WhatsApp

Anton Lavrik

# Tools everyone needs — a reflection on building and running WhatsApp servers

Code Beam SF 2018

# Since our last talk at Erlang Factory in 2014

| | | |
|---|---|---|
| monthly users: | 465M -> 1.5B | x3 |
| daily messages: | 19B -> 60B | x3 |
| daily pics: | 600M -> 4.5B | x7 |
| daily videos: | 100M -> 1B | x10! |

# WhatsApp Server

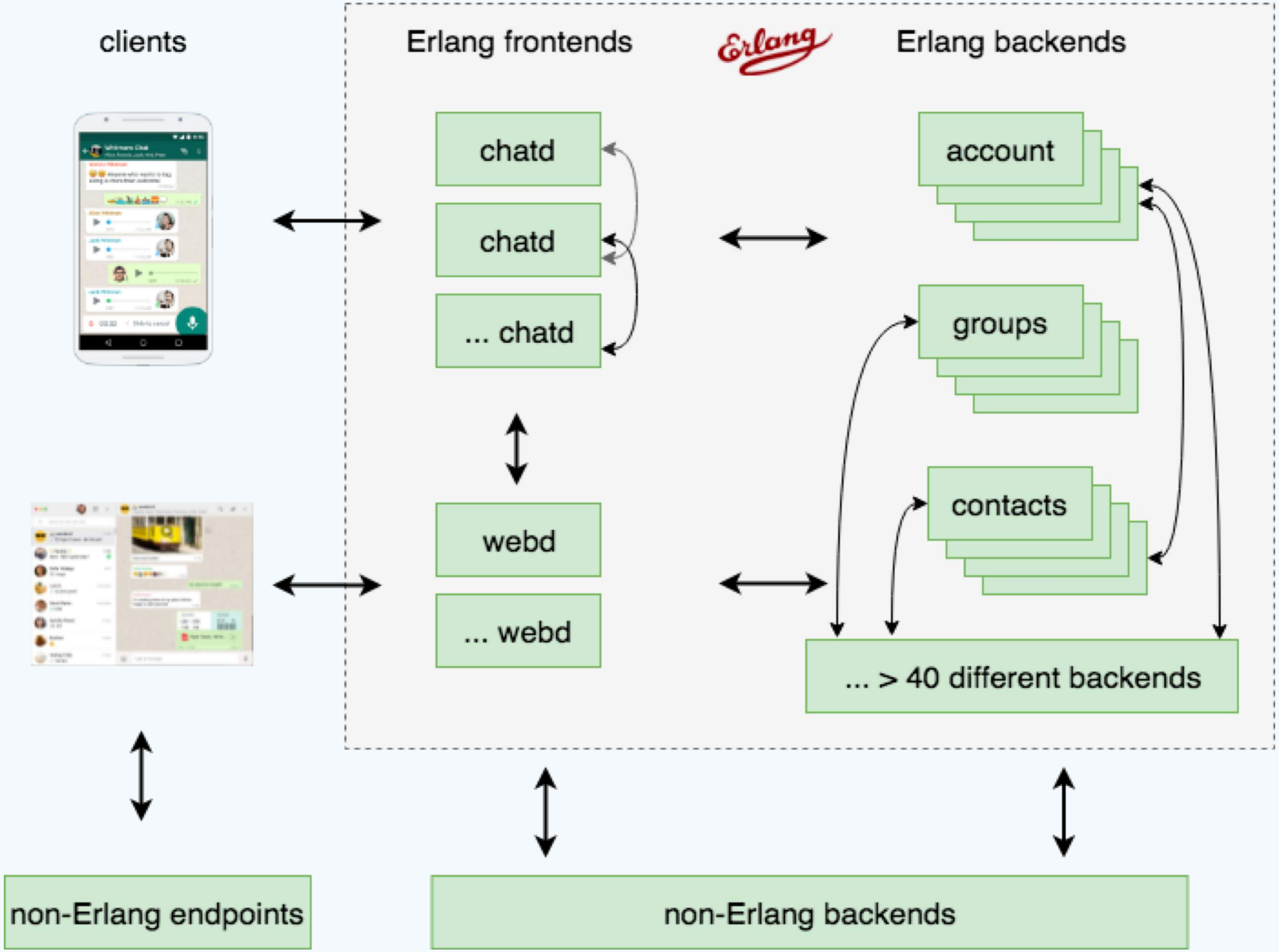
Even more scalable and reliable system

Powered by Erlang

ERLANG

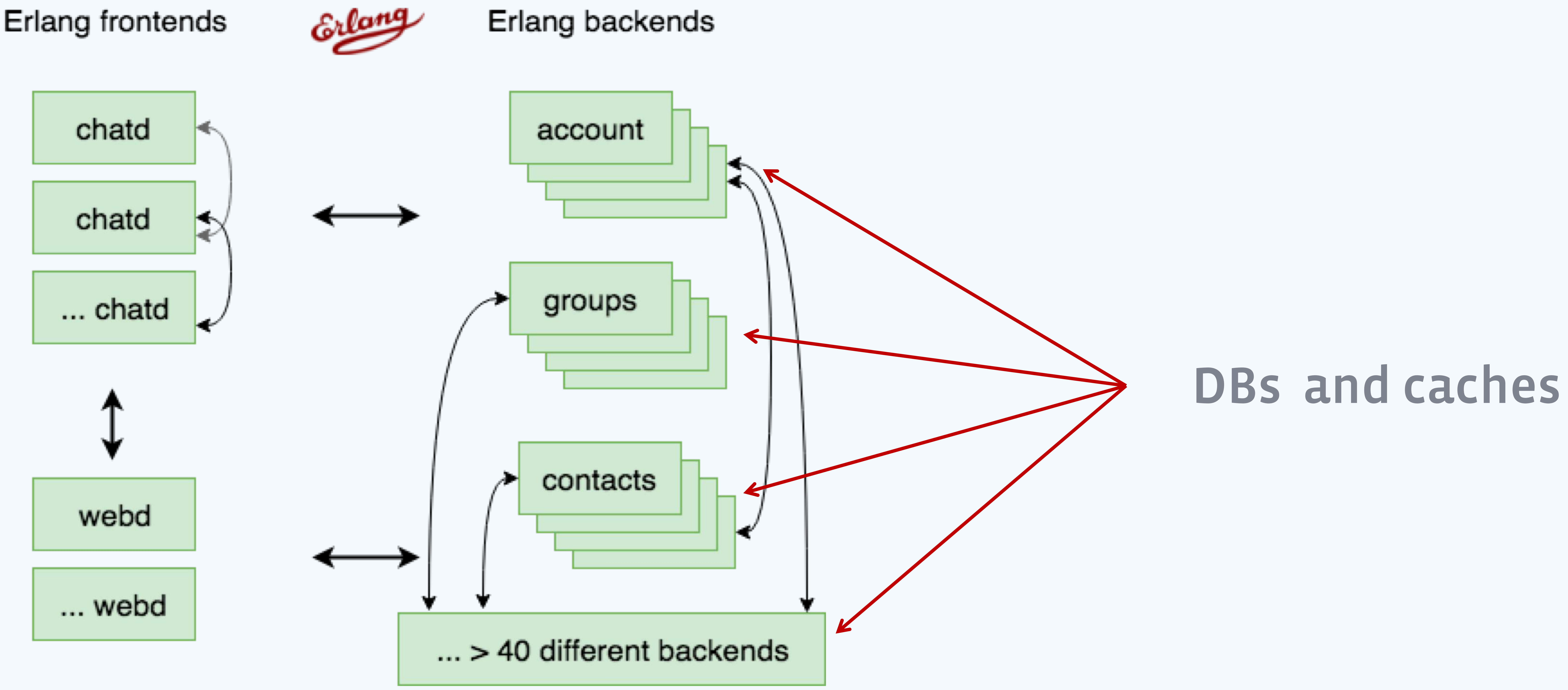
# WhatsApp Server: under the hood

- deliver asynchronous messages reliably in real-time
- keep user messages only until delivered
- highly available service
- handle peak load


clients
Erlang frontends
Erlang backends
chatd
chatd
... chatd
webd
... webd
account
groups
contacts
... > 40 different backends
non-Erlang endpoints
non-Erlang backends

# Databases

# Partitioned embedded DB



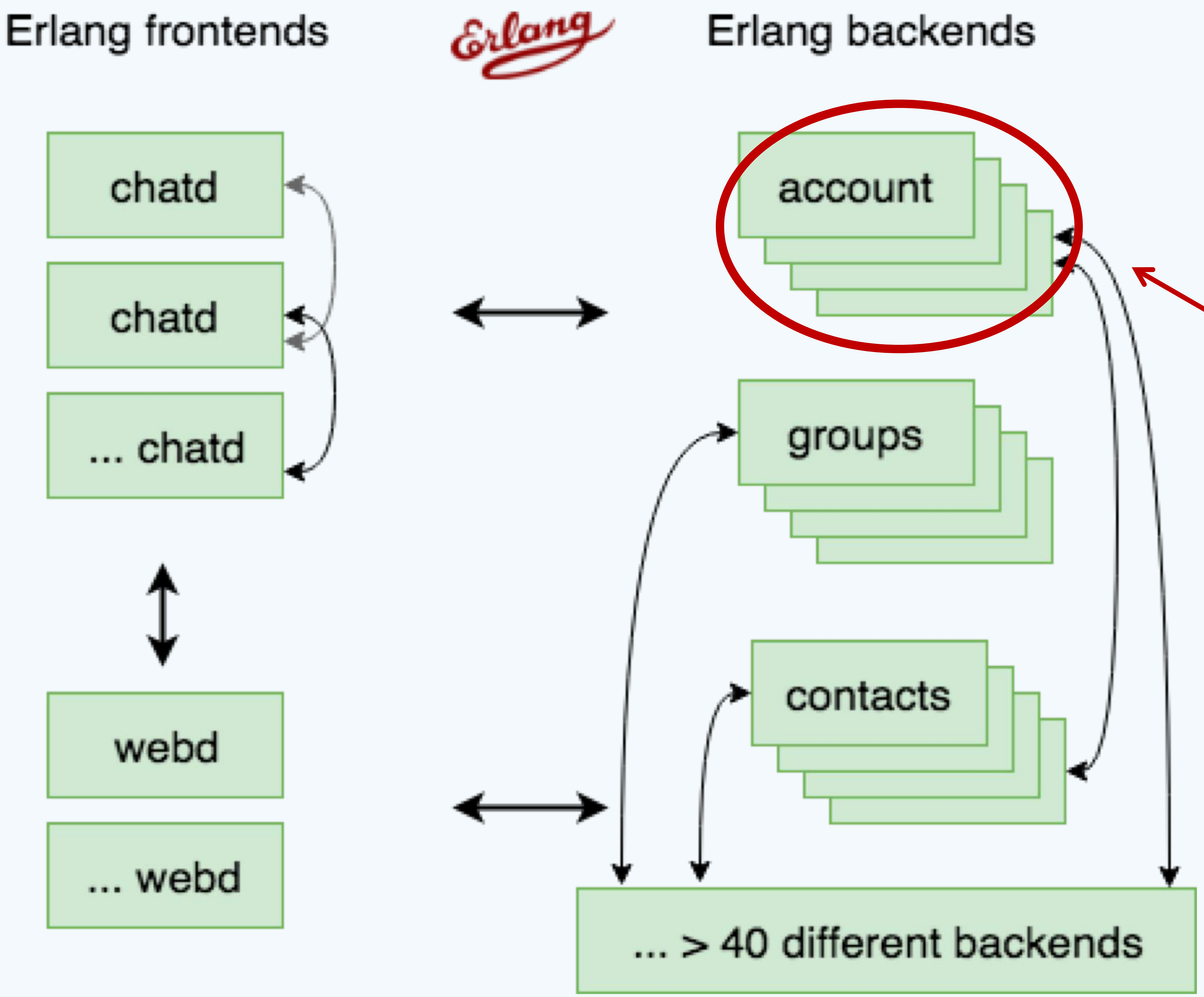
# Our databases

Majority fit in RAM

Data models and access patterns:

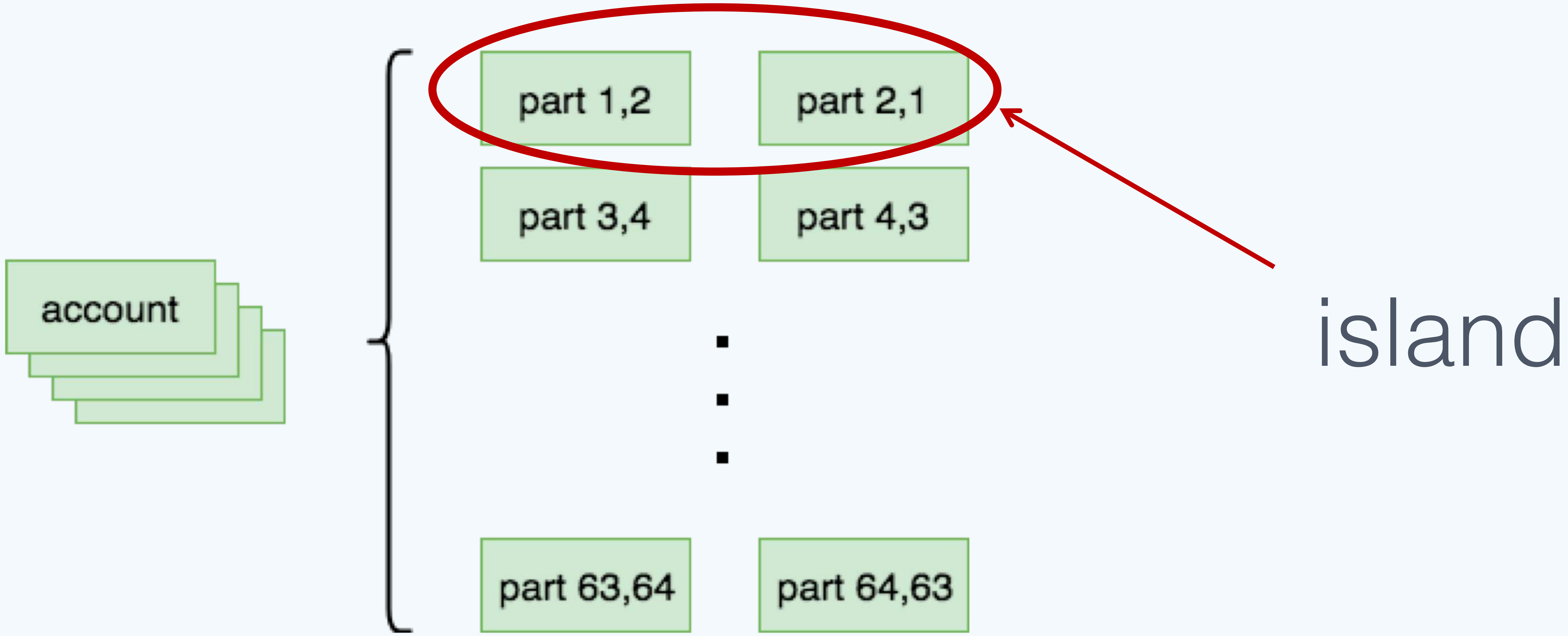
- key-value, read-modify-write
- fast iteration over key space
- graph, e.g. addressbook, group membership

# Partitioned embedded DB




zooming in…

# Partitioning: key to node mapping

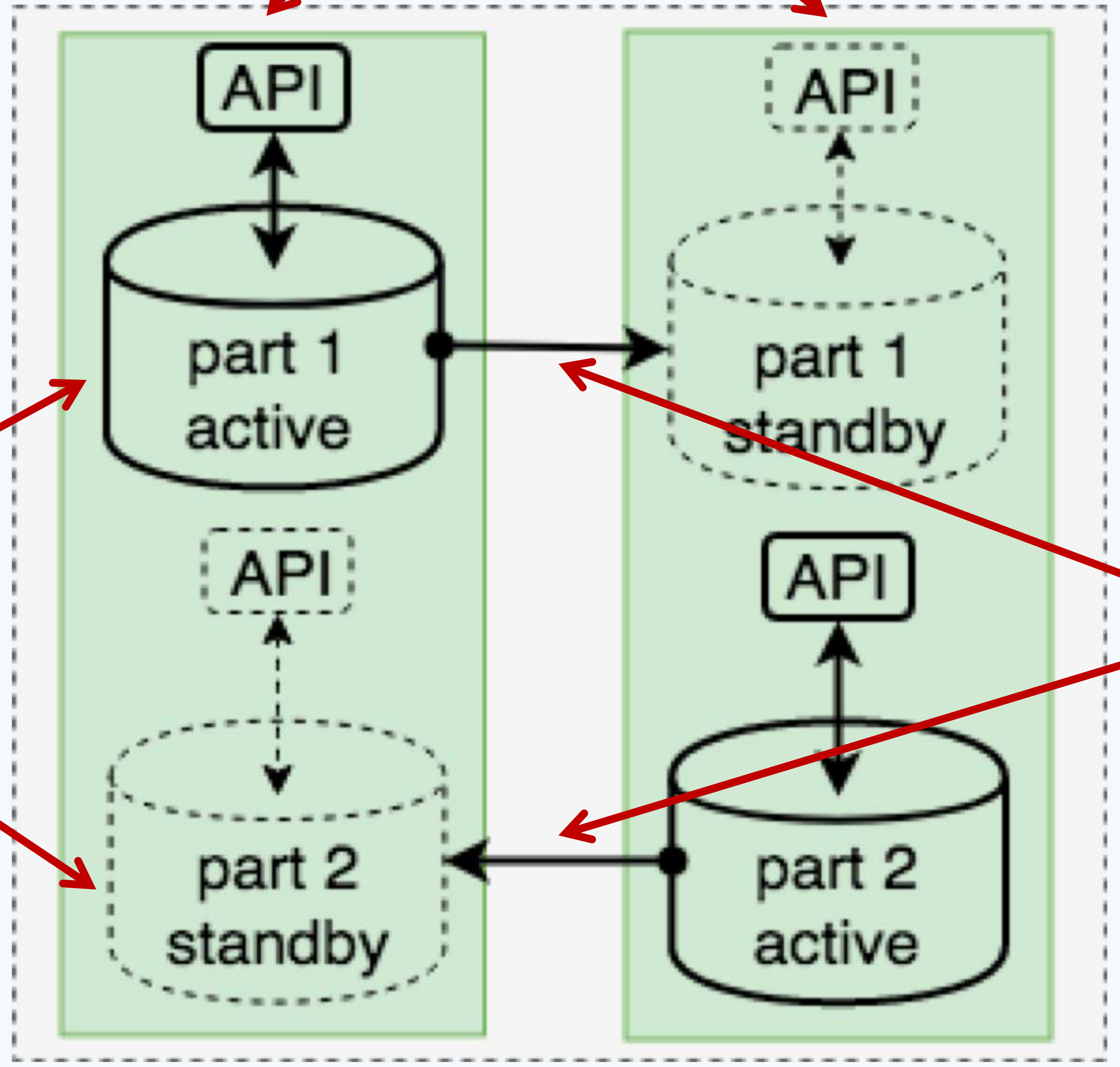



partition number = erlang:phash2(Key, ?NUM_PARTITIONS)

# Backend/DB replicated island

gen_server
registered with pg2

DB partitions

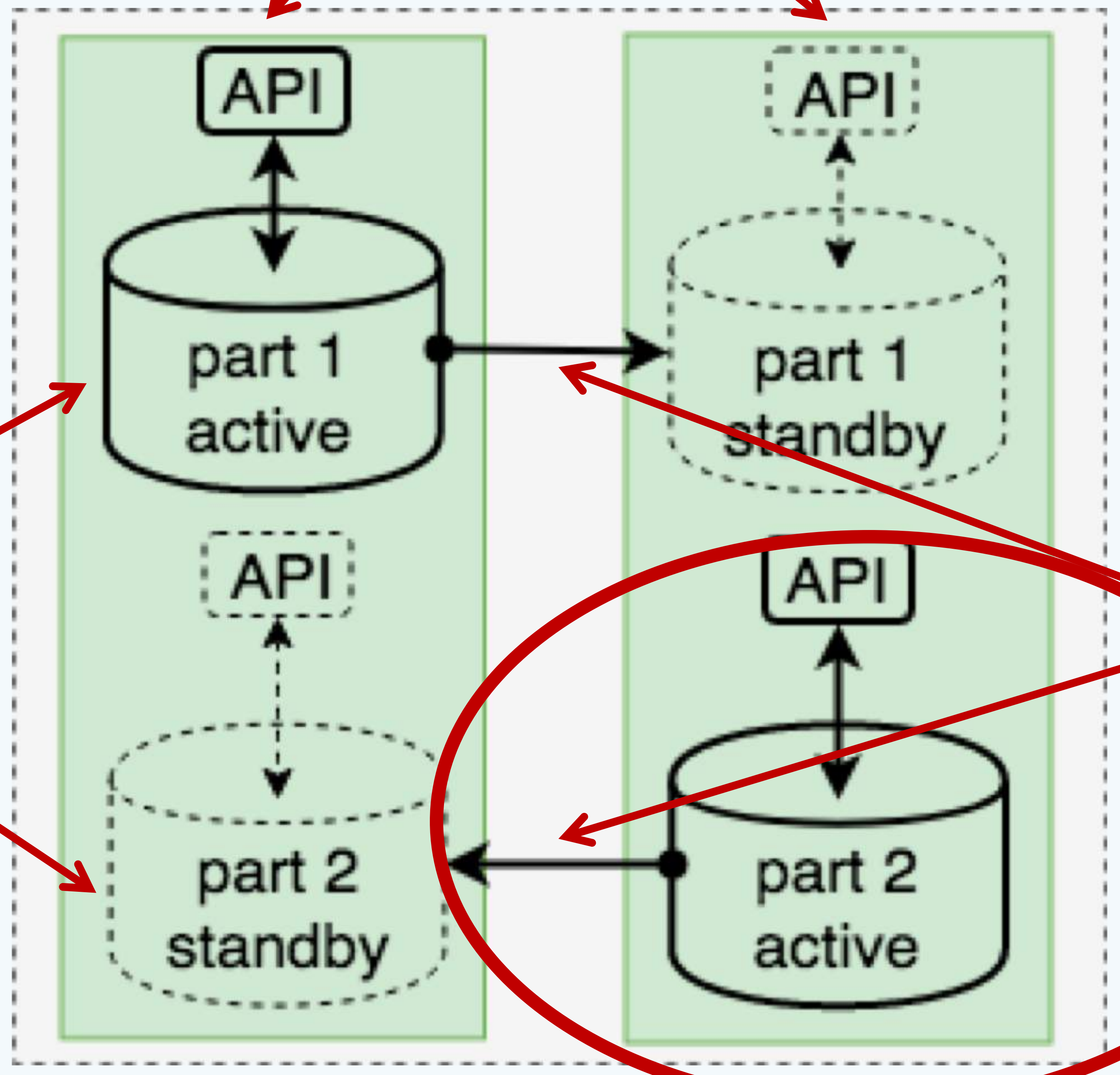



replication

# Backend & DB replicated island

gen_server
registered with pg2

DB partitions



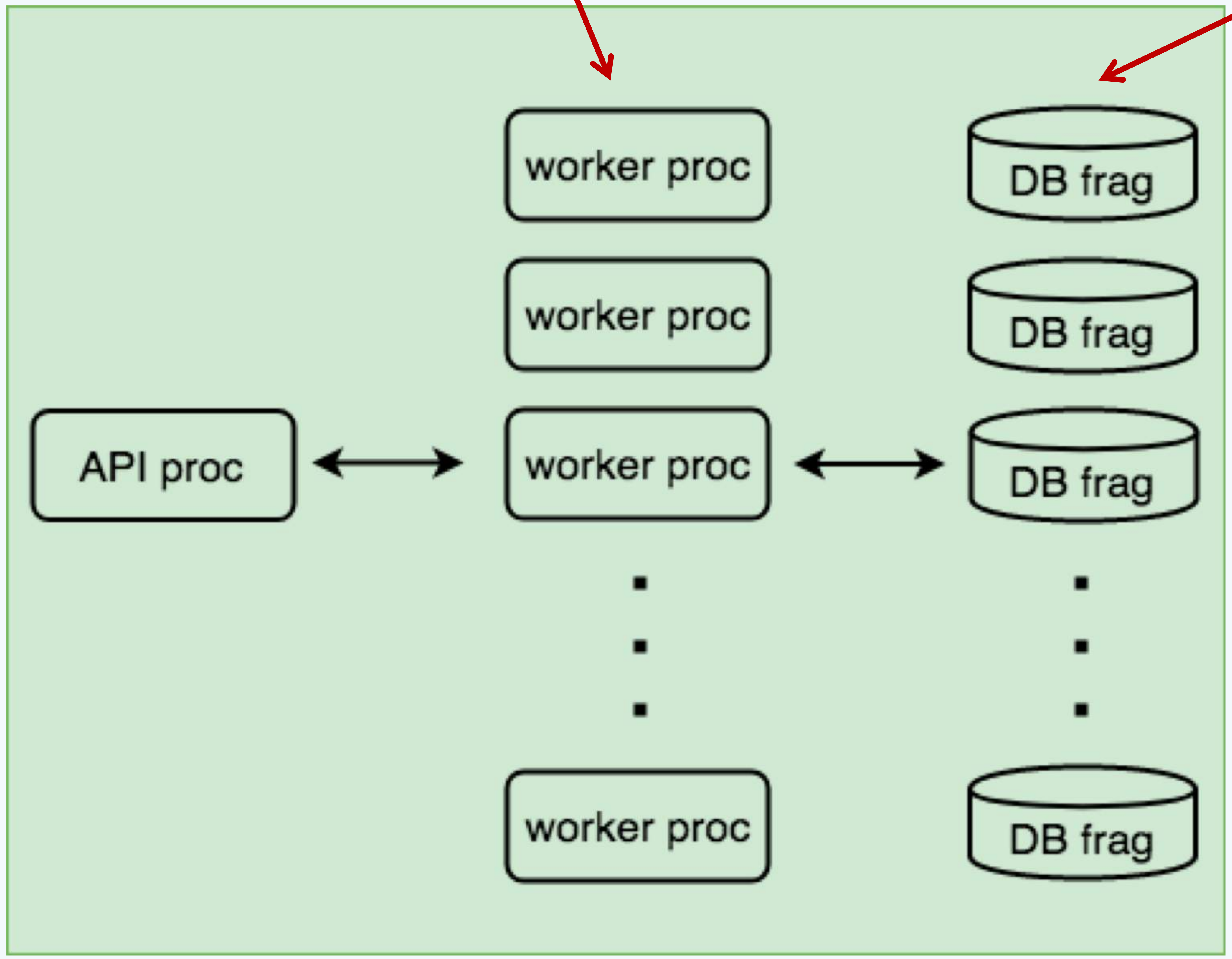

replication

zooming in…

# Backend node internals

application logic

DB partition




# workers, # DB frags are tunable

key -> worker mapping is deterministic:
hash(key, ?NUM_WORKERS)

goal 1: serialize operations for a key to prevent inconsistency

goal 2: minimize lock contention in DB frag on concurrent access

# Key idea

deterministic key -> node -> worker mapping

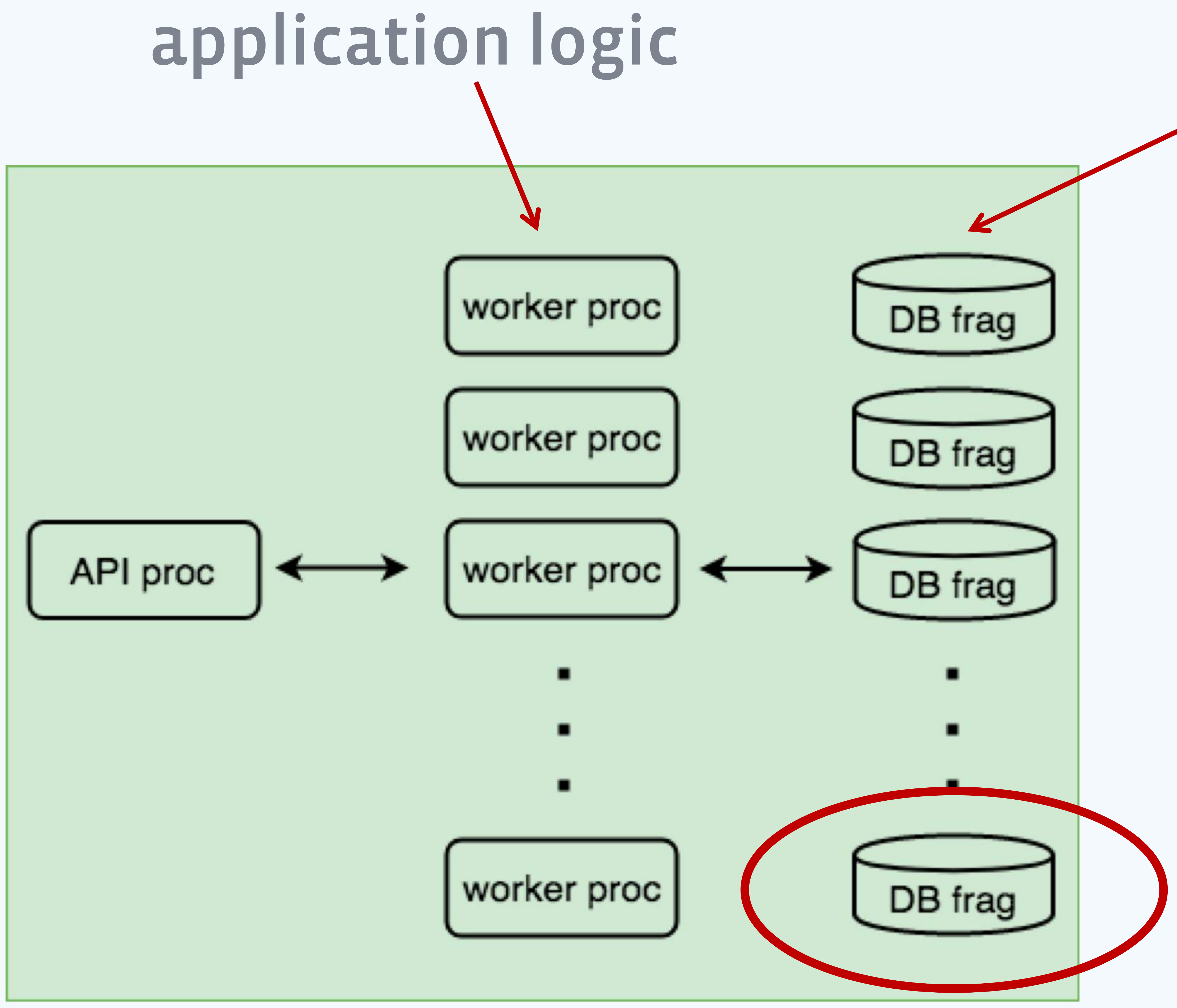
**consistency**

serialize operations for a key to avoid explicit locking

minimize lock contention in DB frag

**efficiency**

# Backend node internals




# workers, # DB frags are tunable

key -> worker mapping is deterministic:
    hash(key, ?NUM_WORKERS)

goal 1: serialize operations for a key to prevent inconsistency

goal 2: minimize lock contention in DB frag on concurrent access

zooming in…

# What is DB frag?

ETS table + replication + persistence

hash table

# But really, what is DB frag?

2 options:

Mnesia with async_dirty

heavily patched
and rigid

ForgETS

new, shiny and
awesome

# ForgETS: clean solution for our use cases

- Resilience to network problems
- Automatic reconciliation
- Easier rebalancing for scaling
- Extra features

don’t miss Mikhail’s talk @

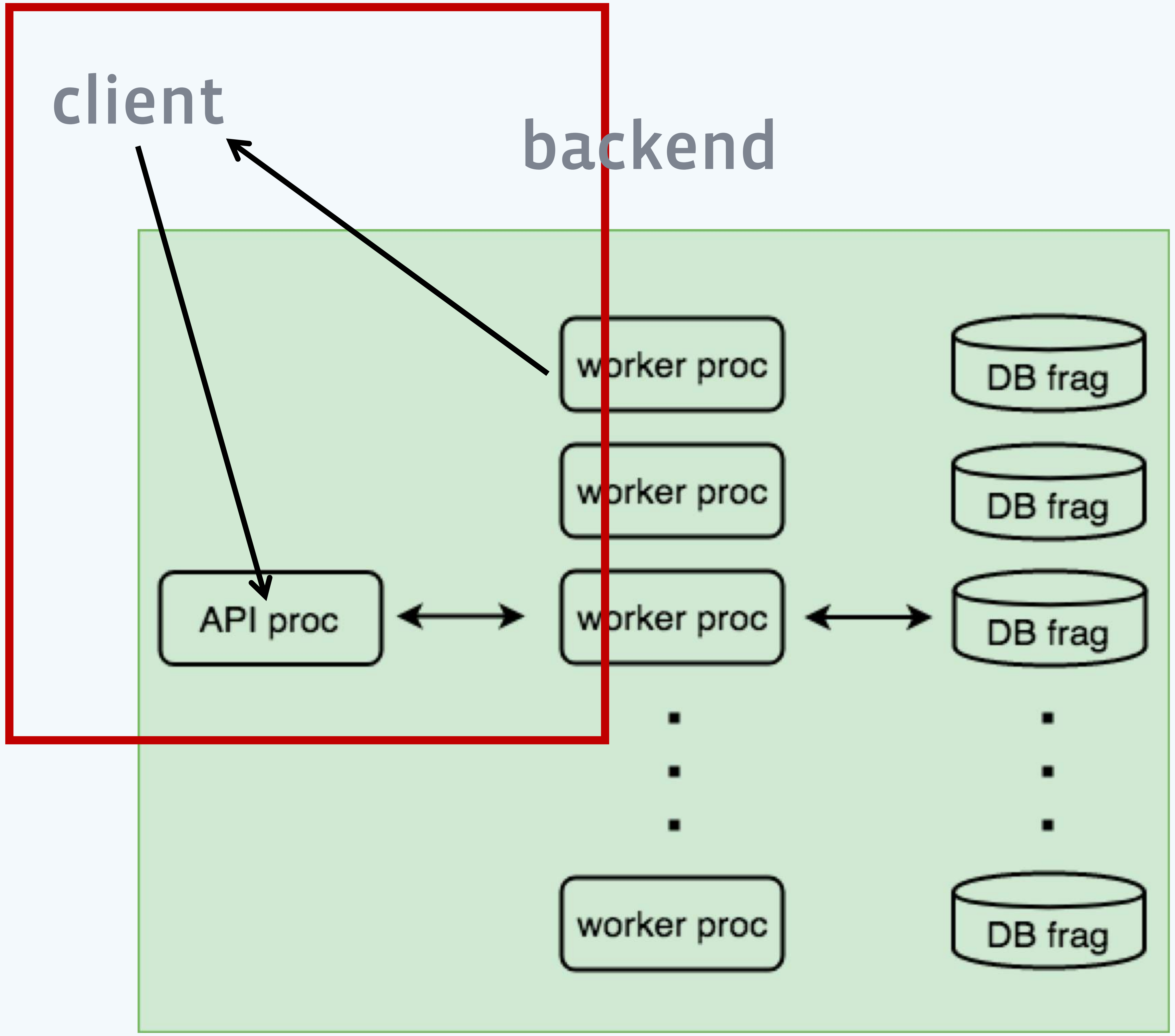
# Benefits Our DB setup

- few moving pieces
- predictable behavior
- efficiency – e.g. short read-modify-write access
- flexible
- scalable
- operated by team who runs the backend

biggest DB was 50B records 2-way replicated across 128 nodes

# Performance

# Optimizing number of messages




client to backend server remote call: 3 messages

sometimes we reduce remote calls down to 2 messages by sending directly to worker

we don’t use gen_server:call/cast cross-node as it requires 2(?) extra roundtrips for remote monitor

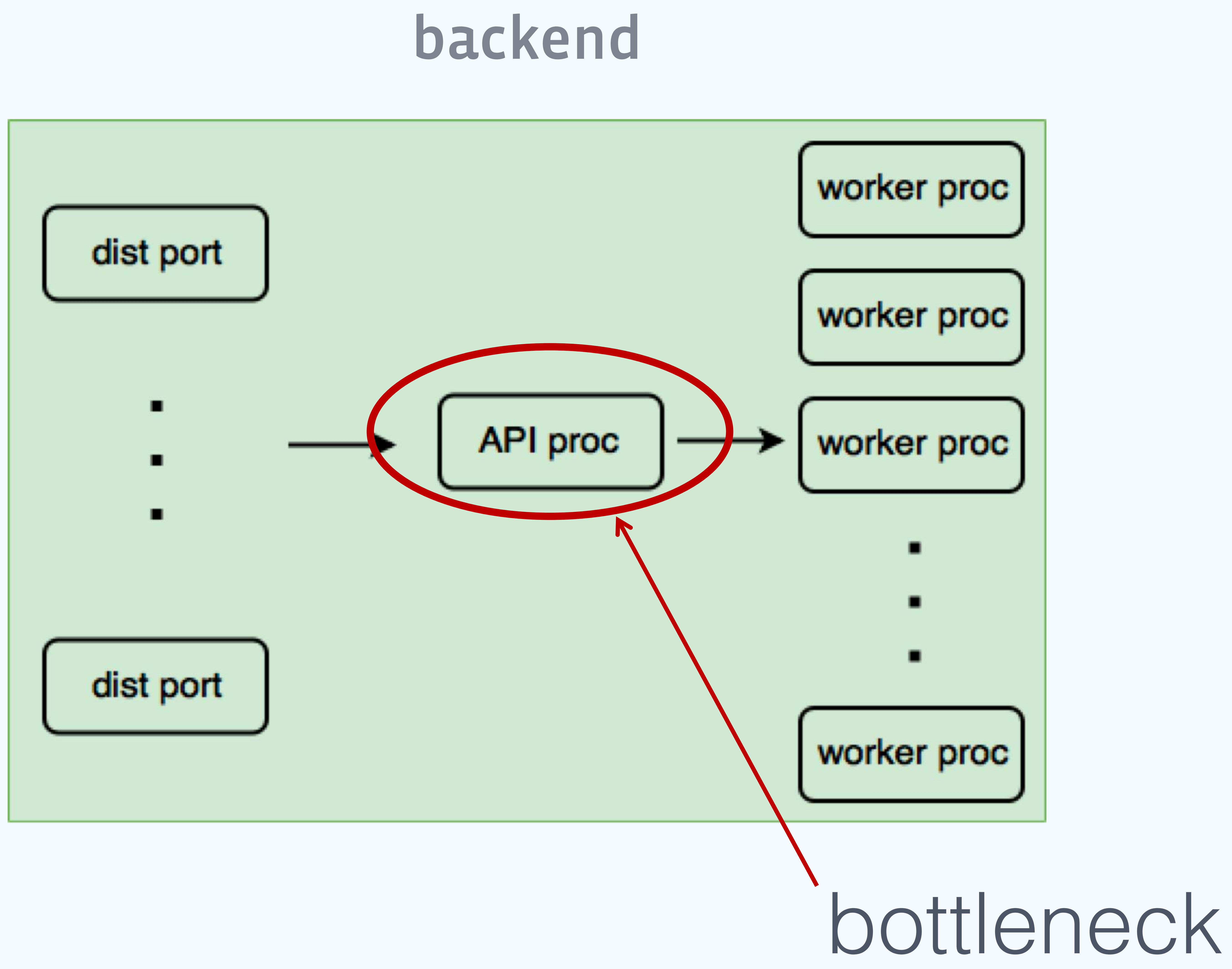
# More on remote calls & round-trips

nodes can become slow, crash, disconnect at any moment, requests can be purposefully dropped

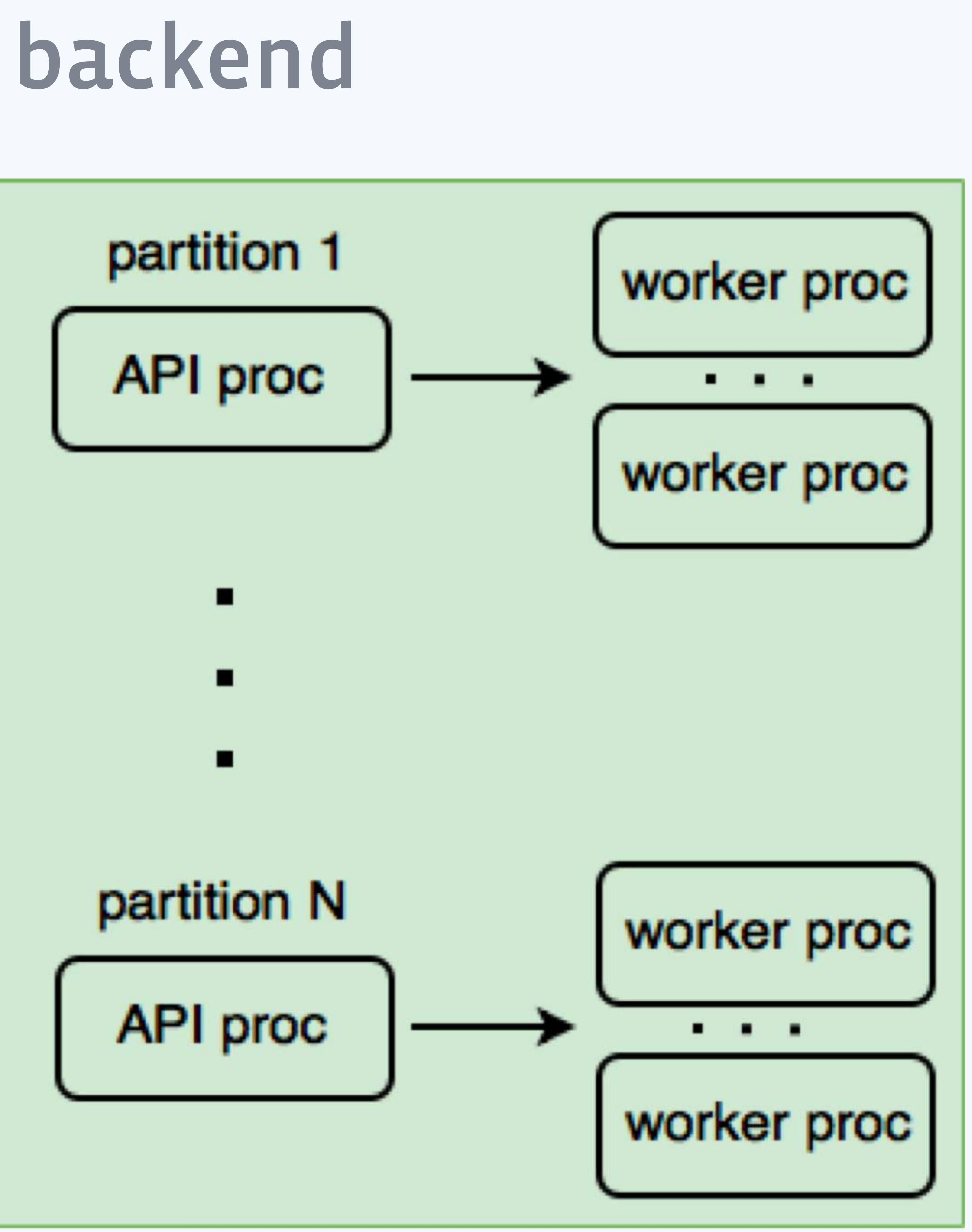
we use timeouts to detect remote calls failures: a single simple model for all type failures

use one-way messages when possible

# Bottleneck example



# Fix bottleneck by parallelizing



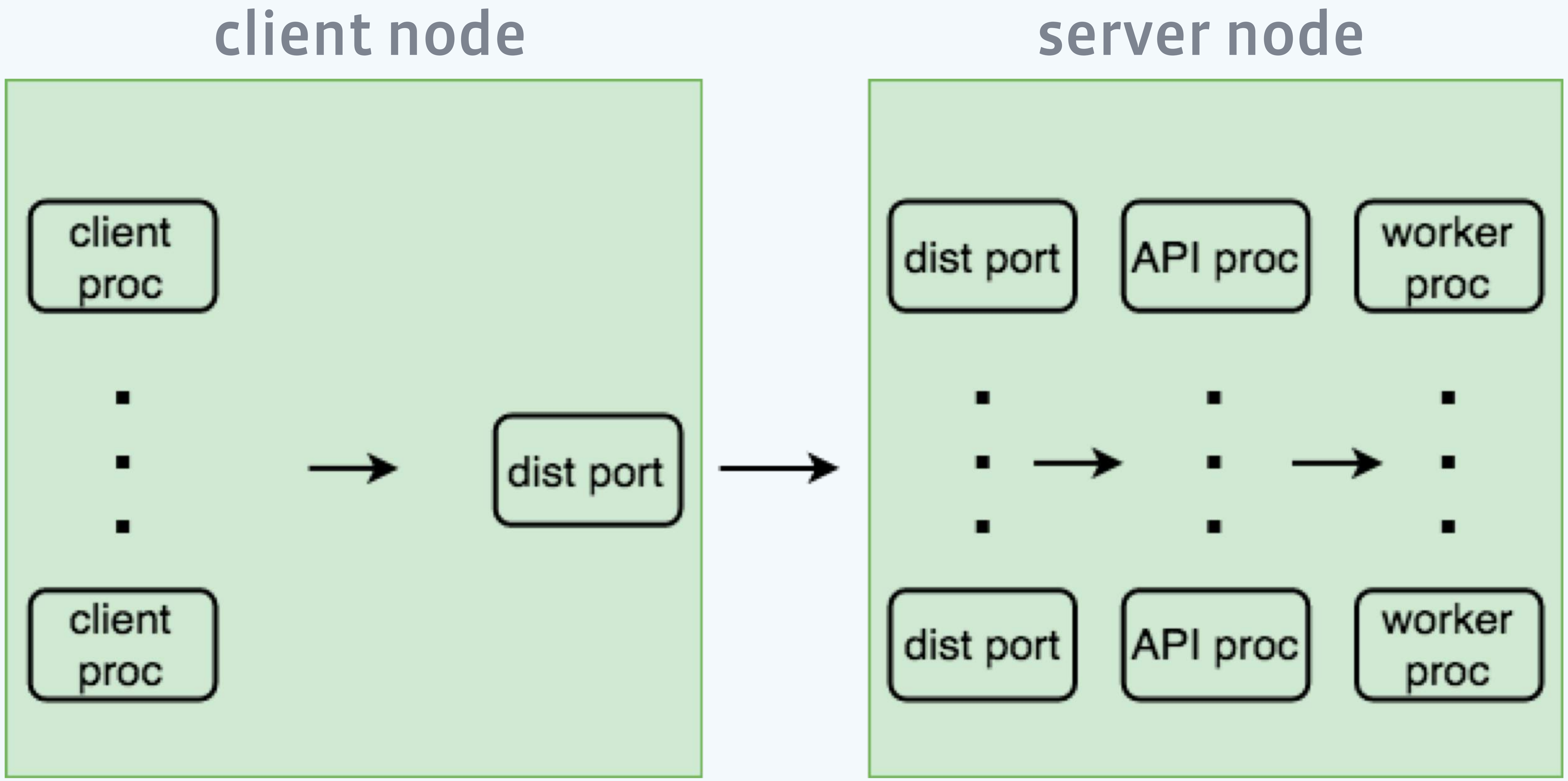
# Handling overload

# Overload: things to consider

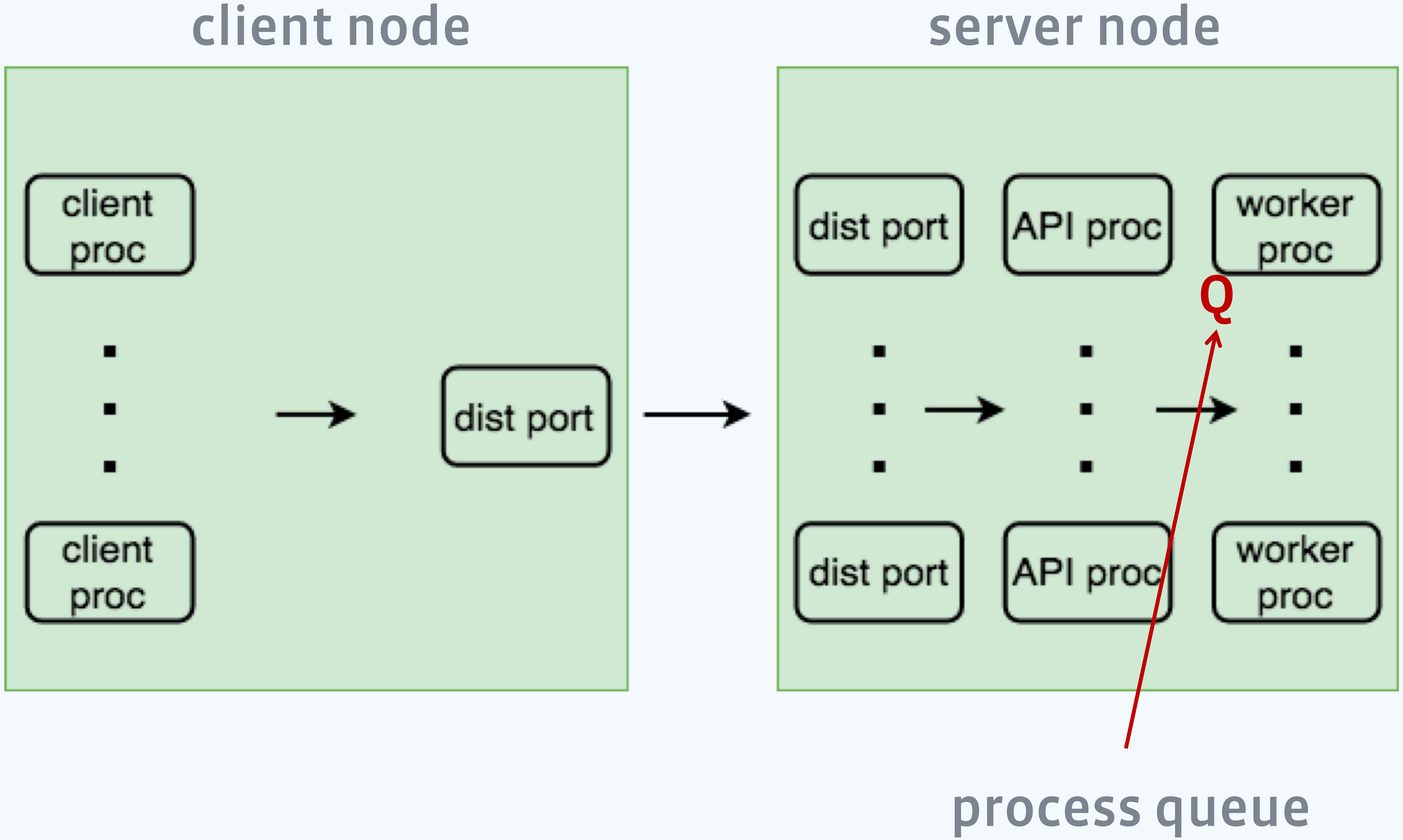
- handle backlog: decide where to queue, how to queue
- decouple: prevent uncontrolled propagation of failures and backpressure through the system

Generally strive to remove all backpressure

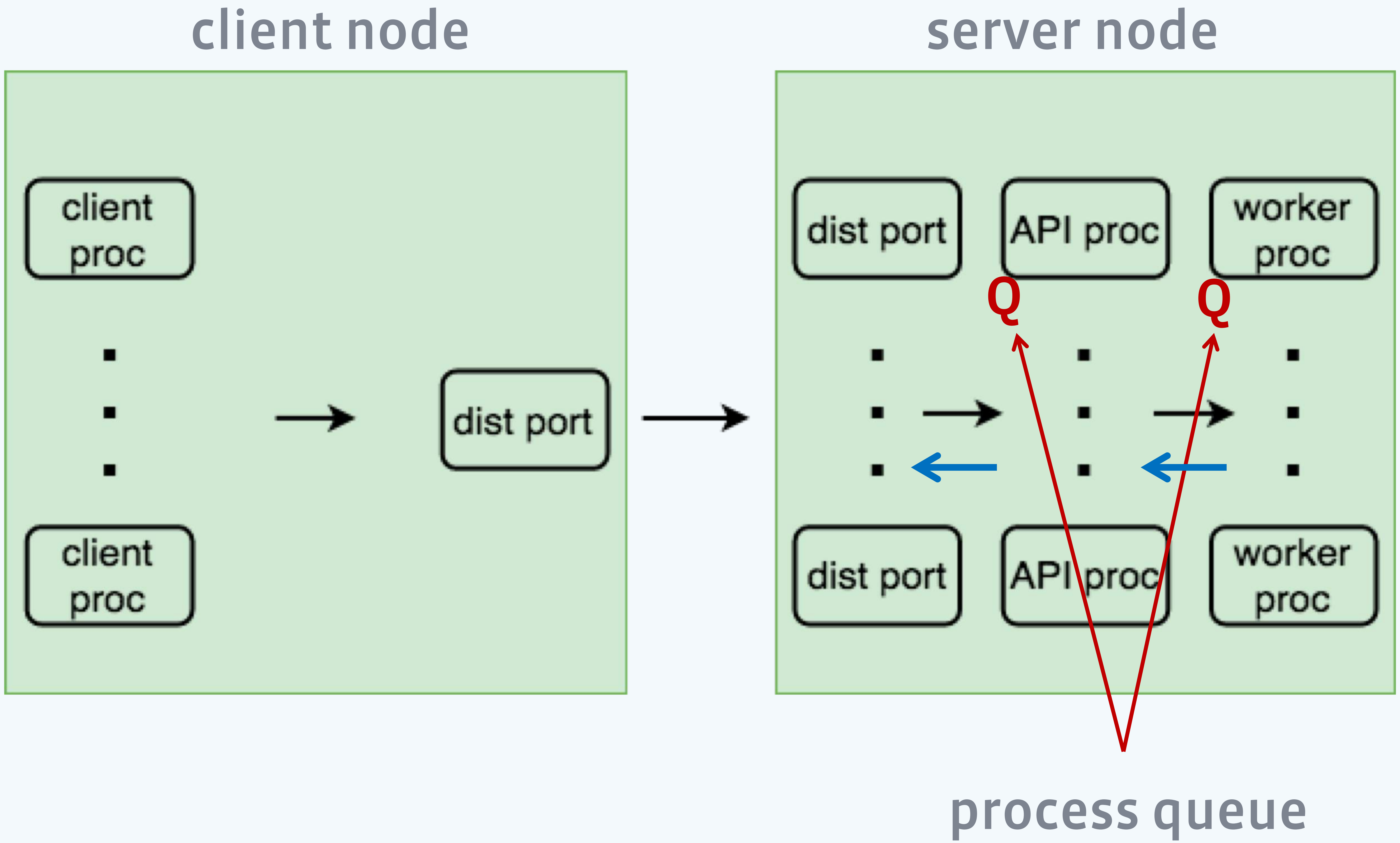
# cross-node request path



# What happens on overload?



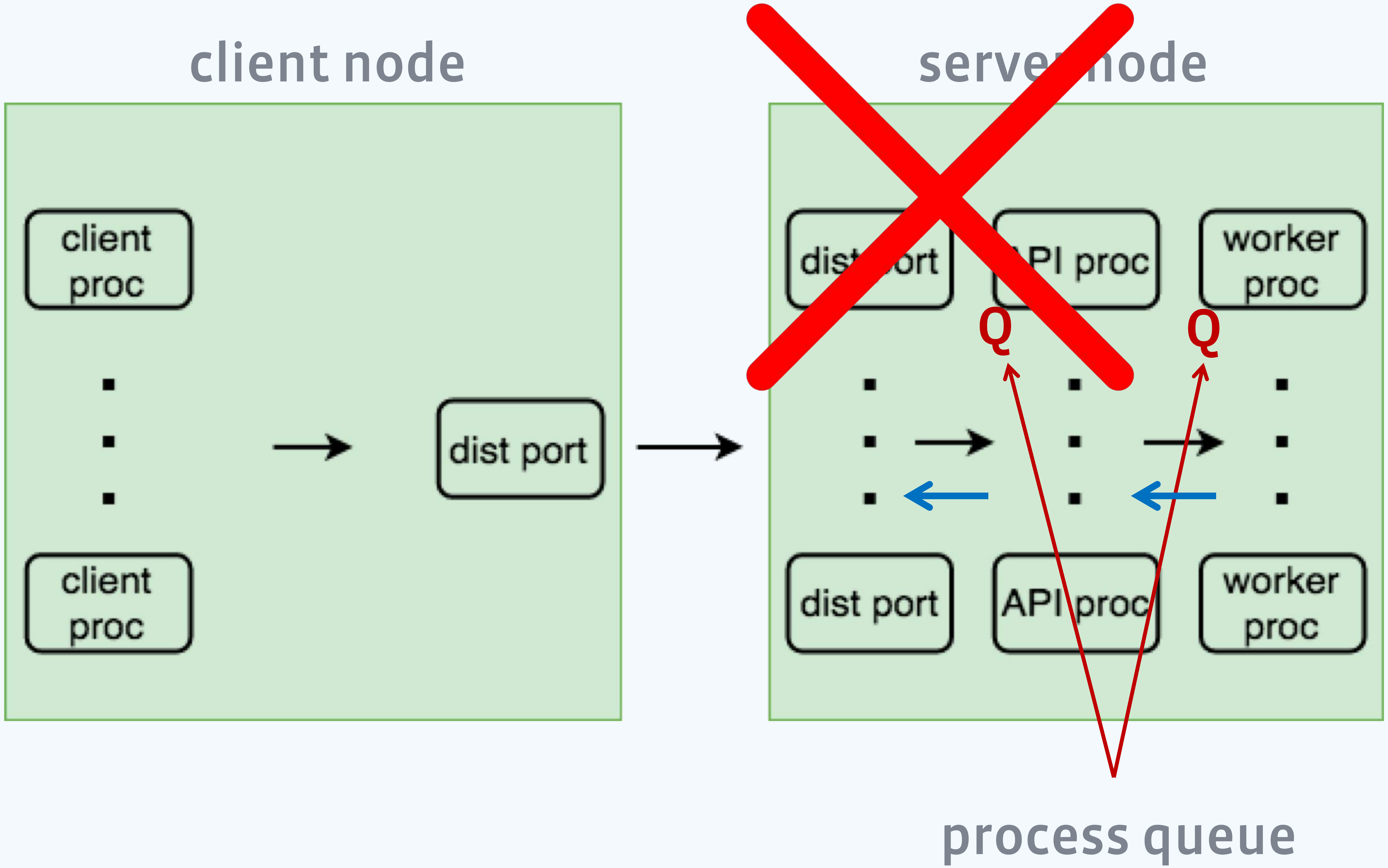
# backpressure and queuing on server node




native Erlang backpressure mechanism:

deprioritizing procs sending into large queue

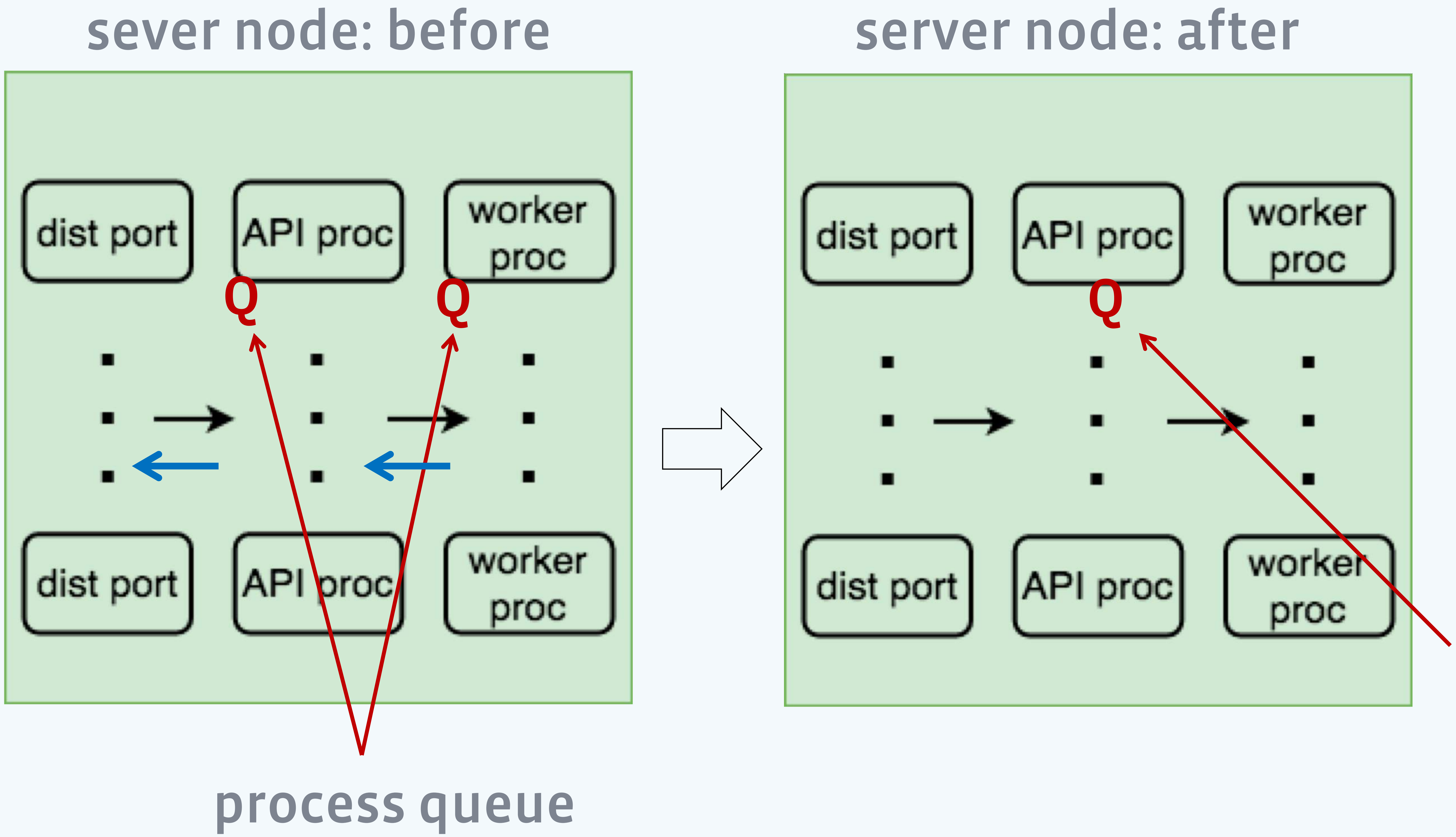
# How it ends: OOM




Erlang process queues are unbounded

With sender penalty, system slows down while queues are growing…

# Solution: custom queue

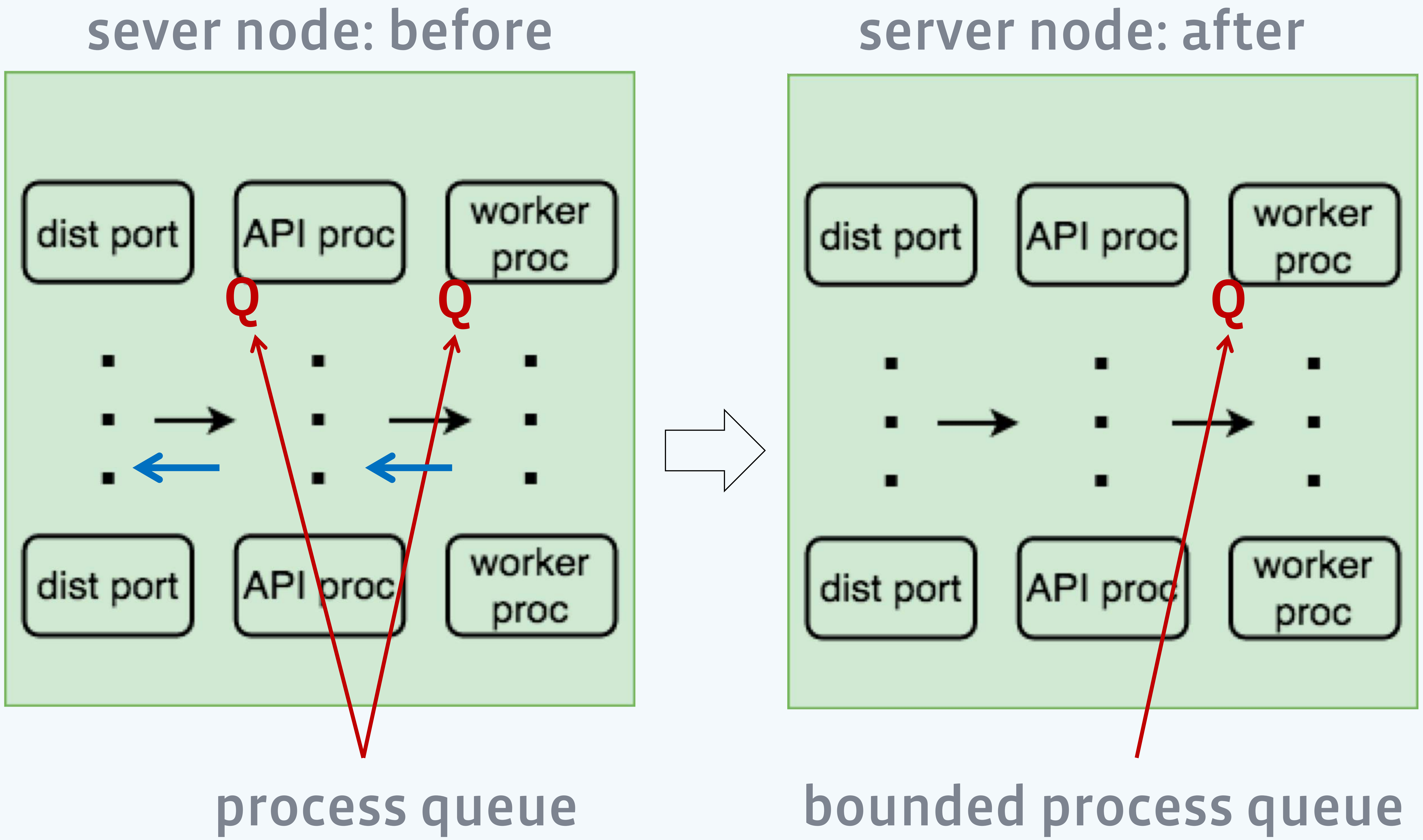



pros:
no backpressure,
can be bounded

cons:
less efficient, we
don’t always use it

worker request
queue

# Solution: bounded worker queue




quickly discard requests when message_queue _len > threshold

pros:
simple and effective

# Solution: discard old requests

keep queue sizes under control by discarding expired requests

based on TTL timeout provided by the client

or based on configurable bound for max request age

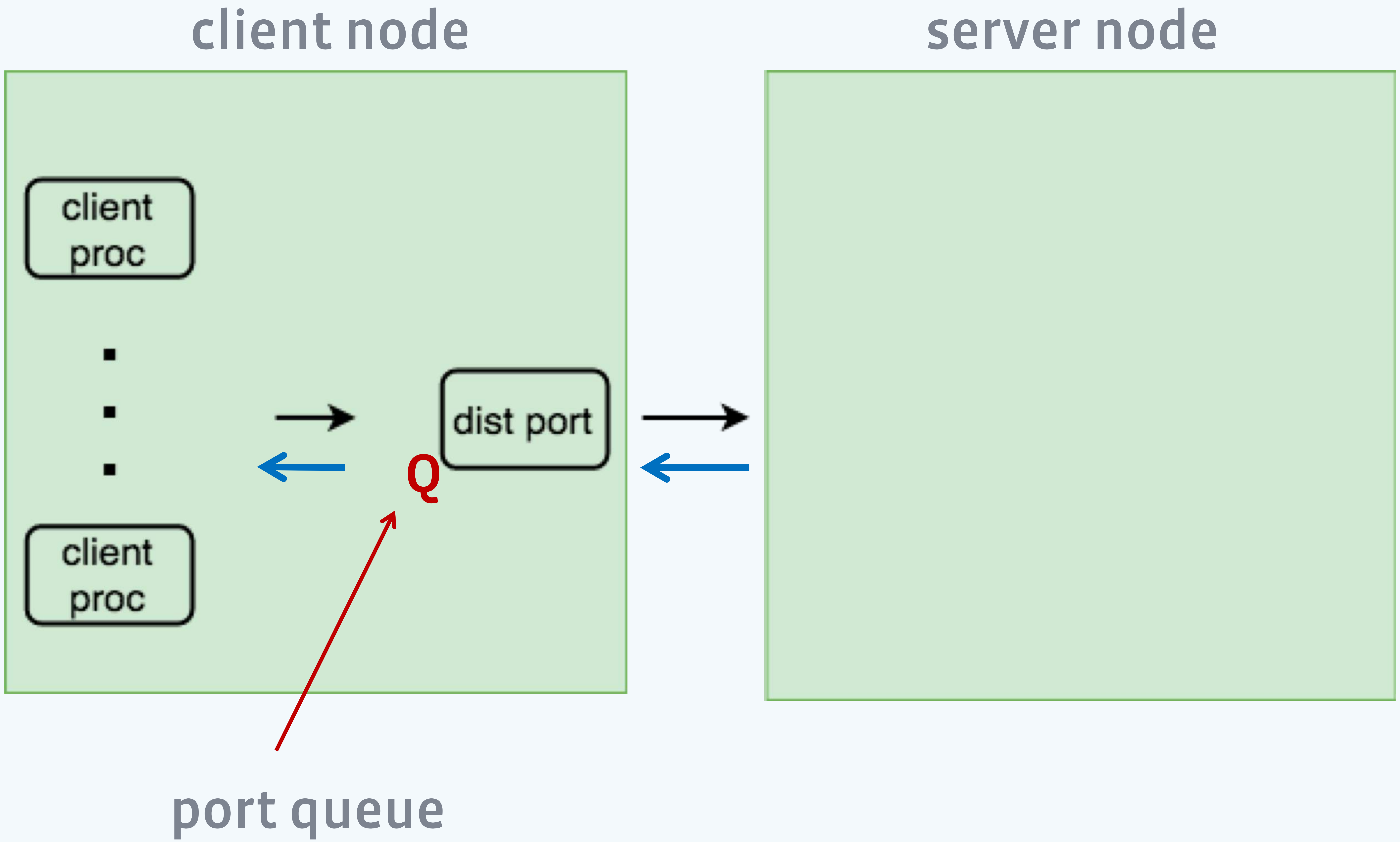
pros: simple and effective
cons: may not always work

# Server toolbox: gen_factory

worker pool with bells and whistles:

- different modes for worker dispatch and key to worker mapping: hash-based, round-robin, first available worker, first available worker with serialization on a key
- bounded queues
- discarding old requests
- workers can have state! — e.g. http client worker pool
- request pipelining
- detecting and killing stuck workers
- integrated with operation and monitoring tools

# backpressure and queuing on client node

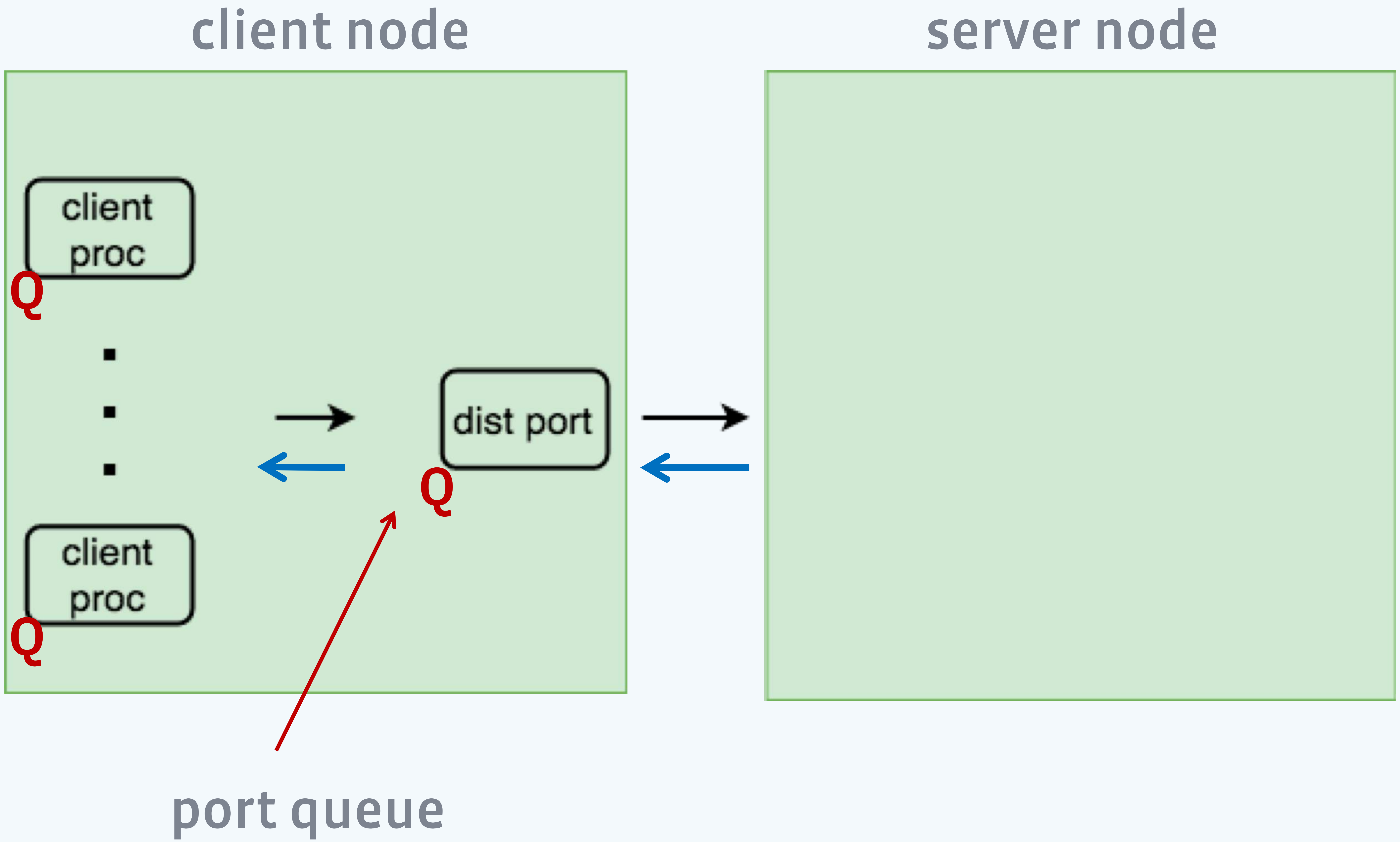



on slow network or slow sever node:

Erlang port queues are bounded

But … suspends clients on busy port

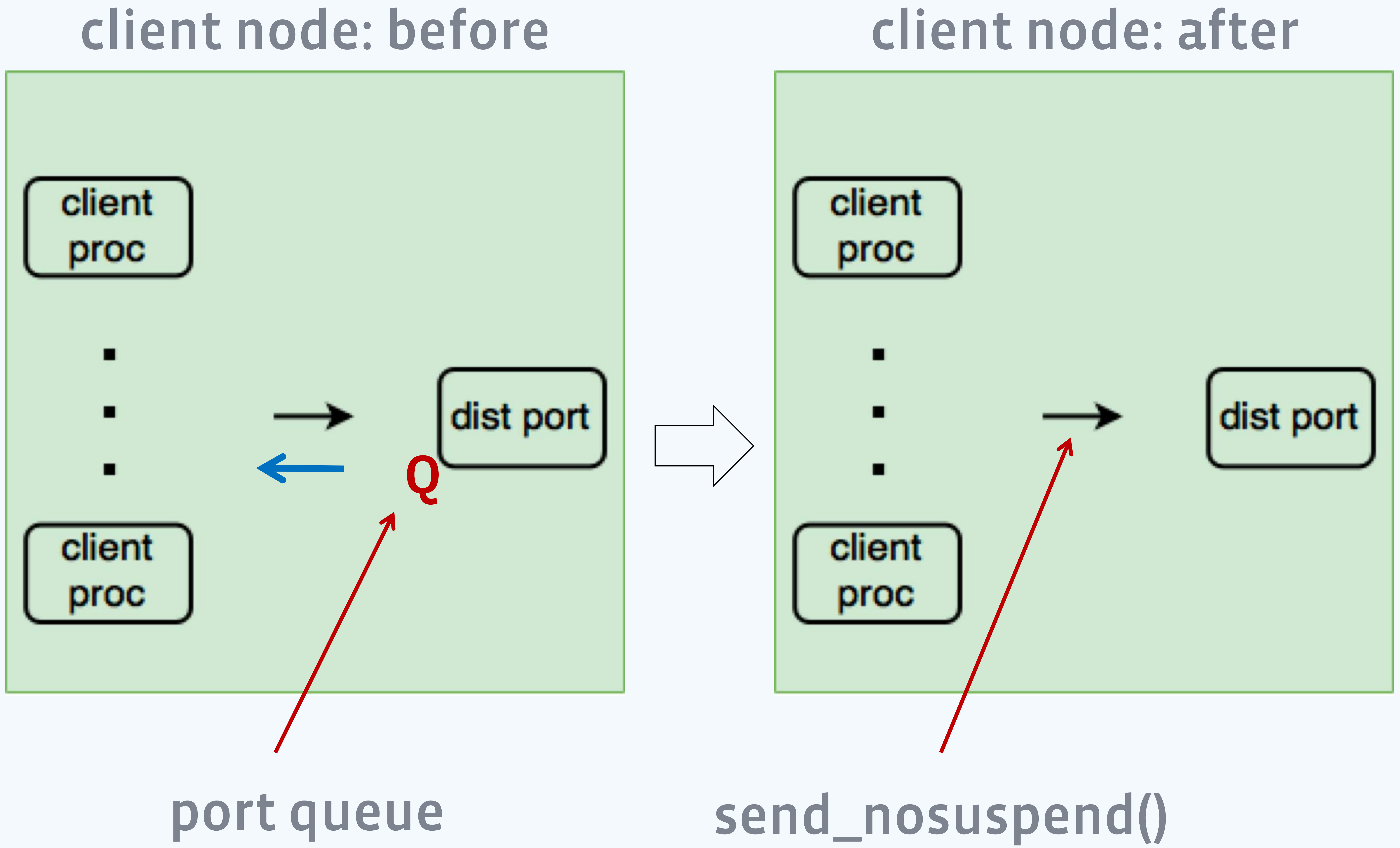
# What could happen?




- OOM because there is a chance there are some unbounded queues waiting on clients

- cascading failure, e.g. in case of chat fronted, clients are also servers

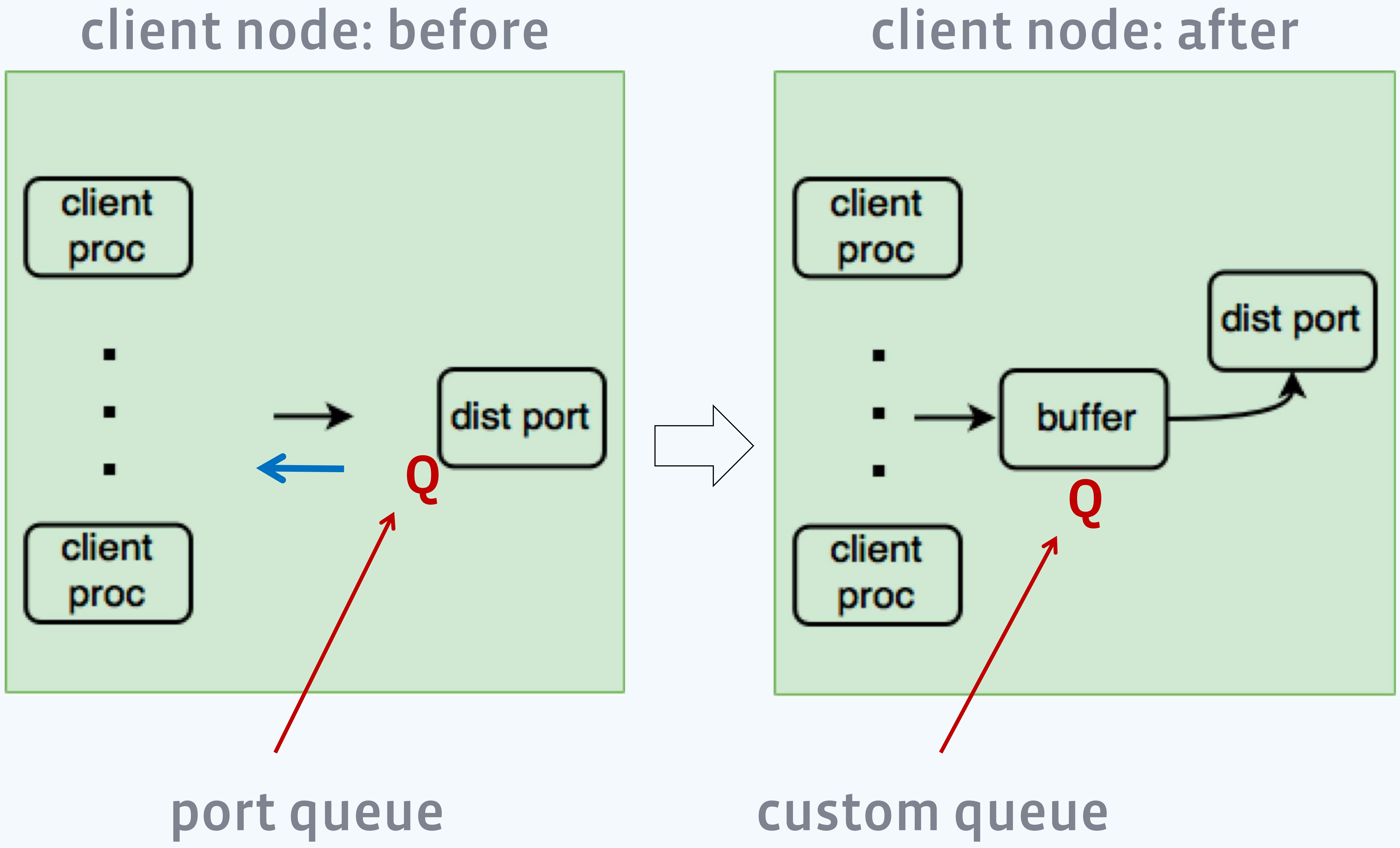
# Solutions: nosuspend, large dist buffers




drop message on busy port, instead of suspending client proc

we also use large dist port buffers

# Solution: custom queue




custom buffer proc in front of dist port

essentially, outbound queue for each destination node

# Solution: wandist

alternative to dist, we use it for connecting dist clusters

- transparently handles TCP reconnect, always connected vs dist. blocking connect
- reliable delivery across reconnects
- always buffering

# Queues and backlog

Type of queues:

- process queue (off_heap in R20)
- port queue
- custom queue (list-based, ets-based)
- process queue, i.e. spawn() – the ultimate concurrent queue
- combination of the above, e.g. worker pool

backlog is the most important operational metric in our system

Erlang makes it easy to reason about it, and handle it*

# How we approach concurrency

Backend:

- gen_factory covers most of our queue management and parallelization use cases
- in some cases we run it with 1000s of workers, to help with throughput when working with external resources
- in any case, concurrency is bounded

Frontend:

- 500K concurrent processes handling mobile client sessions
- gen_factory API

Ad-hoc?

- we rarely spawn() directly, but it is useful sometimes

# Erlang cluster

# Dist

Dist works well for our use case:

```
> length(nodes()).
1203
```

Problems:

- full mesh connectivity
- limited scale, not flexible

# Wandist: scale beyond dist

- only connect clusters that talk to each other
- publish pg2 groups across clusters
- transparently handles, slow network TCP reconnect, always connected
- reliable delivery
- auth & encryption
- implemented in Erlang on top of gen_tcp => less efficient than dist

# Operations

# Common failures and solutions

sick or crashed node: hardware problem, bad code push

quickly crash/stop to allow automatic failover

backlog

less obvious, the goal is to prevent failure propagation

in bad cases, we have to fully gate the system by preventing clients from logging in

# Monitoring: we’ve got backlog!

we get alerted when when there is backlog:

- the node has been running with large queues for some time

- worker queue > threshold

- discarded requests in bounded queues

# Monitoring: queue sizes and why

second-by-second BEAM stats:

- total size of all queues
- queue size of the proc with max queue
- total number of procs with non-zero queue

- internal message rate
- inbound message rate
- outbound message rate
- scheduler utilization

- more granular: gc, scheduler, ports and io

# Monitoring: where is the backlog?

log for each process with queue:

- name & type
- how bad it is: enqueue & dequeue rates, time spent in the queue, estimated time to drain
- extra details: reductions, heap, current_call, initial call

# Deploys: we love hot code loading

takes several minutes to roll out changes even for large clusters

no restart needed: critical with frontend with active user sessions, and backends with embedded DBs

most deploys are small and done by service owners

caveats: load order, state migration, records

# Our BEAM patches

Beam is getting better with each release.

All of it can be done on vanilla Erlang R20: we patch only for scalability (mnesia, pg2), minor performance optimizations, and monitoring.

# Erlang benefits for us

Too many to list…

But ultimately, it allows us to:

- support product features
- scale
- provide highly-available service
- stay very efficient as engineers *!!!*

# Thank you

Interested? Talk to us.