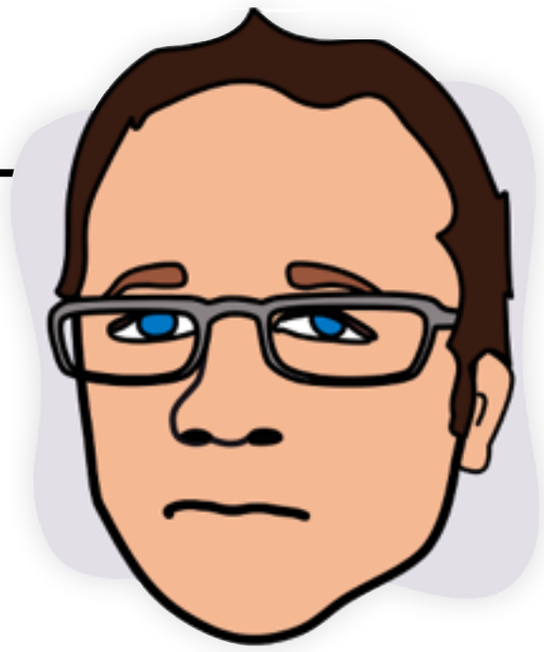
Erlang Factory San Francisco - 2014-03-06

Benoit Chesneau @benoitc

Scaling HTTP connections

http://enki-multimedia.org

About me



- Craftsman
- Working on and over the web
- Building open-sources solutions
- CouchDB committer and PMC member
- Member of the Python foundation, Gunicorn author
- Founder of the refuge project - http://refuge.io

Constraints

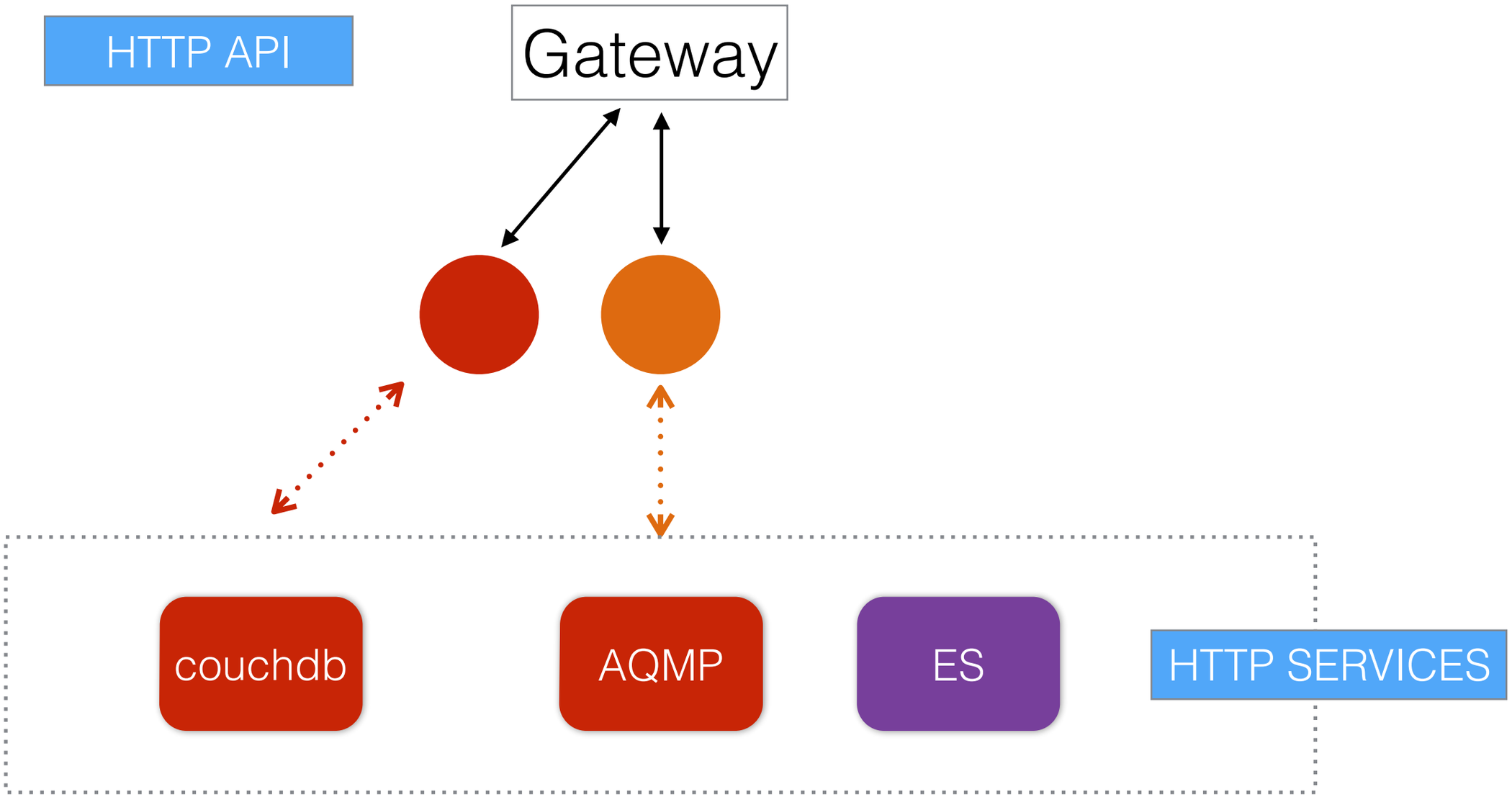
- Building many applications that requires a lot of HTTP connections to external services

- Some built around couchbeam [1], and couchdb [2]

- Other just need a remote or local access to a bunch of HTTP services

[1] http://github.com/benoitc/couchbeam
[2] http://couchdb.apache.org

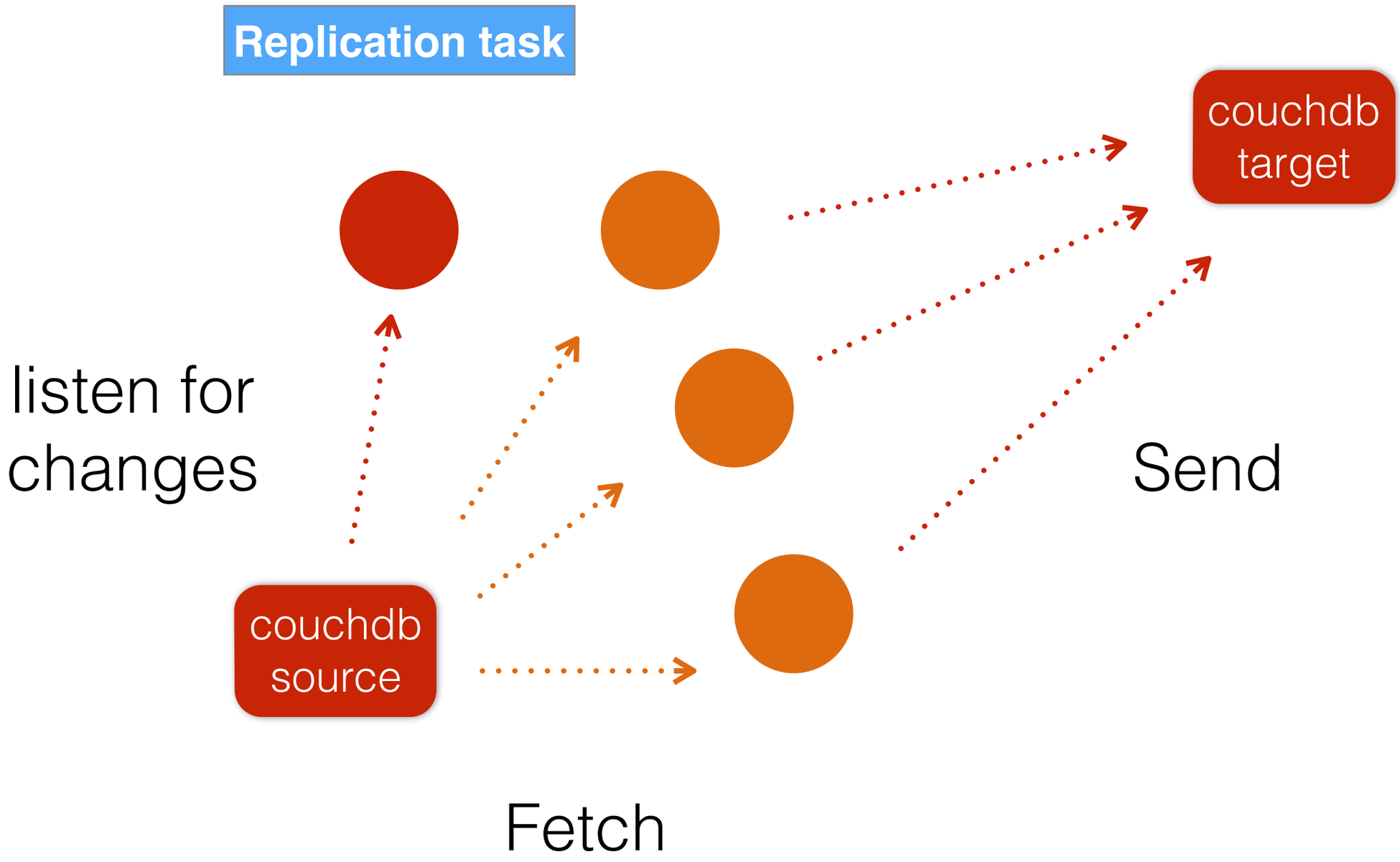
exampe: http resource proxy



exampe: http resource proxy

- allows applications to be built with the resources offered by the proxy
- transformations
- lot of short/long-lived connections
- no keep-alives
- no continuous connections

exampe: couchdb replicator



exampe: couchdb replicator

- specific case when both the source and the target are on different couchdb nodes
- replicate multiple docs, with attachments (blobs)
- thousands of connections (>10K/nodes)
- Continuous short and long-lived connections
- ***crashing far too often***

HTTP connection?

- can be on any transport
- Protocol on top of the transport
- HTTP 1.1 / SPDY / HTTP 2x

Panorama of the different used HTTP clients

- **HTTPC** - HTTP client distributed with Erlang
- **Ibrowse**
  http://github.com/cmullaparthi/ibrowse
- **LHTTPC**
  http://github.com/esl/lhttpc
- **Hackney**
  http://github.com/benoitc/hackney

The C10[0]K problems from the client...

Fight with the system limits

- number of file descriptors is limited
- RAM is limited

When it's limited, reuse….

- To reduce the number of connection we can cache locally
- can be a memory hog
- only get new contents (204/304 status)
- Or try to reuse the connection instead of creating a new one

Control the process

wait for a socket event

```
wait(Socket, KeepAlive) ->
    inets:setopts(Socket, [{active, once}),
    Timer = erlang:send_after(Timeout, self(),
                              {timeout, Socket}),
    receive
        {tcp_closed, Socket} ->
            %% remove from the pool
        {timeout, Socket} ->
            %% remove from the pool
        {checkout, To} ->
            gen_tcp:controlling_process(Socket, To),
            To ! Socket

after KeepAlive ->
    %%


end.
```

give control the socket to a new process

Control the process

- active mode
- can be used to build a pool (using a gen_server for example)
- or reuse the socket in the same process to handle keepalive or pipelining in HTTP1.1
- All the clients are using one technic or another

Limit the concurrency

- Reusing a connection is not enough
- Under load you want to reduce the number of concurrent connections

Limit the concurrency

- queue the connections
- drop the connections

- allows any extra connections until you run out of fds but only reuse some
- lhttpc fork [1] or hackney_dispcount [2] pool

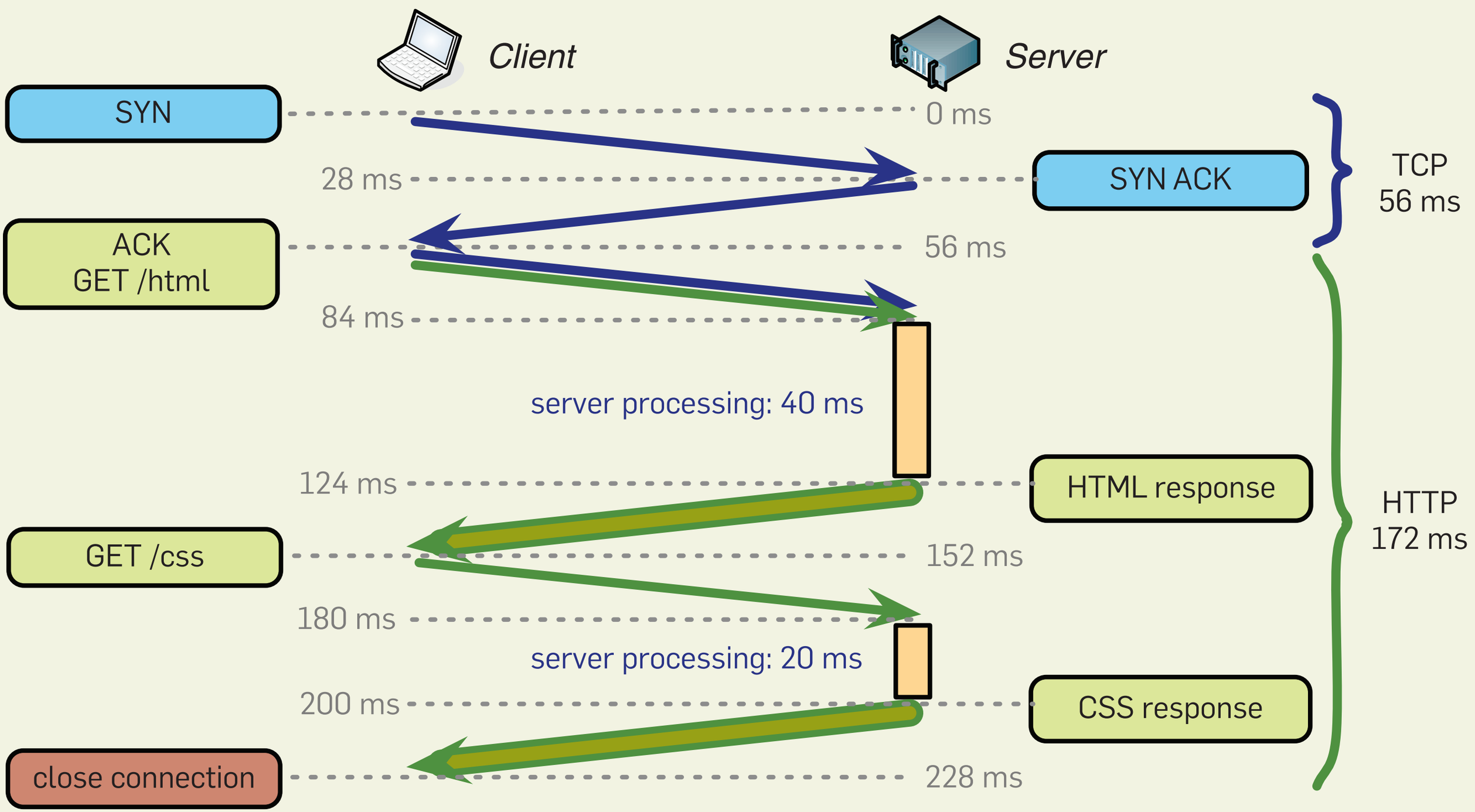
Reduce the memory usage

- memory consumption can be big
- you need to stream when receiving
- but also when you send

The network can be hostile

- a connection can crash
  - at any time.
- A connection can be slow … or too fast.

Figure 1. With 56ms RTT, fetching two files takes approximately 228ms, with 80% of that time in network latency.

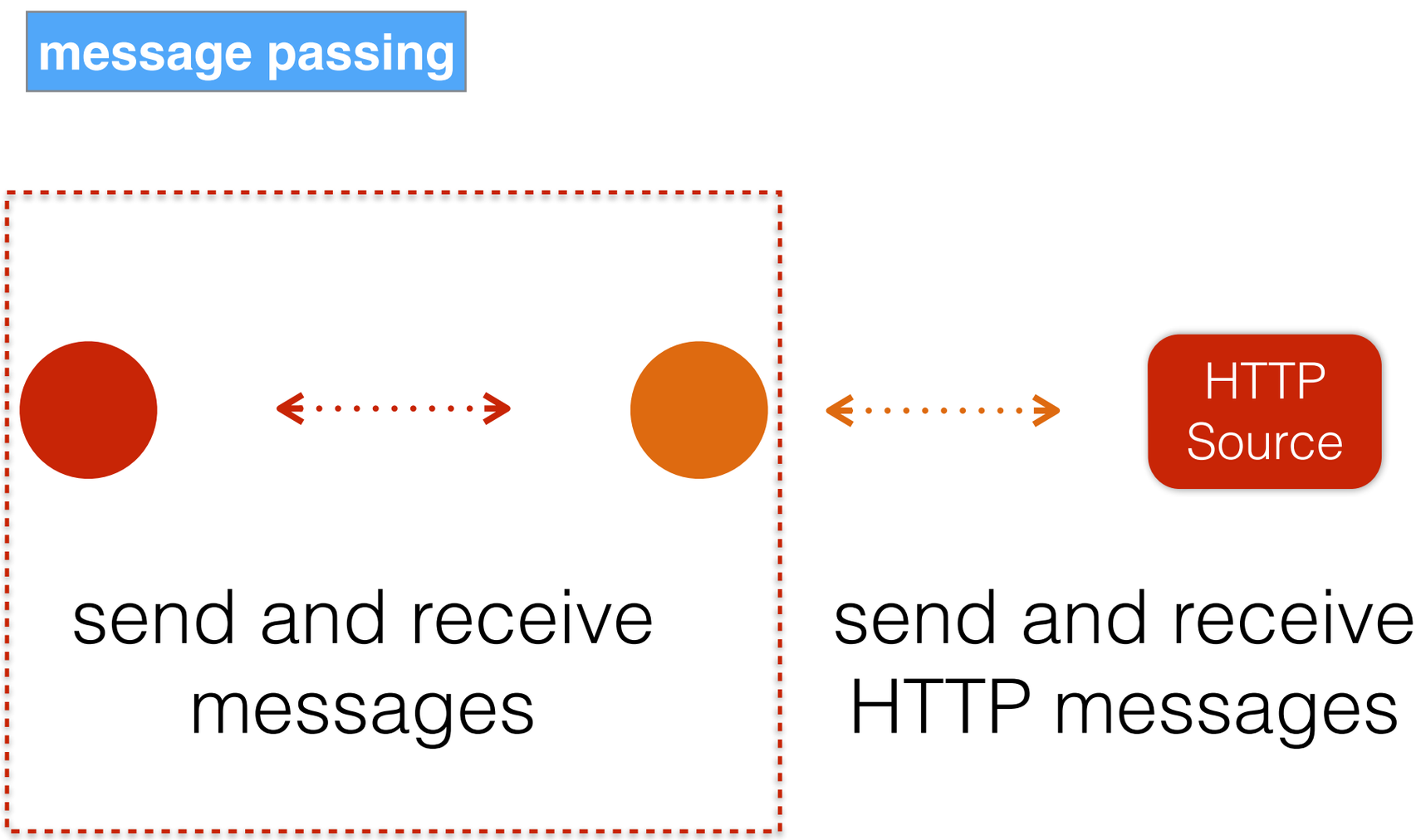


The network can be hostile

- “Expect: 100-continue” by default in hackney
- Fast parser to read headers
- Supervise your requests

Designing an HTTP client

A usual client pattern



A usual client pattern

- A process to maintain the state and dialog with the socket
- Message passing is used to dialog with this process
- The socket is (maybe) fetched from the pool

client patterns - hackney v2 (0.11.1)

send and receive
HTTP messages

Make the API less painful

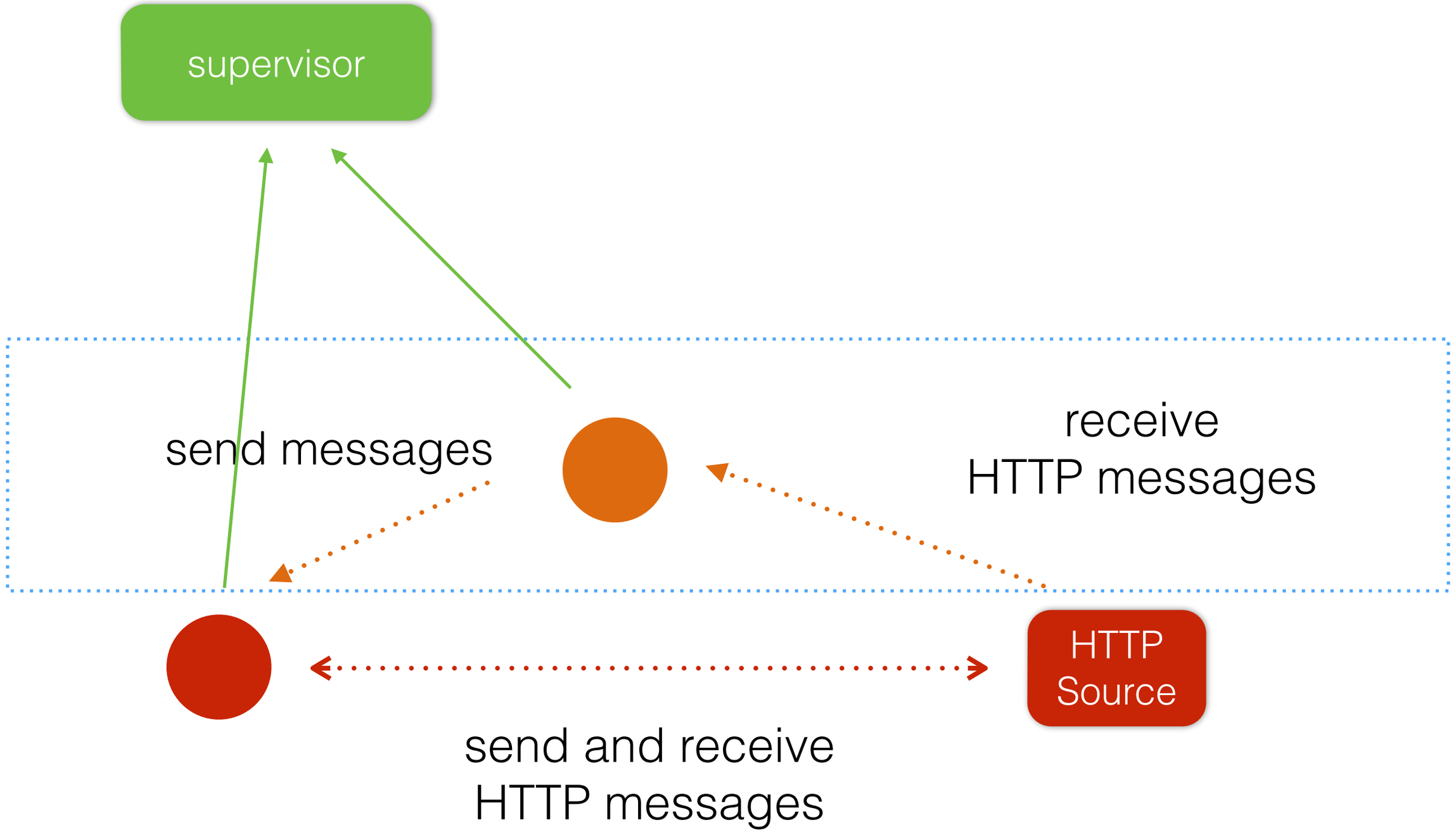
hackney v1

```
{ok, _, _, Ctx} = hackney:request(get, <<“http”//friendpaste.com”>>),
{ok, Chunk, Ctx1} = hackney:recv_body(Ctx)
```

hackney v2

```
{ok, _, _, Ref} = hackney:request(get, <<“http”//friendpaste.com”>>),
{ok, Chunk} = hackney:recv_body(Ref)
```

client patterns - hackney v2 (0.11.1)



hackney v2 (0.11.1)

- All requests (active connections) have a ref ID
- no message passing by default
- The intermediate non parsed buffer (state) is kept in an ETS while reading the response
- Only async connections open a new process

copy data

- When you send a message:
- data is copied to the other process
- When the binary size is > 64K only a reference is passed.
- The reference is kept around, until all the process that have accessed to the reference has been garbage collected (ref count)

hackney v2 (0.11.1) - status

- solved my garbage collection problem
- simple API
- Easily handle multiple connections
- hackney_lib: extract the parsers and HTTP protocol helpers

HTTP 2 designed for Erlang

- Stream—a bidirectional flow of bytes, or a virtual channel, within a connection. Each stream has a relative priority value and a unique integer identifier.

- Message—a complete sequence of frames that maps to a logical message such as an HTTP request or a response.

- Frame

hackney v3

- hackney_connect: a connection manager allowing different policies. Sort of specialised pool for connections

- connection event handler

- Embrace HTTP 2 - abstract the protocol in Erlang messages

- While we are here add the websockets support

?

@benoitc