# Implementing BitTorrent in Elixir

September 8, 2016

Erlang User Conference 2016

# Goal of the Talk

- No new ground breaking BitTorrent research here; only the basics—BitTorrent Protocol (BEP-0003)
- Establish some tools for building non-trivial systems on the BEAM

Presentation focus: BEP-3 — the basic BitTorrent implementation

The talk is geared: towards Elixir developers who mostly works with a framework like Phoenix

I think it is impossible to cover EVERYTHING that goes into an actual BitTorrent implementation in 40 minutes; this is more of a highlight reel!

[]

«Do you agree that one should use the right tool for the job ?»

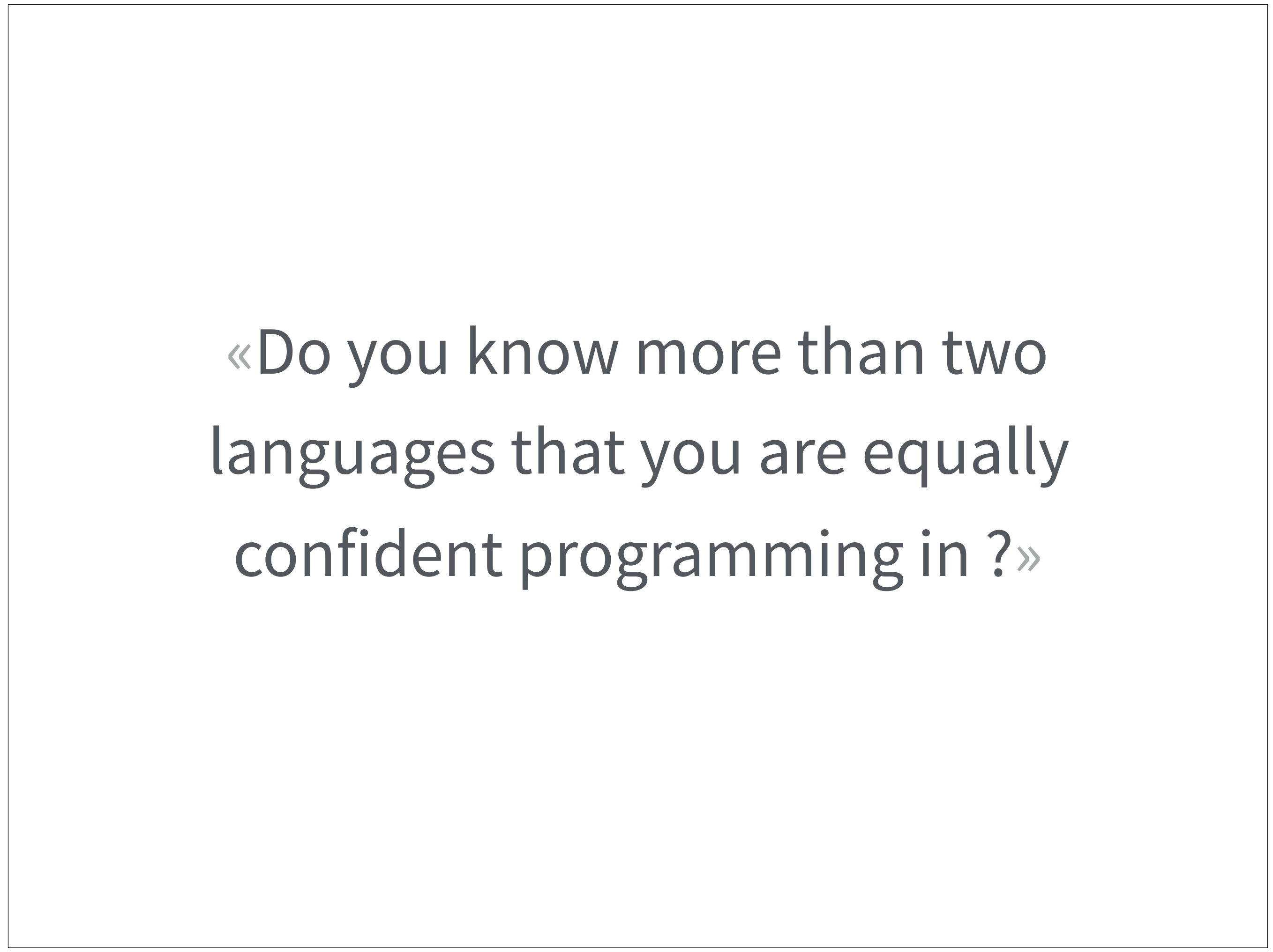
I remember the first Erlang presentation I saw…back in Copenhagen Torben Hoffmann told the story of WhatsApp.

Torben started off by asking the following question: …

> A notion that one can only agree to

He asked us to keep our hand up and take it down if we couldn't answer yes to his follow up question:

> Do you know more than two languages?




How can we have a feeling for the best tool if we only know one or two?

«The right tool for the job»

There must be jobs that are best solved using «the right tool»

«To fully understand a programming language you must implement something non-trivial with it»

–Jesper Louis Andersen

Erlang User Conference 2012 «Combinatorrent - a Haskell Case Study»

Further more, if we follow Jesper Louis' saying…

(…Fibonacci doesn't count.)

It seems…

We must find a **non-trivial task**
that is **best solved**
using **the tool we want to learn**

So, who is up for a hard challenge?

So let's hear it: Who's up for a hard challenge?
…
I hate hard challenges.
Hard challenges are nasty. They feel like work.

# Implementing BitTorrent


- BitTorrent is hard
- We'll probably not get it right the first time
- But if we choose the right tool

—–

- But we will be well on our way…

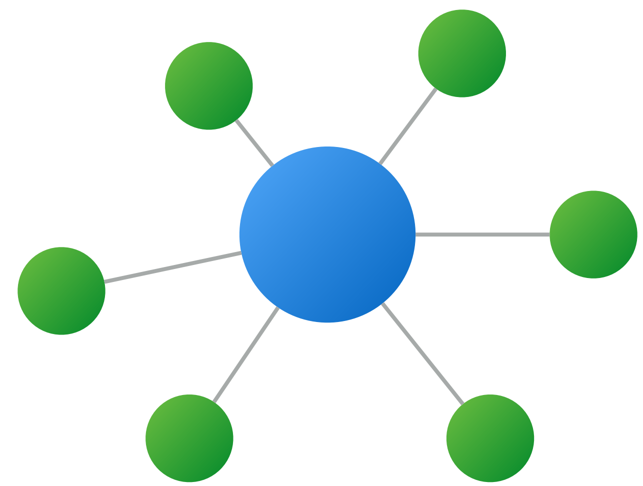
—–

- We want a manageable task
- one sort of outside of our comfort zone

# How BitTorrent Works

Let's talk about the problem…

# HTTP



Let's compare with HTTP

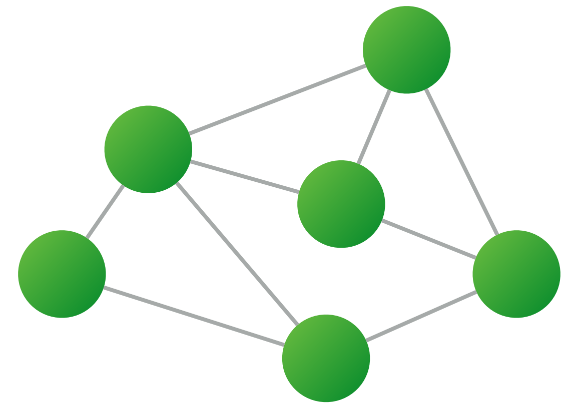
Content distribution!

Simple

Server/Client model

Single point of failure

# BitTorrent



Content distribution!

Complex

Peers are **connected in a mesh**—referred to as **The Swarm**

Data is shuffled around amongst the peers

Every client is a server
—very hard to remove
**—no single point of failure**

The BitTorrent lifecycle consist of three stages

Identity / Discovery / Exchange

# Identity

The Info Hash

https://www2.example.com/foo

HTTP

Again: compared with HTTP

- The HTTP **identity**: the URI
- Points to a resource on the internet
- Stuff can change
- Stuff can disappear

```
<<114, 114, 247, 45, 24,
  160, 165, 233, 159, 236,
  48, 155, 105, 138, 168,
  184, 97, 62, 251, 12>>
```

BitTorrent (sha1)

- In contrast… the **BitTorrent identity** is a **sha1 hash**—the **result of a one way function** that **takes some data and digest it**
- It is not that human friendly…
- This makes changes impossible: one change = new hash, the identity will always point to the same data.

Where does this «info hash» come from ?

[] We find a file we want to share

The file is an array of bytes…

2) We split the array of bytes into a list of pieces, of (let's say) 256 KB


3) we calculate the sha1-hash of every piece

```
[<<160, 52, 165, 235, 44, 208, 74, 31, 216, 244,
  212, 123, 216, 54, 11, 66, 72, 27, 21, 31>>,
 <<108, 181, 136, 15, 57, 63, 173, 140, 198, 244,
  159, 92, 196, 68, 111, 127, 244, 55, 156, 151>>,
 <<39, 192, 32, 96, 18, 99, 243, 167, 101, 167,
  59, 155, 42, 88, 201, 153, 151, 149, 88, 85>>,
 <<7, 104, 159, 184, 98, 80, 189, 182, 173, 156,
  68, 114, 108, 242, 218, 152, 137, 128, 62, 73>>,
 <<80, 141, 98, 154, 153, 142, 228, 110, 6, 231,
  130, 18, 34, 170, 32, 77, 148, 48, 125, 210>>,
 <<27, 51, 243, 228, 253, 165, 243, 242, 44, 171,
  116, 184, 119, 255, 157, 168, 226, 39, 189, 100>>]

|> IO.iodata_to_binary()
```

Giving us a **list of hashes**

By concatenating this list into one binary, *by using our trusty iodata_to_binary-function*, we will get *a blob of hashes*.

# pieces =

```
<<160, 52, 165, 235, 44, 208, 74, 31, 216, 244, 212, 123,
216, 54, 11, 66, 72, 27, 21, 31, 108, 181, 136, 15, 57,
63, 173, 140, 198, 244, 159, 92, 196, 68, 111, 127, 244,
55, 156, 151, 39, 192, 32, 96, 18, 99, 243, 167, 101,
167, 59, 155, 42, 88, 201, 153, 151, 149, 88, 85, 7, 104,
159, 184, 98, 80, 189, 182, 173, 156, 68, 114, 108, 242,
218, 152, 137, 128, 62, 73, 80, 141, 98, 154, 153, 142,
228, 110, 6, 231, 130, 18, 34, 170, 32, 77, 148, 48, 125,
210, 27, 51, 243, 228, 253, 165, 243, 242, 44, 171, 116,
184, 119, 255, 157, 168, 226, 39, 189, 100>>
```

We refer to this **concatenated series of hashes** as «**pieces**»

# The info dictionary

```
info =
  %{"name" => "Unnamed torrent 1",
    "pieces" => pieces,
    "length" => 1441792,
    "piece length" => 262144}
  |> Bencode.encode!()

info_hash =
  :crypto.hash(:sha, Bencode.encode!(info))
```

- This data is put into a dictionary, along with some meta data describing the file size and the size we chose for the pieces;

---

- This data is encoded into b-encode,
- B-encode is kinda like JSON, only very compact

---

- And we will use the Erlang crypto module to generate the hash.

This is the info hash

# The info hash

```
Meta-data

+

The hashes of the pieces

of the file consist of
```

The brilliance about this is that we can:

*Verify the pieces*
*— as we receive them*
*— and start sharing them*
*— without having the whole file*

# Discovery

Finding Peers

- Given an info hash
- we need to find peers participating in the sharing the file

# Query a DHT*

or

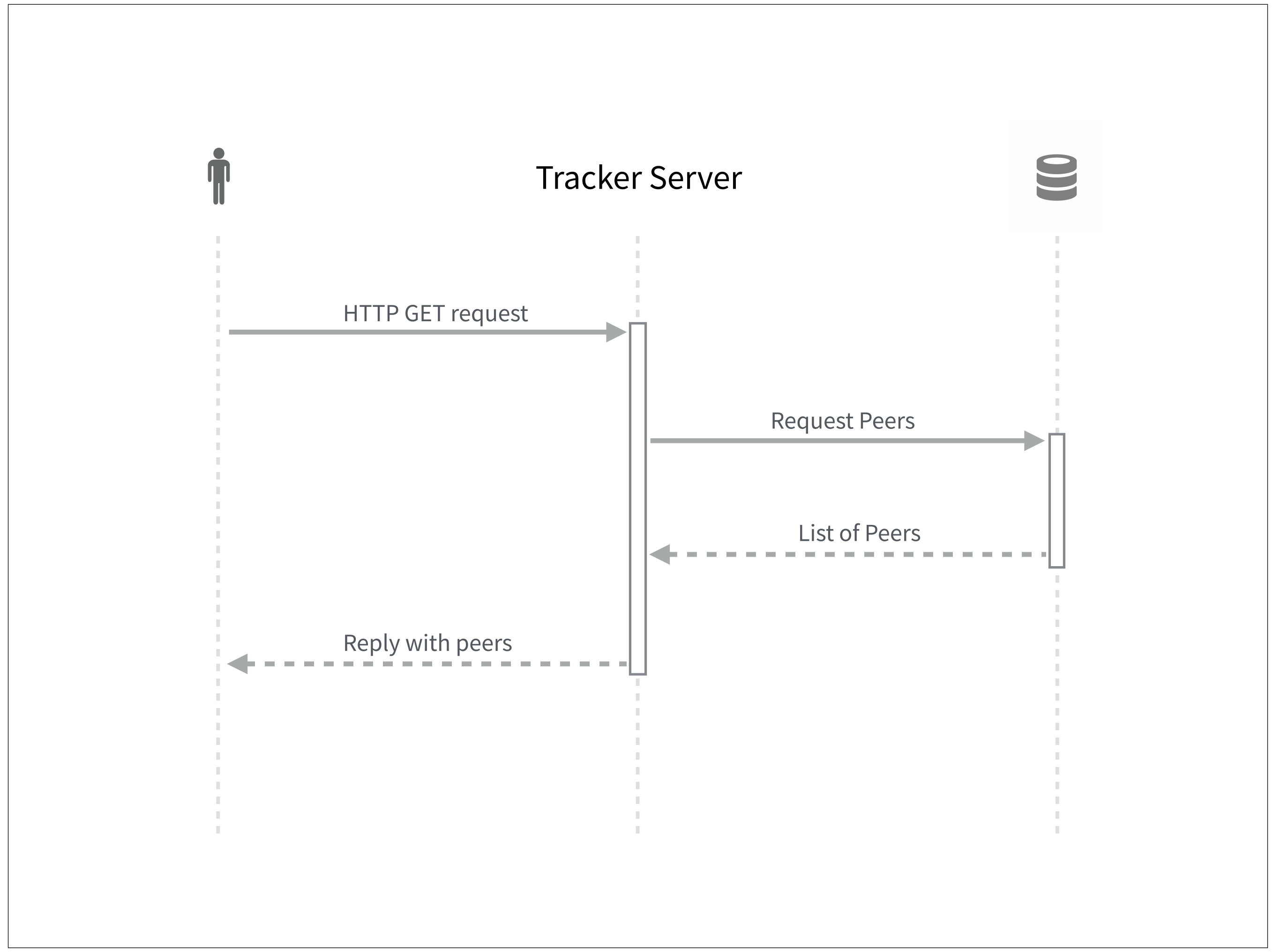
# Ask a Tracker Server

* Distributed Hash Table

There are some ways to go about this: DHT or a tracker server

# Ask a Tracker Server

- Let's focus on the simplest possible thing: the tracker server
- In the .torrent file we will find a list of tracker servers

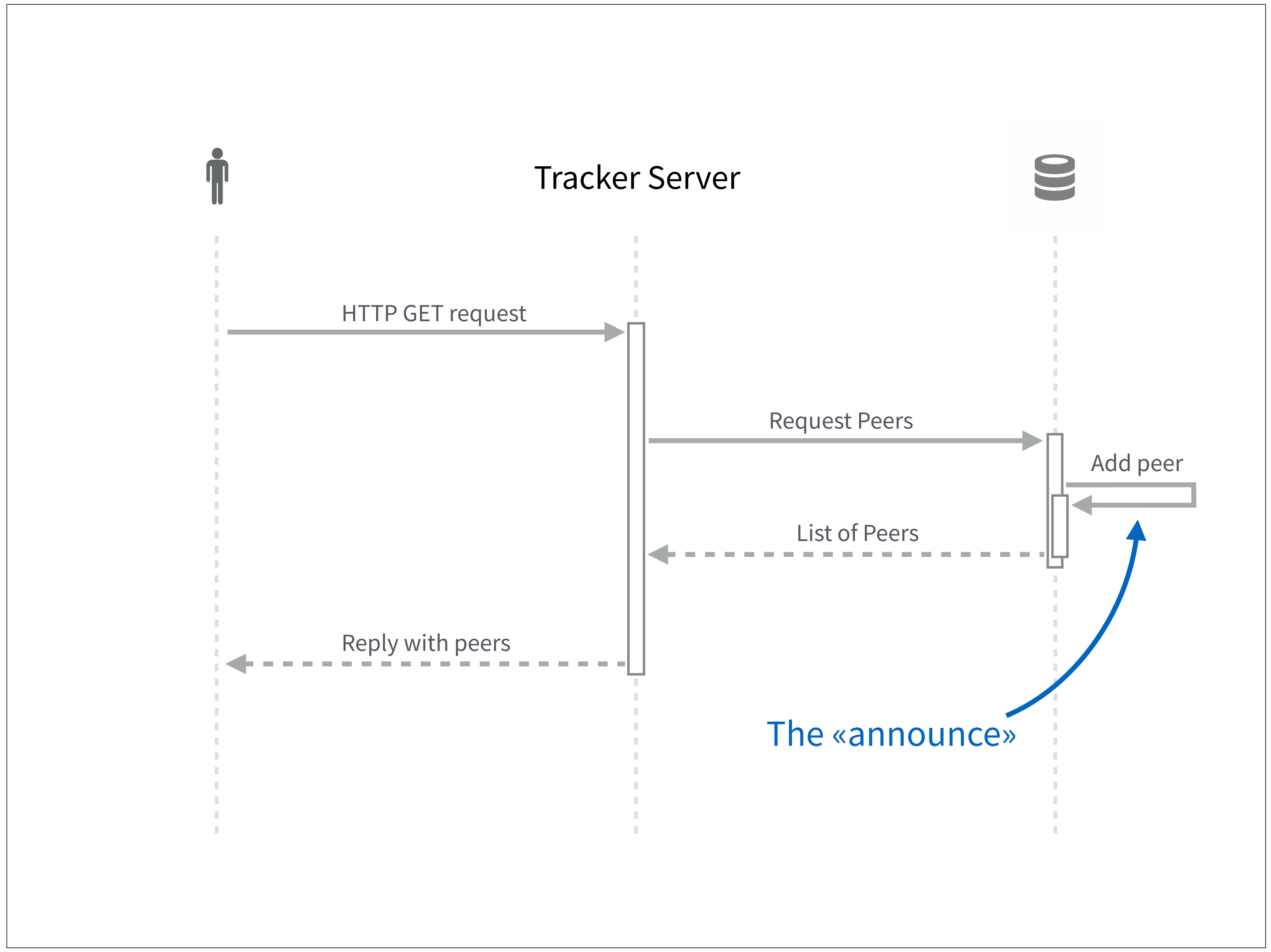



It's a HTTP-service

- We send a HTTP GET request to the tracker server. We have some params on this request.

- Params:
    - a self assigned peer id,
    - the info hash,
    - the port that we accept connections on,
    - status (connecting, stopping, complete)

<click>
The tracker server will fetch a list of peers from its store and send it back to us
<click>

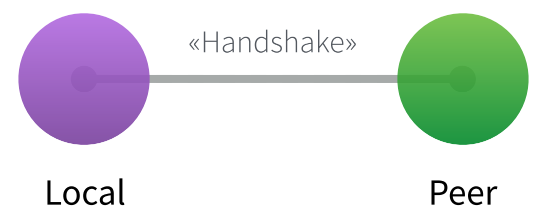



The server will then store the information about us — ready for other peers

This is called an «announce»

Then:
We will now have a list of peers to connect to, and others will get to know us when they do an announce.

# Connecting




We will try to open connections to peers from the list using TCP/IP

As soon as we get a connection to one we will send a BitTorrent handshake

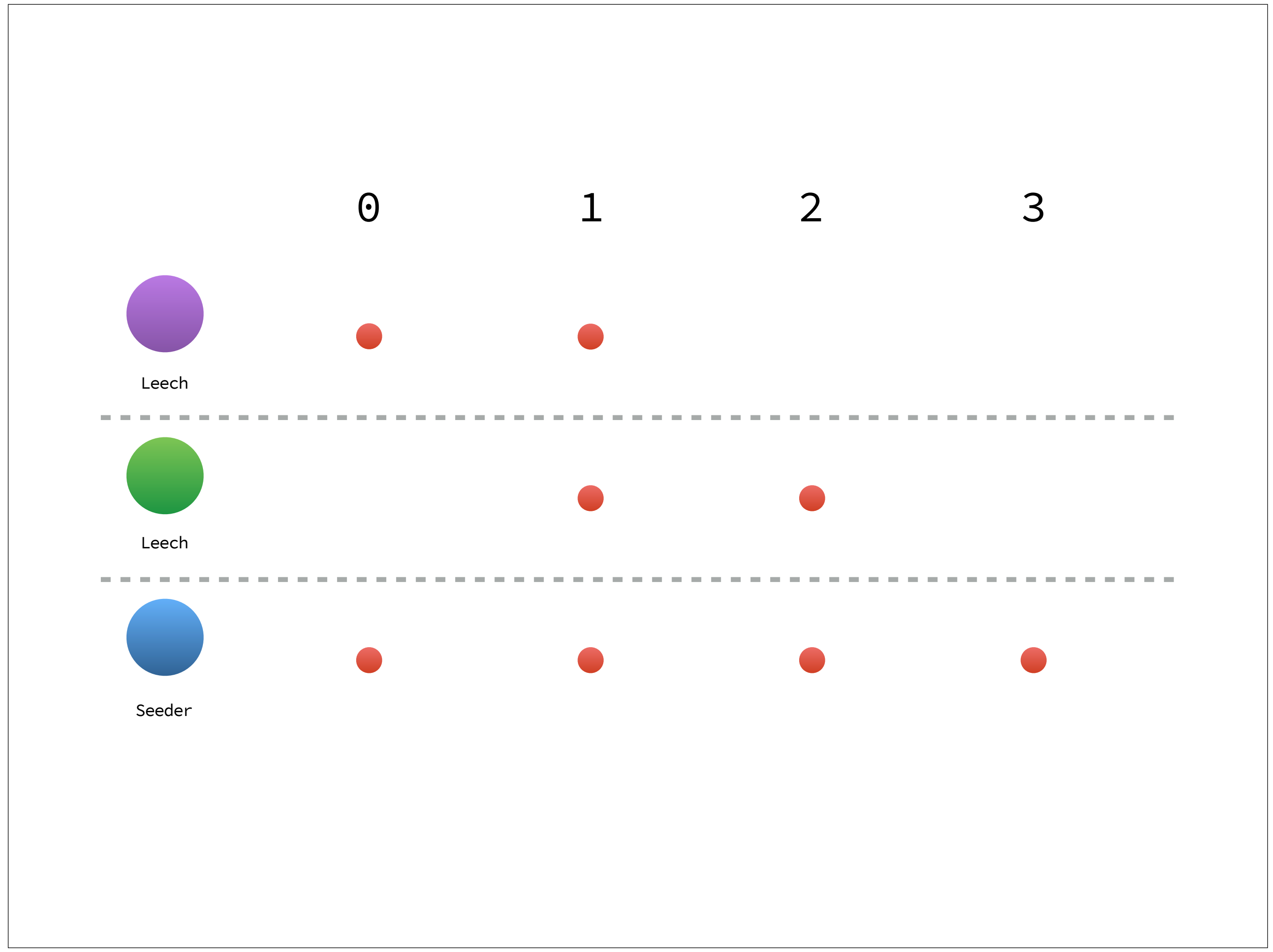
# Exchange

The Peer Wire Protocol

# The Wire

The peer wire protocol support messages for **communicating**

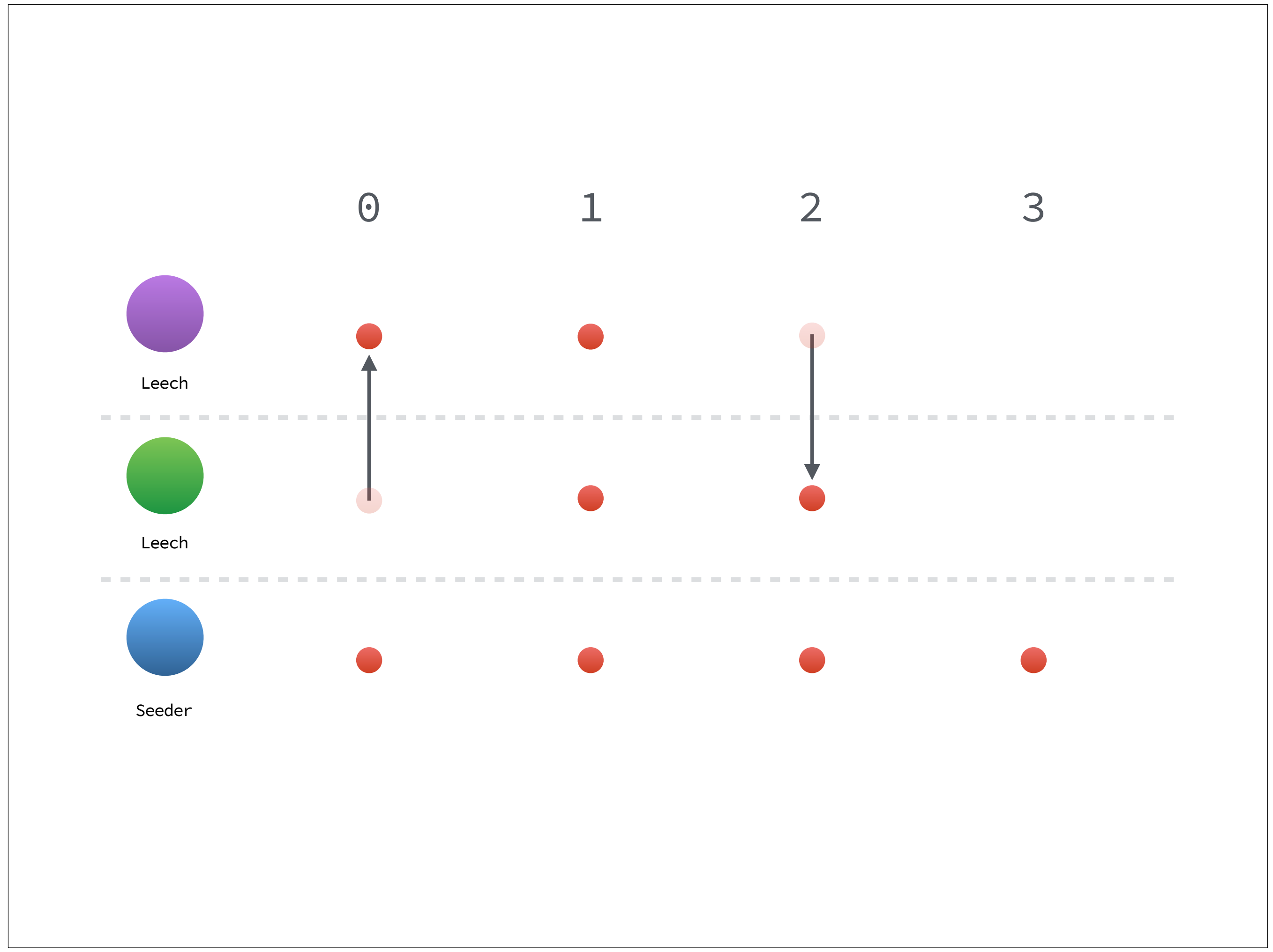
- a peers **internal state**
- as well as its **intentions.** (accepting to share or not)
- and placing piece requests




We have 3 peers

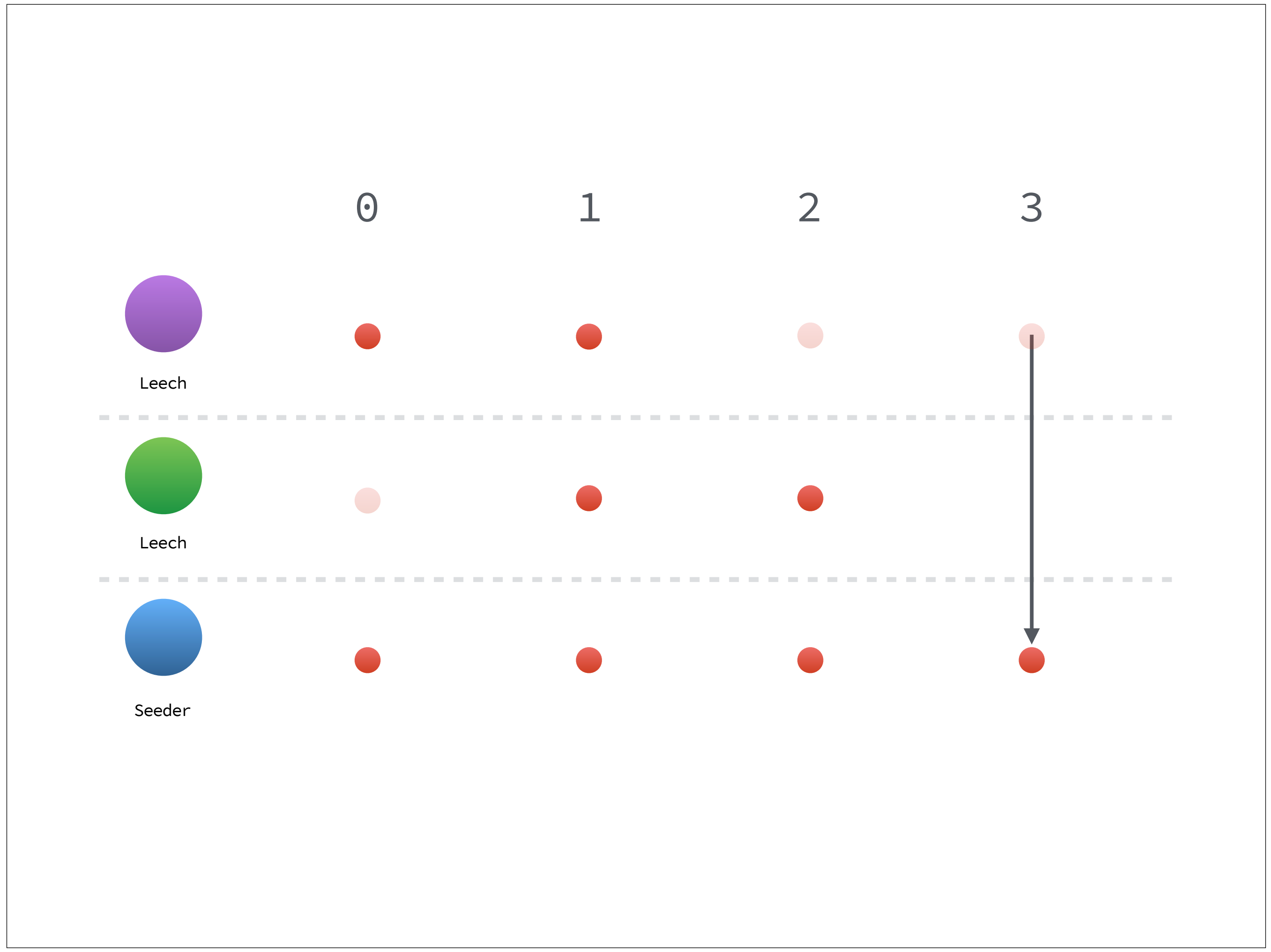
we find a peer who has a piece that we are interested in

Green seems interesting; we place a request for piece 2



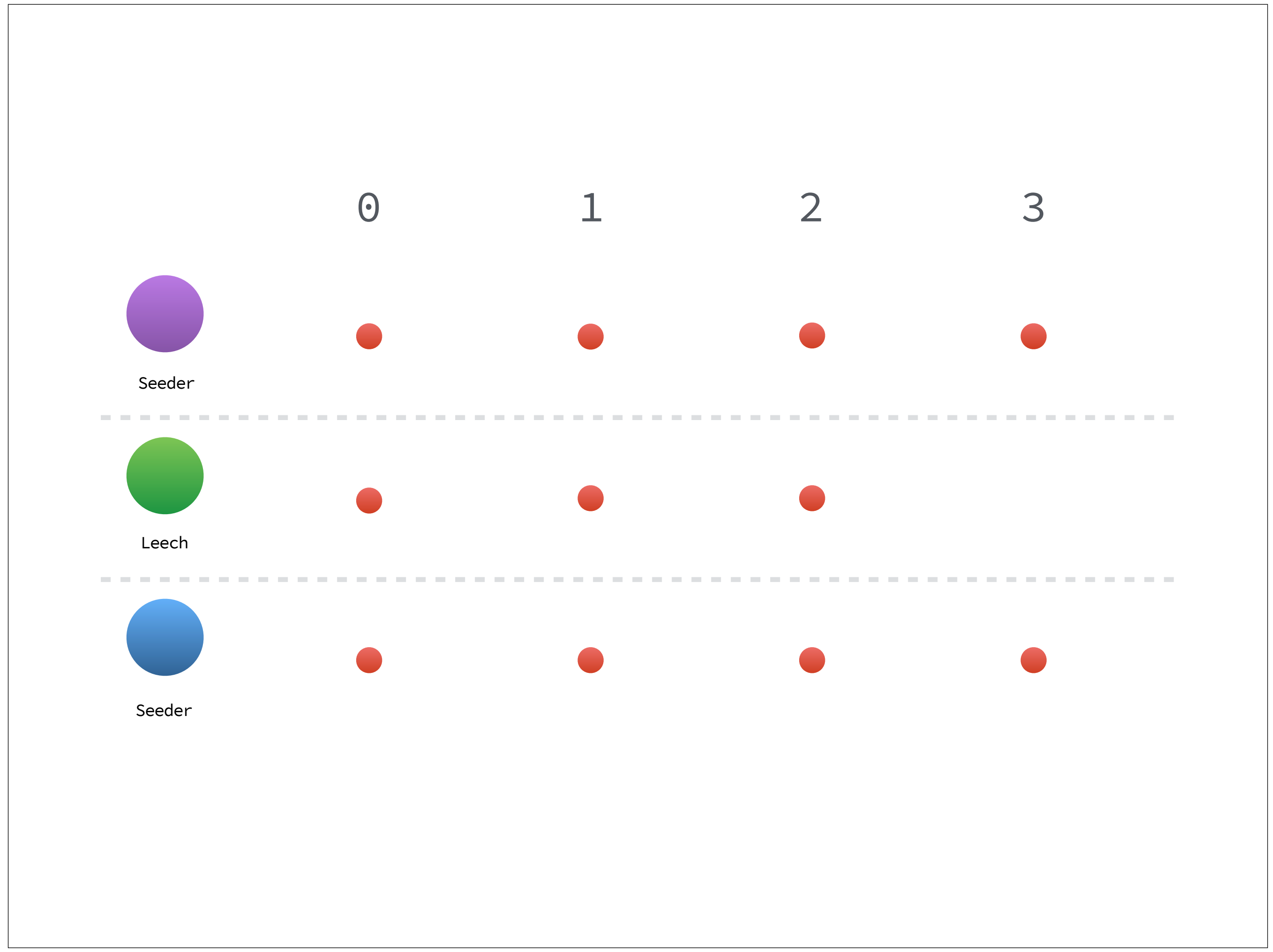

In return they will ask for piece zero

We upload while we download; building good relationships are key




While we get piece 2 from the green peer we will place a request for piece 3 from the blue

Have multiple concurrent downloads going on

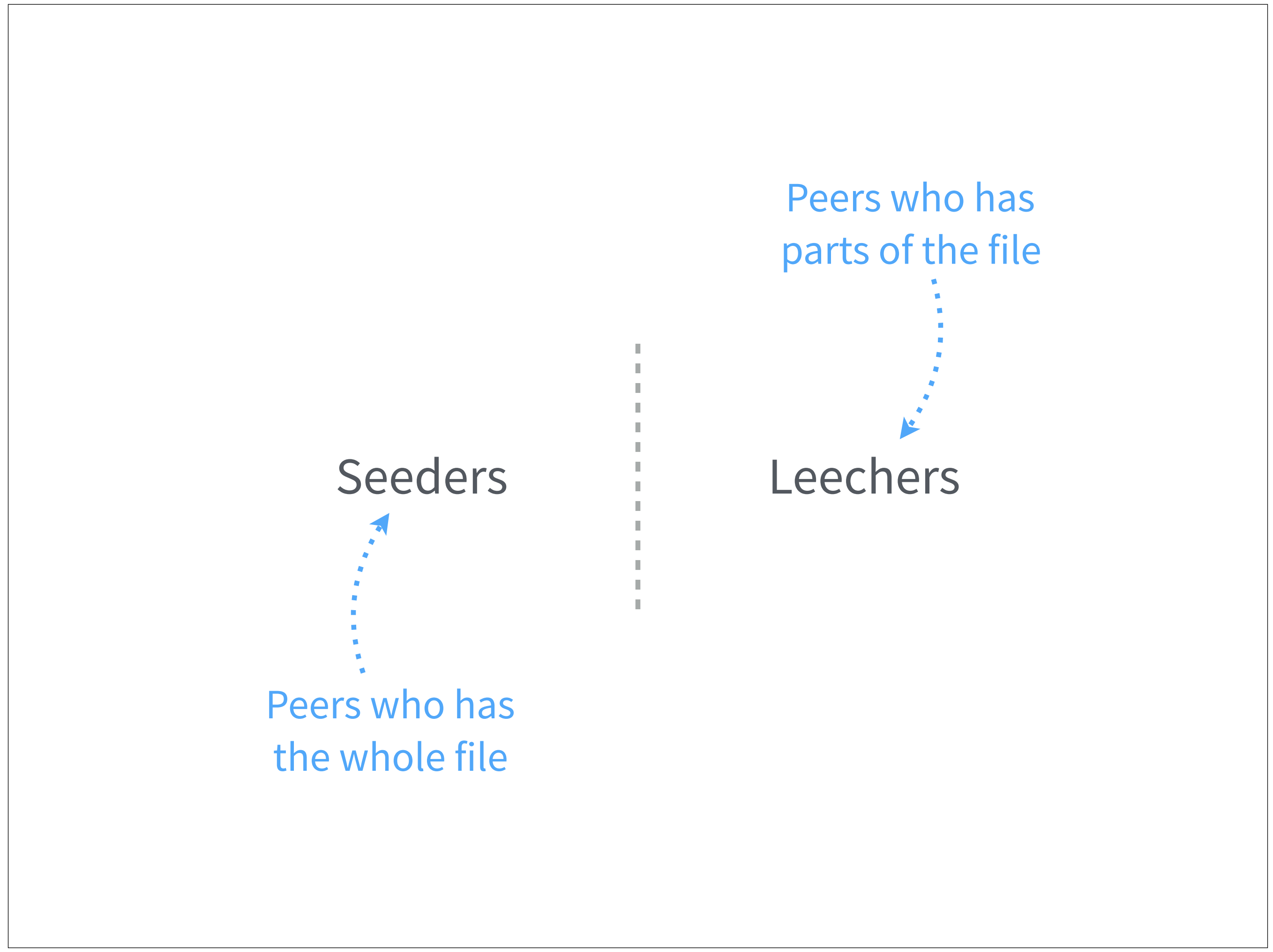



Eventually we will have the entire file!

Conclusion: we can **download** pieces from **random peers** because we can **verify each piece we receive** using the **hash corresponding to the piece.**

but …

«How do we—*who has nothing to share*—get others to share with us?»



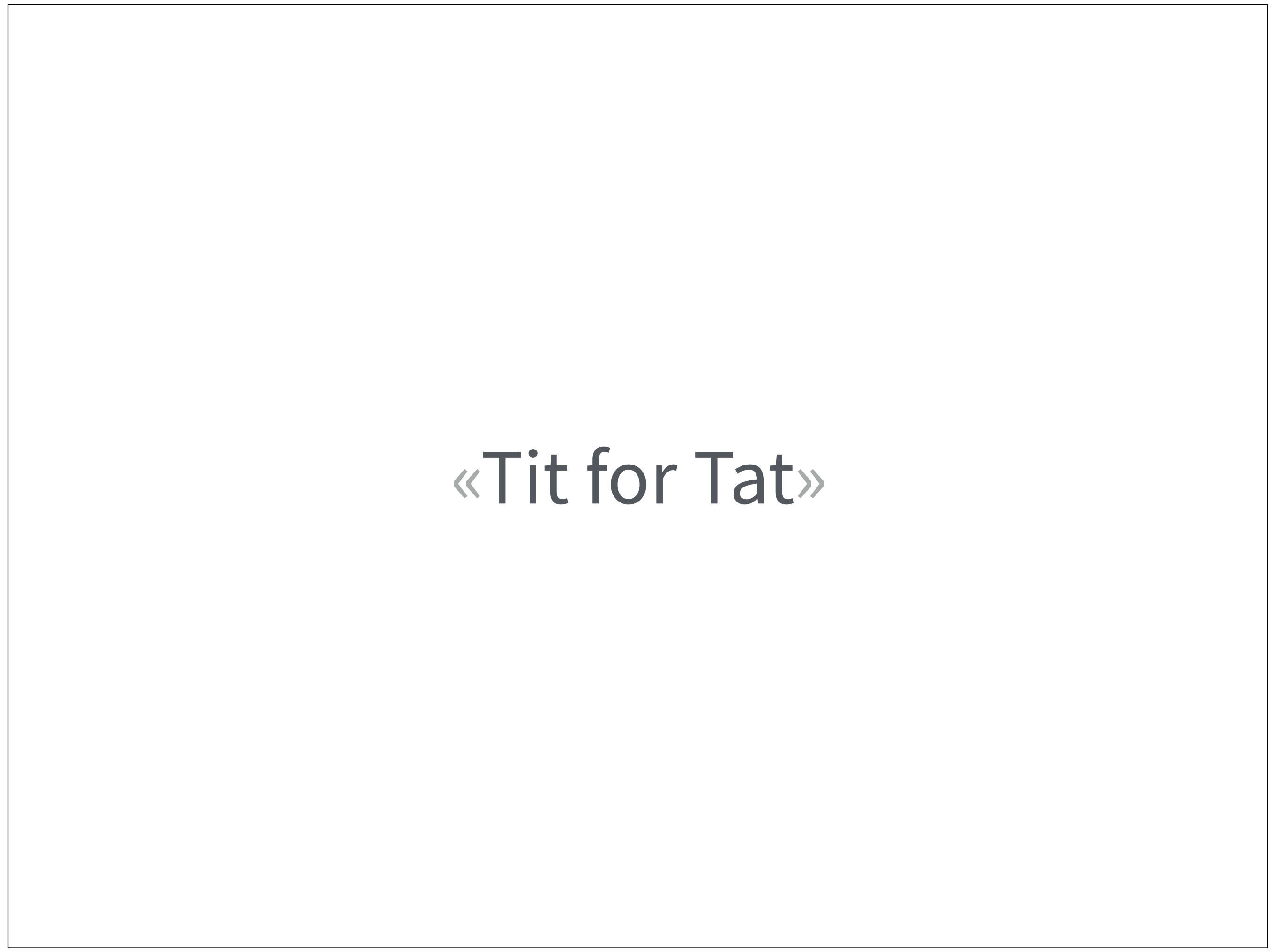

click **Seeders** are peers that has the whole file

**The seeders interest:** making the file public;
**Keeping the file alive**.
They want to **upload as much as possible**.

---

click **Leechers** are peers who has parts of the file

Their interest: **Get the file**,
**Download as much as possible**;
They need to upload as much as possible.




**The more you upload the more you get**

What is good economy for you is good economy for everyone

this economy is referred to as «tit-for-tat»

So why attempt a BitTorrent implementation using Elixir?

- Implementing binary protocols is our home turf
- It is a good exercise for implementing highly stateful systems
- Async messages between many independent and isolated actors
- Peers can disappear at any given time—for whatever reason

«Sounds kinda familiar…»

«BitTorrent is a specialisation of Erlang process semantics»

–Jesper Louis Andersen

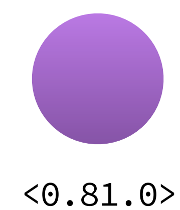
Elixir* just might be the right tool for the job

* any language running on the BEAM, really…

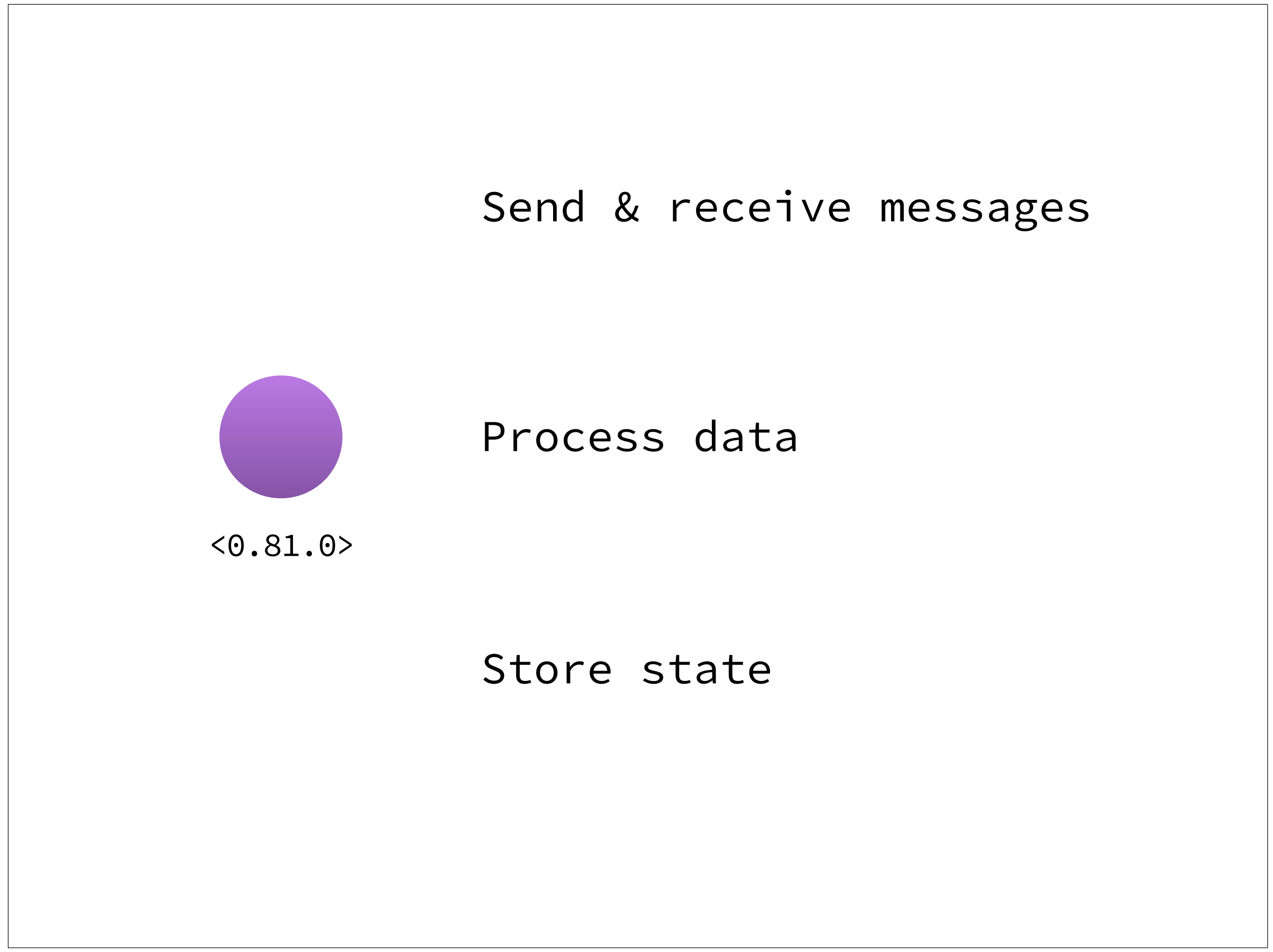
Thus the problem is a good candidate for **Learning Erlang Process Semantics**

# Modelling BitTorrent
## in an actor based system

# Processes and State




It is essential that we talk about our friends: the processes




We got a system that implements the actor model:

That means processes that can:

- Send and receive messages
- process data
- store state

and we get told to…

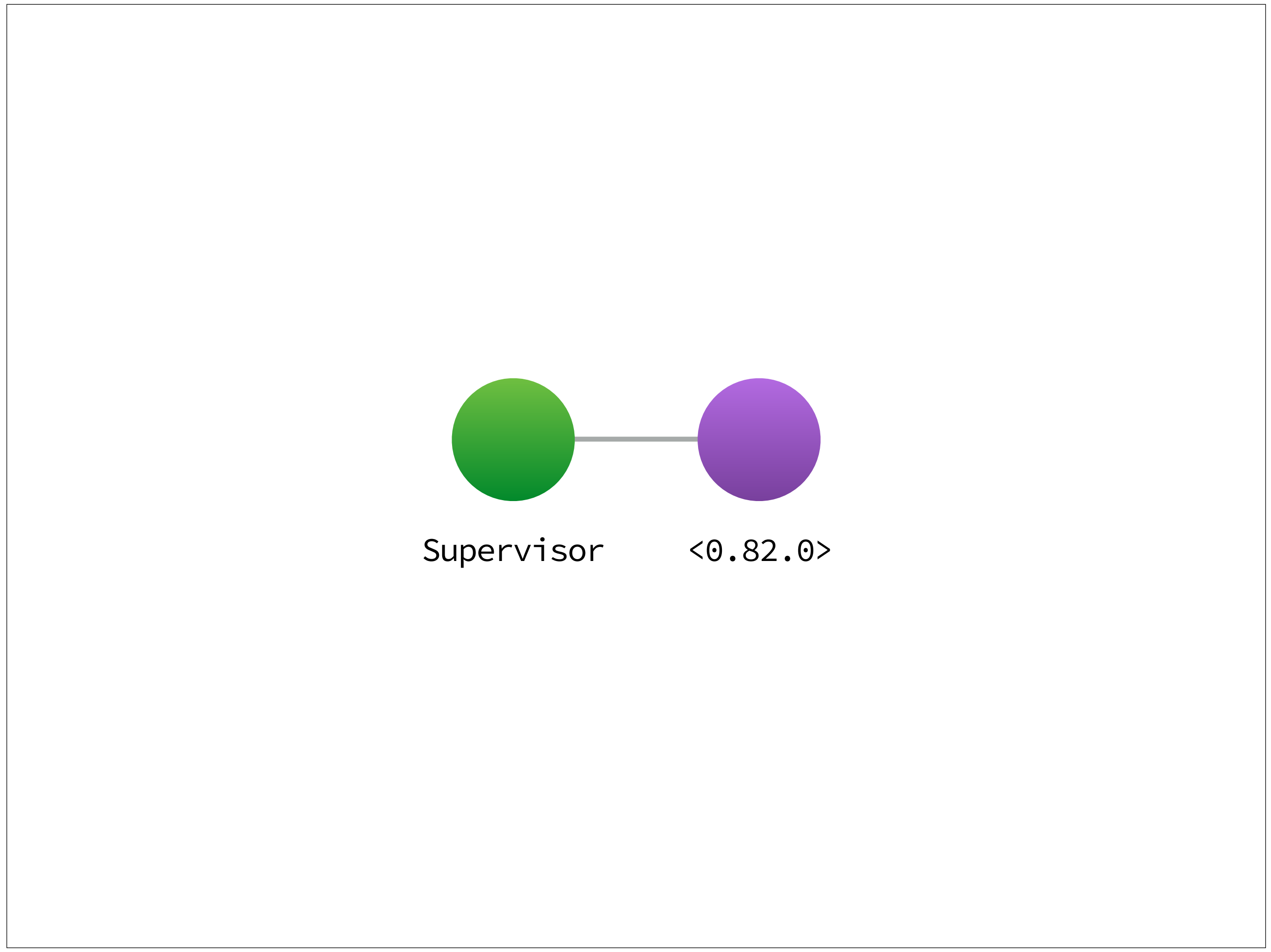

Keep calm?

How could I keep calm?

The process is gone—state and everything!

The messages I sent to the process—gone!

The computing it was doing?—gone.



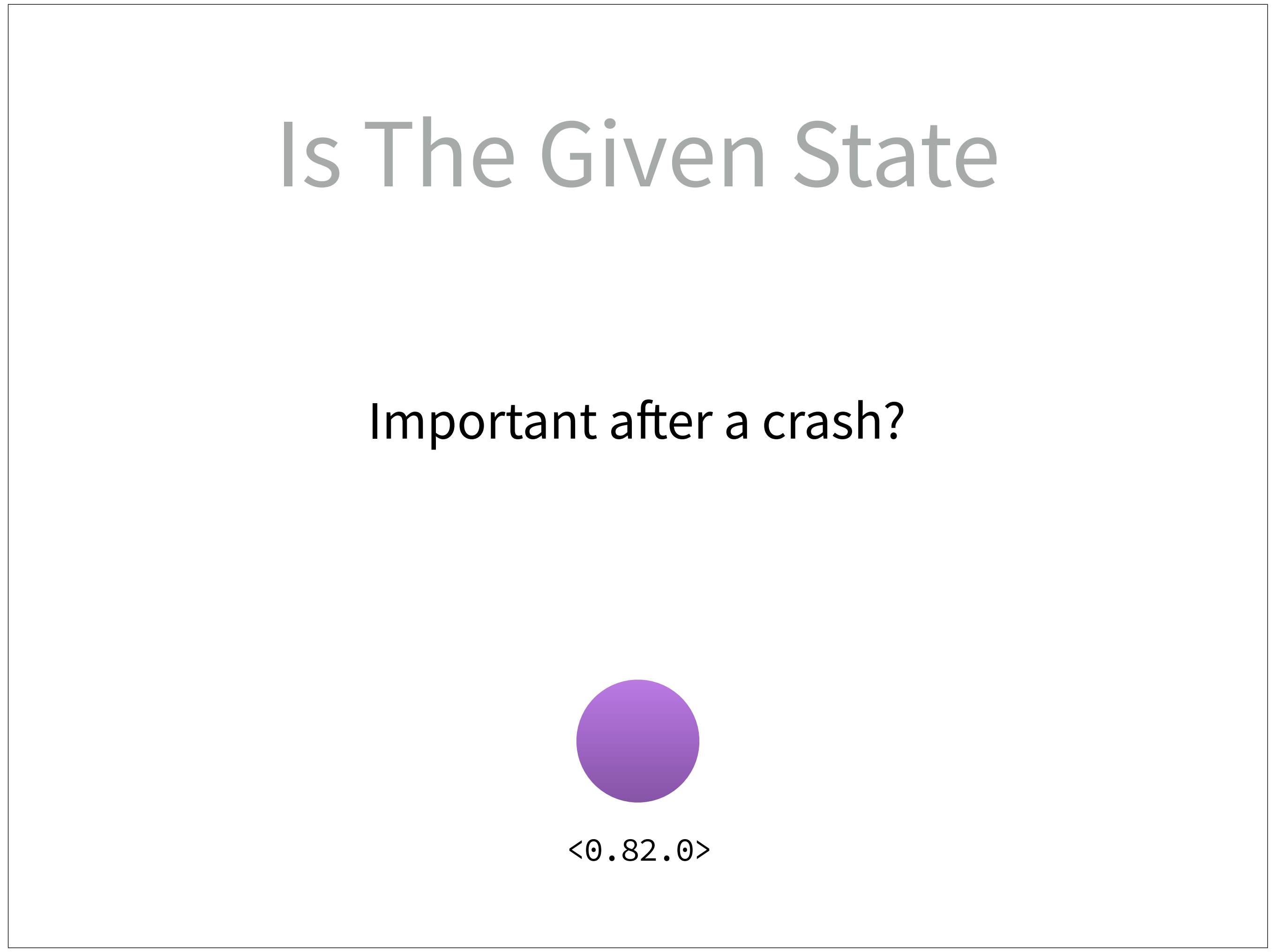

- We use supervising to mitigate process crashes

- Two purposes
  —restart processes
  —contain crashes

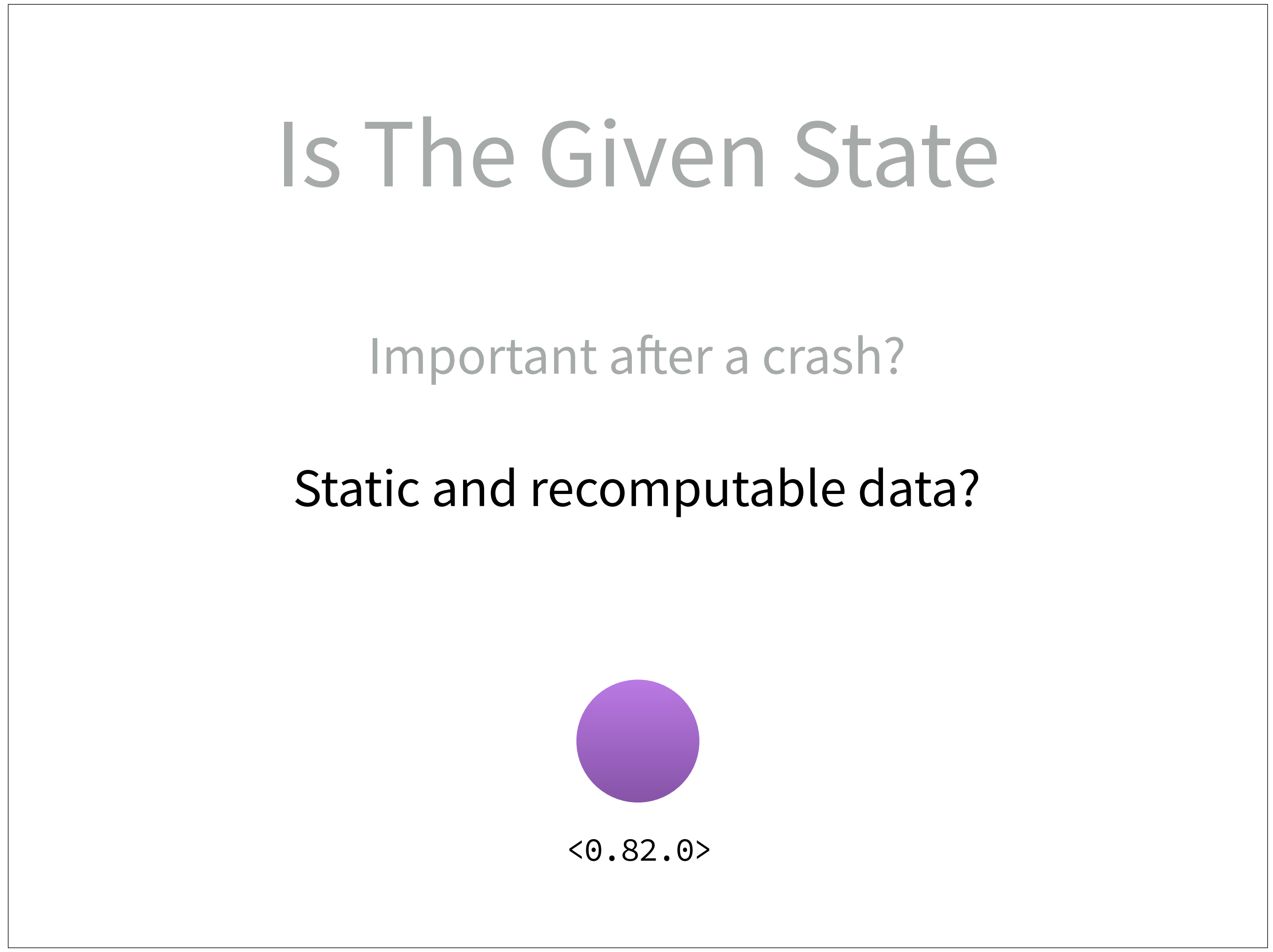
- Restart is good, but **what about the state?**

- We need to **identify the nature of state**,

let's ask some questions about state




- **Important?**
- Did the process store some intermediate result for some another process?
- **Just let it crash and let the controlling process re-spawn it**



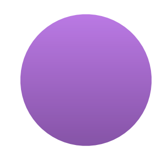

- Did the process hold **static data** that **doesn't change over time?**
  **—**(«*configuration data*» given at initialisation)
- The supervisor will take care of this, **just restart it**
- **initialisation params should restart** that process **and it should rebuild its state.**

# Is The Given State

Important after a crash?

Static and recomputable data?

Accumulated over time?

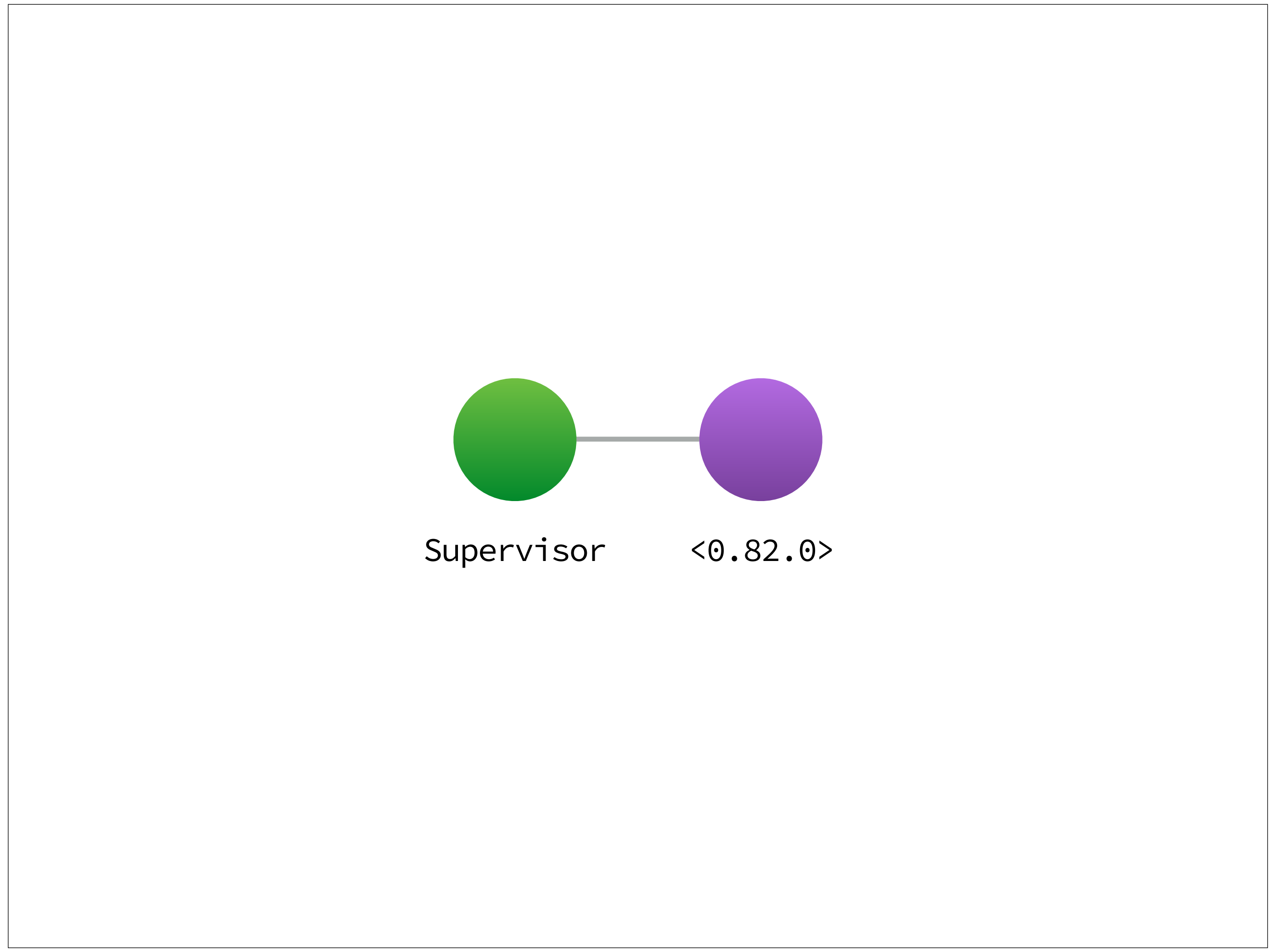



- Is the state **generated and changed over time**?
  —input from users or sensors.

- If this state originates from other *processes in our system* we can *recompute it by querying those processes*.

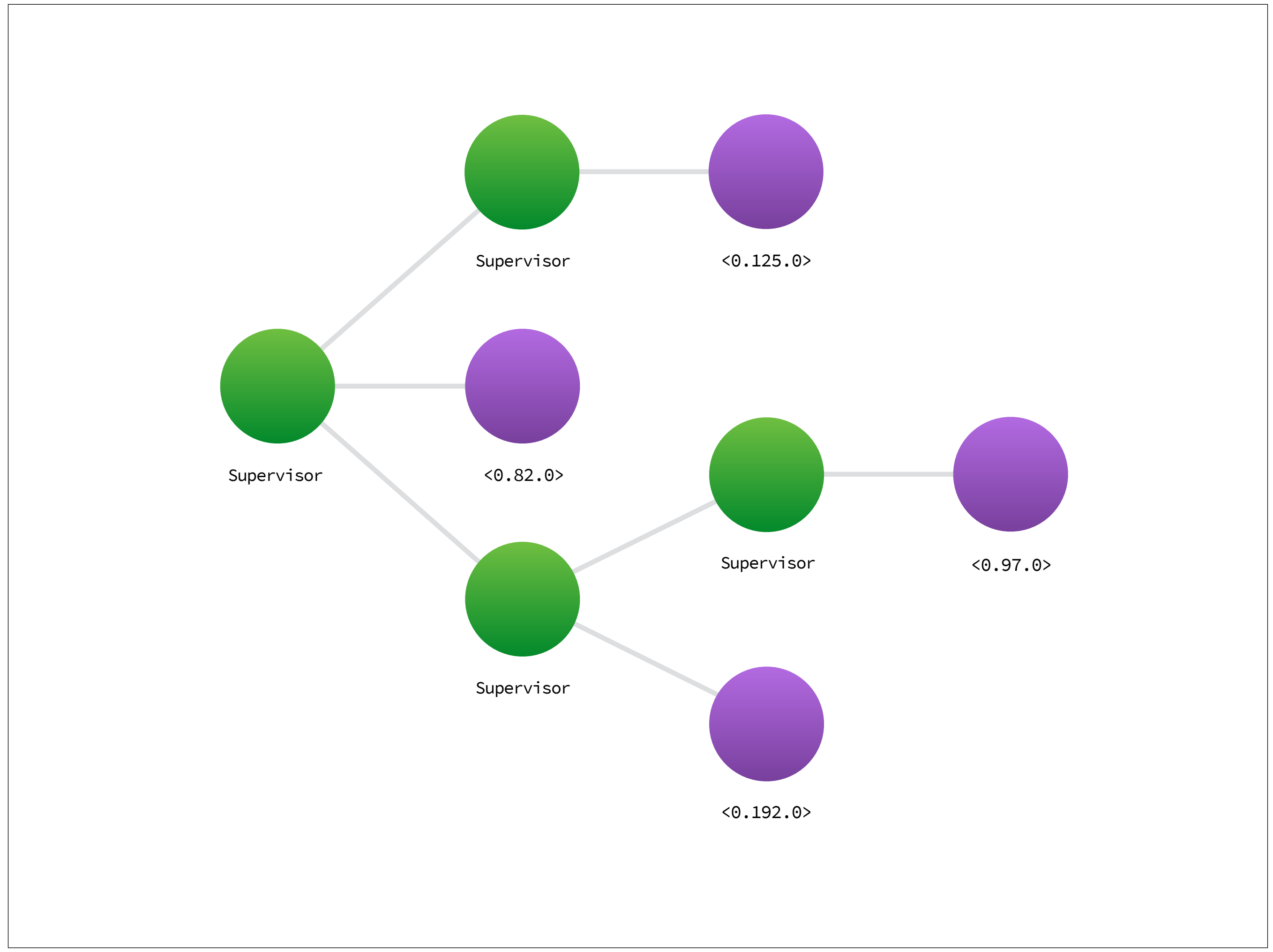
- **If the data can not be computed**; **persist it**.

- **Beware of the supervision restart strategy**; the process crashed for a reason. We need to restart the process from a **«known working state.»**




- Identify category of state helps
  —but we need processes to work together

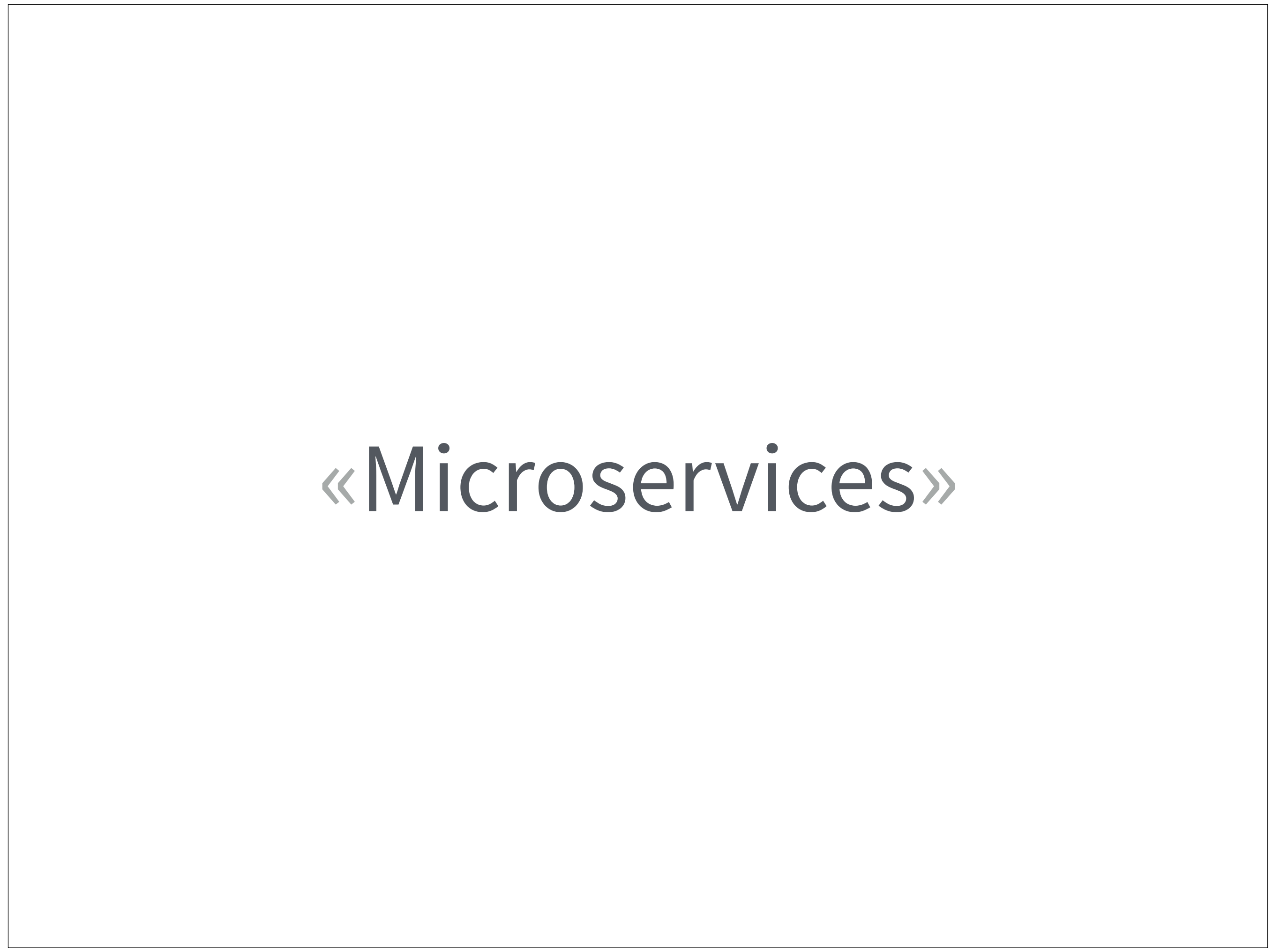
- We need to identify the Error Kernel, the part of the program that *must be correct* for correct operation <click>




- We group related processes together **in compounds** using supervision trees, allowing them to crash together

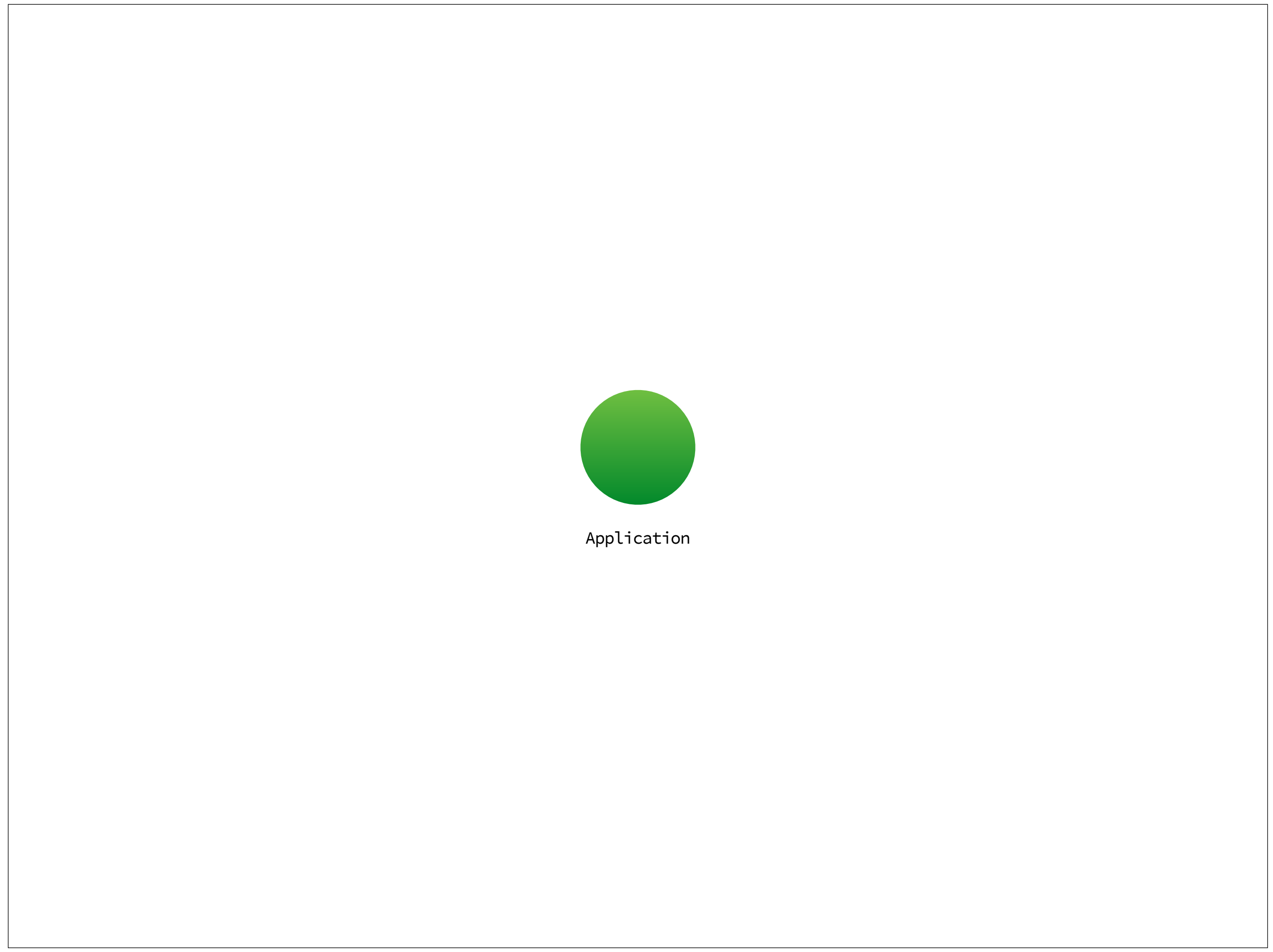
Without affecting the rest of the system

- Every time we have a dangerous operation we will offload it to another process



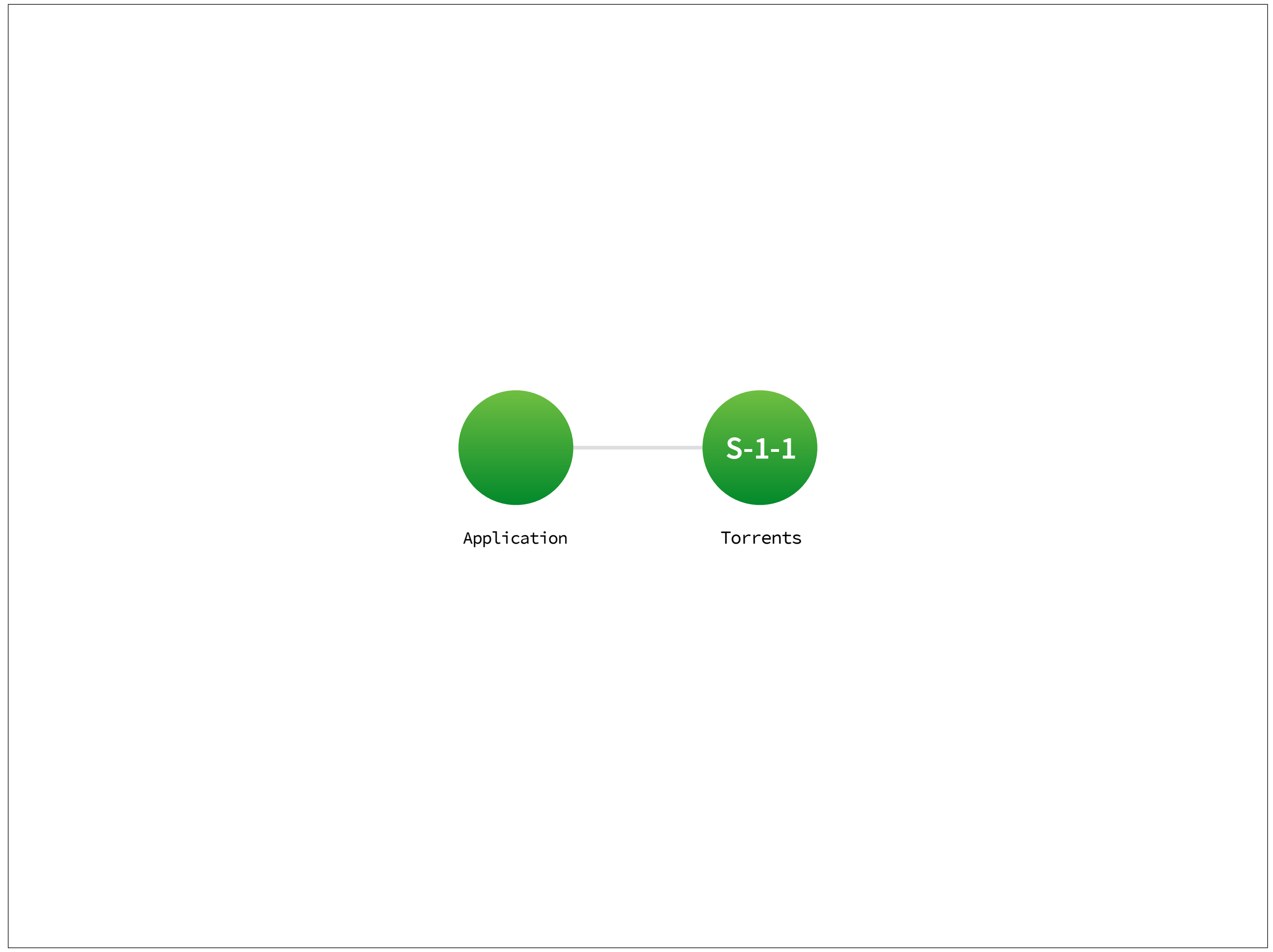

Let's try to apply that knowledge…

Thinking in Micro-services




We will start out with our top most supervisor—the application.

Crash = game over.

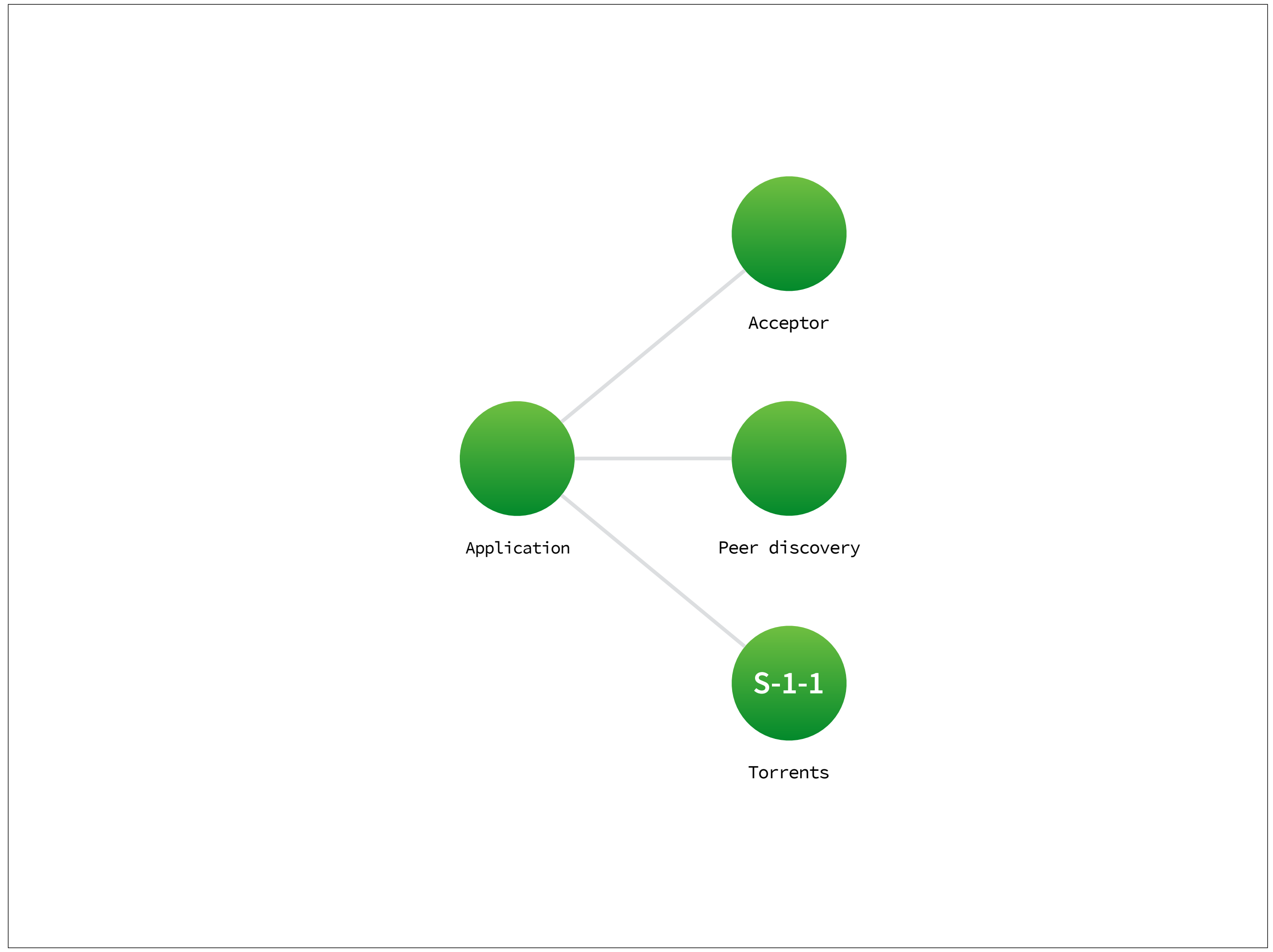



We want to be able to participate in **multiple torrents** at the same time.

A **simple-one-for-one** supervisor seems like the process for the job

this service will **supervise all the torrents we participate in**


We need a service to handle the *peer discovery*




…and ***to be a server*** we need a «acceptor» service
—to **handle incoming connections**

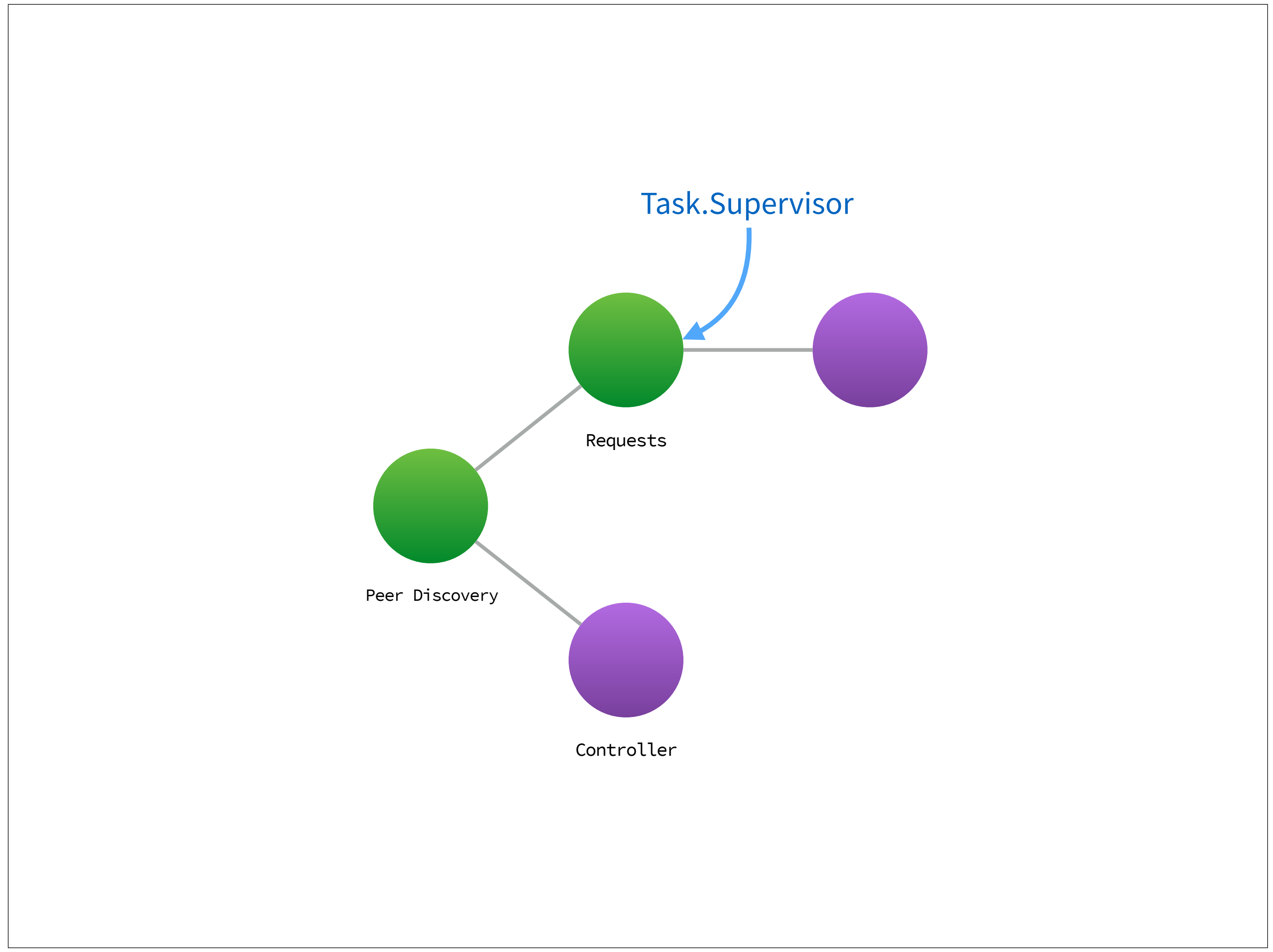
# Peer Discovery

First, briefly, peer discovery

# Responsibilities

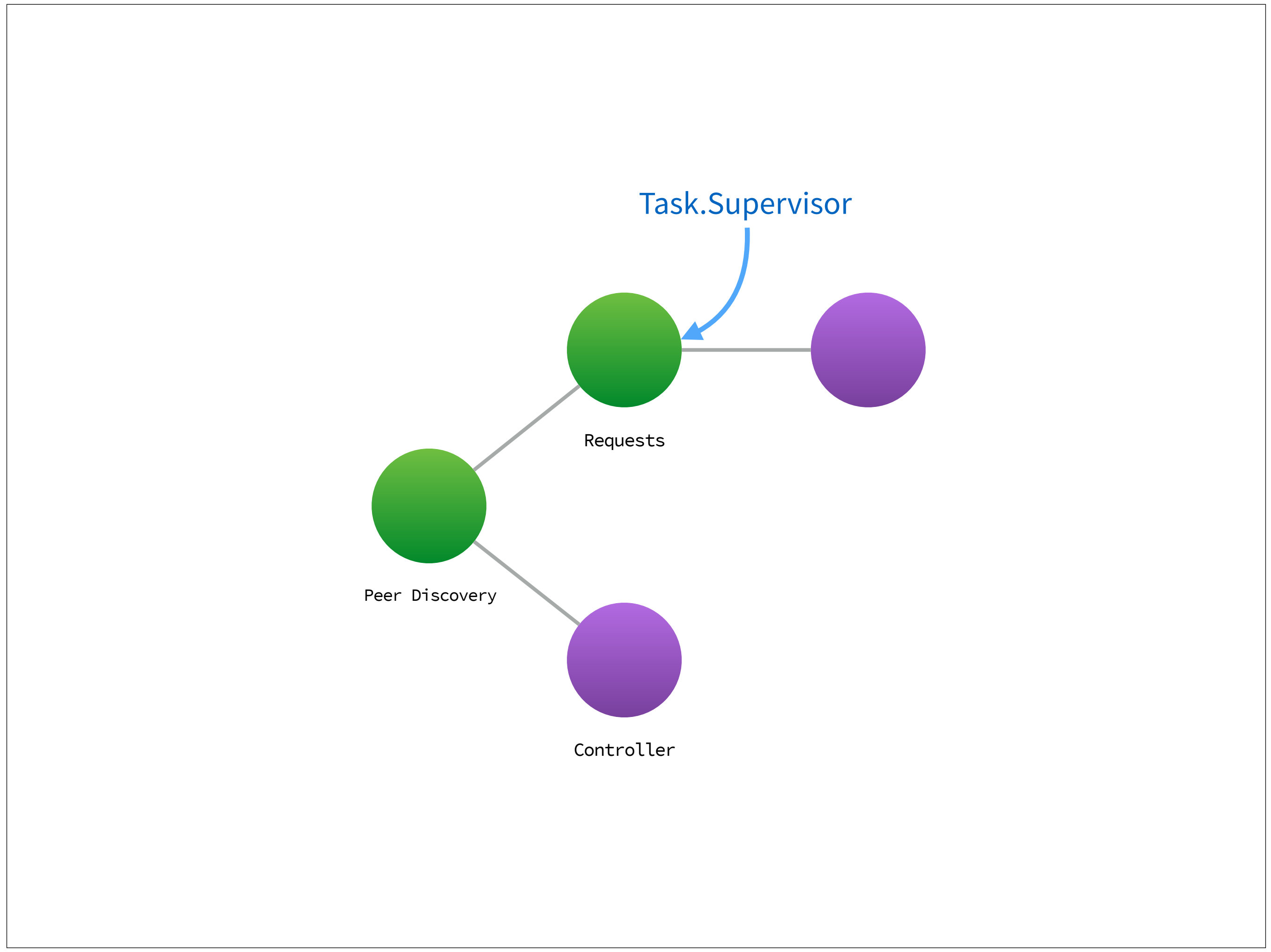
- Obviously, we should be able to ask the service for peers
- Call a tracker service once every 10 minutes—it should not spam the tracker service
- Keep the list of peers it received the last time

We would like to ping the server with our presence once every 10 minutes or so to let them know that we are still there




We could handle this by using a HTTP-client in a task

Controller could call itself every ten minutes
—start a request-task;
—and, keep the peers from the last response in its state, ready if someone asks for it




Crashes happening in requests will be contained

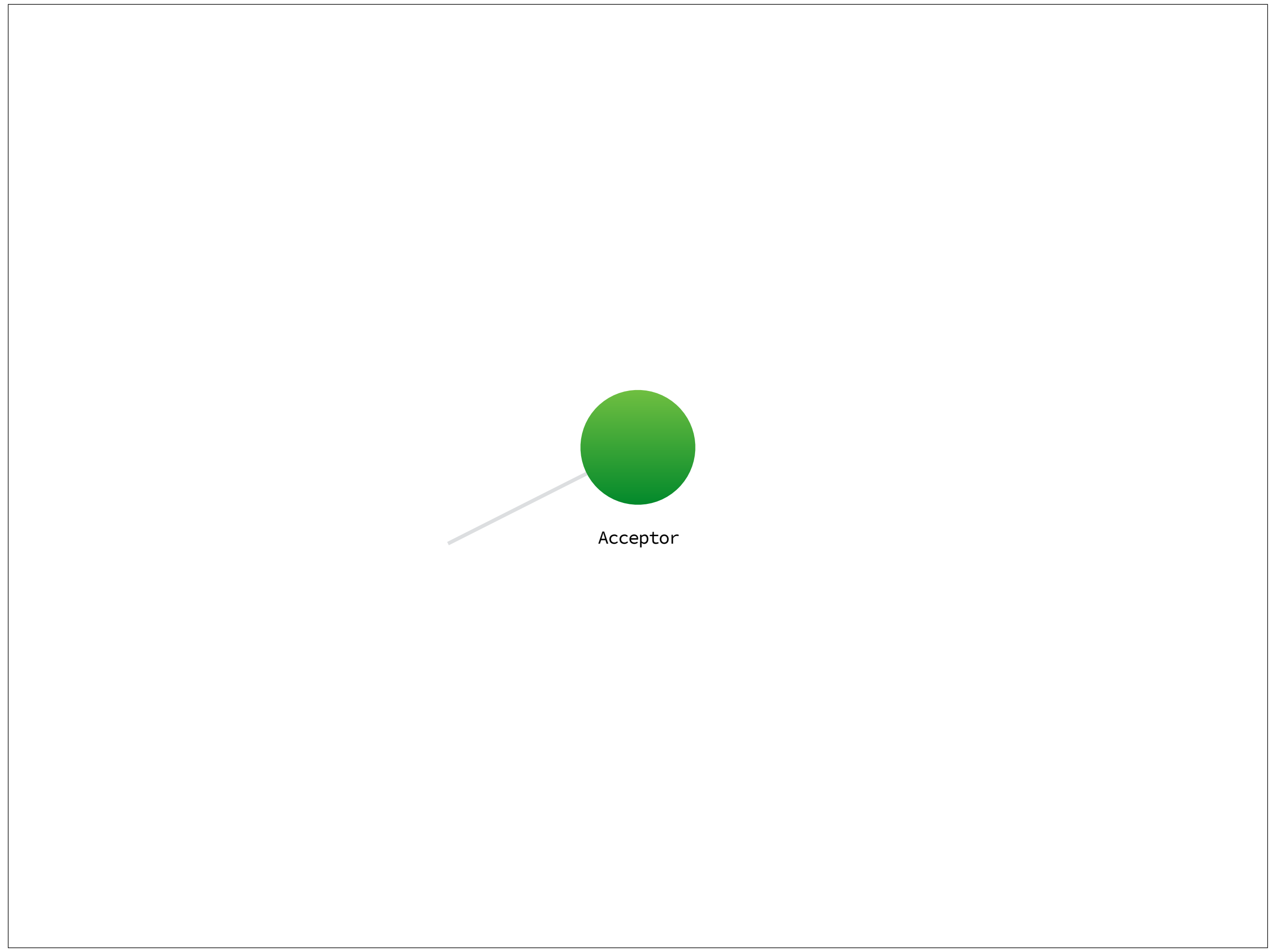
No state to care about:
—the state will rebuild itself on requests
—it is *very* temporary.

in the interest of time…let's keep it at that; the requests needs to use a http-client, there are plenty of them on Hex

# TCP/IP
# Socket Acceptor

Having announced ourself to the world…

we need to **accept incoming connections**




We need the service to implement a TCP socket acceptor pool

> a service that hangs around—accepting connections from the network.

# Responsibilities

- Listen on a port for incoming connections
- Reply to these with the protocol handshake
- If accepted it should create a new «peer process» and hand it the network socket


We could build our own For that I recommend reading:
—«Buckets of Sockets» in «Learn you some Erlang»

**But!**

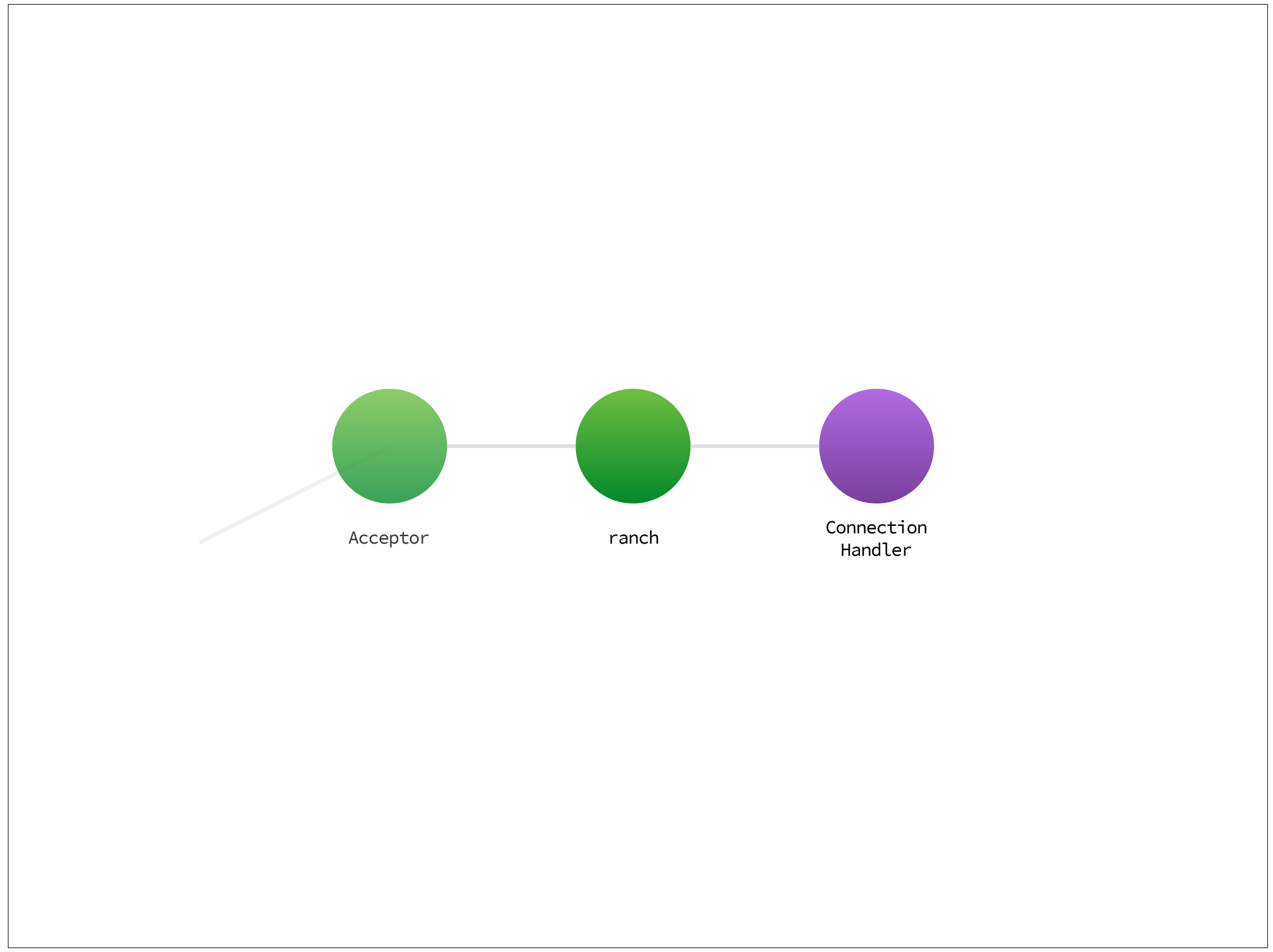
> We want to build a BitTorrent implementation, so let's just add a dependency…

# :ranch

Ranch is a socket acceptor pool for Erlang.

It has fine documentation and there are a tons of blog posts on the topic.

So I'll just give you a quick overview.




**The connection handler**

Ranch acts like a supervisor
—that spawns a connection handler process

---

This process allows us to define our own connection handler module

Here we can **specify what it means** to **accept or decline connections** for our use-case

# The Connection Handler

- Check if the given peer is blacklisted
- Make sure the peer perform a valid BT handshake
- Make sure we are tracking the requested torrent
- Check if we already have enough peers for the given torrent

let's look at the BitTorrent Handshake

# BitTorrent Handshake

`68 bytes`

it's exactly 68-bytes, always

# BitTorrent Handshake

```
<<19, "BitTorrent Protocol",
  0, 0, 0, 0, 0, 0, 0, 0,
  info_hash::binary-size(20),
  peer_id::binary-size(20)>>
```

1) a length prefix, **19 bytes**,
2) followed by the string "BitTorrent Protocol"

The designer did this in the hope other protocols would adopt it
—this makes protocols really **easy to distinguish** from each other

# BitTorrent Handshake

```
<<19, "BitTorrent Protocol",
  0, 0, 0, 0, 0, 0, 0, 0,
  info_hash::binary-size(20),
  peer_id::binary-size(20)>>
```

3) Then 8 reserved bytes.

Used to extend the BitTorrent protocol with new capabilities.

The peers can negotiate rules of protocol communication here.

But this is outside the scope of this talk…

# BitTorrent Handshake

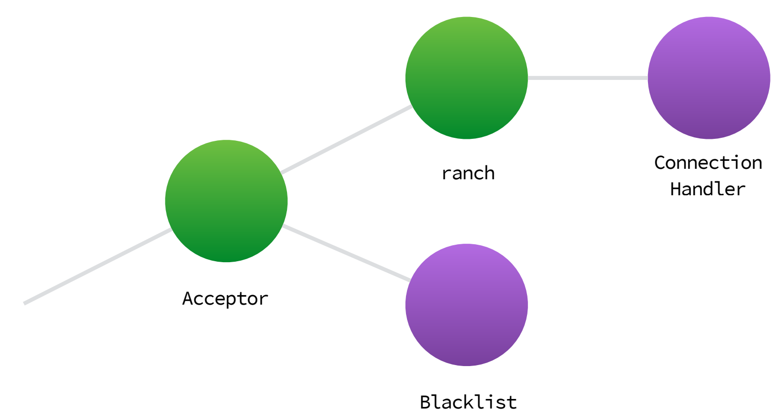
```
<<19, "BitTorrent Protocol",
  0, 0, 0, 0, 0, 0, 0, 0,
  info_hash::binary-size(20),
  peer_id::binary-size(20)>>
```

Then we get the **info-hash** the remote want to communicate
—and their self assigned **peer id**.

—–

*If we know about the requested info_hash*
—we will send them a similar handshake
—and accept the connection

This should be it for the handshake.




…Getting us to this.

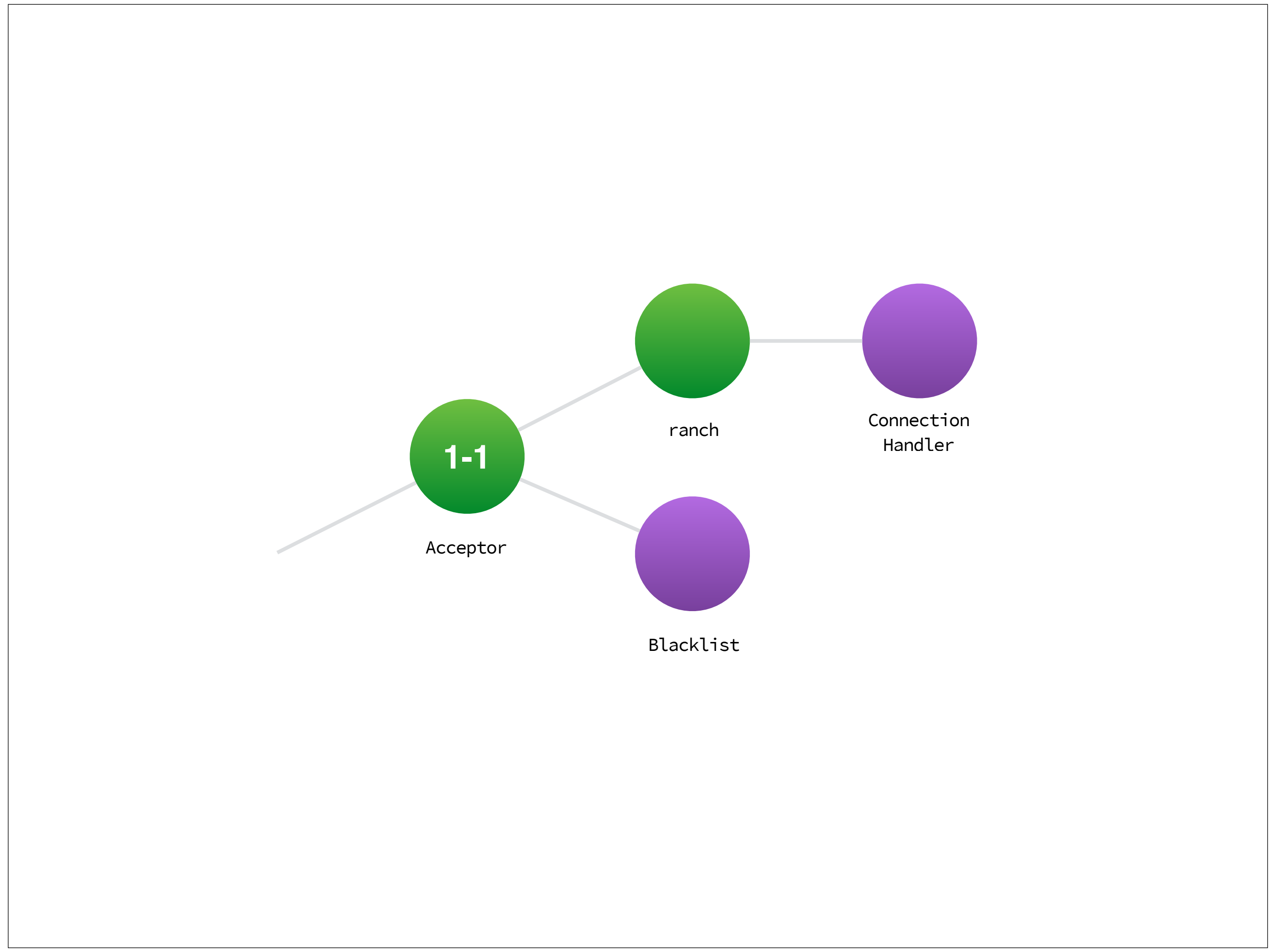
We have a **Blacklist** process, which *can be* an **Agent** storing a **Map**.
—If a peer violates the protocol they should get on this list
—connection handler will consult this list when accepting peers

___

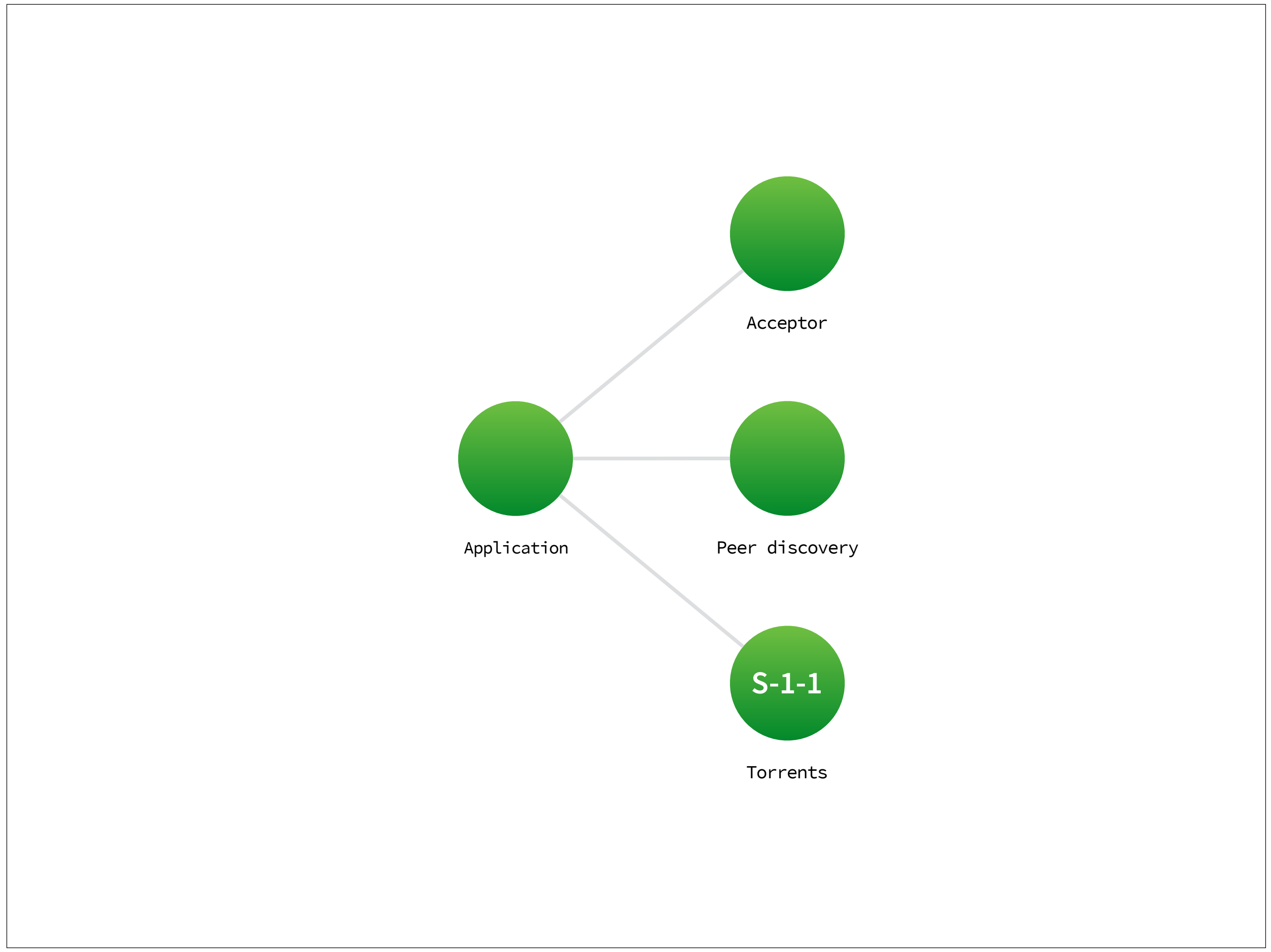
**Let's review failure cases**
Ranch crash = *static config data, just restart it as if nothing happened.*

Connection handlers = *off loaded to another process*

Blacklist crash = *not the end of the world…just start a new black list*




**One for one** should be a sufficient restart strategy for the acceptor



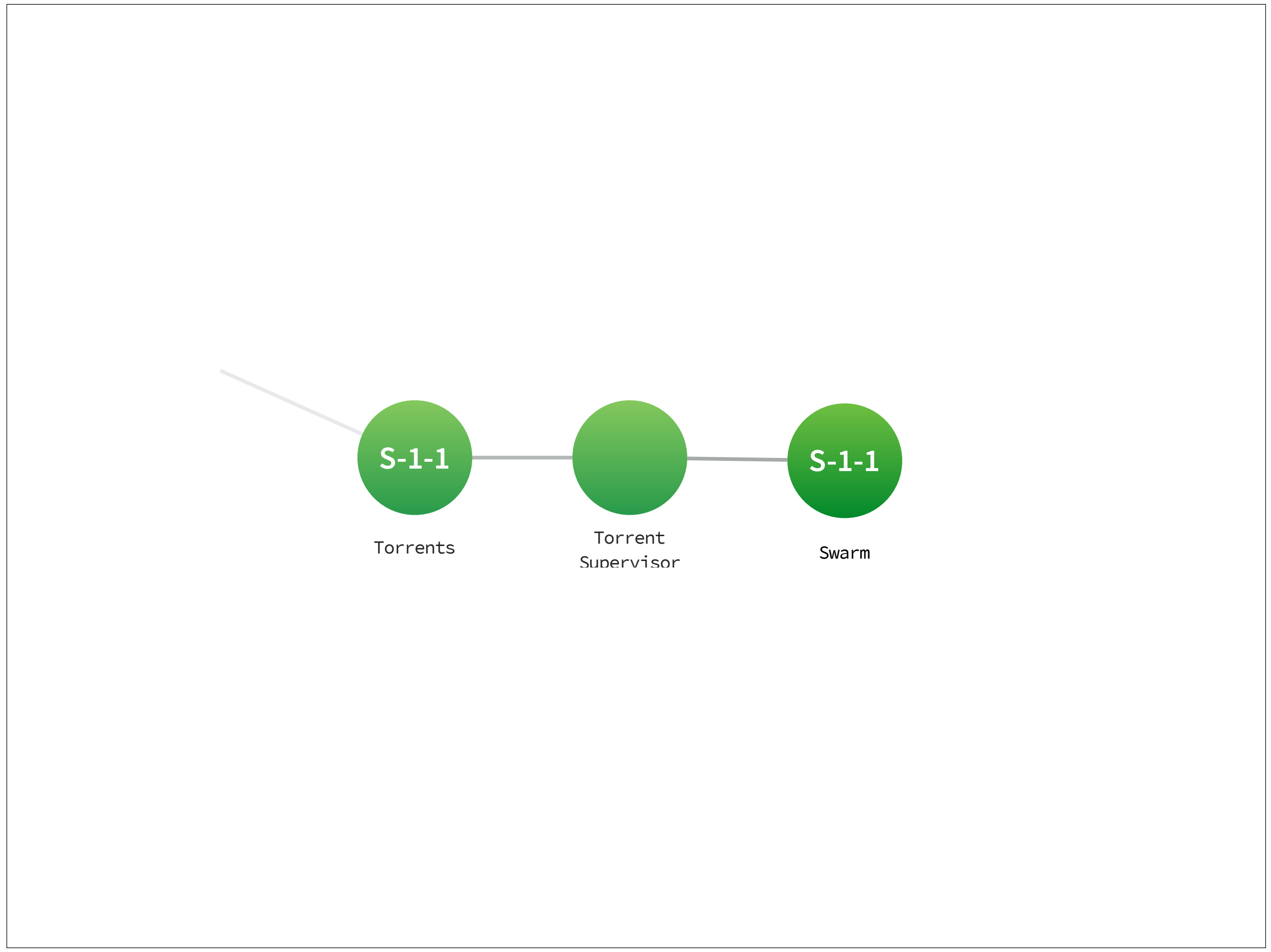

**This should take care of accepting inbound connections.**

We will now need to hand over the peer to the correct swarm

…so let's look at the torrent-service

# The Torrents Service

The torrents service will consist of some sub services

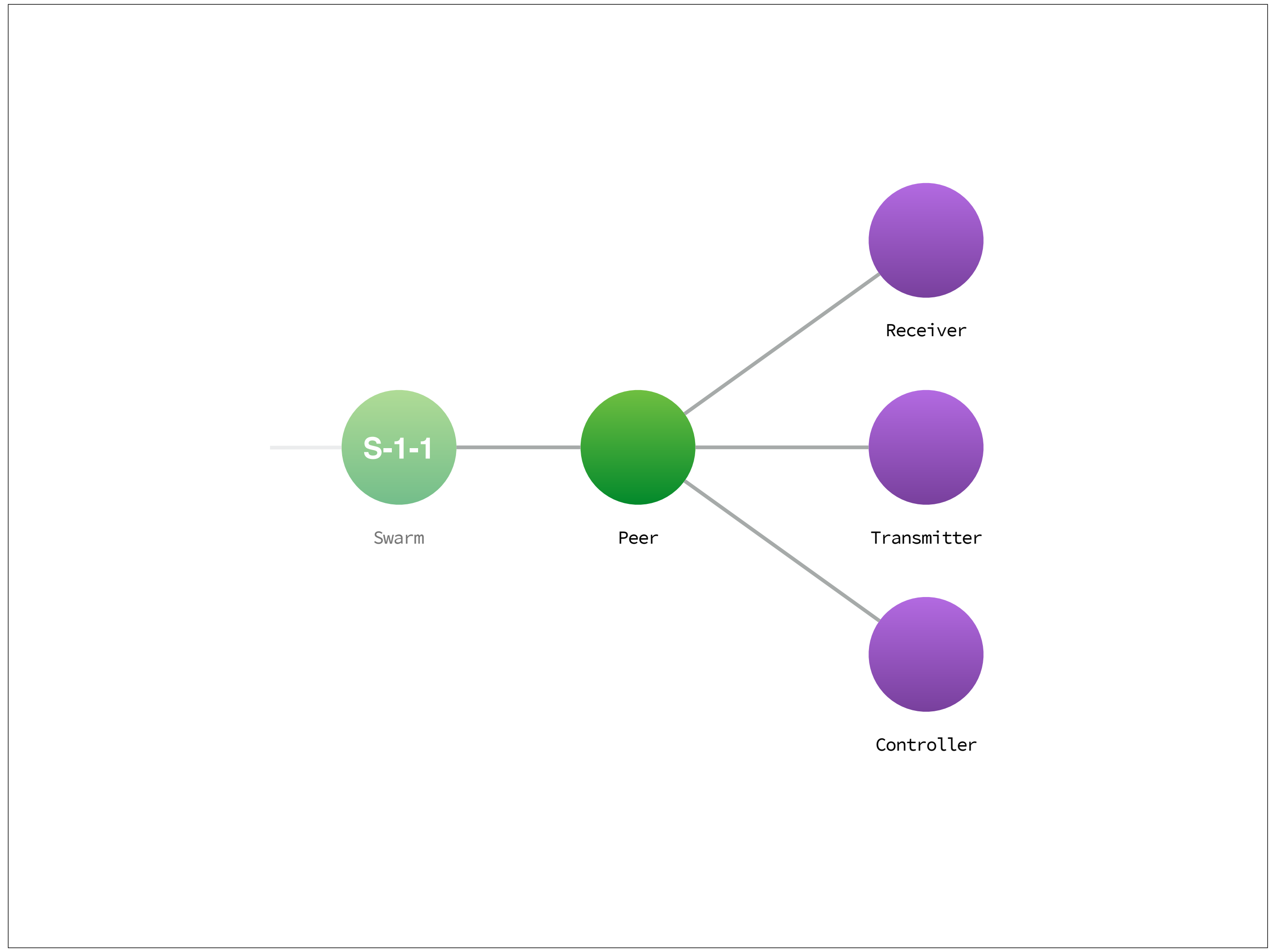



We need a place for the connected peers, and many of them

> let's call it «the **swarm**»

# Swarm and the Peer

First the service handling communication with the peers




Peers will consist of 3 processes:

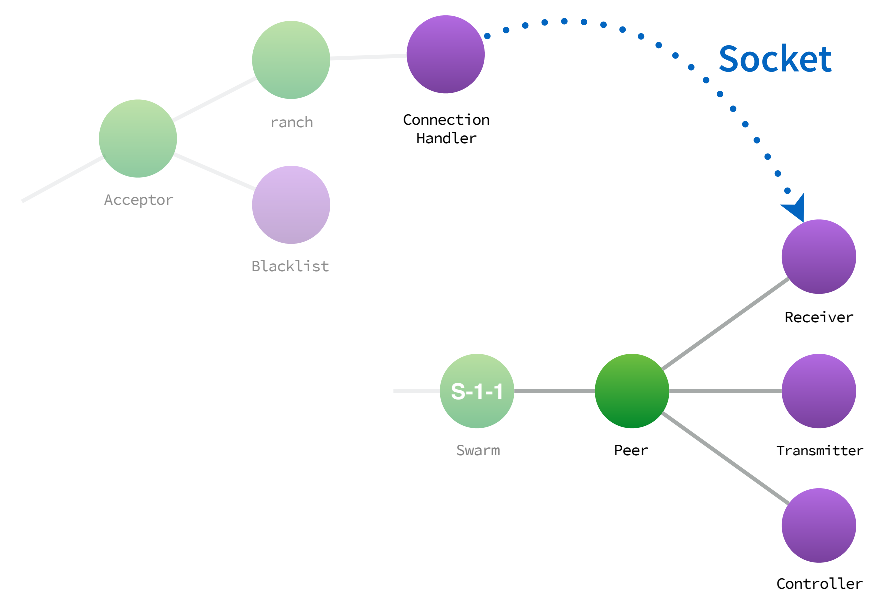
**A receiver and a transmitter**
—because we would like to receive and transmit at the same time

**The controller**
— will keep peer state
— and make decisions about incoming data

# The Receiver

To be able to handle retrieval of data…

# :gen_tcp. controlling_process/2




We need to hand over the network socket from the accepter

**This will allow it to read from the socket,**
—and errors will be handled by the process


and;

We will set the socket to **passive mode**, allowing us to specify how much data we accept, and when.

**And be sure to accept binary data**
—So we can pattern match on the data

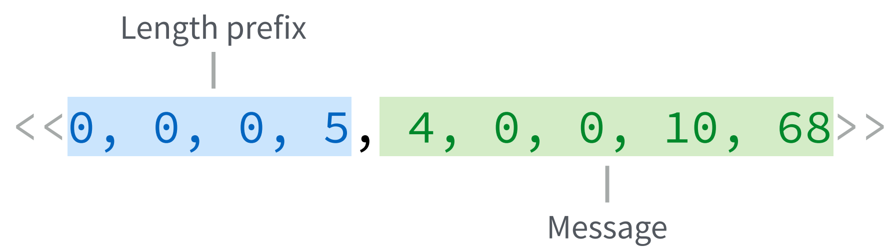
# Peer Wire Messages

```
<<0, 0, 0, 5, 4, 0, 0, 10, 68>>
```

So, let's have a look at the data we are going to receive

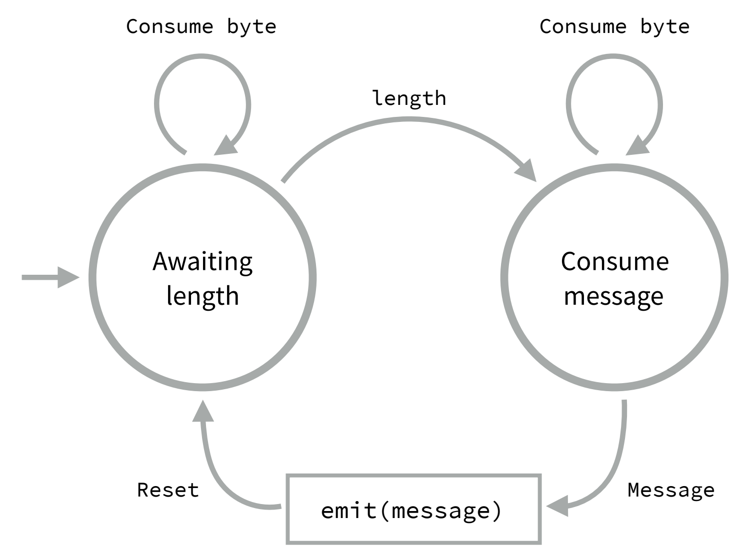
> *We will receive an endless stream of length prefixed messages*

# Peer Wire Messages




First bytes
—a four byte big-endian—
describes the length of a coming message.

Followed by the message

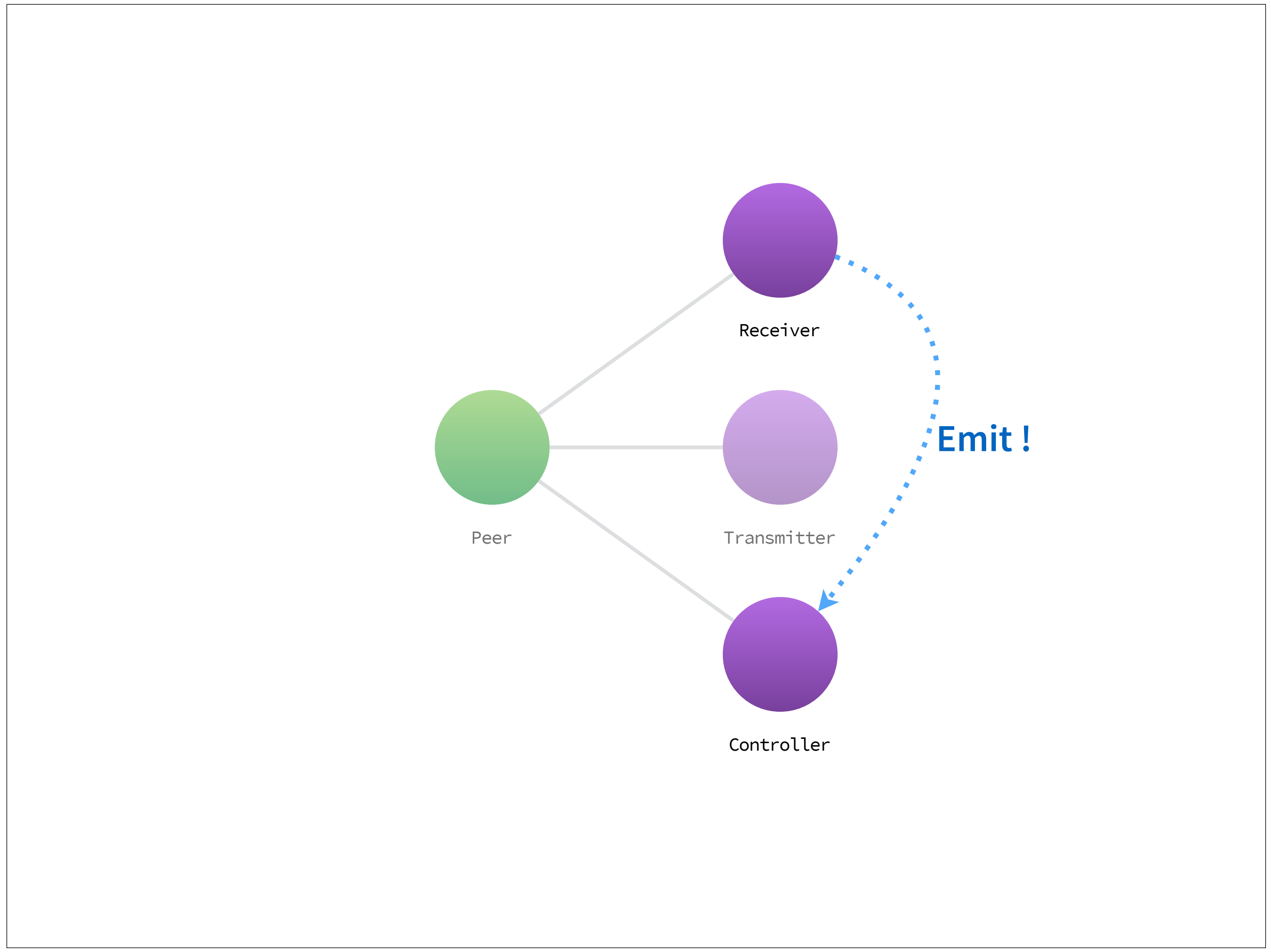



- Personally I like to use a state machine for this kind of thing

—

We start out: Looking for length
when we know the length we consume the «message»—of length bytes

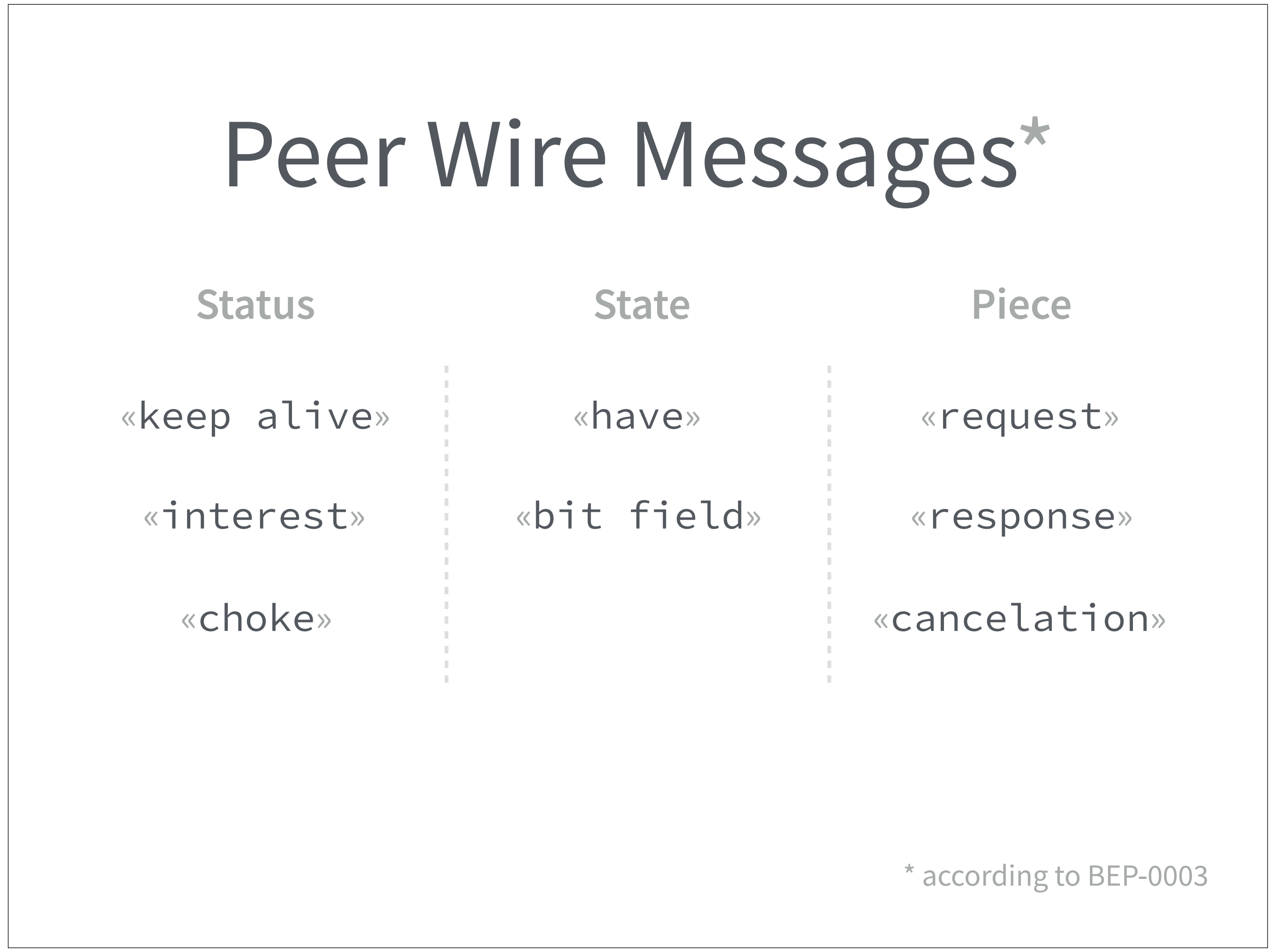
on retrieval we will;
— decode it
— send it to the controller.
— reset the state machine—wait for the next message




Now we have a flow of peer wire messages: receiver -> controller

# The Peer Controller

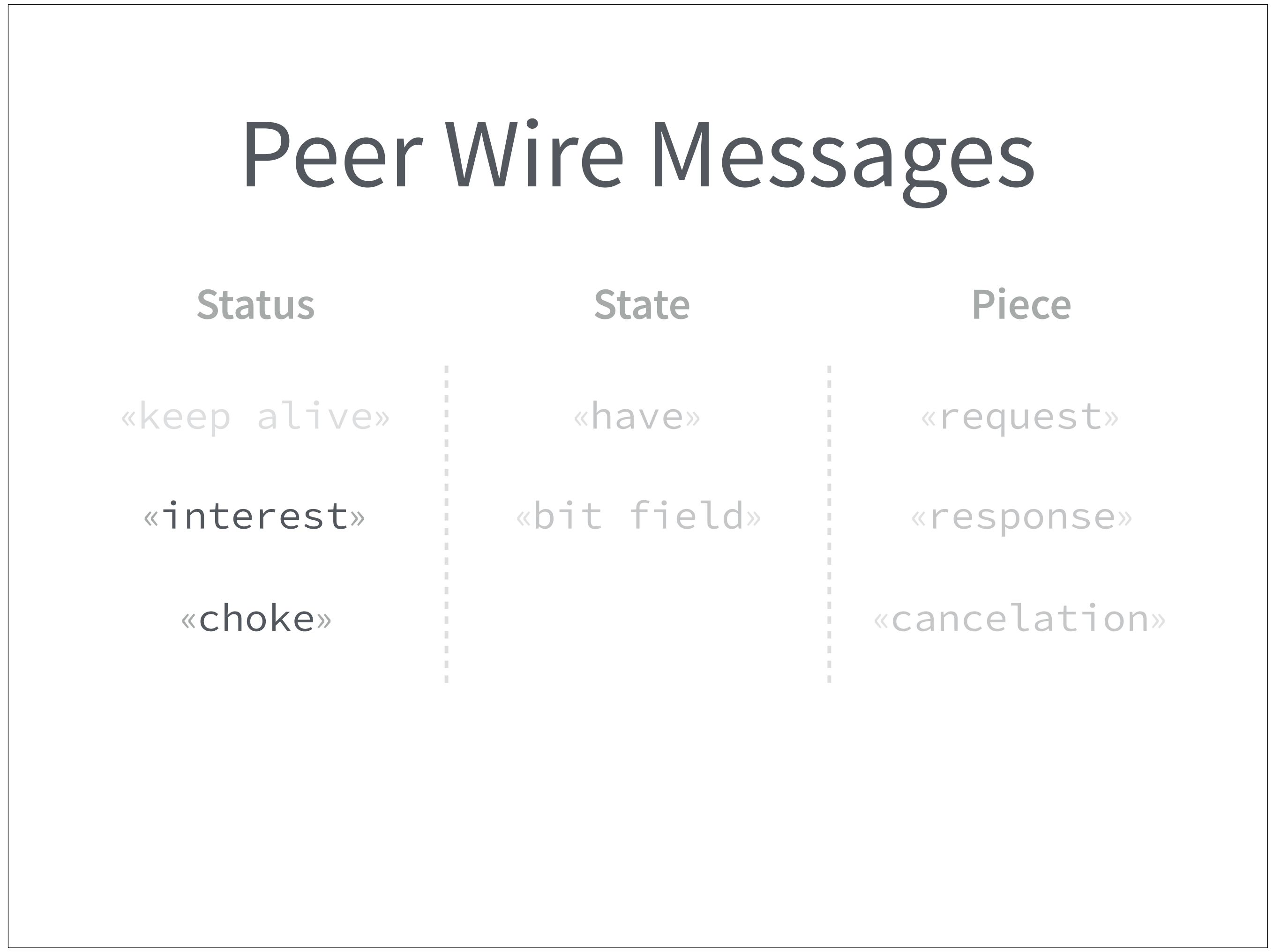
The controller will be the brains of the operation




This is a confusing slide of all the peer wire messages

One can categorise them in:

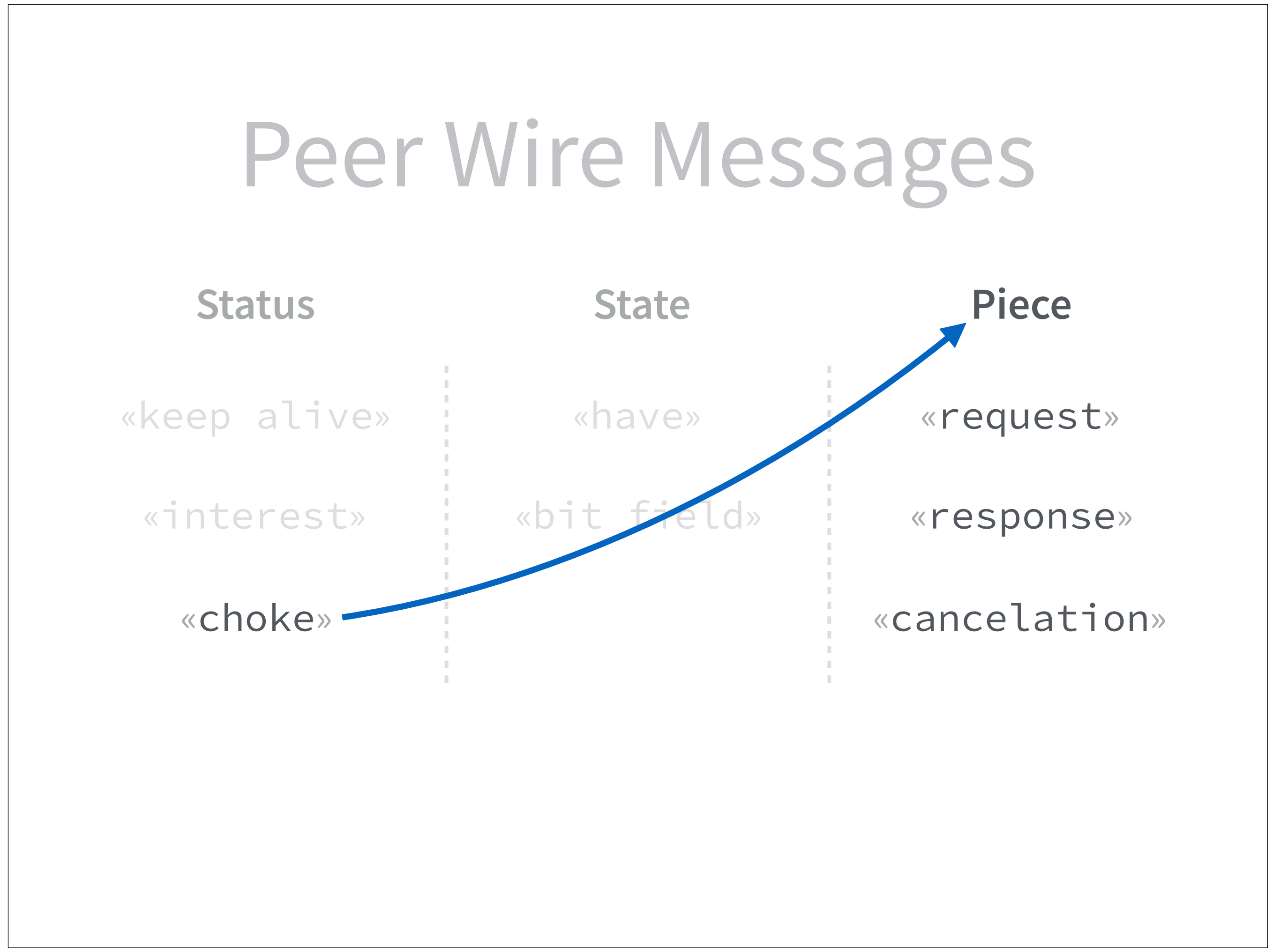
**—Status messages**
**—Piece state message**
**—Piece requests messages**



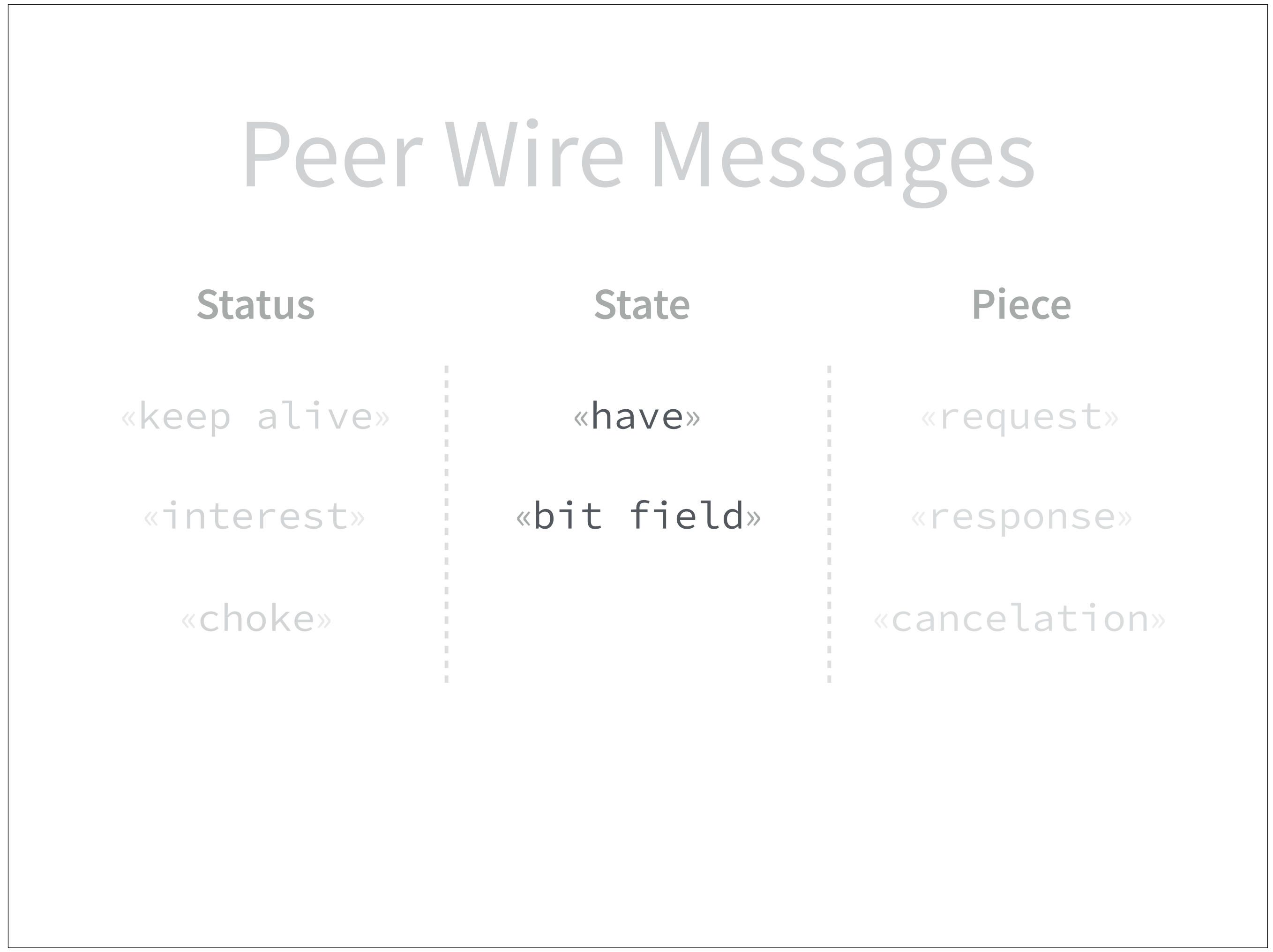

The controller needs to take care of some bookkeeping

We want to track **interest** and **choke** status (both boolean states)

Choke can be a bit hard to envision, as it sounds like a violent act…



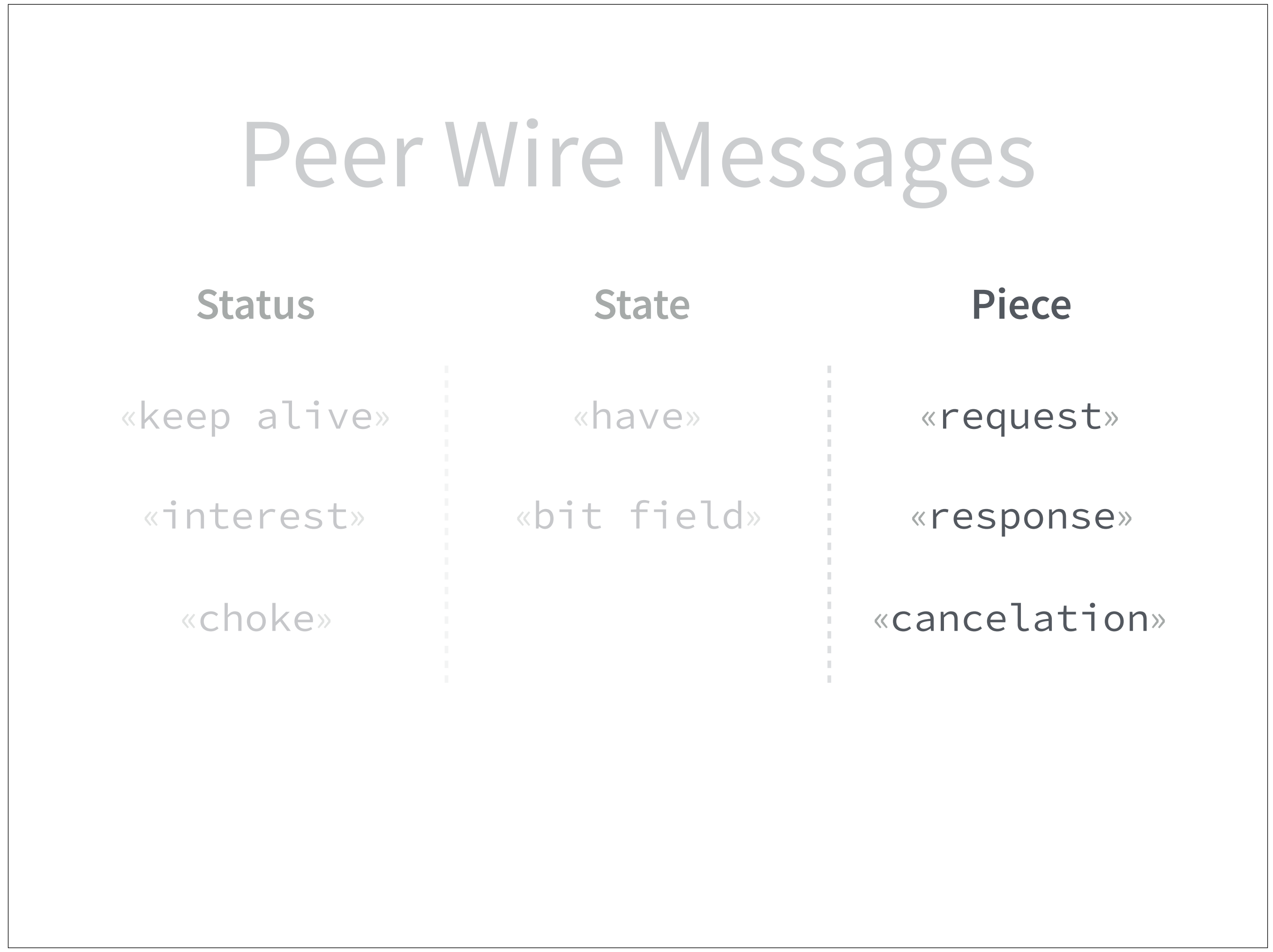

*When we are not being choked*
we are allowed to
*perform piece requests*
*and the remote would respond with data*




The peer will communicate which pieces it has in its possession

— **First by sending a bit field**
— then by sending a **have-message** every time it receives a new piece

We need to keep a set of pieces and update it

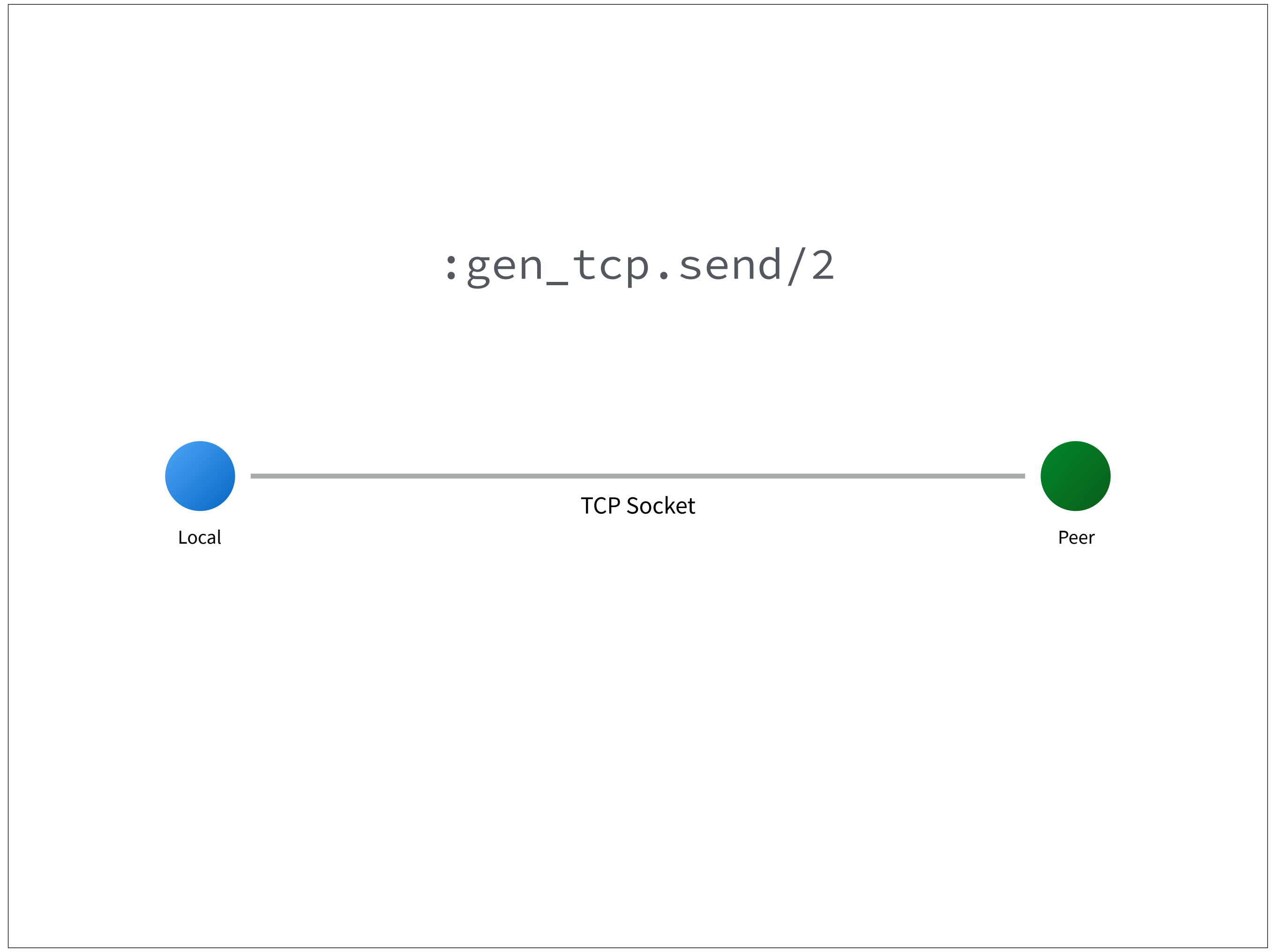



Lastly we have the piece requests;
—used to place,
—fulfil,
—and cancel orders

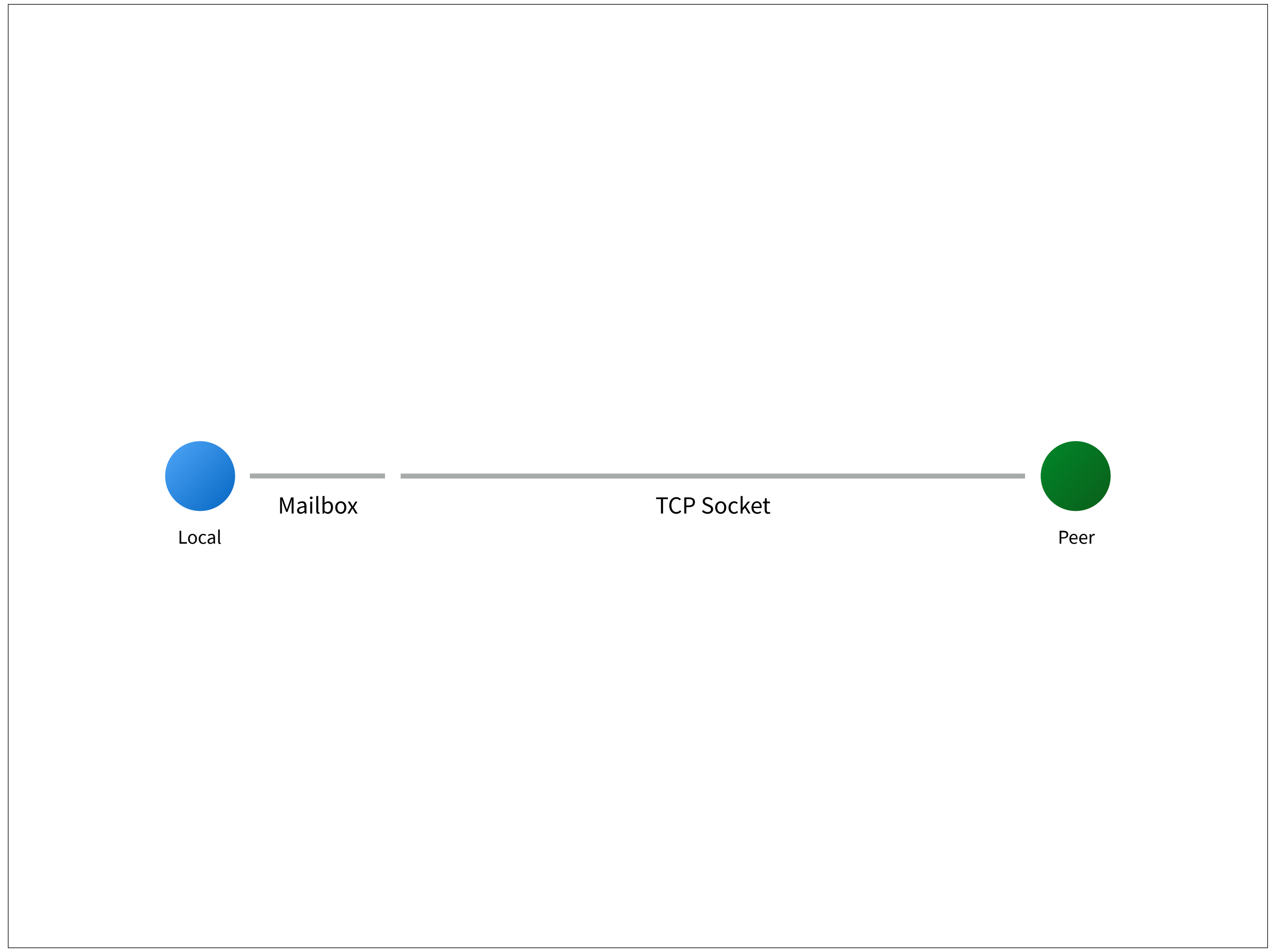
# Responsibility

- Bookkeeping: interest / choke status / available pieces
- Decide what to do with the incoming messages
- Rely messages from the system to the transmitter
- This can be handled by a GenServer

# The Transmitter




- Unlike the receiver we know what data goes onto the wire
- It is really just a matter of encoding the messages to peer wire and relay them



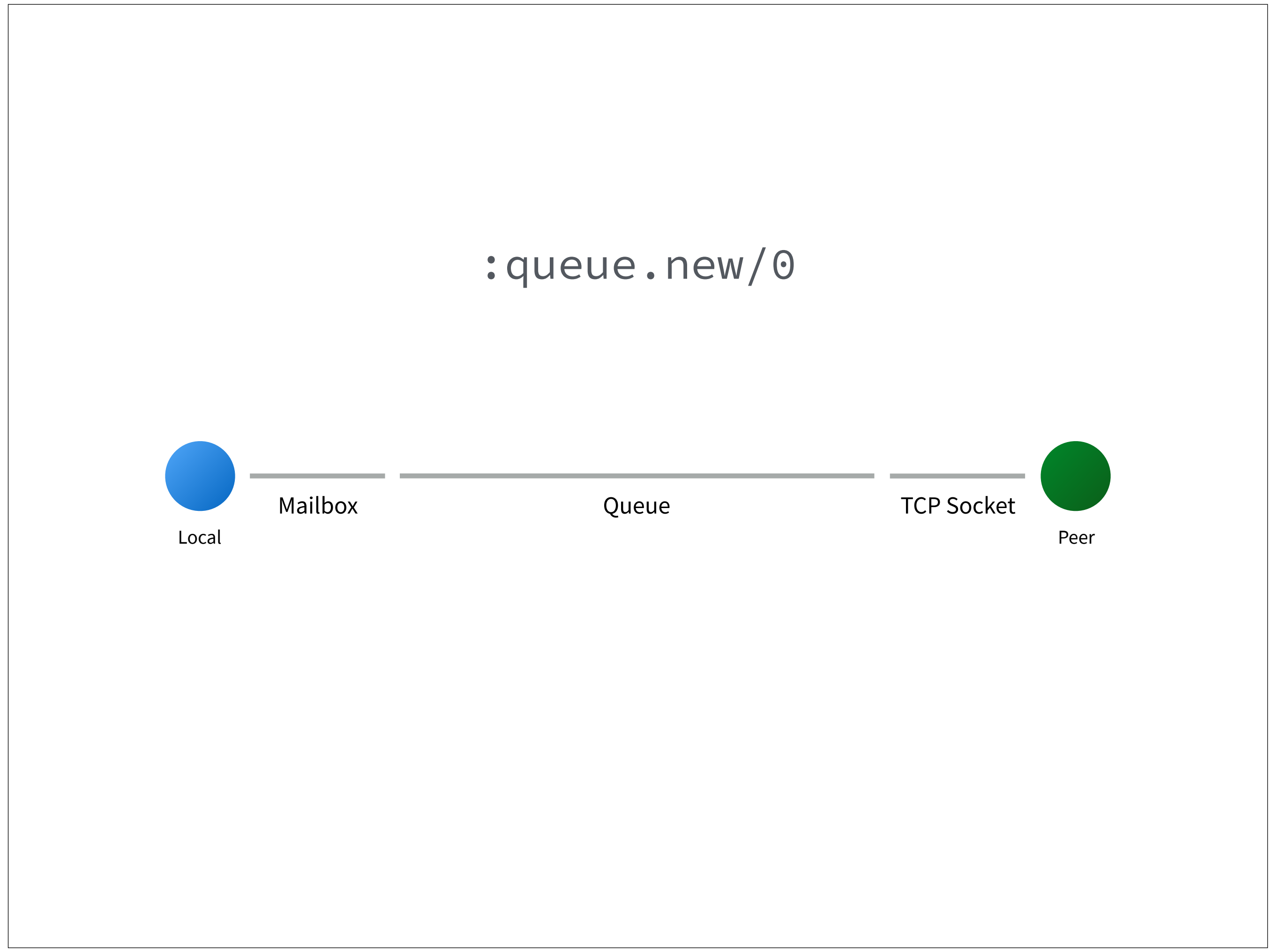

- We could just take messages from the mailbox,
  —peer wire encode,
  —and send them to the remote peer

- **But this could be wasteful**;

  What if we send the **same message twice?**;

  What if we want to change choke status and have a bunch of piece requests lined up?

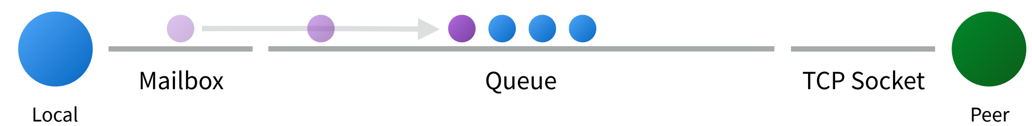
- we can introduce a middle man



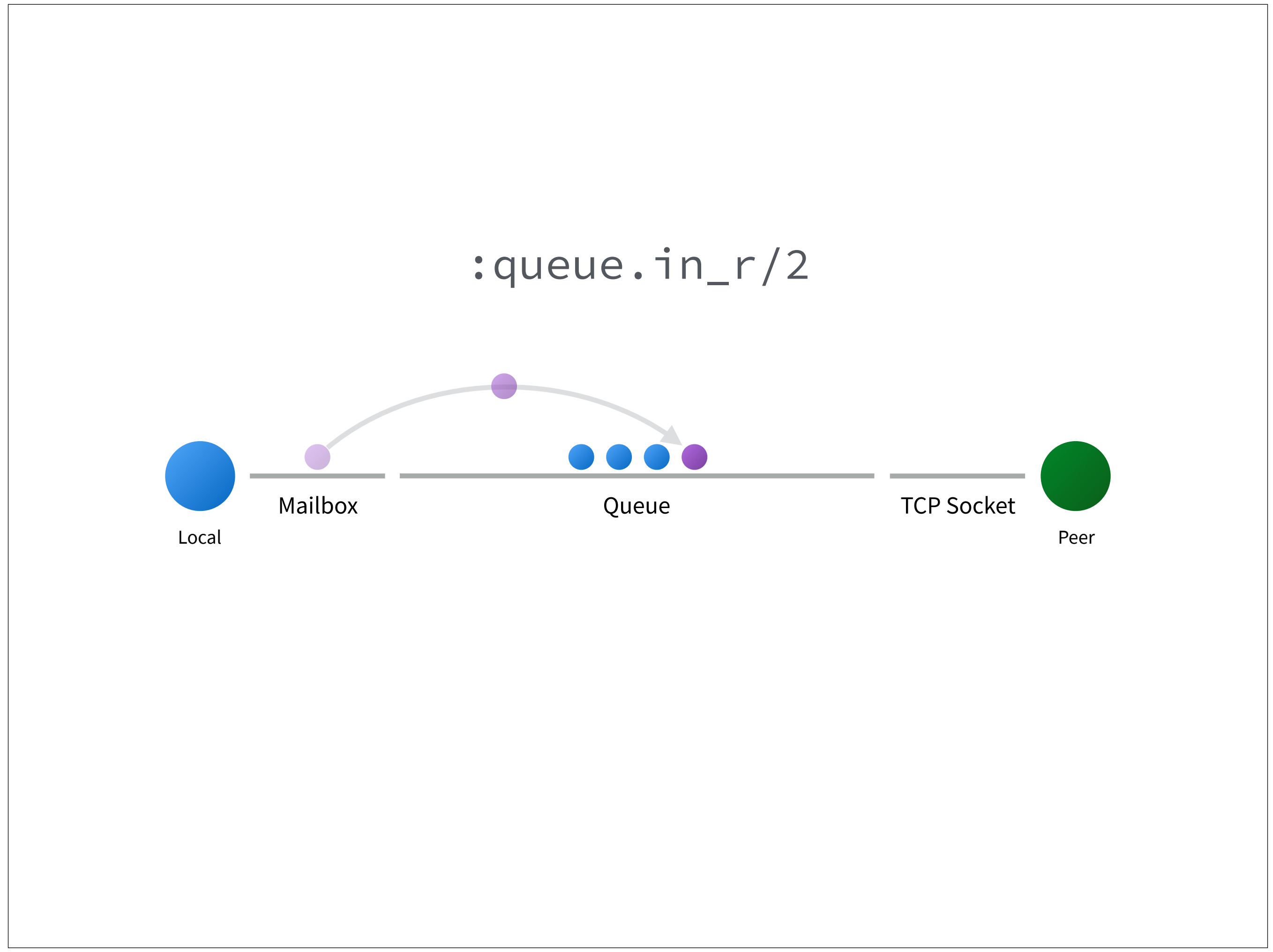

- The Erlang standard library has a queue that allow first-in-first-out operations
  —and manipulating the queue

- This way we could cancel requests before they even hit the network.

:queue.in/2

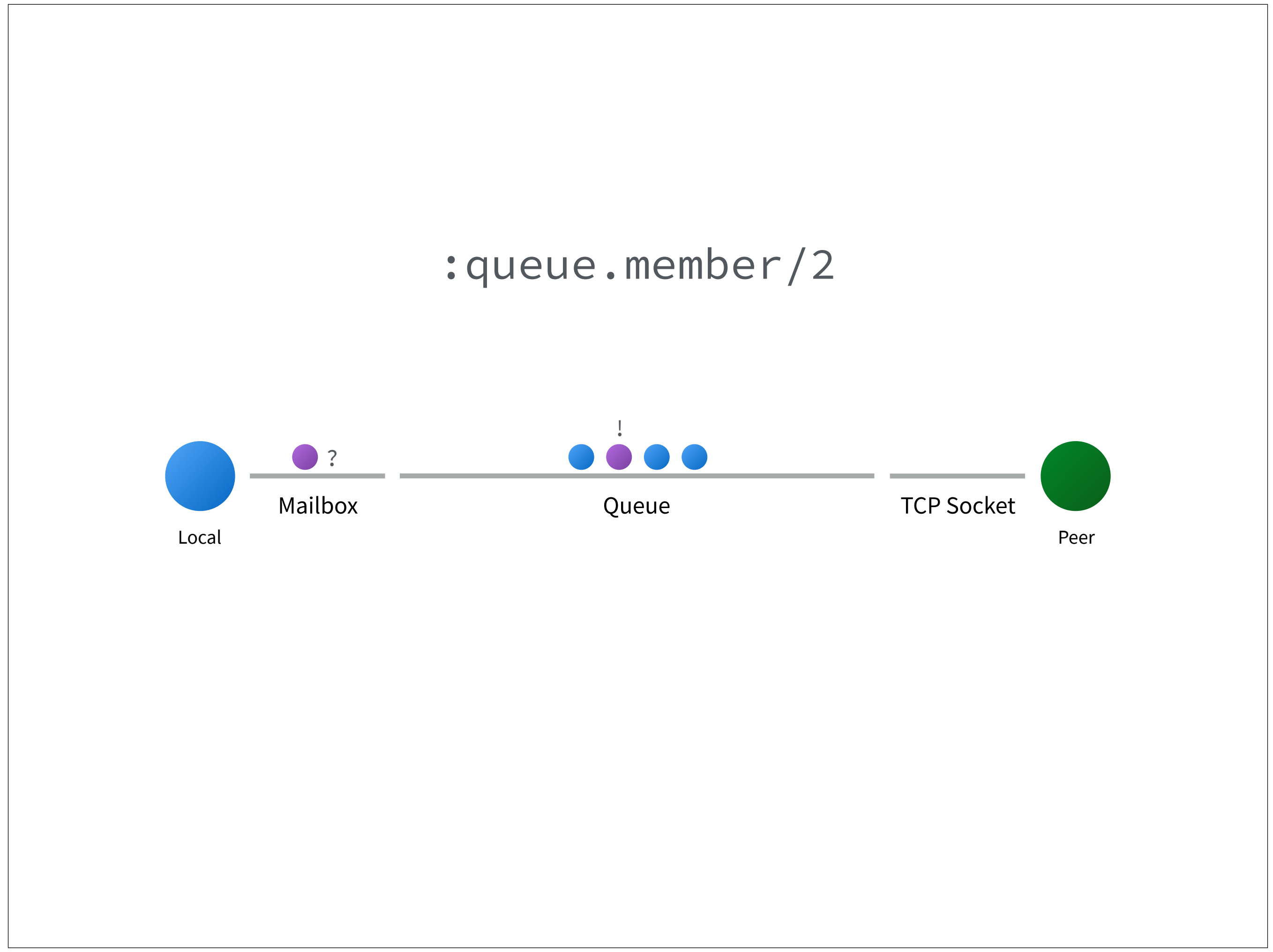



queue in will add the message to the back of the queue



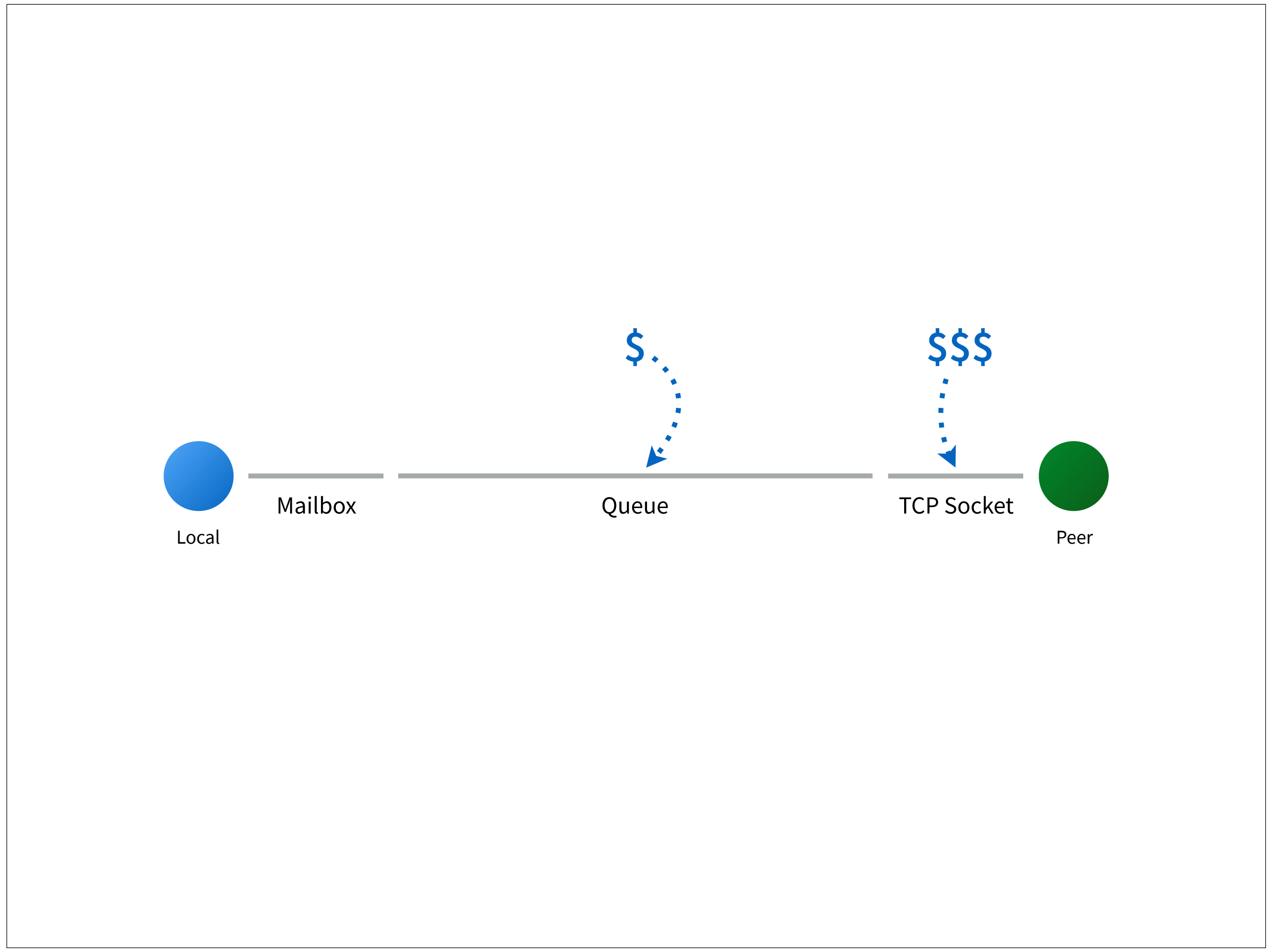

queue R will put the message at the front of the queue

…«queue reverse» ?

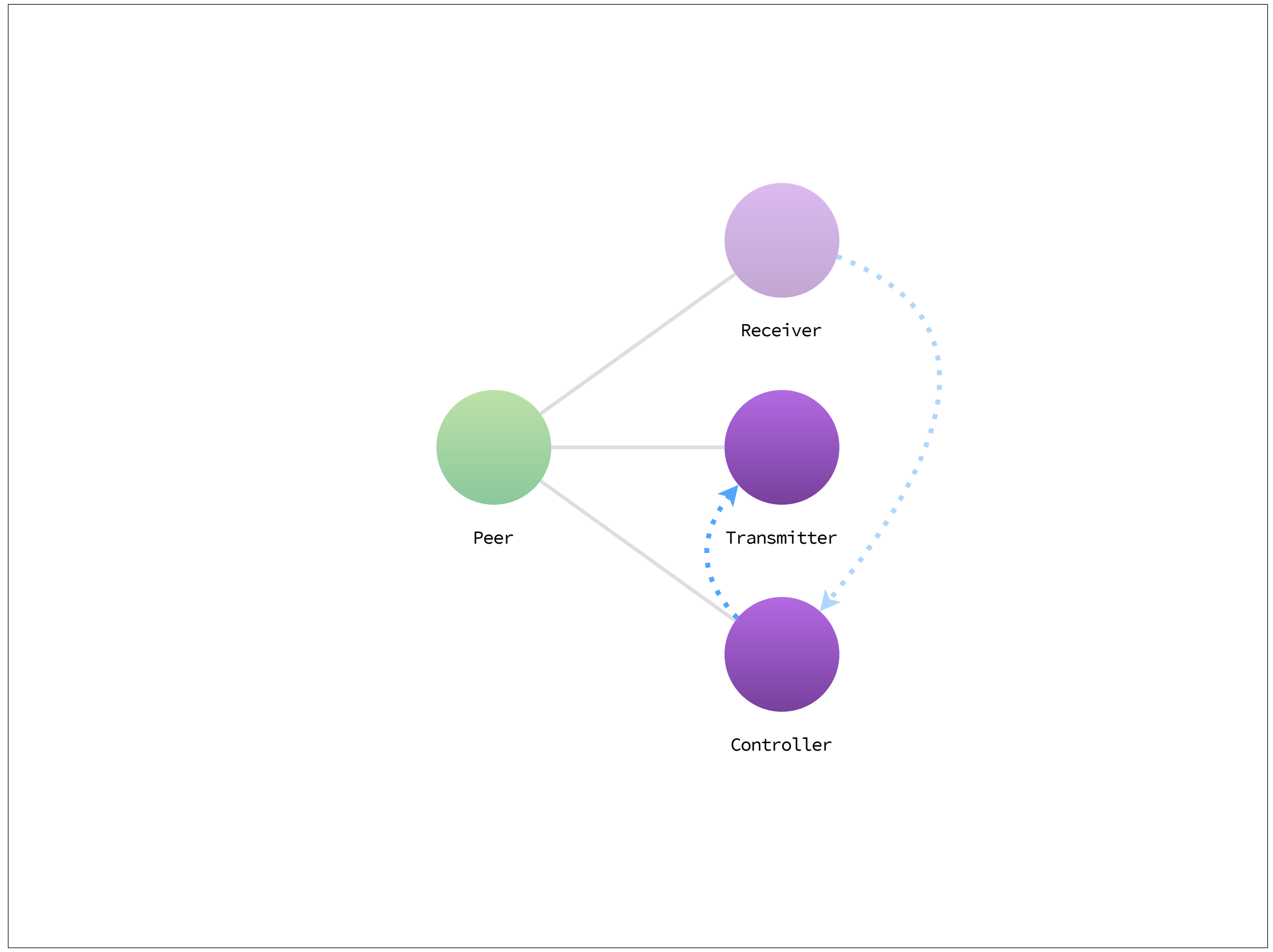



We can also see if a message is already enqueued and drop it

the important thing…




Altering a queue is a ton cheaper than hitting the network




This should take care of the communication


Error handling:

**Strategy: One for All**, if the transmitter or receiver crashes the peer should crash.

We should not attempt to restart a failed peer; they are temporary

If they reconnect we will get a new socket and they will send their resent their bit field making it possible to rebuild the state

If they failed because of a protocol violation we should tell the acceptor blacklist about the incident

You might be wondering:

«We sure have a ton of processes to keep track of by now: How do we locate them?»

# :gproc

by using :gproc

its an extended process registry

`:my_module_name`

Normally we can only name a process using an atom

{:peer_controller, peer_id, info_hash}

:gproc allows us to add structure to the process name

We can then locate peers by known data, such as a peer_id

Besides that it allows you to save metadata to an ETS table

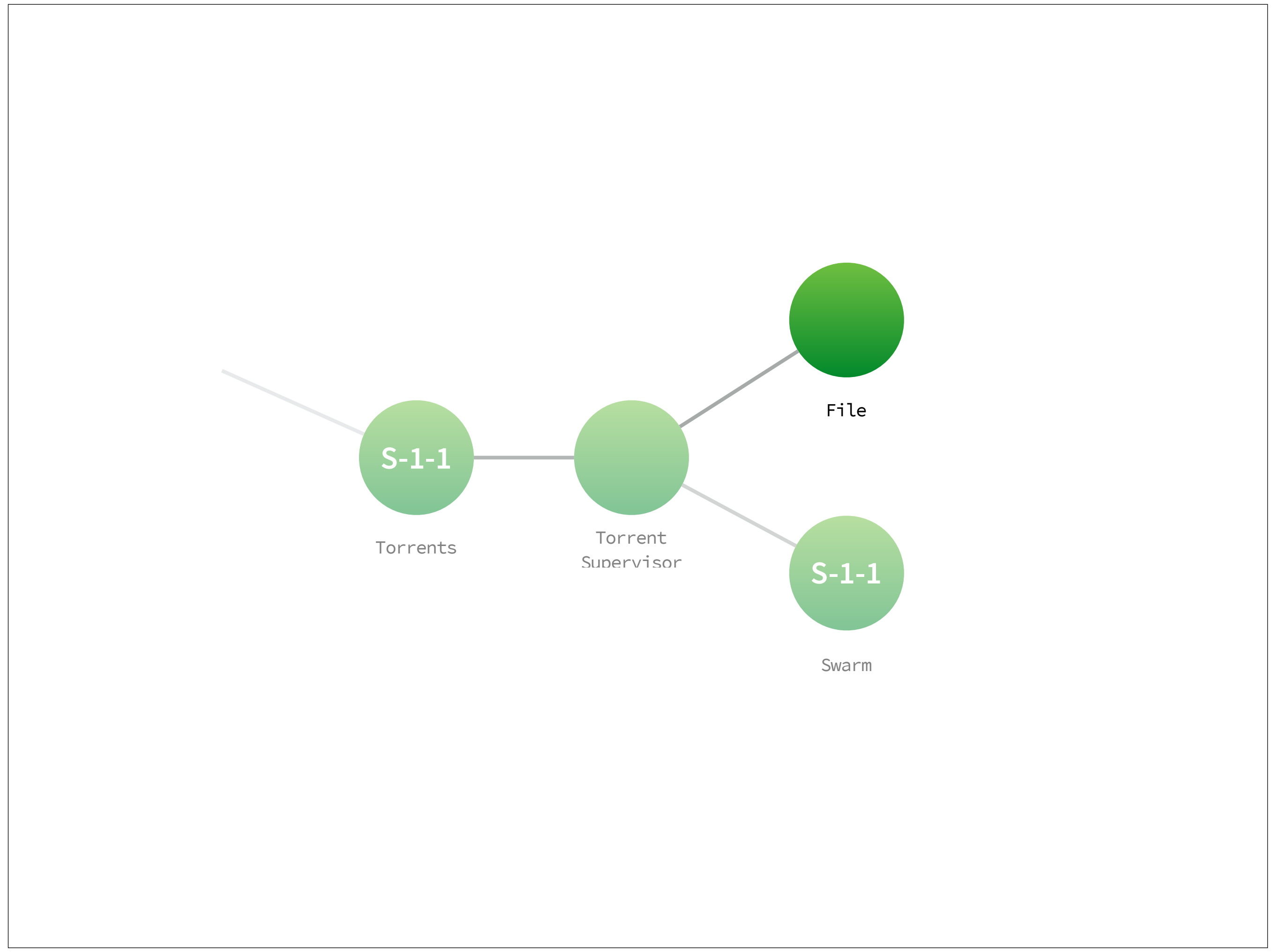
```
iex(1)> Swarm.find_non_choking_peers(info_hash)

# list of non choking peer PIDs
```

This allows us to:
—Search for processes matching criteria—such as choking status
—we can then define a helper function that returns a list of peer process ids

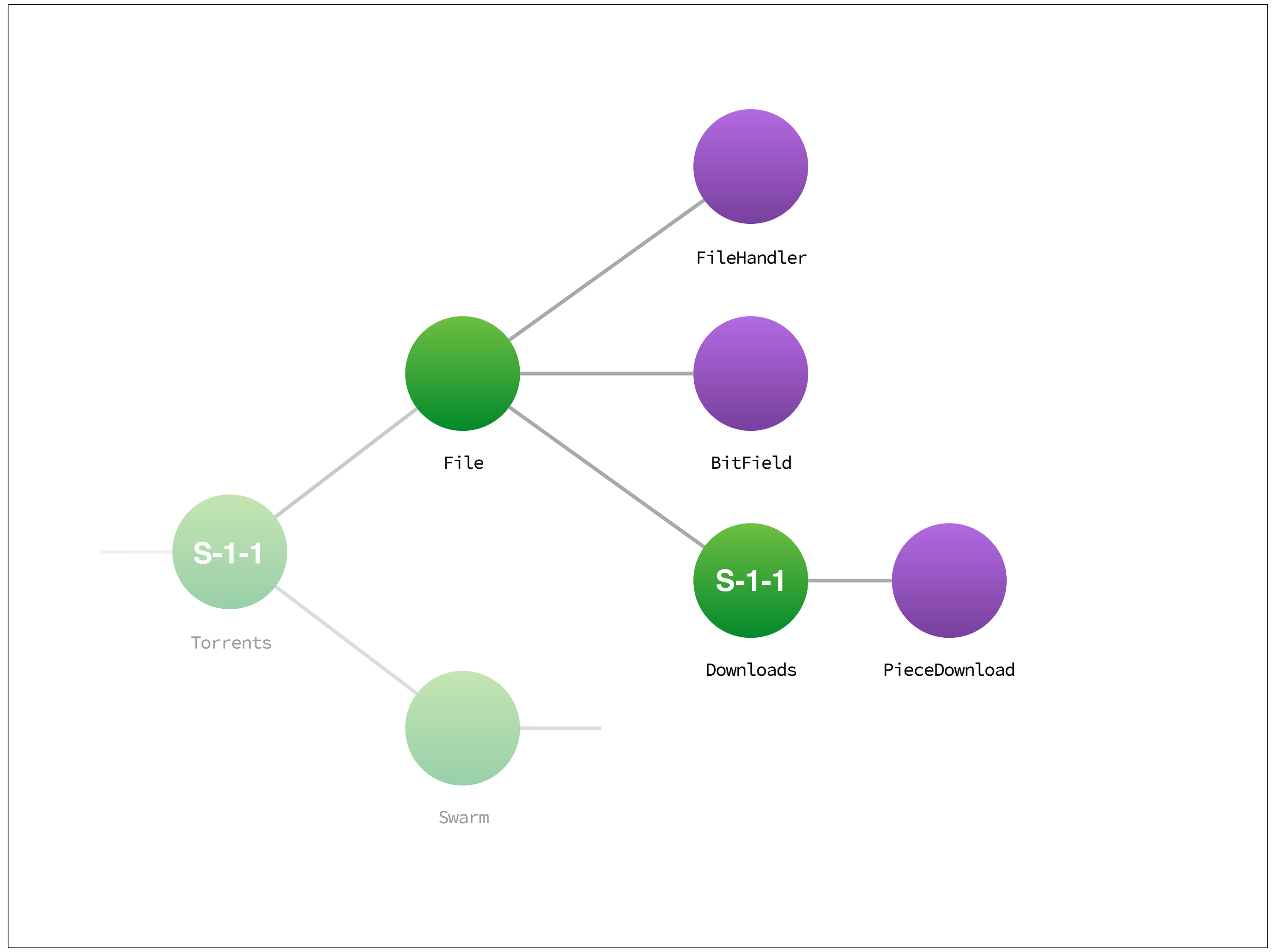
We can then select a candidate to download pieces from

# Data store

For that we need a service that handle storing the file




The file belongs to a torrent process, so keeping it here makes sense




Only one process should have access to the actual file, the «file handler»

—

We would need to update and access a bit field, so that goes in a process—could be an **Agent**

—

and lastly we need to manage and control the download of the individual pieces, and have many processes doing so

# The File Handler

First writing and getting data

# Responsibilities

- Write and read bytes to disk as we get data
- Be able to retrieve ranges of a given file piece; based on offset and byte length
- Run checksums on «pieces» when they are fully downloaded and written to the file buffer

…opening a file,
—reading from that file
—writing to that file
—how hard can it be?

```
File.open(file_name, [:read, :write])
```

We open a file in read and write mode, all good…

that will give us an io_device we can write and read data from

```
File.open(file_name, [:read, :write])

IO.binread/2 + IO.binwrite/2
```

But we have to look into the IO-module for actual reading and writing

They will let us **read and write n-bytes from the IO device**

…from where ever the cursor is in the file, which is great is you are consuming a file, or streaming its content

We need something a bit more low-level

```
File.open(file_name, [:read, :write])

IO.binread/2 + IO.binwrite/2

:file.position(io_device, position)
```

To manipulate the cursor position we need to use the Erlang standard library

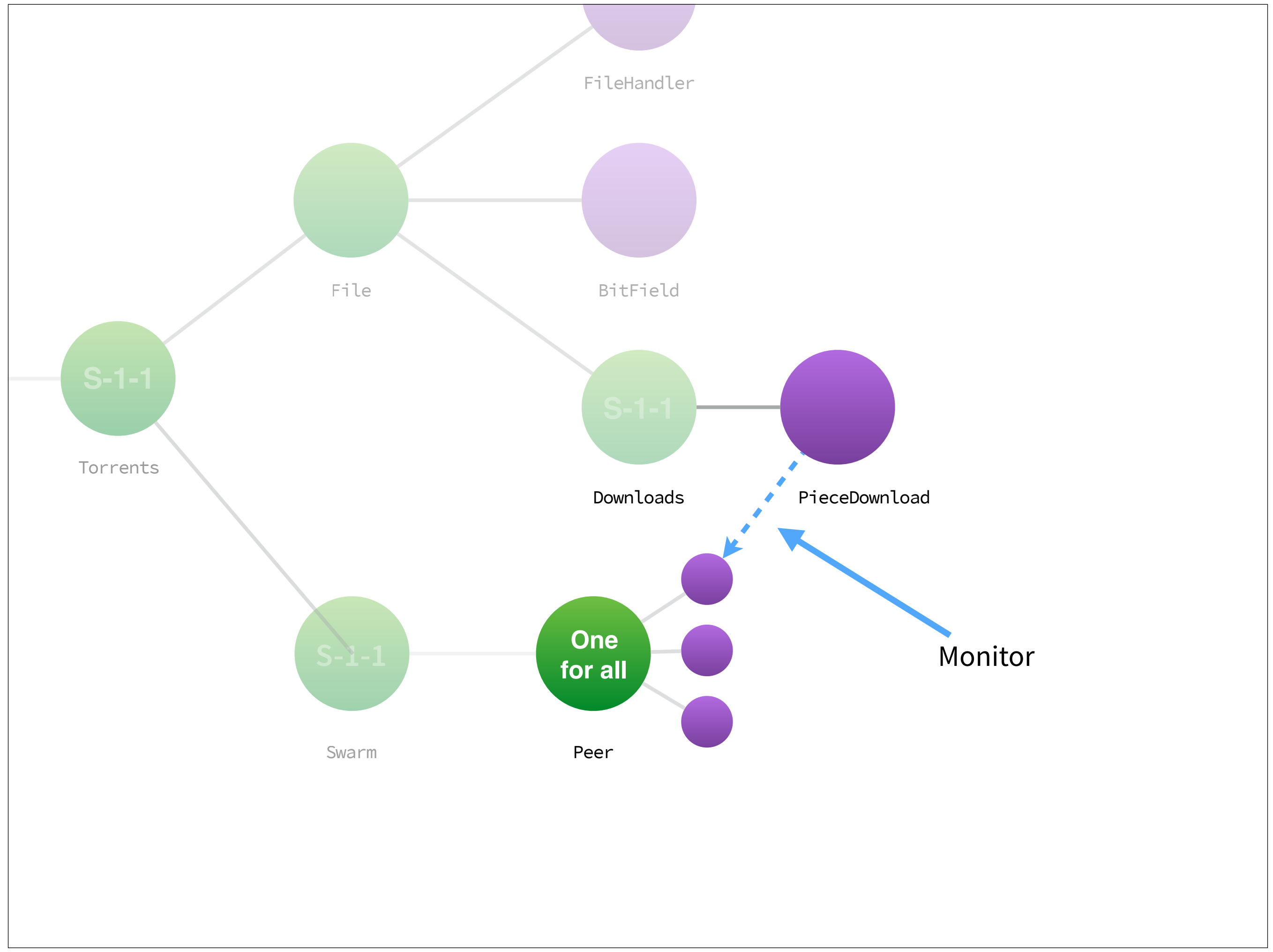
To my knowledge Elixir does not export this functionality

this should allow us to read and write data in a random access form, in an effective manner

# Piece Downloads

- Make a plan of ranges (of ~16 Kb each) to download for the given piece
- Request a non-choking peer with the given piece from the swarm service
- Send the planned ranges to the peer service
- Receive blocks as they come in from the peer—and write them instantly to the disk using the file handler
- Verify the piece when it has been downloaded against its checksum
- Update the BitField and broadcast a have to all the peers in the swarm

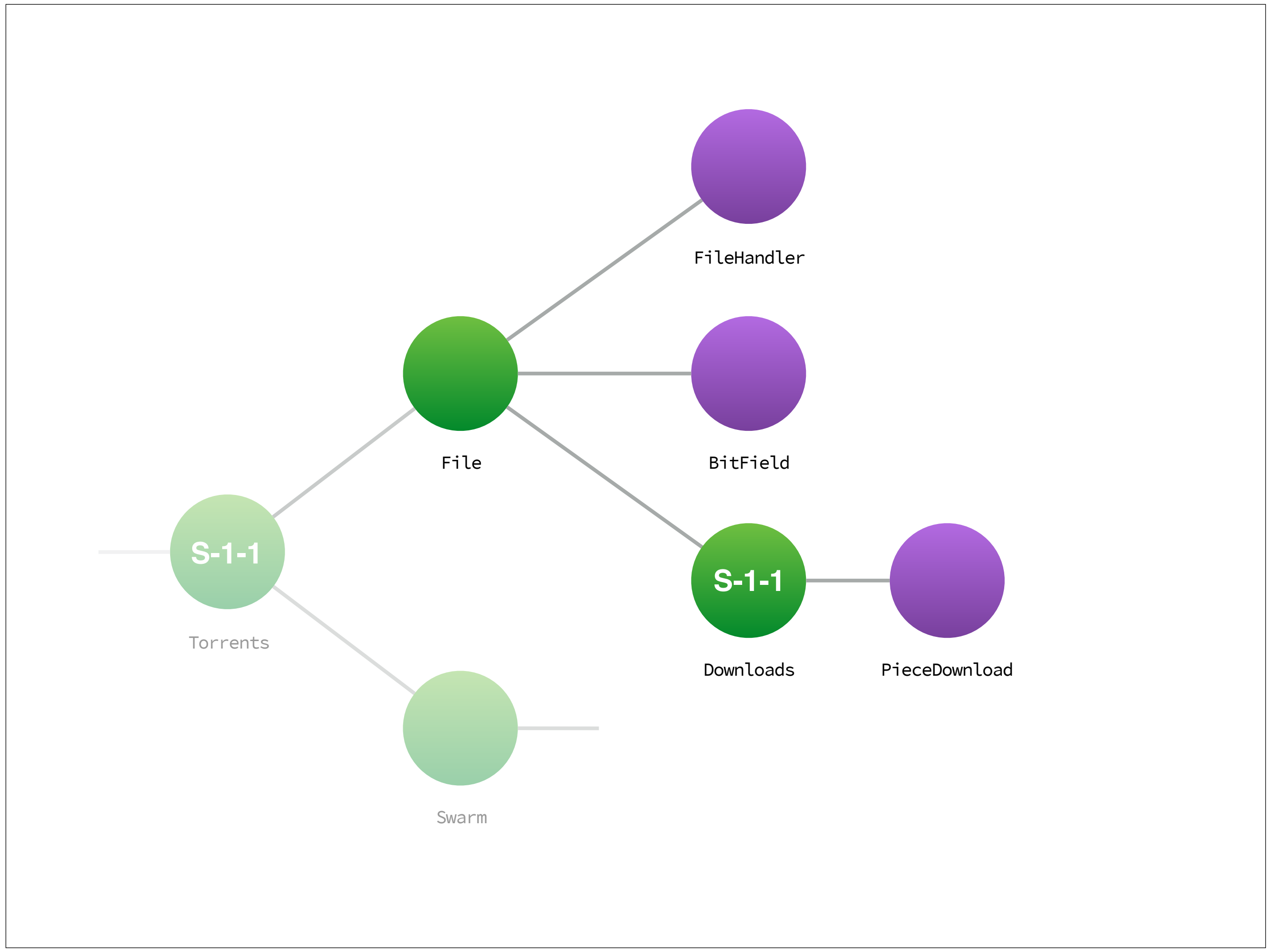
…one does not just ask for a piece; one has to break a piece request up in chunks of 16 kilobytes…




**That was the happy path!**

**What if the peer goes away before we get the entire file?**

We will **monitor** the peer and listen for an exit;
The state machine should flip back to looking for peers and continue downloading




Handling Failure

Every time **we get a block we will save it to disk**, so the **piece downloader** is **temporary**; it will **just restart and find a new peer**

**The BitField state is dynamic**, and we **can generate it** by asking the **FileHandler**

…and the **FileHandler** can **rebuild its state from the file system**; it should run through the pieces and **verify them using the piece hashes**

# …and now for the main Torrent Controller

…and then we have to create a main controller
—that decide what to do next;
—using the various services…

that will be another day; I only gave the gist of it

Torrent Controller; Peer Bandwidth Throttling; Download strategies; Distributed Hash Tables;

Tons of big and small details . . .

Lots to be solved!

## In conclusion

Split the task in small focused services.

Ask Questions; Attempt to solve the problems;
Rinse, repeat.

…and eventually we have a BitTorrent client

I hope I made a case for why it is fun to think about the BitTorrent Problem in Elixir

Please ask questions on communities like Stack Overflow to raise awareness…

- The BitTorrent Protocol Specification — http://www.bittorrent.org/beps/bep_0003.html
- «Theory» Community BitTorrent Specification — https://wiki.theory.org/BitTorrentSpecification
- More on Erlang and State — http://jlouisramblings.blogspot.dk/2012/05/more-on-erlang-and-state.html
- Combinatorrent - a Haskell Case Study: Jesper Louis Andersen—https://www.youtube.com/watch?v=qJL-9LKQoq4
- eTorrent - writing P2P clients in Erlang — http://www.erlang-factory.com/upload/presentations/499/jlouis-efsf2012.pdf
- Learn You Some Erlang for great good! — http://learnyousomeerlang.com
- Some icons used from: https://www.iconfinder.com/iconsets/octicons