SBroker

SBroker - Adapting TCP Algorithms to Schedule Tasks

How long is someone going to wait?

- Expectation
- Pay off
- Concentration

Response time in man-computer conversational transactions - Miller 1968

- Response to key press or other movement: 0.1 seconds
- Simple enquiry: 2 seconds
- Impediment blocking activity: 2 seconds

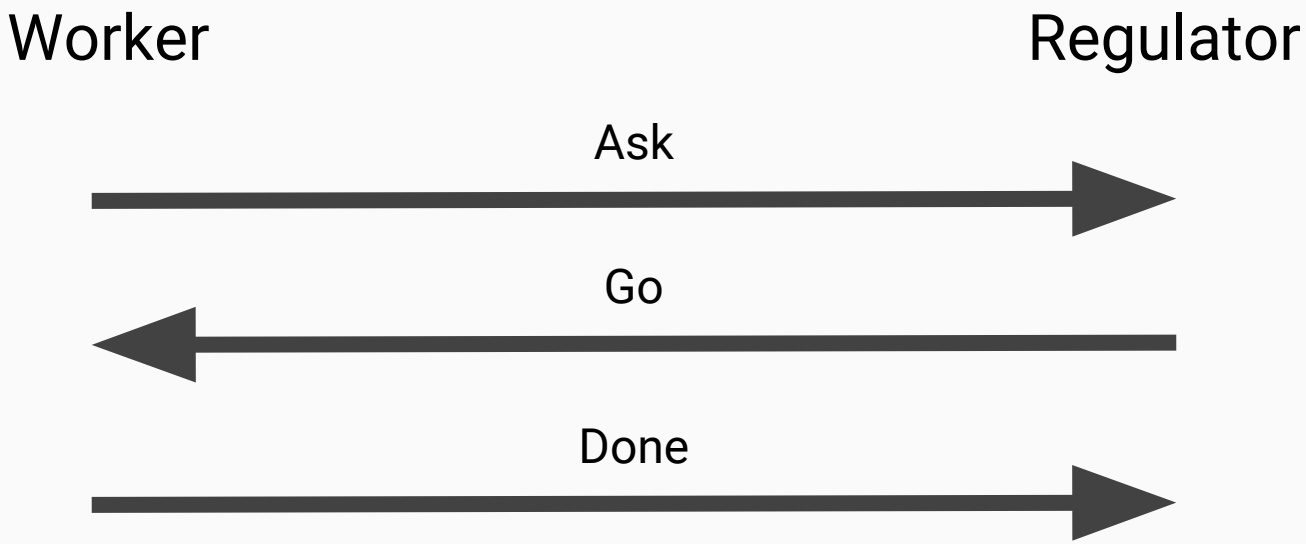
Usability Engineering - Nielsen 1993

- 0.1 second: instantaneous
- 1 second: continuous flow of thought
- 10 seconds: continuous attention

Job Scheduling

- Regulate load
- Prevent overload
- Detect underload?

Job Scheduling



Job Scheduling

- github.com/uwiger/jobs
- github.com/jlouis/safetyvalve

sregulator

```erlang
{go, Ref, RegPid, _, WaitTime} =
        sregulator:ask(Regulator),
% Carry out task
sregulator:done(RegPid, Ref).

{drop, WaitTime} = sregulator:ask(Regulator).
```

```elixir
{:go, ref, reg_pid, _, wait_time} =
        :sregulator.ask(regulator)
# Carry out task
:sregulator.done(reg_pid, ref)

{:drop, wait_time} = :sregulator.ask(regulator).
```

sregulator

```
init(_) ->
 QueueSpec = {sbroker_timeout_queue,
       #{timeout => 2000}},
 ValveSpec = {sregulator_open_valve, #{}},
 MeterSpec = {sbroker_overload_meter,
       #{alarm => {overload, ?MODULE}}},
 {ok, {QueueSpec, ValveSpec, [MeterSpec]}}.
```

```
def init(_) do
 queue_spec = {:sbroker_timeout_queue,
       %{timeout: 2000}}
 valve_spec = {:sregulator_open_valve,
       %{max: 10}}
 meter_spec = {:sbroker_overload_meter,
       %{alarm: {:overload, __MODULE__}}}
 {:ok, {queue_spec, valve_spec, [meter_spec]}}
end
```

sregulator_open_valve

- Limits concurrent tasks to maximum capacity
- Simple “valve”

sregulator_open_valve

```
#{max => non_neg_integer() | infinity}.
```

```
%{optional(:max) =>
    non_neg_integer | :infinity}
```

sregulator_rate_valve

- Limits tasks per time interval when above minimum capacity
- Tasks don’t count towards limit below minimum capacity
- Limits concurrent tasks to maximum capacity

sregulator_rate_valve

```
#{limit    => non_neg_integer(),
  interval => pos_integer(),
  min      => non_neg_integer(),
  max      => non_neg_integer() | infinity}.
```

```
%{optional(:limit)    => non_neg_integer,
  optional(:interval) => pos_integer,
  optional(:min)      => non_neg_integer,
  optional(:max)      =>
    non_neg_integer | :infinity}
```

BBR: Congestion-Based Congestion Control - Cardwell, Cheng, Gunn, Yeganeh, Jacobson 2016

- Find the minimum round trip time at the maximum bandwidth
- First find bottleneck bandwidth by increasing rate to find maximum rate
- Then enter cycles of gain, drain and cruise to probe bandwidth
- If minimum round trip time unchanged, throttle to probe round trip time
- Delay between sends with packet size / bottleneck rate

sregulator_bbr_valve

- Adjusts interval between when tasks start to find minimum latency for maximum throughput when above minimum capacity
- Tasks don’t count towards limit below minimum capacity
- Limits concurrent tasks to maximum capacity

sregulator_bbr_valve

e032aaf619d9483c14d2124b318db30ff4956c12

sregulator_bbr_valve

```
#{target    => non_neg_integer(),
  window    => pos_integer(),
  min       => non_neg_integer(),
  max       => non_neg_integer() | infinity}.
```

```
%{optional(:target)    => non_neg_integer,
  optional(:window)    => pos_integer,
  optional(:min)       => non_neg_integer,
  optional(:max)       =>
    non_neg_integer | :infinity}
```

sbroker_drop_queue

- FIFO or LIFO queue
- Head drop or tail drop on maximum size
- Simple queue

sbroker_drop_queue

```
#{out      => out | out_r,
  drop     => drop | drop_r,
  max      => non_neg_integer() | infinity}
```

```
%{optional(:out)      => :out | :out_r,
  optional(:drop)     => :drop | :drop_r,
  optional(:max)      =>
    non_neg_integer | :infinity}
```

sbroker_timeout_queue

- FIFO or LIFO queue
- Head drop or tail drop on maximum size
- Drops requests in queue for longer than timeout

sbroker_timeout_queue

```
#{timeout => non_neg_integer() | infinity,
  out     => out | out_r,
  drop    => drop | drop_r,
  max     => non_neg_integer() | infinity}
```

```
%{optional(:timeout)  =>
      non_neg_integer | :infinity,
  optional(:out)      => :out | :out_r,
  optional(:drop)     => :drop | :drop_r,
  optional(:max)      =>
      non_neg_integer | :infinity}
```

Controlling Queue Delay - Nichols, Jacobson 2012

- Assumes a standing queue of target time is acceptable
- Tracks minimum queue delay
- Drops a packet if minimum delay above target for an interval
- Decreases interval and repeats until minimum delay meets target
- Interval should be approximately 95%-99% percentile RTT latency
- Target should be approximately 5%-10% of interval

sbroker_codel_queue

- FIFO or LIFO queue
- Head drop or tail drop on maximum size
- Drops requests using CoDel algorithm

sbroker_codel_queue

```
#{target   => non_neg_integer(),
  interval => pos_integer(),
  out       => out | out_r,
  drop     => drop | drop_r,
  max      => non_neg_integer() | infinity}
```

```
%{optional(:target)    => non_neg_integer,
  optional(:interval)  => pos_integer,
  optional(:out)       => :out | :out_r,
  optional(:drop)      => :drop | :drop_r,
  optional(:max)       =>
    non_neg_integer | :infinity}
```

On Packet Switches with Infinite Storage - Nagle 1985

- Pretend a network switch could have infinite storage
- One queue per flow of packets
- Dequeue from each queue in turn for fairness between hosts

sbroker_fair_queue

- Wraps (infinitely) many queues with same configuration
- Round robin dequeues from each active queue
- Garbage collects empty queues

sbroker_fair_queue

```
{module(), term(), index()}

-type index() :: key() |
      {hash, key(), 1..4294967296}.
-type key() :: application | node | pid | value |
            {element, pos_integer()}.
```

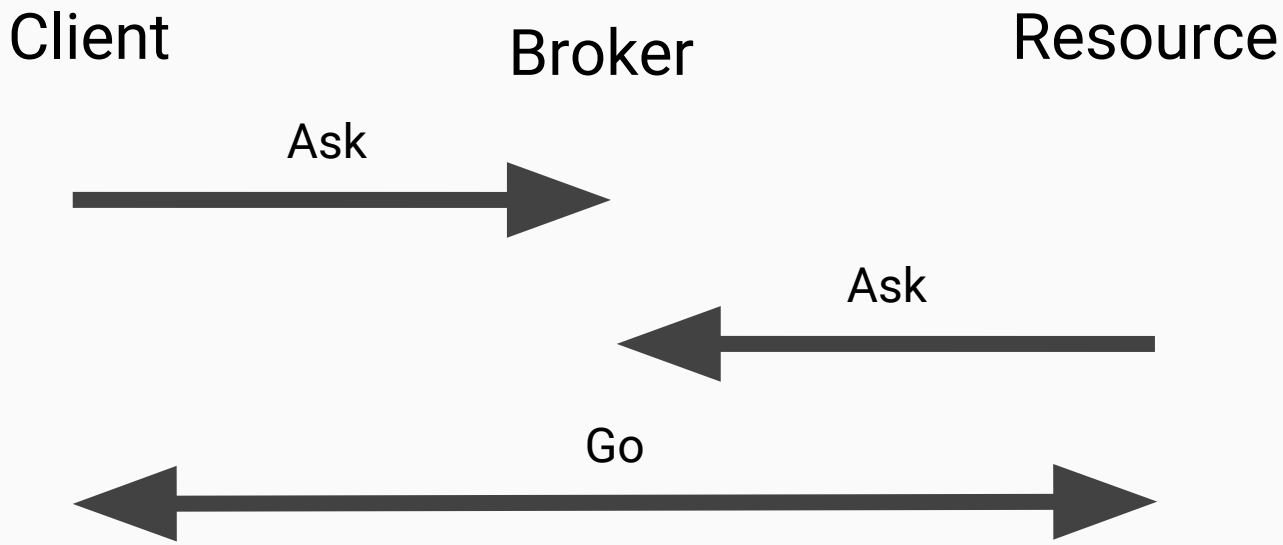
```
{module, term, index}

@type index() :: key |
      {:hash, key, 1..4294967296}
@type key() :: :application | :node | :pid |
      :value  | {:element, pos_integer}
```

Job Matching

- Match client with resource to handle task
- Regulate load
- Prevent overload

Job Matching



sbroker

```
% Client
{go, Ref, Resource, _, WaitTime} =
        sbroker:ask(Broker).
% Owner of resource
{go, Ref, Client, _, _} =
        sbroker:ask_r(Broker, Resource).

{drop, WaitTime} = sbroker:ask(Broker).
```

```
# Client
{:go, ref, resource, _, wait_time} =
        :sbroker.ask(broker)
# Owner of resource
{:go, ref, client_, _} =
        :sbroker.ask_r(broker, resource)

{:drop, wait_time} = :sbroker.ask(broker).
```

sbroker

```
init(_) ->
 AskSpec = {sbroker_timeout_queue,
       #{timeout => 2000}},
 AskRSpec = {sbroker_timeout_queue,
       #{timeout => 2000}},
 MeterSpec = {sbroker_overload_meter,
       #{alarm => {overload, ?MODULE}}},
 {ok, {AskSpec, AskRSpec, [MeterSpec]}}.
```

```
def init(_) do
 ask_spec = {:sbroker_timeout_queue,
       %{timeout: 2000}}
 ask_r_spec = {:sbroker_timeout_queue,
       %{timeout: 2000}}
 meter_spec = {:sbroker_overload_meter,
       %{alarm: {:overload, __MODULE__}}}
 {:ok, {ask_spec, ask_r_spec, [meter_spec]}}
end
```

Alarms

- sbroker_overload_meter
- sregulator_underload_meter

Load Balancing and Overload Protection

- srand
- sscheduler
- sbetter (sbetter_meter)
- sprotector (sprotector_pie_meter)