Building a Task Queue with GenStage, Ecto & PostgreSQL

Evadne Wu
Faria Education Group
ev@radi.ws; twitter.com/evadne; github.com/evadne

REVISION DATE
17 April 2018

Outline

📝 Use Cases

💡 Basic Scheduling Concepts

🏗️ Designing Task Systems

🚦 Regulating Resource Use

🔍 Monitoring Task Systems

📦 Scaling Task Systems

🔮 Future State

Expected Benefits

You will have a generic understanding of how to design Tasks and Executors, with focus on their use within Web applications.

You will receive some anecdotal evidence on performance and operational characteristics of particular solutions or components, that you may use.

You will see some code snippets that may be of use, but things that are more or less standard will not be repeated in this deck.

1

Use Cases

Our Use Cases

Ingest Documents and send them out for pre-processing

- Another service takes the Conversion Requests and feeds results back asynchronously

Import Documents and Annotations from external services

- Migrations, done in batches

Broadcast Events regarding newly created Annotations and Comments

- To live user sessions, and Services that have subscribed for these Events

…and many other uses for Job Queues

My Team

Commercial

- Embeddable Document Previews & Annotation for Web Applications
- Coursework Collection System with Deep Content Introspection
- System operations for things we built (exposed as "Managed Services")
- Other Java, Ruby, C pixie dust as required

Non-Commercial

- Various OSS Pull Requests and Projects

2

Basic Scheduling Concepts

Task Deadlines

Type	If Missed	Applicable?
Soft e.g. Send Emails	🙁 People unhappy	Yes
Firm e.g. Compute Balances	☹️ Results useless	Yes
Hard e.g. Guide Missiles	🔥 Very bad things	No

Task Scheduling: Approaches

Type	Scheduling	Efficiency
Schedule Driven	Hardcoded	N/A
Time Sharing	% of CPU Time Slice	Depending on # of Context Switches
Priority Driven	By Task Priority	70% - 100%

Task Scheduling Approaches

Clock-Driven: based on hardcoded schedules

Processor-Sharing: Based on preemptive time-boxes

Priority-Driven: based on priorities

- Priorities determined by programmer (Fixed Priority)
- Priorities determined dynamically by the program (Dynamic Priority)

Clock-Driven

All tasks are arranged on a schedule by the programmer

Scheduler runs tasks, and sleeps in between

Not adaptive and there are no good ways to recover from faults

- Potentially solved by implementing watchdog process

Time Sharing (Processor-Sharing)

Each task is given a certain fraction of time slices to execute

- Quite similar to preemptive multi-tasking.
- However context-switching is not zero-cost.
- The smaller time slices are, the more time is spent context-switching

Priority Driven

Each task is given a priority.

- Highest priority task runs.

Dynamic Priority Example

- **Earliest Deadline First (EDF):** task with least time left to run first.

Fixed Priority Examples

- **Rate-Monotonic (RM):** Tasks with smallest period first.
- **Deadline-Monotonic (DM):** Tasks with closest deadline first.

Task Scheduling: In Practice

Aspect	Non-Web	Web
Slowdowns	Taken seriously. Systems reworked.	#YOLO (Probably no SLA).
Breaches	Taken seriously. Avoid lawsuits.	Apologise on Twitter.
Validation	Check all tasks run and completed by deadline.	Check if app feels snappy on dev laptop.

Trickling Down of Robustness

Observation: We operate in a world with drastically higher tolerance for faults, and drastically lower cost of recovering from errors.

- Hug Ops works when there is no loss of human life involved
- Our worst case scenario is mere inconvenience or embarrassment

Hypothesis: This should make it quite easy to do an acceptable job.

- We can therefore aim for a well done job.

Old vs. New World Mindset

Aspect	Old World	New World
Place	On-Premises mostly capex	Cloud mostly opex
Capacity	Mostly Fixed not without costs	Mostly Elastic to a certain point
Intervention	Optimise Code try to fit code in servers	Autoscale get more servers to run code

Old vs. New World Mindset

Cheap and on-demand compute makes most problems solvable with money.

- Solution caters to the “rest of us” with a reasonable amount of resources.

Externalities of poor software architecture and performance are greatly deferred.

- Once painfully expensive at an early stage, these problems are now deferred further
- They are almost invisible for more use cases that previously required larger investment.

Conclusion: We are now freely able to achieve greater scale.

- **Caveat:** until it is no longer feasible.

3

Designing Task Systems

3

Designing Task Systems

Classifying and Designing Tasks

Basic Classification of Tasks

Type	Example	Can Parallelise?
One-Off Unordered	Send Email	Easily
One-Off Serial	Publish Changes	By Entity
Periodic (Batch)	Rubbish Removal	Probably

One-Off Unordered Tasks

Tasks either do not have side effects, or are idempotent.

- Validity not coupled to state.
- Failures can be retried without extensive eligibility checks.
- Good candidate for first attempts.

Tasks do not need to be executed in-order.

- Easily parallelised.

One-Off Serial Tasks

Tasks may have side effects or build upon each other.

- May require ordered execution.
- Retrying a single task out-of-order may cause issues.

Not easily parallelised, but partially amenable to partitioning.

- Perhaps by unit of absolute isolation, e.g. Customer Account.
- Probably the same unit of isolation used in multi-tenant applications.

Periodic Tasks

Often operate on a large number of entities at once.

- May not be feasible to identify the underlying entities ahead-of-time.
- May be too expensive to enqueue one-off tasks (due to context switching costs).
- May require all-or-nothing among the entire batch.

Probably parallelised by batch.

- Depending on whether partial completion is acceptable.
- Depending on expected rollback behaviour as well.

Other Considerations

Cancellation: can a task be cancelled in the future?

- These tasks may need to be One-Off Ordered.

Repetition: should a task repeat itself?

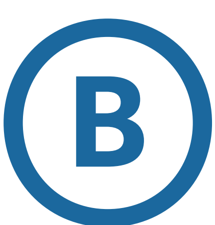
- Repetitive One-Off Tasks can probably be simplified, and re-implemented as Periodic.

Retrying: should a task be re-run if it fails the first time?

- What happens if retries never succeeded?

3

Designing Task Systems



Creating One-Off Tasks

Creating One-Off Tasks

Approach	Scalability	Consistency	Robustness
Application after transaction commit	😄	👻	☠️
Application within transaction	😀	🙂	🙂
Database with trigger function	👌	🙂	🙂

One-Off Tasks: No Transaction

Exactly what happens if you store jobs outside your RDBMS:

- Changes committed to database from application.
- Job enqueued to queue from application.

You can lose jobs to unhandled crashes

- Could mitigate with nonces and tight retries.
- Could sacrifice best case latency to delay processing.
- These are ugly workarounds at most, and should be designed away.

One-Off Tasks: No Transaction

Example: Insert a Job outside of a Transaction

```
def MyApp.Web.DocumentController do
  def create(conn, params) do
    with {:ok, document} <- Document.ingest(params) do
      spawn(&(DocumentProcessor.perform(document)))
    end
  end
end
```

One-Off Tasks: With Transaction

Implies that jobs are stored in the same RDBMS as the changes.

- Either use `Repo.multi` or custom code in `Repo.transaction`.

Easy to copy and paste, but very difficult to simplify.

- Copy and paste is not good, as it can introduce unwanted errors.
- More unwanted complexity does not solve the issue of code duplication.

One-Off Tasks: With Transaction

Example: Insert a Job within a Transaction

```
Repo.transaction(fn ->
  with {:ok, document} <- Document.process(params) do
    Repo.insert(%DocumentJob{document_id: document.id})
  end
end)
```

One-Off Tasks: From Application

Avoid creating One-Off Tasks from application code.

- No matter whether tasks are enqueued within the same transaction or not.
- They can be missed if direct changes were made to the database.
- They might be missed if the application crashes.

Do not treat databases as exclusive, untyped, dumb data stores

- People will try to integrate your databases directly, regardless of design intent.
- Try to encode essential constraints directly into the database.

One-Off Tasks: From Database

Use a trigger function, which writes to the jobs table upon mutation.

- Good for jobs that can not be inferred, or requires retrying.
- See Transactionally Staged Job Drain, by @brandur.

Guaranteed to run in most cases, but can be skipped if needed.

- `SET session_replication_role = replica;`

One-Off Tasks: From Database

Example 1: Remove Remote Objects

```
CREATE FUNCTION jobs.enqueue_remove_objects() RETURNS trigger AS $$
  BEGIN
    INSERT INTO jobs.remove_objects (id, object_reference)
      VALUES (OLD.id, OLD.object_reference);
    RETURN OLD;
  END;
$$ LANGUAGE plpgsql;
```

One-Off Tasks: From Database

Example 1: Remove Remote Objects

```
CREATE TRIGGER enqueue_cleanup_job
  AFTER DELETE ON documents
FOR EACH ROW
  WHEN old.object_reference IS NOT NULL
EXECUTE PROCEDURE
  jobs.enqueue_remove_objects();
```

One-Off Tasks: From Database

Example 2: Issue Charges

```
CREATE FUNCTION jobs.enqueue_process_charges() RETURNS trigger AS $$
  BEGIN
    INSERT INTO jobs.process_charges (id)
      VALUES (OLD.id);
    RETURN OLD;
  END;
$$ LANGUAGE plpgsql;
```

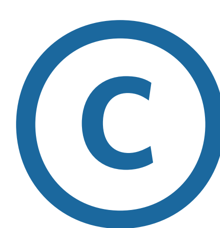
One-Off Tasks: From Database

Example 2: Issue Charges

```
CREATE TRIGGER enqueue_process_charges
  AFTER UPDATE ON purchases
FOR EACH ROW
  WHEN NEW.status = 'processing'
EXECUTE PROCEDURE
  jobs.enqueue_process_charges();
```

3

Designing Task Systems



Designing the Executor

Deciding the Concurrency Model

Single Executor, Single Priority

- Each Consumer takes work; easiest implementation

Single Executor, Multiple Priorities

- Can be controlled by Weighted Polling of Tasks
- Can also be controlled by latency, to avoid lower priority tasks not being run under load
    - Possibly a better fit than Weighted Polling in some cases

Deciding the Concurrency Model

Multiple Executor, each having Single Priority

- Each Executor takes work; basically many copies of Single Priority Executors.

Multiple Executor, each having Multiple Priorities

- A bit more tricky

Levels of Sharing: Overview

Approach	Complexity	Isolation	Contention	Robustness
No Sharing task tables split by type	High	High	Low	OK
Share Everything all tasks in same table	Low	Low	High	Meh

Levels of Sharing: No Sharing

Each Task type has its own dedicated Executor.

- Isolated and easy to monitor, as any metric would already be split by relevant dimensions.
- Each Executor can be scaled independently, as needed.

Each One-Off Task can be represented precisely.

- Arguments represented with correct, dedicated types (or domains).
- Eliminates runtime casting to/from JSONB, JSON or HStore representations.

Levels of Sharing: Share Everything

All tasks held in the same table

- Need to be conscious of page churn
- Any schema change or full vacuum blocks all tasks

No typing support from database

- All task arguments will need to be serialised to/from JSONB, etc.
- Application upgrades that cause task definition changes are now harder to do

Proposed Architecture

One GenStage pipeline for each Executor

- Multiple Executors, each Single Priority
- Clear and defined data flow

Pipelines to dynamically scale, as required

- Orchestrator to observe queue lengths and sojourn time
- Each Pipeline to scale according to suggestions made by Flow Orchestrator
- Allows offloading of decisions that can not be made with only local information

Proposed Architecture

Each Pipeline to expose a load determination function

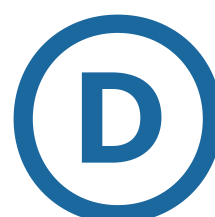
- Returns total amount of workload (normalised) and latency
- Also return expected reduction in latency per workload processed

Orchestrator to start with weighted presets and re-adjust over time

- Initial preset determined by relative priority constant, set at startup
- Takes in observations from Pipelines
- Tells each Pipeline what the target buffer size (normalised) should be

3

Designing Task Systems



Populating the Executor

Fulfilling Initial Demand

Each GenStage Consumer start with a fixed demand size

- Usually scales with number of cores in the system, by default.

Initial demand can easily be fulfilled (in full, in part, or not at all) by queries

- Select up to `N` jobs (with obligatory `SKIP LOCKED`) then update their states.

Unmet demand needs to be fulfilled at a later time when work is available

- Producer must be made aware of additional work that has become available.

Fulfilling Unmet Demand

Unmet Demand must be fulfilled at a later time when more work is available.

- Schedule a tick after N seconds.
- On tick, run a query to grab up to J jobs.
- If J jobs were returned, fulfil all demand and re-schedule a tick after N seconds.
- Otherwise, fulfil any demand outstanding and re-schedule a tick after N + P seconds.

Jobs that were created between Polling Intervals will be processed late

- Worst case latency = MAX(processing time) + MAX(jitter) + polling interval

Async Notifications

Postgres exposes `LISTEN` and `NOTIFY` commands.

- Can be used to make the Producer aware of incoming jobs between Polling Intervals
- Supported in Postgrex.

Asynchronous Notifications are exposed as (Topic, Payload) tuples.

- It can also be used in lieu of polling for new jobs.
- Payload can be useful as well.

Async Notifications: Example

Aspect 1: Trigger Function, which sends Notifications with the NEW row in Payload

```
CREATE FUNCTION new_job_notify() RETURNS trigger AS $$
  BEGIN
    PERFORM pg_notify('new_job', row_to_json(NEW)::text);
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Async Notifications: Example

Aspect 2: Trigger Definition, which calls the Trigger Function on Job creation

```
CREATE TRIGGER notify
  AFTER INSERT ON jobs
FOR EACH ROW
  WHEN NEW.status = 'pending'
EXECUTE PROCEDURE
  new_job_notify();
```

Async Notifications: Example

Aspect 3: Listening for Notifications with Postgrex

```
config = Repo.config |> Keyword.merge(pool_size: 1)
channel = "new_jobs"
{:ok, pid} = Postgrex.Notifications.start_link(config)
{:ok, ref} = Postgrex.Notifications.listen(pid, channel)
receive do
  {:notification, connection_pid, ref, channel, payload} -> # ?
end
```

Async Notifications: Caveats

PostgreSQL uses a single global queue for all Async Notifications.

- See `src/backend/commands/async.c` for implementation details.
- Single global queue, backed by disk, with hot pages mapped in memory.

All listening backends get all notifications and filter out the ones they don't want

- Performance degrades as listeners are added.

Maximum payload size is capped by SLRU (Simple Least-Recently Used) page size

- Probably don't want to send large messages in payloads this way, anyway.

Async Notifications: Caveats

Example: Testing Maximum Notification Payload Size

```
evadne=# select pg_notify('test', string_agg(substr('a', 1), '')) from
generate_series(1, 7999);
 pg_notify
-----------


(1 row)
```

Async Notifications: Caveats

Example: Testing Maximum Notification Payload Size

```
evadne=# select pg_notify('test', string_agg(substr('a', 1), '')) from
generate_series(1, 8000);
ERROR:  payload string too long
```

3

Designing Task Systems

Handling Errors & Timeouts

Errors, Timeouts, etc

Things can go wrong

- Usually we want to capture stack traces on error (exceptions)
- We may also want to see what the Task is waiting on, causing the timeout

Task timeouts can be implemented with `Task.yield`

- On timeout, grab stack trace, kill Task, and have the Task retried later

Stack traces available from `Process.info`

- Many other pieces of information also available

Errors, Timeouts, etc

Aspect 1: Starting a linked Task which runs and monitors the actual Task

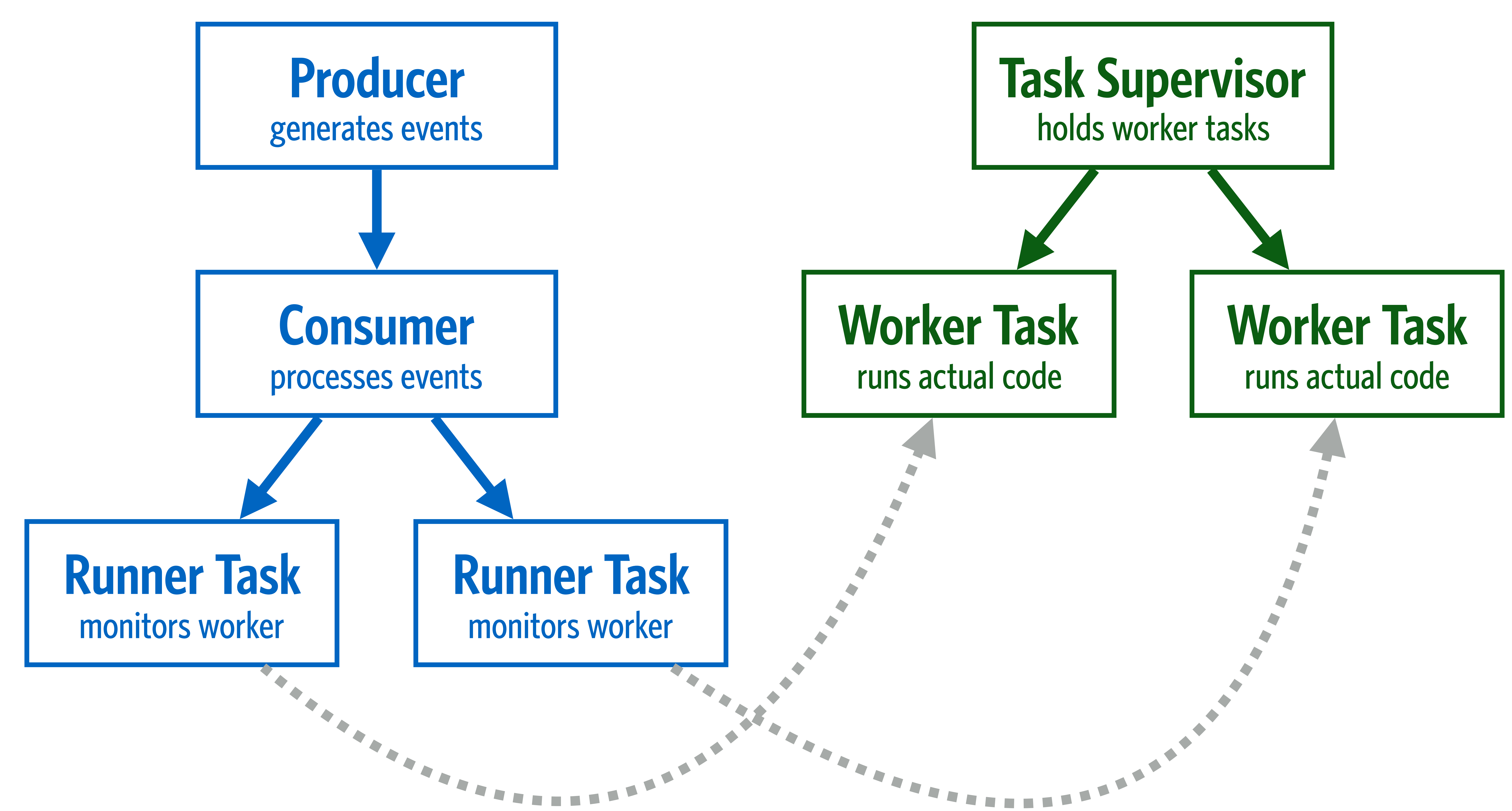
```
Task.start_link(fn ->
  task = Task.Supervisor.async_nolink(TaskSupervisor, &(run(event)))
  result = Task.yield(task, @timeout)
  handle_result(event, task, result)
end)
```

Errors, Timeouts, etc

Aspect 2: Utility function to get Current Stacktrace for a given PID

```
defp stacktrace_for(pid) do
  {_, entries} = Process.info(pid, :current_stacktrace)
  Enum.map(entries, &Exception.format_stacktrace_entry/1)
end
```

Consumer & Runner Supervision



3

Designing Task Systems

Maintaining Periodic Tasks

Periodic Tasks: Characteristics

They frequently depend on driving queries

- They impact a large amount of entities
- They may also generate a large amount of data

They have looser deadlines, and their results may be all-or-nothing in nature

- Faster availability of partial results may or may not be welcomed

Periodic Tasks: Examples

Example 1: Remove Rubbish

- Clean up soft-deleted objects past their retention periods

Example 2: Generate Reports

- Month end reporting, CARR rollups, etc.

Example 3: Restart Services

- Hopefully not necessary

Periodic Tasks: Considerations

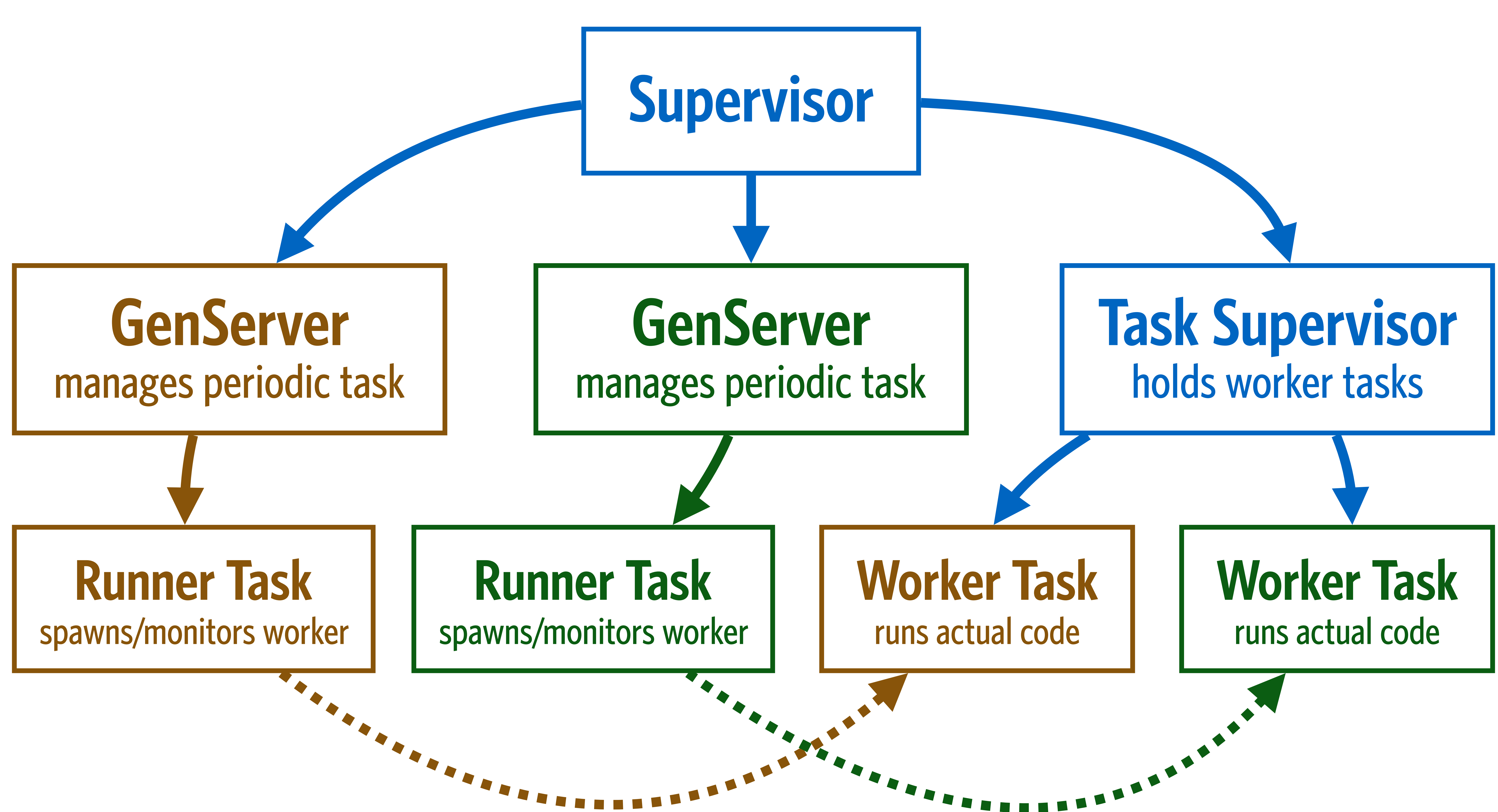
Handling Schedule Overruns

- Kill and restart task
- Postpone next period

Thundering Herd Problem

- Causes of hourly peaks, especially observable when running at scale
- Either vary start time within period, or vary wait time between runs

Periodic Tasks: Implementation



4

Regulating Resource Use

Regulation: Contentious Things

Resource Type	Exhaustion Consequences
Host Resources CPU, Memory, Disk	Service Degradation heightened latency and/or error rate
Service Rate Limits first- or third-party	Service Degradation and/or Monetary charges
Upstream Capacity first-party, expensive processes	Service Degradation and/or Monetary charges
Users' Patience e.g. notifications/day	Unhappy Users and/or churn (long-term)

Regulation: Approaches

The proper approach depends on whether level of resource usage is dynamic

- Sometimes this can be inferred before running the task
- Pre-processing of a file will require the same space on disk as the HTTP request indicates
- May need to spend cycles teasing the origin for appropriate boundaries

Sometimes the Pipeline does not need changing

- Pooling results is enough in these cases
- Example: 1-Click Ordering holds the cart in read/write state, rollup email of updates, etc.

Regulation: External Processes

External OS Processes can be limited using OS tools

- Use `nice` to control prioritisation
- Use `cpulimit` to pin a process to a particular CPU
- Use Control Groups to impose hard limit on resources

Some External Processes naturally degrade over time

- Memory leaks due to bugs or natural fragmentation
- Recycled every X runs or when memory usage exceeds expectations

Regulation: Discrete Resources

Some Resources are discrete

- Number of concurrent database connections
- Number of concurrent sessions to other systems
- …

They can be regulated by use of a Pool and/or a Regulator

- Each Resource is represented by one Process
- The Process is actually used to communicate with the Resource

Regulation: Non-Discrete Resources

Some Resources are not discrete

- CPU time, memory, disk...

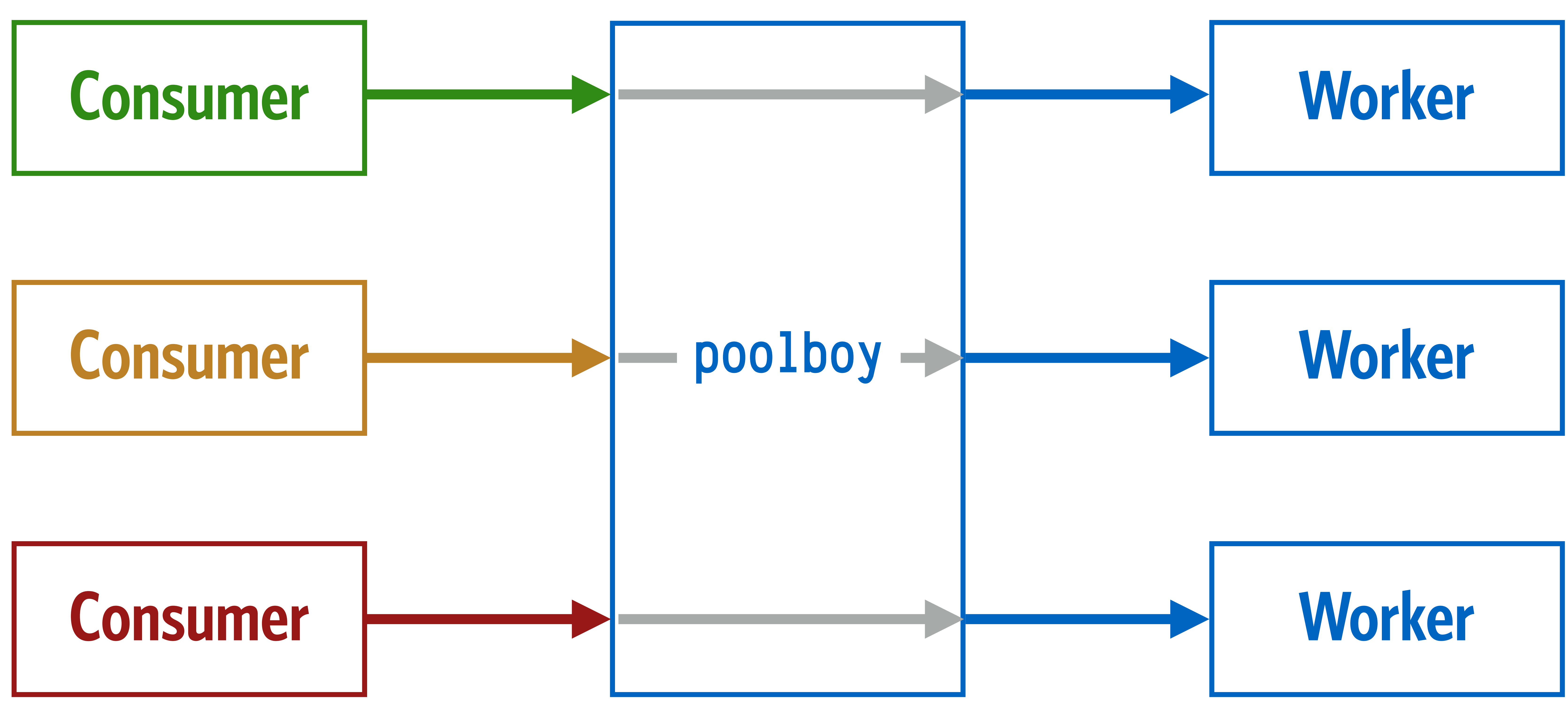
They can still be regulated in the same manner

- Each Resource is segmented into X smaller chunks, each represented by one Process.
- The Process is checked out, but just as a placeholder.

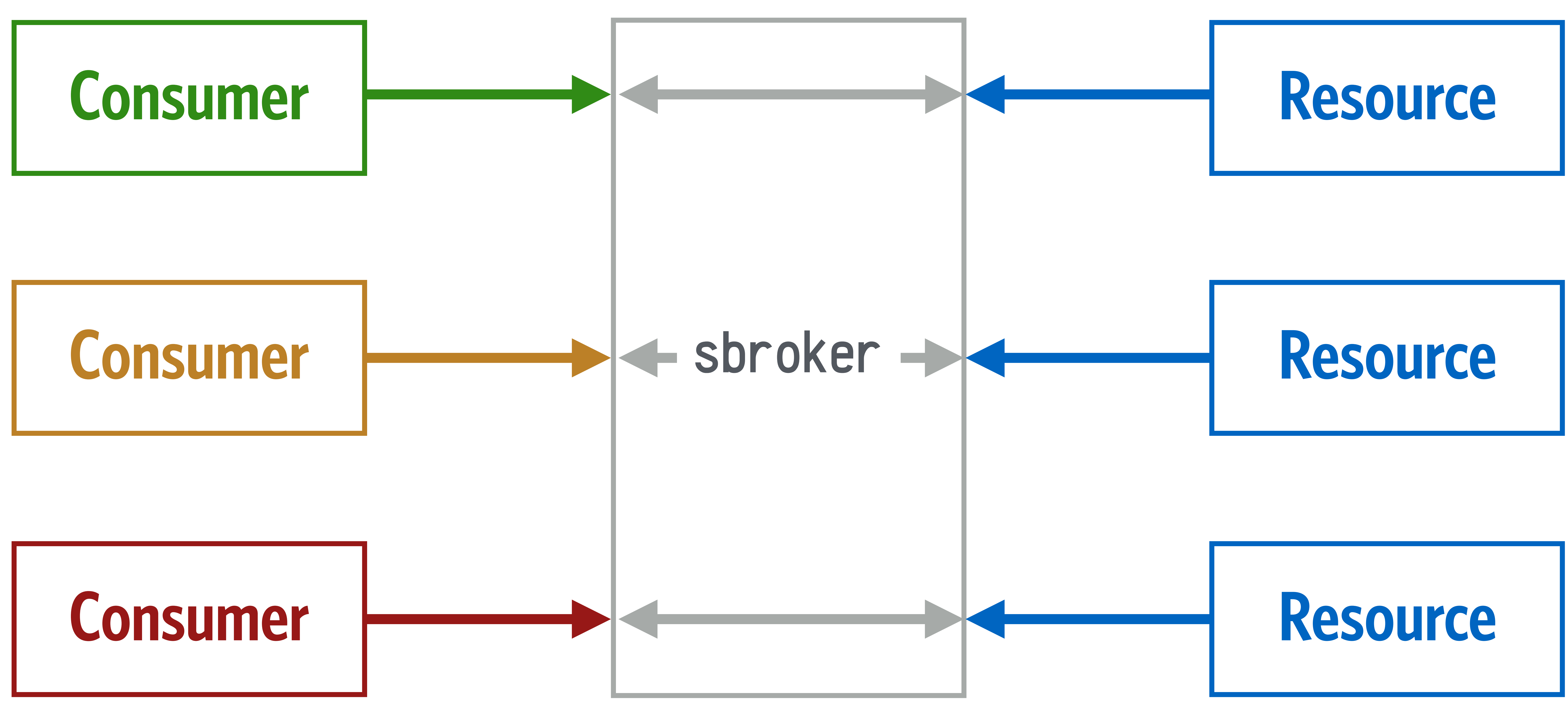
This is a Bin Packing Problem in disguise

- Find most efficient way to pack workloads together, minimising overall latency.

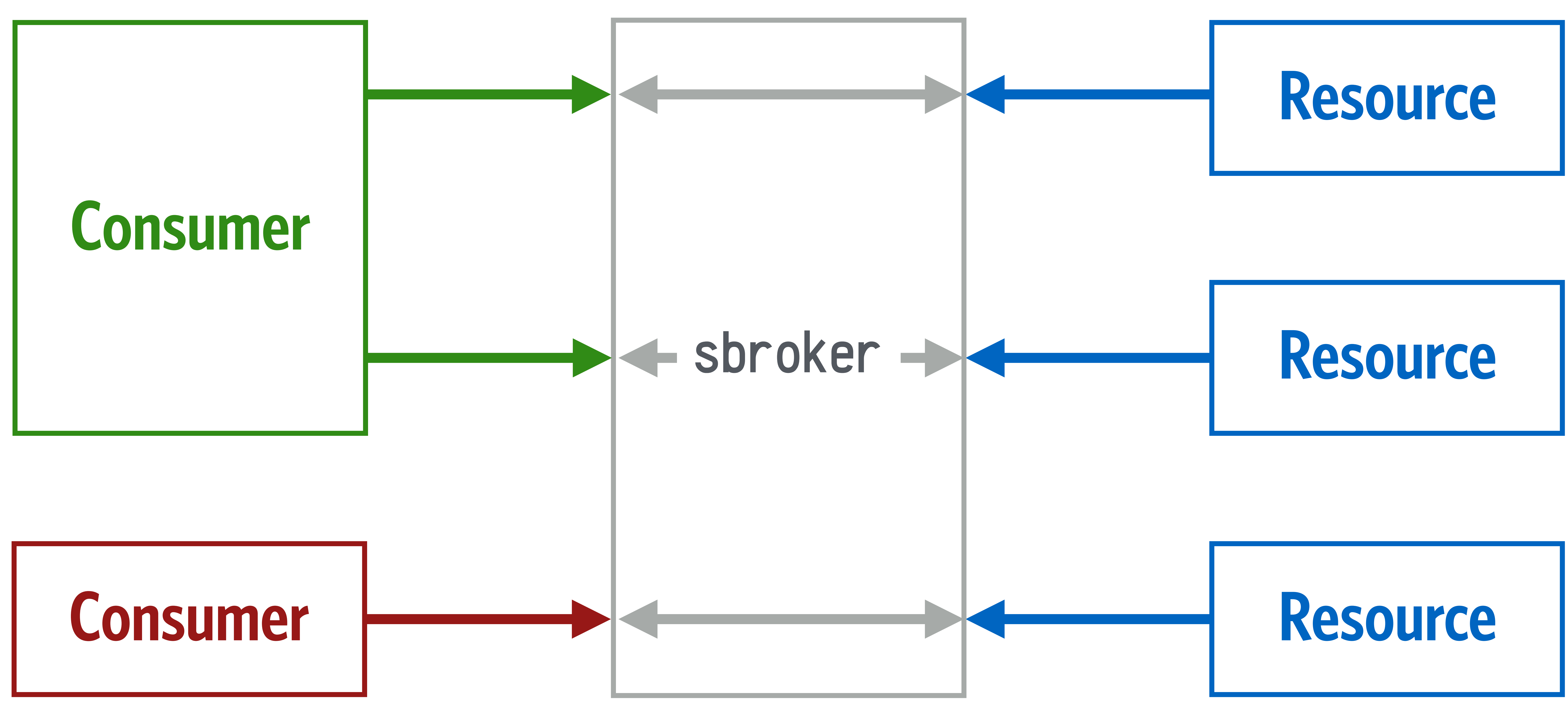
Regulation: With `poolboy`



Regulation: With sbroker



Regulation: With sbroker



5

Monitoring Task Systems

Monitoring: Motivation

Identify Executors that are experiencing delays.

- The number of tasks in each queue is a poor proxy metric.
- Queue Length, Latency and Sojourn Time are different aspects although inter-related.

Need to be able to do both just-in-time monitoring and post-facto analysis.

- Some patterns only emerge over time and can advise preemptive up/down scaling.

Auto-scaling depends on good metrics.

- Good monitoring begets good metrics.

Monitoring: Suggested Approach

Report from every Task Executor.

- One time series per (Task Executor, Host) tuple.
- Can be further aggregated in upstream monitoring systems.

Use Exometer.

- `:histogram` type for task duration and latency; `:spiral` type for task count.

Push metrics to CloudWatch and/or other Monitoring Solutions, like WombatOAM.

- Allows visibility of Infrastructure and Application in the same place.

Monitoring: Suggested Key Metrics

Metrics to Watch Intently:

- Latency
  - Queue Time
  - Processing Time
- Error Rates
- Number of Retries
- Throughput probably matters

6

Scaling Task Systems

6

Scaling Task Systems

Scaling Elixir

Scaling Elixir: Motivation

Desired Attribute	Protection Against	Expected Benefits
Resiliency	Host Failures	Application not dying when a single server dies.
Scalability	Resource Exhaustion	Application not limited to resources of a single server.
Consistent QoS	Large Customers	Single customer unable to degrade service for all.
Easy Deployment	Mutable Hosts	Higher quality of life with Rolling Updates.

Scaling Elixir: Aspects

Preparing Infrastructure

- Needs to allow intra-BEAM networking (as well as epmd).

Establishing Cluster

- Needs to be dynamic if auto-scaling is in place, may require service discovery.

Distributing Work

- Easy for One-Off Unordered tasks but a bit more fiddly for the rest.
- However it is eminently possible.

Scaling Elixir: Infrastructure Prep

Allow traffic to flow correctly

- Probably best to leave epmd in place.
- Ports used by BEAM intra-node communication configurable.
- Assuming EC2 deployment, this is easy to do.
- Assuming ECS deployment, **awsvpc** task network mode attaches ENI to each container.

Scaling Elixir: Clustering

Establish Peers

- The gossip-based mechanism in libcluster can be used.
- Many other ways to get an idea of which nodes to connect to.
  - Can also write a custom strategy to query ECS for running jobs in service.
  - Also possible via Route 53 private hosted zones.
- Select the approach which is suitable for your organisation.

Scaling Elixir: Distributing Work

Expected Benefits

- Server A runs tasks for Customers A, B, C; server B runs tasks for Customers D, E, etc, and they do not require any manual configuration when nodes were added or removed.

Sample Scenarios

- You want to have more than one server without doing the same jobs twice.
- You have One-Off Ordered Tasks that can't be split between 2 Executors.
- You have other forms contention which requires even allocation of Periodic Tasks.

Scaling Elixir: Distributing Work

Work can be scheduled by abusing (repurposing) Swarm.

- Standing on the shoulders of @bitwalker.
- In short, Swarm distributes processed based on a consistent hashing algorithm.
  - Each name is hashed to a particular segment in the ring.
  - Each node is responsible for parts of the ring, which is partitioned across all nodes.
  - Each server runs the relevant processes accordingly.
- These processes do not necessarily need to do any work, their presence is sufficient.

Scaling Elixir: Distributing Work

Node Startup Sequence

- Each node is to run the whole GenStage pipeline
  - Producers pull nothing.
  - Producers are to be told which Accounts to pull down work for.
- Each node is assumed to be in an Erlang Cluster, which requires Clustering to be done.
- Each node boots up, starts, gains membership in Swarm and wait.
- Each node also runs the Enforcer (custom process)

Scaling Elixir: Distributing Work

The Enforcer

- Solves the cold boot problem in case of system hard down: there will be no existing state
- Tells Swarm which Workers are supposed to be running
- The missing Workers are then created by Swarm somewhere in the cluster.
- Essentially runs `SELECT id FROM accounts;` in a loop

Scaling Elixir: Distributing Work

Aspect 1: Enforcer polls for Accounts that should have Workers periodically

```
defp poll(interval) do
  Enum.each(account_ids(), &(enforce(:account, &1)))
  to_interval = round(:rand.uniform * interval + interval)
  Process.send_after(self(), :poll, to_interval)
end
```

Scaling Elixir: Distributing Work

Aspect 2: Enforcer queries Swarm status to determine whether to start a Worker

```
defp enforce(scope, value) do
  name = {scope, value}
  case Swarm.whereis_name(name) do
    :undefined -> register(name)
    _ -> :ok
  end
end
```

Scaling Elixir: Distributing Work

Aspect 3: Enforcer starts a Worker somewhere in the cluster via Swarm

```
defp register(name) do
  supervisor = MyApp.Swarm.Supervisor
  case Swarm.register_name(name, supervisor, :register, [name]) do
    {:ok, _} -> :ok
    {:error, {:already_registered, _}} -> :ok
  end
end
```

Scaling Elixir: Distributing Work

The Swarm Supervisor

- Holds all Swarm Workers
- Supervisor with **:simple_one_for_one** strategy

Scaling Elixir: Distributing Work

Example: The Swarm Supervisor is a Simple One-For-One Supervisor

```
defmodule MyApp.Swarm.Supervisor do
  def init(_) do
    [worker(MyApp.Swarm.Worker, [], restart: :temporary)]
    |> supervise([strategy: :simple_one_for_one])
  end
  def register(name), do: Supervisor.start_child(__MODULE__, [name])
end
```

Scaling Elixir: Distributing Work

The Swarm Worker

- Represents an Account; purely a conduit to various Producers
- Adds the Account to all Producers during init.
- Removes the Account from all Producers during handoff.

Scaling Elixir: Distributing Work

Example: The Swarm Worker monitors and messages the Producer on init

```
def init({scope, id}) do
  target_name = target(scope)
  target_ref = Process.monitor(target_name)
  _ = GenServer.cast(target_name, {:add, scope, id})
  Process.send_after(self(), :join, 0)
  {:ok, {scope, id, target_ref}}
end
```

Scaling Elixir: Distributing Work

Example: The Swarm Worker calls Swarm.join asynchronously

```
def handle_info(:join, {scope, _, _} = state) do
  Swarm.join(scope, self())
  {:noreply, state}
end
```

Scaling Elixir: Distributing Work

Example: The Swarm Worker handles Handoffs

```
def handle_call({:swarm, :begin_handoff}, _from, {scope, id, ref}) do
  _ = GenServer.cast(target(scope), {:remove, scope, id})
  {:reply, :restart, {scope, id, ref}}
end

def handle_call({:swarm, :end_handoff}, state), do:
  {:noreply, state}
```

Scaling Elixir: Distributing Work

Example: The Swarm Worker does no conflict resolution whatsoever

```
def handle_cast({:swarm, :resolve_conflict, _delay}, state), do:
  {:noreply, state}
```

Scaling Elixir: Distributing Work

Example: The Swarm Worker prefers to die quickly

```
def handle_info({:swarm, :die}, state), do:
  {:stop, :shutdown, state}
def handle_info({:DOWN, ref, :process, _, _}, {_, _, ref} = state), do:
  {:stop, :shutdown, state}

def handle_info({:EXIT, _, _}, _), do:
  Process.exit(self(), :kill)
```

Scaling Elixir: Distributing Work

The Producers

- Each Producer holds a MapSet which contains Account IDs.
- The MapSet determines which Tasks the Producer pulls down.
- So far so good.

6

Scaling Task Systems

B

Scaling Postgres

Scaling Postgres: Motivation

Vertical Scaling Still Limited

- There are still global bottlenecks that get worse with usage
  - Notifications are dealt with globally for example

The Maximum Concentration Ratio is fixed

- Number of Application Servers ÷ Number of Database Servers
- Essentially a Single Point of Failure, even if clustered with failover
- Jesus Nut of Software Engineering

Scaling Postgres: Motivation

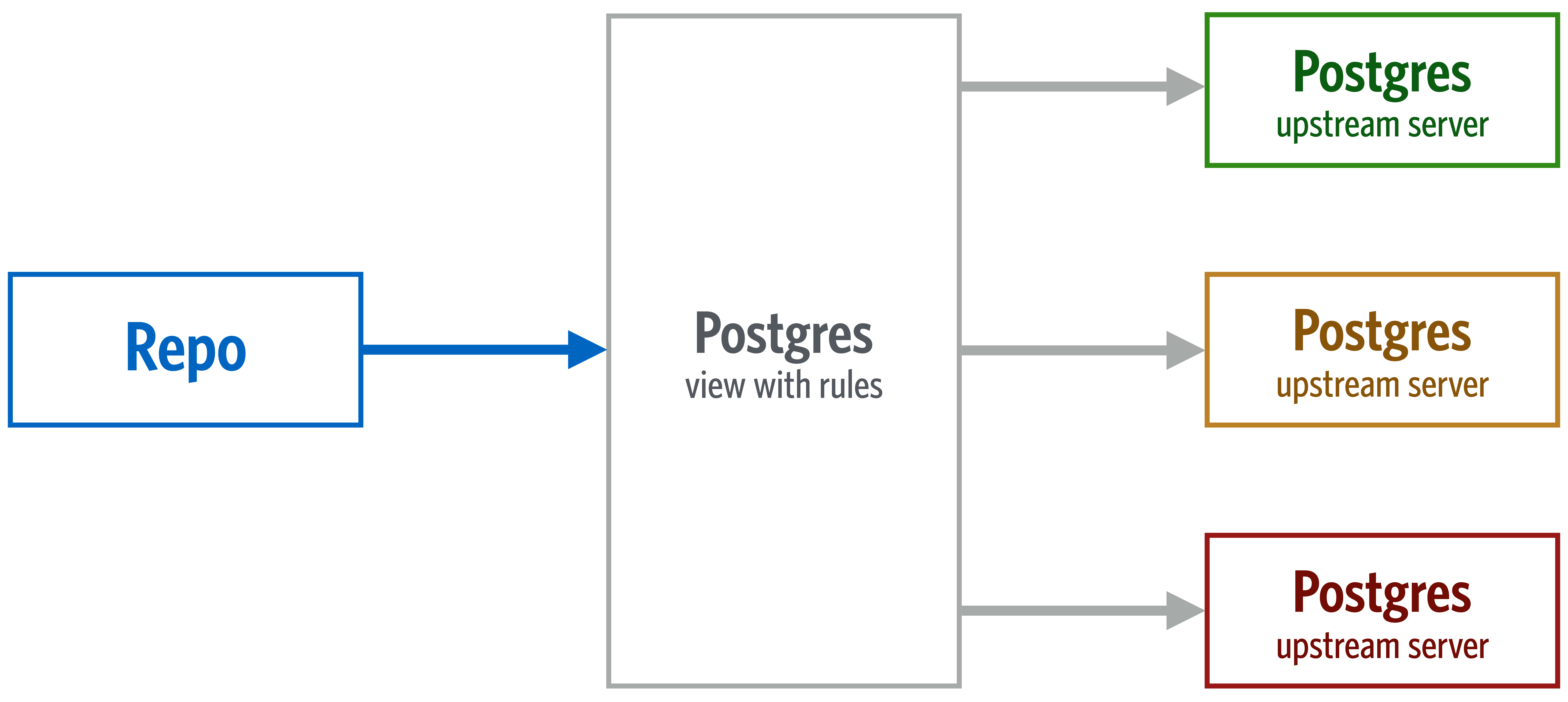
Effort put into Horizontal Scaling pays dividends

- Allows isolated A/B testing of new database engines
- Obviates global outages that accompany even a minor version bump
  - Postgres only guarantees on-disk representation stability among minor versions
- Obviates even partial outages if data is quietly migrated ahead of time
- Allows adding additional resources dedicated for specific Accounts
  - One of many ways of achieving Enterprise-tier SLA

Scaling Postgres: Ready Solutions

Project	Type	Approach
xDB MMR EnterpriseDB	Commercial free trial available	Multi-Master
Postgres-XL NTT	OSS	Multi-Master
pg_partman keithf4	OSS	Partitioning
Postrgres 10 native partitioning	OSS	Partitioning

Scaling Postgres: Single Repo



Scaling Postgres: Single Repo

All tables are actually views, backed by foreign tables

- Parent table can be built with a view that uses UNION + INSTEAD OF update/delete rules.
- **Caution:** FDWs forward per session, so N sessions per shard, with M shards = M*N connections from master instance.

Other Problems (~Postgres 10)

- Views are not really backed by shards and the solution is really wonky.
- DDL operations (migrations) are not run against shards so this is even more wonky.

Scaling Postgres: Single Repo

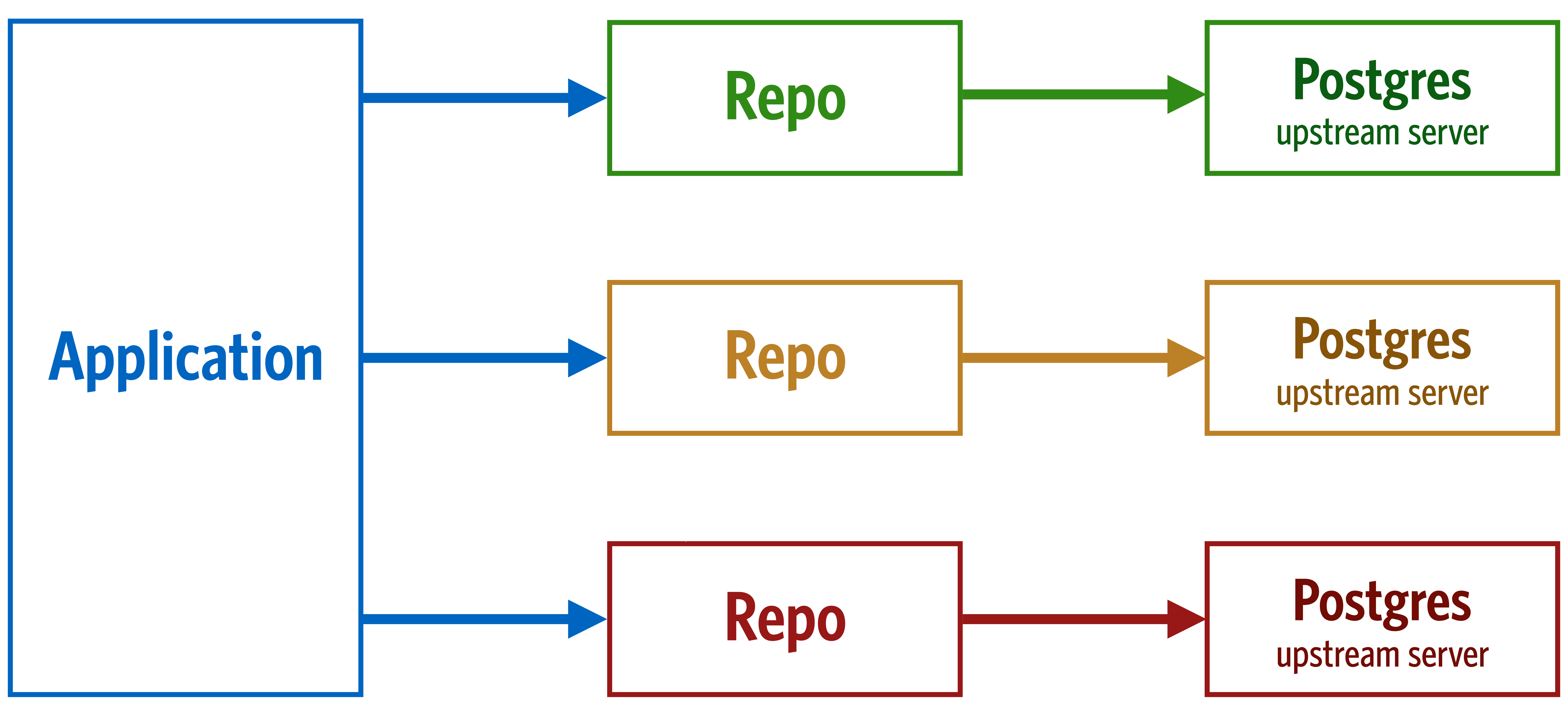
The Future of Postgres Sharding *[March 2018]*

- Shard management will be added to the partitioning syntax, which was added in Postgres 10.

Suggestion

- Buy some time until native support for shard management is added to Postgres.
- Wait until RDS also supports it and the right number of support cases have been filed.
- Don't try to hide the fact that data is now stored across shards.

Scaling Postgres: Many Repos



Scaling Postgres: Many Repos

Each repo corresponds to a separate Postgres instance.

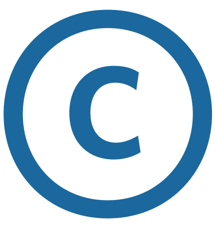
- You can still tie them together with FDWs.
- Useful if you want a single access point for debugging purposes.

You may want to also add a "master" repo for data that is global in nature

- Each shard can still refer to it via FDWs.
- Useful if you have globally defined truths that unfortunately differ, but not per-customer.
- Per-customer data can always be stored in their own shards.

6

Scaling Task Systems



Scaling Infrastructure

Scaling Infrastructure: Motivation

Servers can scale to fit workload

- Not only up but also down (to save money)

Sometimes an extra server buys weeks of engineering time for rework

- Outages are not only annoying, they have technical and human cost
- Fixes made in the heat of the moment are seldom good

Scaling Infrastructure: Approach

Scaling should be based on Host Resource utilisation

- Utilising Upstream Capacity will also raise utilisation of Host Resources.

Scaling should not be based on Executor metrics

- Retain alarms on Executor metrics but don't scale based on it.

Scaling should not be used without Resource Use Regulation

- Calling a service from more servers can leave more concurrent sessions opened.

Scaling Infrastructure: Suggestion

Base Scaling Decisions on Saturation

- i.e. how many normalised hosts remain free (total free % ÷ total %)

Suggested Approach

- Assuming ECS deployment, adjust Desired Count on the Service based on this metric.
  - Target Tracking Scaling Policy + Metric Math, as needed.
- Assuming EC2 target type, adjust Desired Count for ASG or Spot Fleet.
  - Based on Reservation %.

7

Future State

8

References

Reference: Elixir/Erlang Code

Alex Kira & co.: exq

Mike Buhot & co.: ecto_job

Basho: sidejob

Michael Shapiro & co.: honeydew

James Fish & co.: sbroker

Bernard Duggan: Exometer to CloudWatch (gist)

Reference: Articles

@brandur: Transactionally Staged Job Drains in Postgres

@brandur: Implementing Stripe-like Idempotency Keys in Postgres

Chris Hanks: Turning PostgreSQL into a queue serving 10,000 jobs per second

Craig Ringer: What is SKIP LOCKED for in PostgreSQL 9.5?

Robert Haas: Did I Say 32 Cores? How about 64?

Michael Schaefermeyer: Monitoring Phoenix

Hamidreza Soleimani: Erlang Scheduler Details

Reference: Other References

Paper: Real-Time Scheduling Analysis (PDF)

Paper: Errata: A New Algorithm for Scheduling Periodic, Real-Time Tasks (PDF)

Postgres Wiki: Built-In Sharding, Distributed Transaction Manager, Table Partitioning

Postgres Mailing List: Transactions Involving Multiple Postgres Foreign Servers

Bruce Momjian: The Future of PostgreSQL Sharding (PDF; Video)