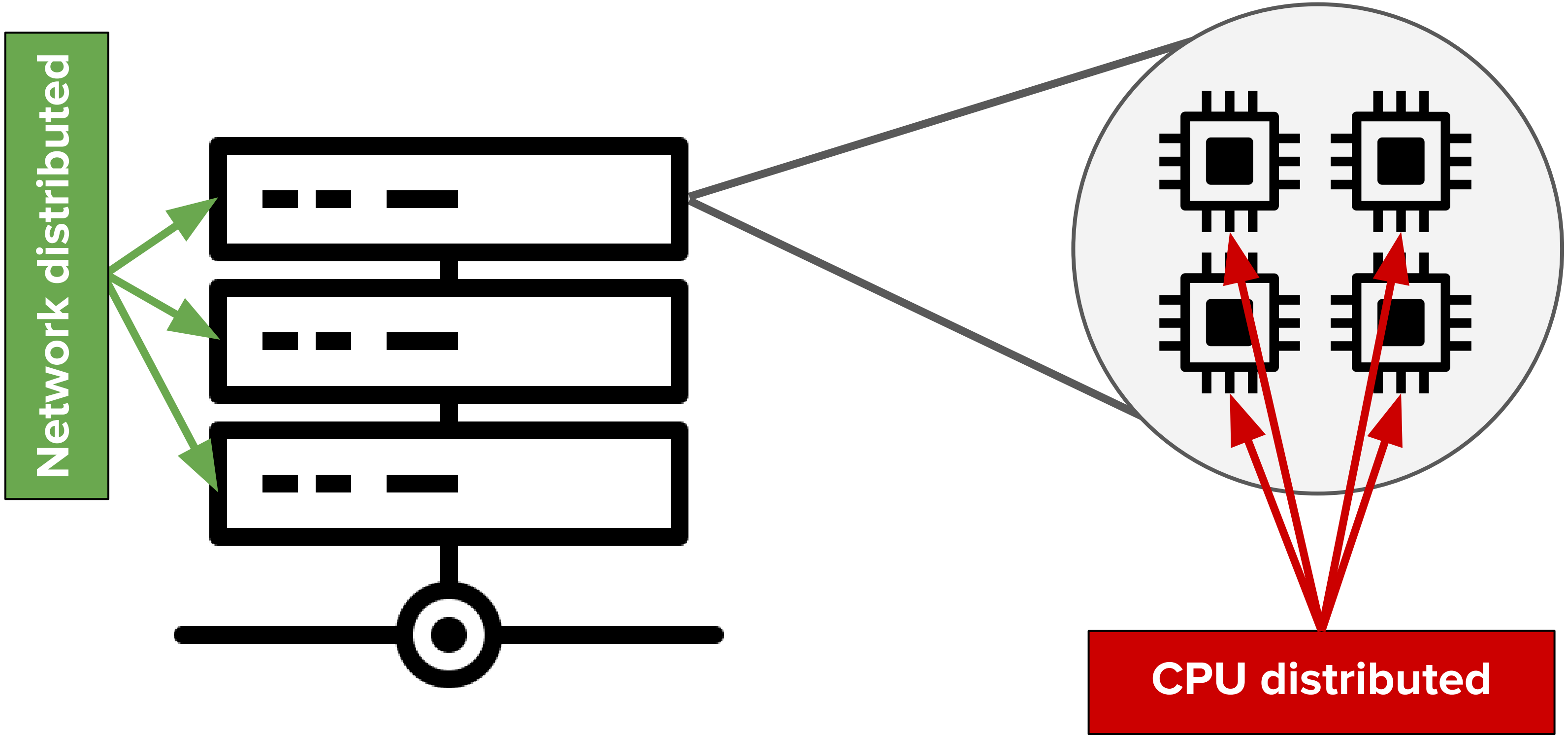
I **hack**, **create** and **contribute** to FOSS projects:

“the capability of a system, network, or process to handle a growing amount of work”

https://en.wikipedia.org/wiki/Scalability

Use more resources

(efficiently)



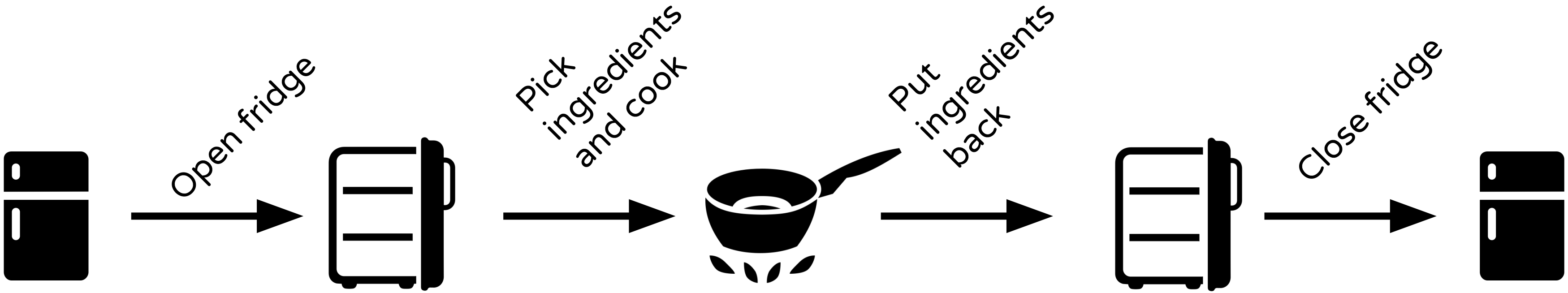
Network distributed

CPU distributed

Table of contents

- Concurrency
- Parallelism
- Threading
- Processes
- Network distribution
- Queues
- Retrying
- Consistent hashring
- Caching

Concurrency



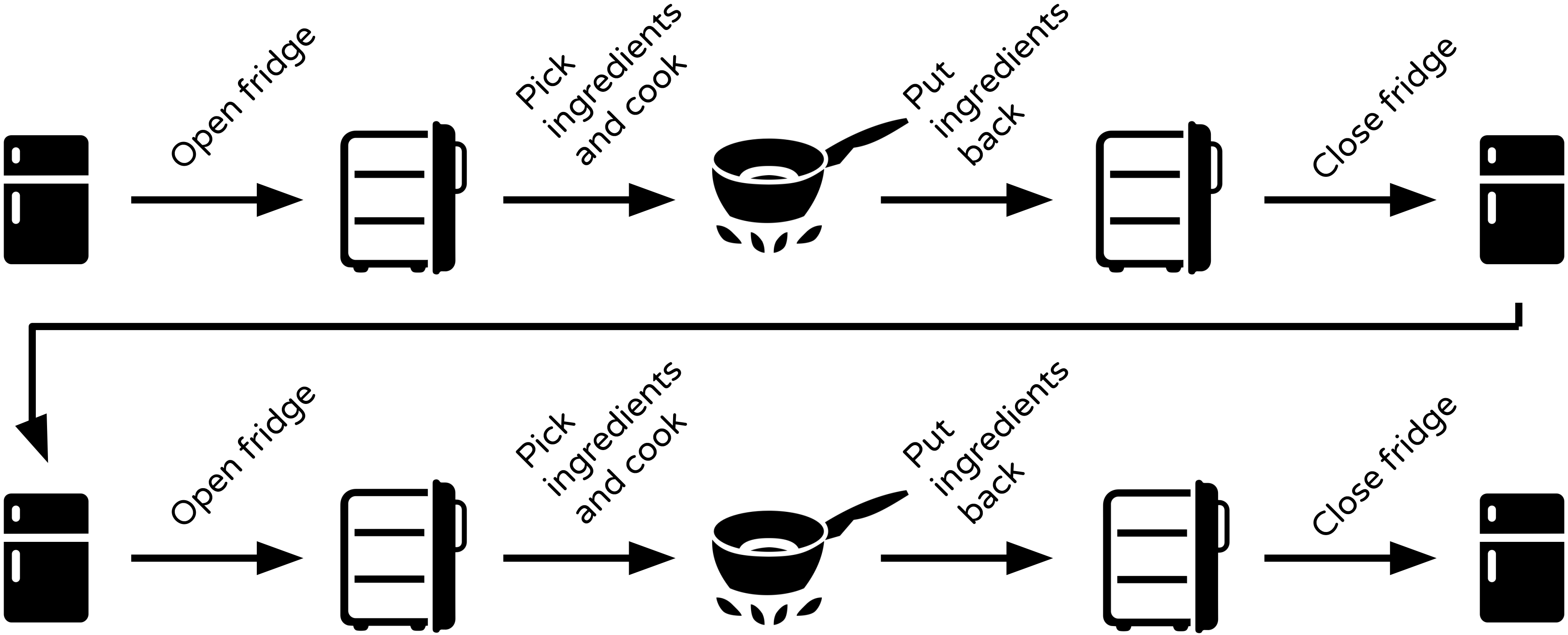
Open fridge

Pick ingredients and cook

Put ingredients back

Close fridge

Cooking a dish



Open fridge

Pick ingredients and cook

Put ingredients back

Close fridge

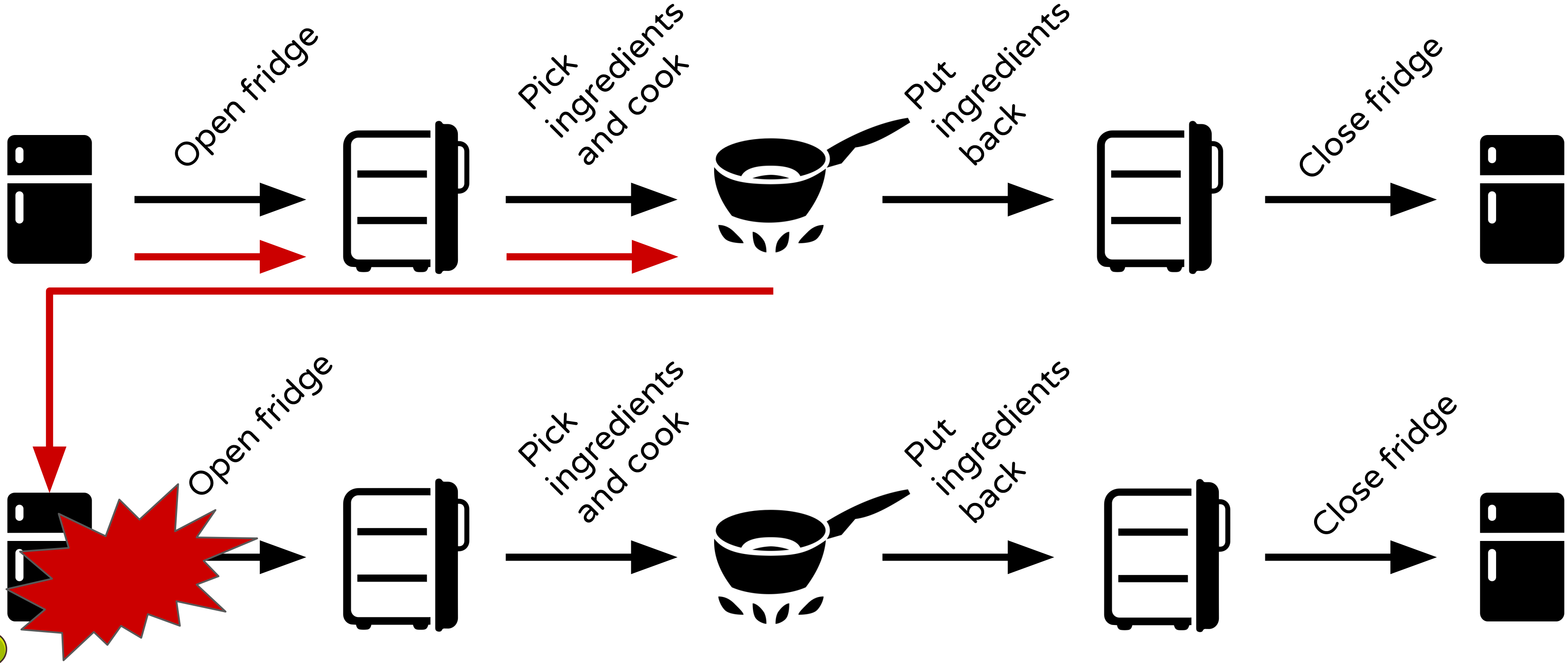
Open fridge

Pick ingredients and cook

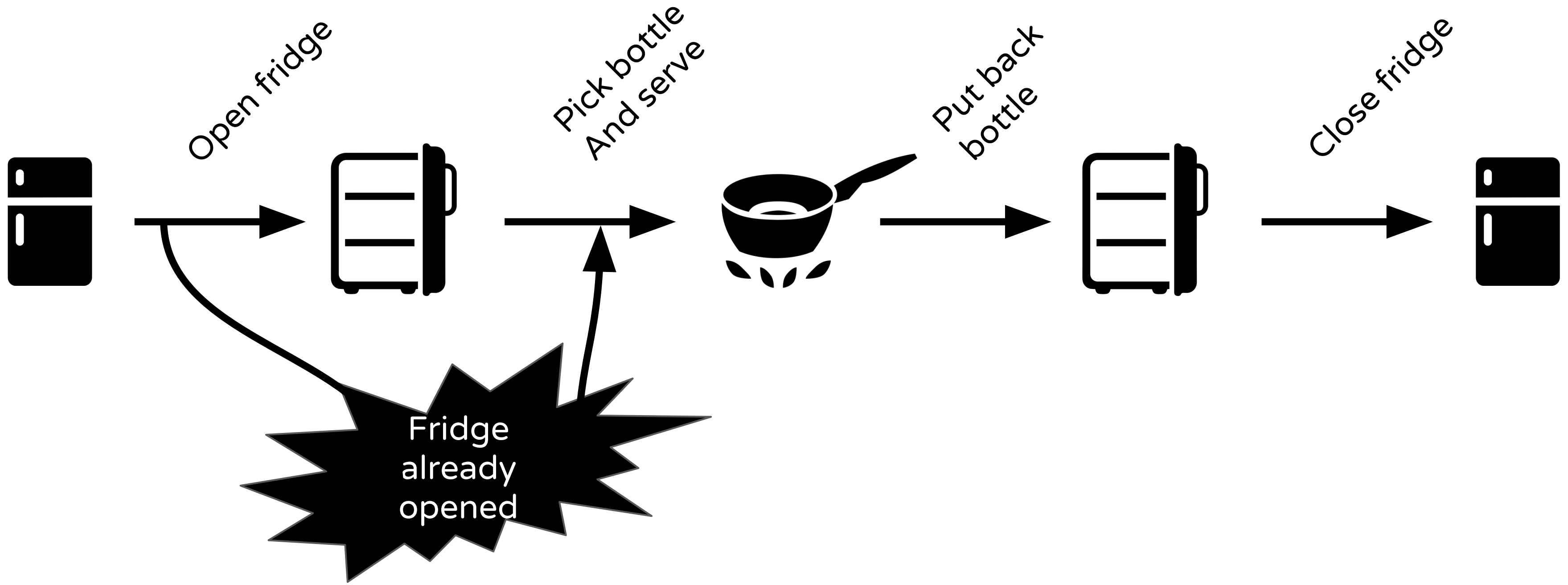
Put ingredients back

Close fridge

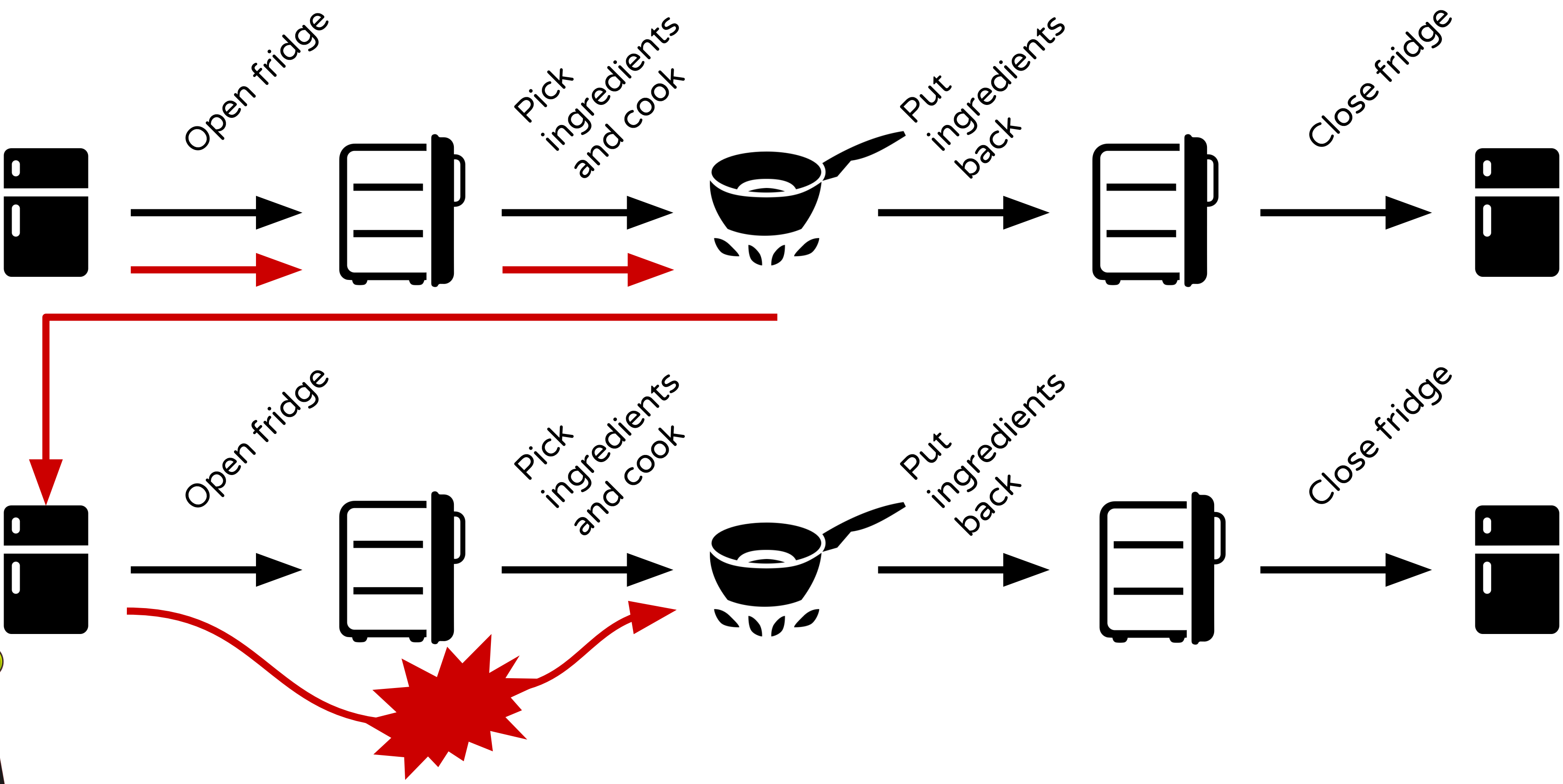
Cooking 2 dishes



Cooking 2 dishes concurrently



Cooking one dish, handling exception



Open fridge

Pick ingredients and cook

Put ingredients back

Close fridge

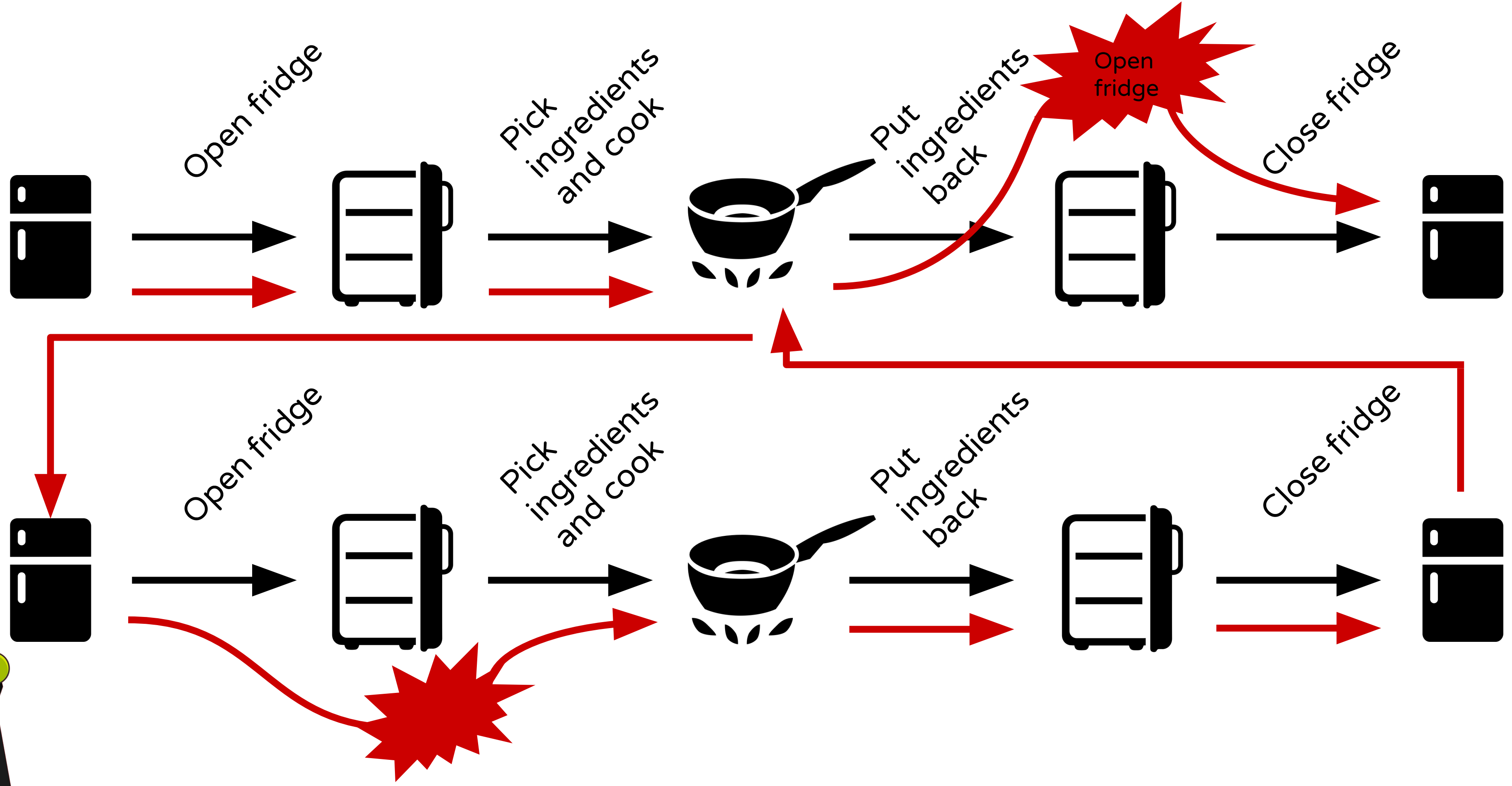
Open fridge

Pick ingredients and cook

Put ingredients back

Close fridge

Cooking two dishes concurrently handling fridge opened



Cooking two dishes concurrently handling fridge opened then closed

Parallelism

Multi-threading

Multi-threading

How to

- Shared global state
- Modules
  - Threading
  - Concurrent.futures
    - threading + pool
  - Futurist
    - concurrent.futures with backlog control and statistics
- threading.Lock

Global Interpreter Lock

Use multi-threading for

- I/O intensive workload that can be parallelized and asynchronous
- Computing stuff without accessing Python data structures
- Running C extensions in parallel

Multi-processes

Multi-processing

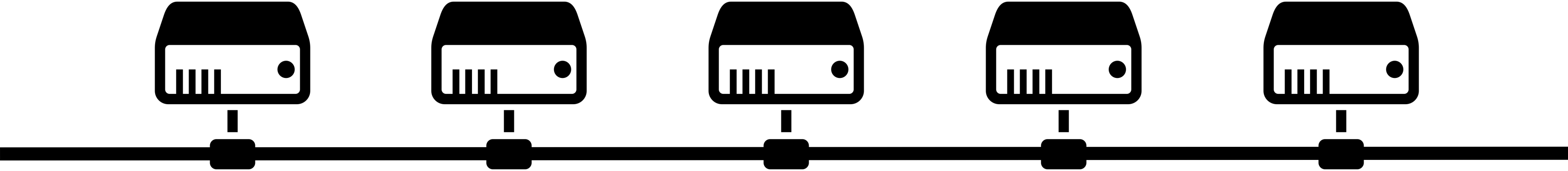
How to

- No shared global state
- 100% CPUs usage!
  - Independent GILs
- Modules
  - Multiprocessing
  - Concurrent.futures
    - Multiprocessing + pool
  - Cotyledon
    - Daemons
- Locks
  - `multiprocessing.Lock`
  - POSIX/SysV IPC
  - File-based locks (fasteners)
- No need for **Go**
  - multiprocessing.Manager().Queue()

Use multi-processing for

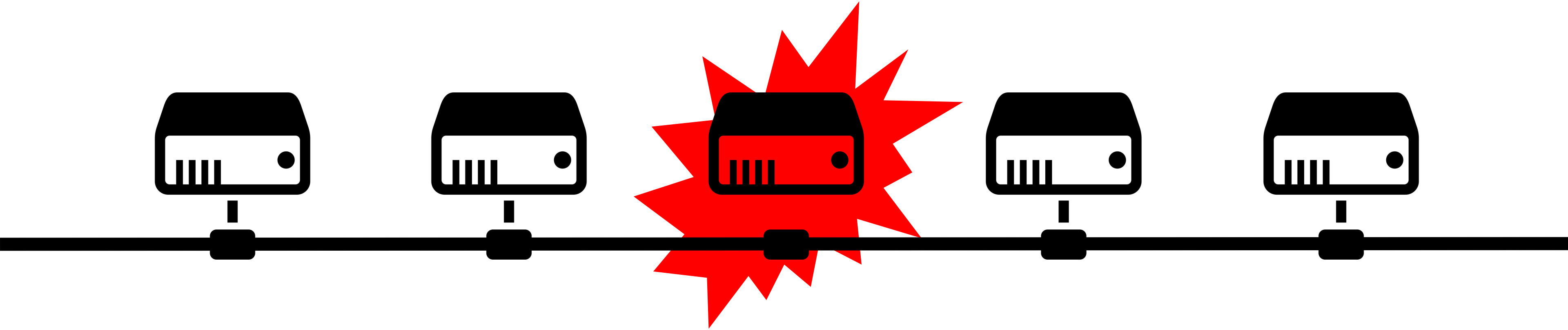
- Daemons, long-running tasks
- Stateless job processing
- First step toward a distributed system

Distributed system



Distributed across network

Multiple servers



What can fail will fail.

Multiple servers

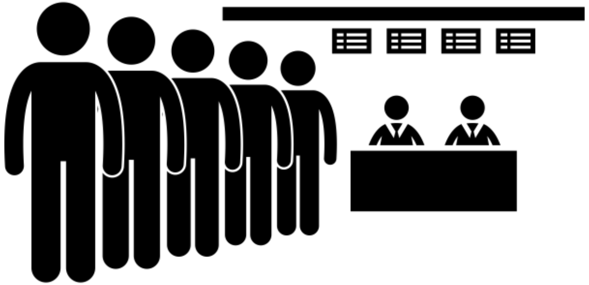
Pros

- No single-point-of-failure
- Horizontal scalability

Cons

- Failure of a node
  - Make failure a default scenario
- Failure of network
- Latency
  - Asynchronicity

Use queues based systems



- Easy to scale
- Easy to implement
- Easy to debug
- … because they are **functional (no-side effect)**

RQ

Easy job queues
for python

rq

- Based on Redis
- Low barrier of entry
- Easy scalability

Celery

- Multi-broker
- Widely used
- Language agnostic
- Advanced features:
  - Rate limits
  - Routing

Distributed locks

Not all are equals.

- ZooKeeper
- etcd
- Redis
- Consul
- memcached
- PostgreSQL
- MySQL

Tooz implements all of them, give it a try!

etcd

```
import etcd3

client = etcd3.client()
lock = client.lock("foobar")
with lock:
    print("do something")
```

Don’t forget

- Client disconnections, crashes, timeouts…
- Concurrency!
- Reliability
- Distributed lock manager crash (SPOF)
- Network latency
- Network outage

Designing for failure

```
while True:
    try:
        do_something()
    except:
        pass
    else:
        break
```

Need to handle:

- When to retry
- How often to retry
- What to do before retrying
- What to do after retrying

tenacity

```
pip install tenacity
```

- Define when to retry
- Define how often to retry
- Define what to do before retrying
- Define what to do after retrying

Retry on exceptions and stop at some point

```
import tenacity

@tenacity.retry(
    wait=wait.wait_fixed(1),
    stop=stop.stop_after_delay(60),
    retry=(retry.retry_if_exception_type(IOError) |
           retry.retry_if_result(lambda result: result == None))
def do_something_and_retry():
    do_something()
```

Group membership

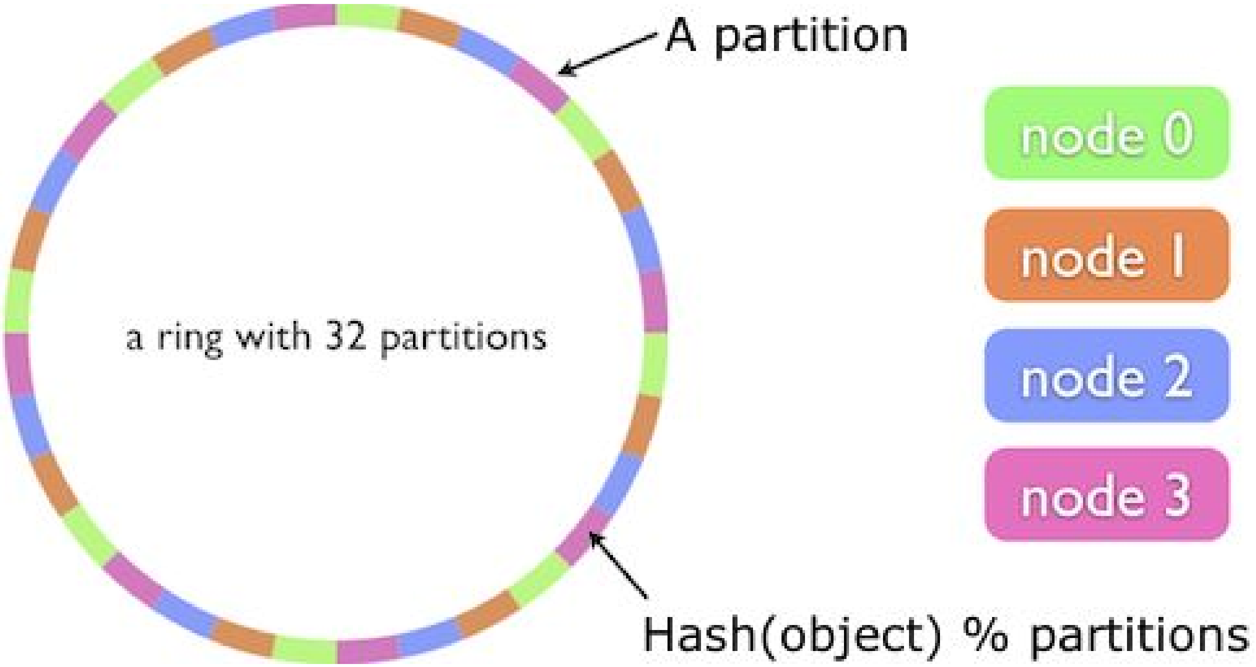
Pick a coordinator

- ZooKeeper
- etcd
- Redis
- Consul
- memcached

Tooz implements all of them, give it a try!

Use cases

- Workload distribution
- Aliveness check
- Consistent hashring



Consistent hashring

Caching

Memoization or remote cache

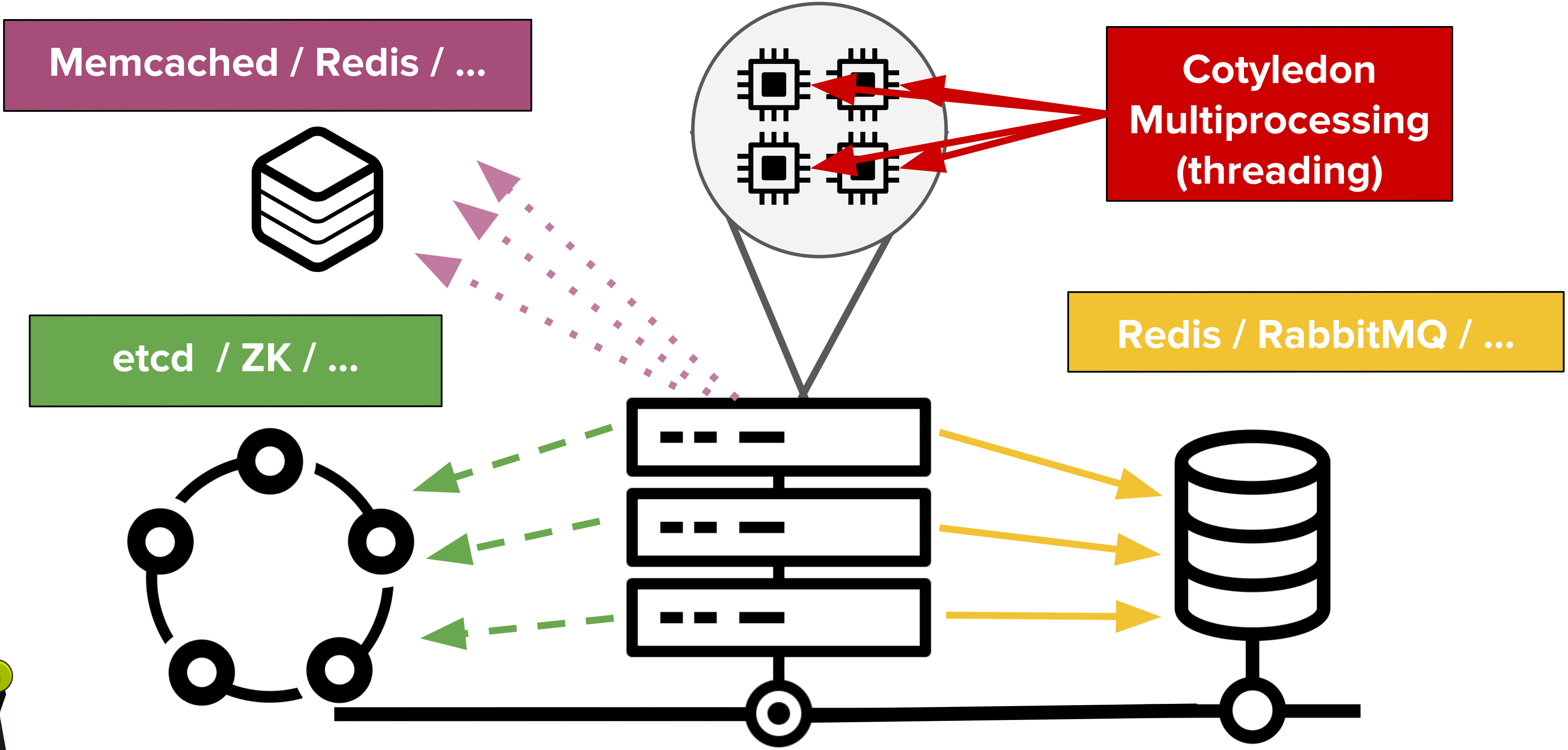
- High cost of computing data
- High latency of accessing data

- cachetools
- functools.lru_cache
- dogpile.cache

- **Cache invalidation**

cachetools

```
>>> import cachetools.func
>>> import math
>>> import time
>>> memoized_sin = cachetools.func.ttl_cache(ttl=5)(math.sin)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=128, currsize=0)
>>> time.sleep(5)
>>> memoized_sin.cache_info()
>>> CacheInfo(hits=1, misses=1, maxsize=128, currsize=0)
```



Memcached / Redis / ...

Cotyledon Multiprocessing (threading)

etcd / ZK / ...

Redis / RabbitMQ / ...

The Big Picture

Questions, feedback:

julien@danjou.info

Blog:

https://julien.danjou.info

Twitter:

@juldanjou

Talk inspired by http://scaling-python.com