Distributed Producer/Consumer Framework with Guaranteed Message Delivery

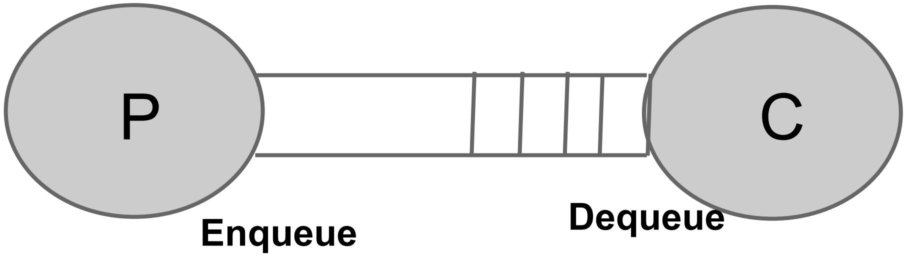
guanhua ye, TigerText Inc
gye@tigertext.com

tigertext

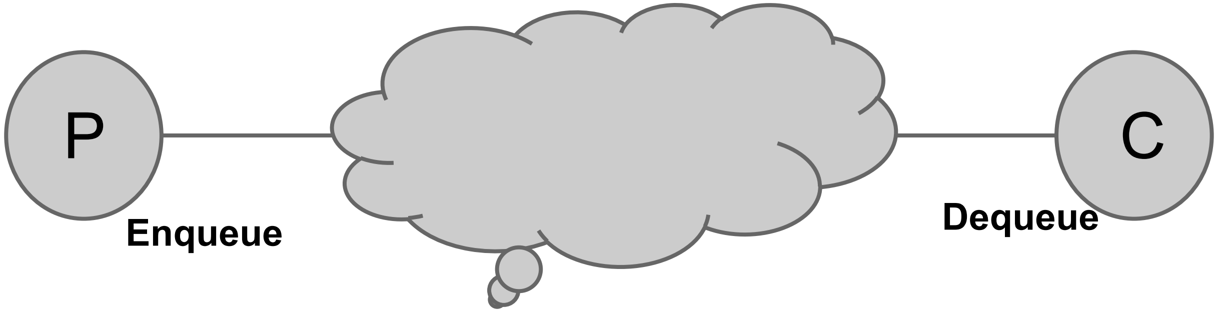
Agenda

- Overview
- System Design
- Component details
- Demo

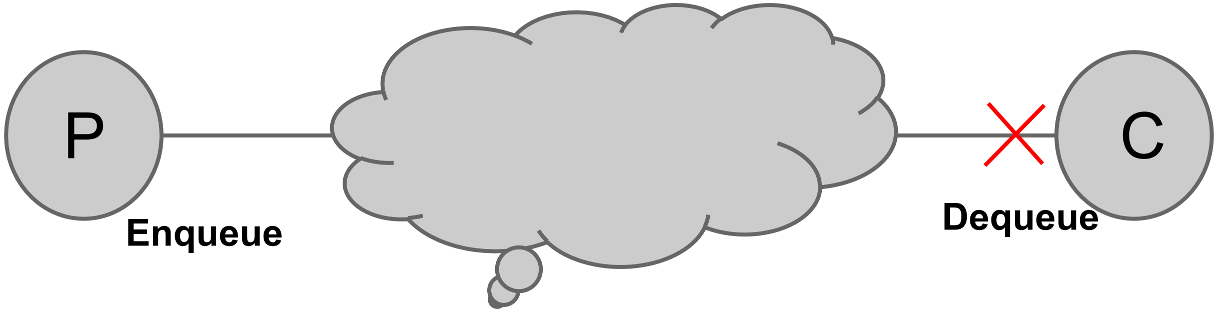
Classic producer-consumer problem



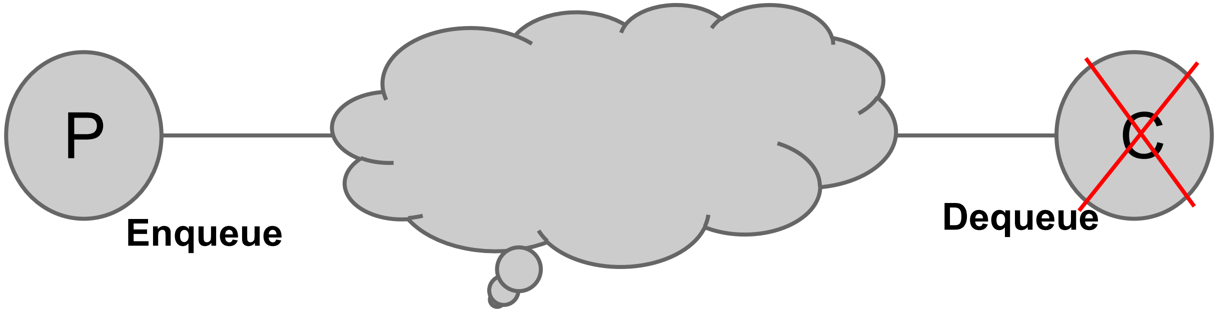
Distributed Producer/Consumer



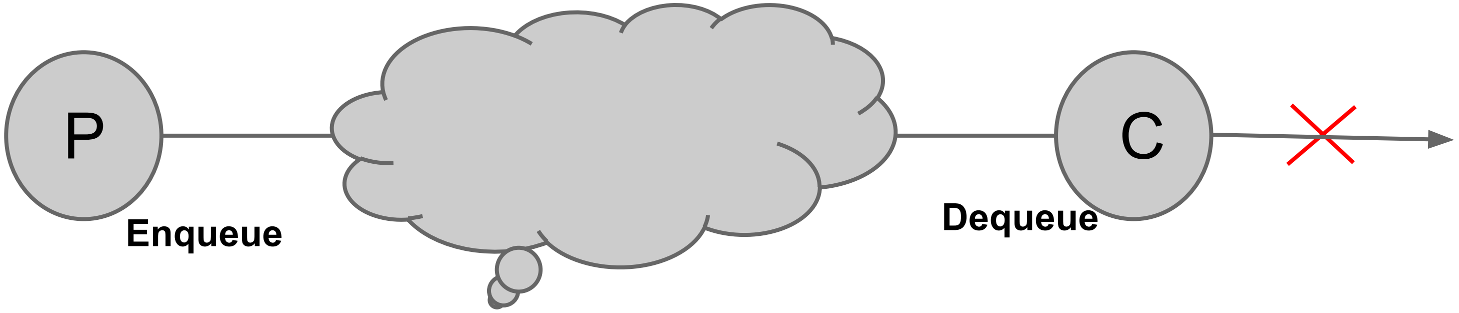
Distributed Producer/Consumer



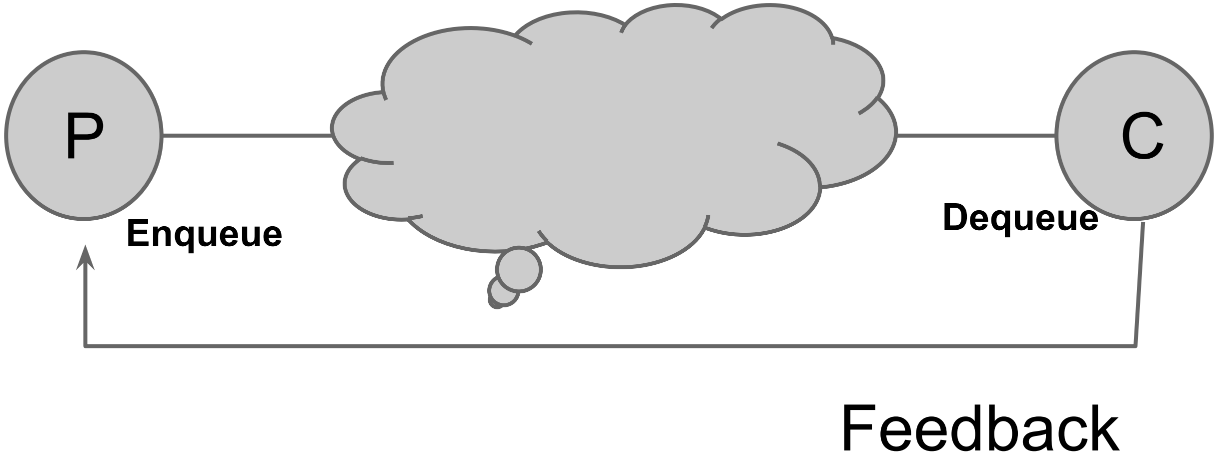
Distributed Producer/Consumer



Distributed Producer/Consumer



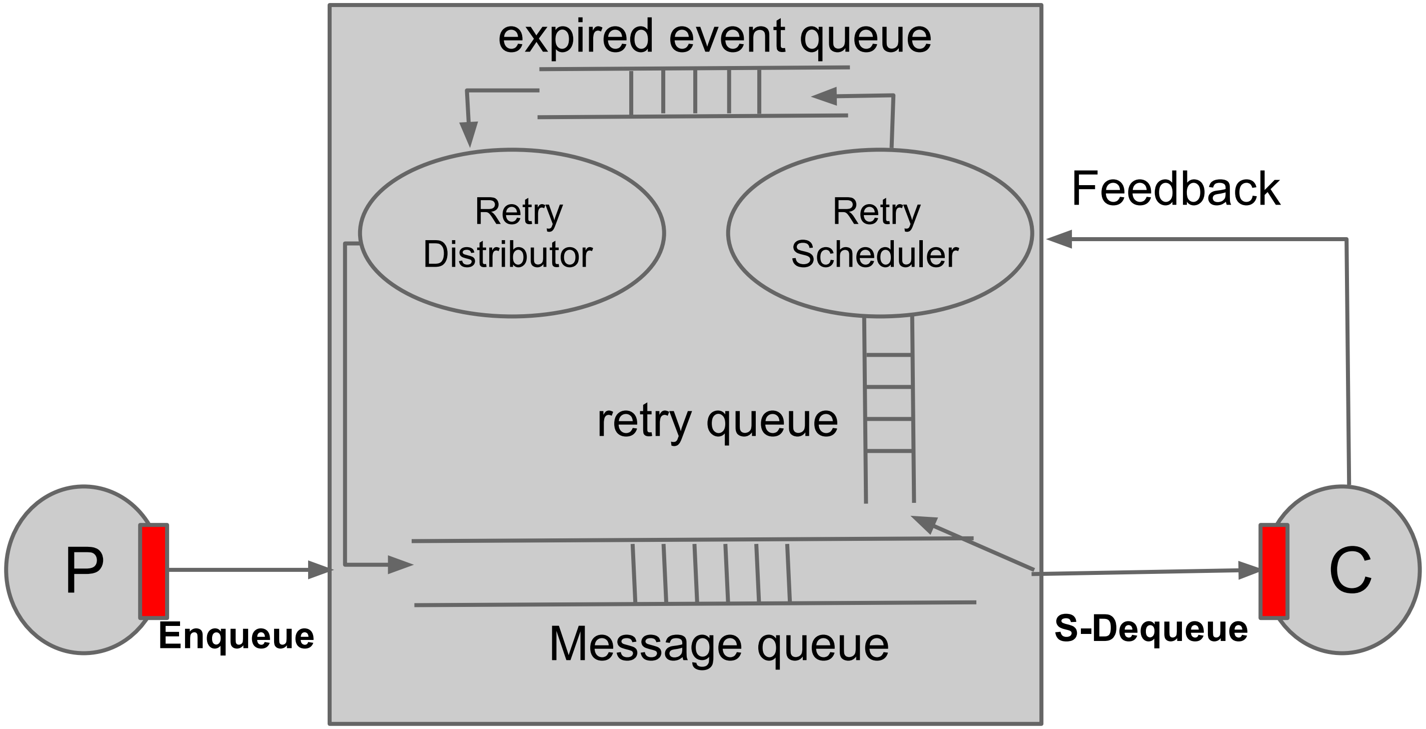
Producer/Consumer with Feedback



Design Goals

- Simple producer/consumer operation
- No location limitation
- No limit on the number of producer or consumer
- Self-provisioning, no configuration required when adding new types of producer/consumer
- Use off the shelf technologies

Distributed Producer/Consumer with Guaranteed Message Delivery



🟥 - Client lib for producer/consumer in javascript and erlang

Why Redis?

- stable
- very fast
- atomic operation, transaction and server side scripting
- Technology we familiar with
- High confidence on operations

Self-provisioning Addressing

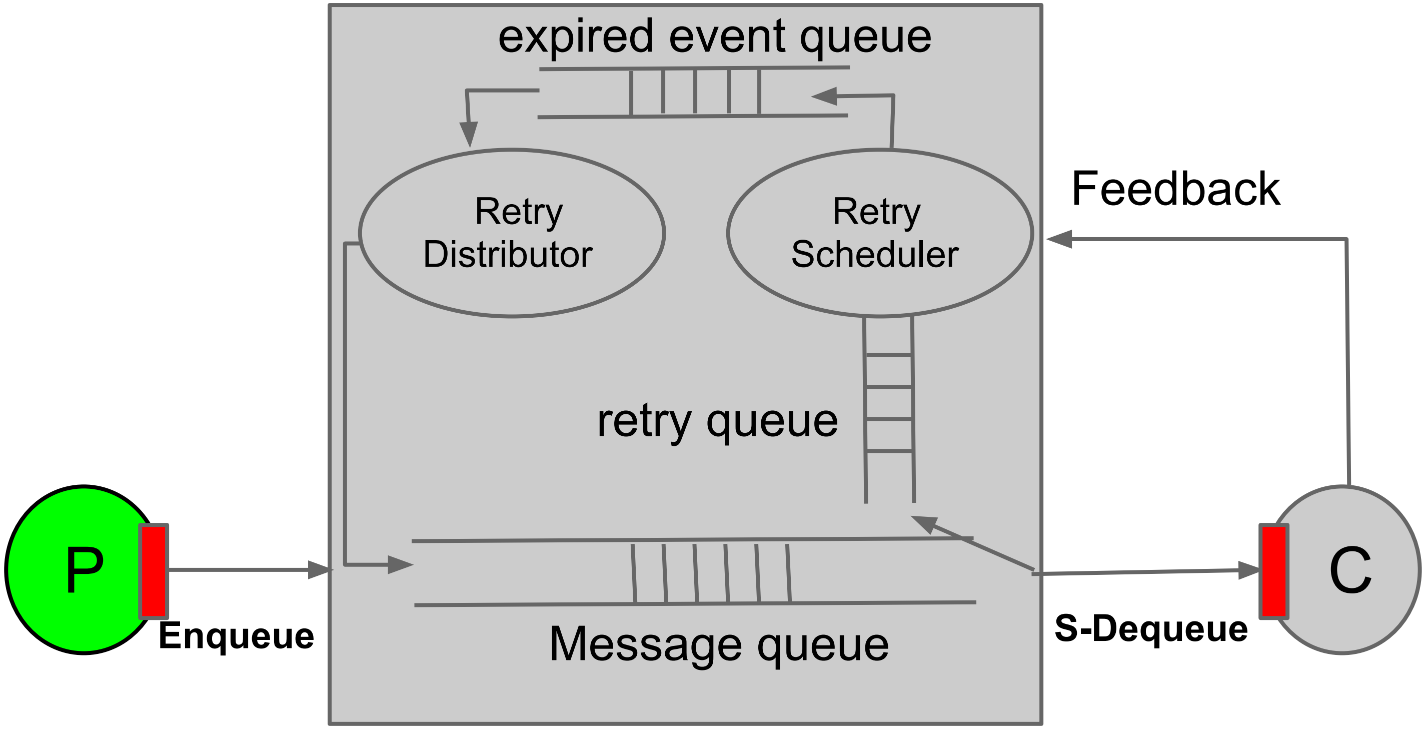
Each event that producer generates contains:

- **Service Name**: Producer/Consumer use service name to identify corresponding message queue
- **Timer Id/event id**: UUID for each timer/event

Example:

service:test_service:timer:bc0e88e1-37ff-4ce8-a7ce-6af26d768a9d

Distributed Producer/Consumer with Guaranteed Message Delivery



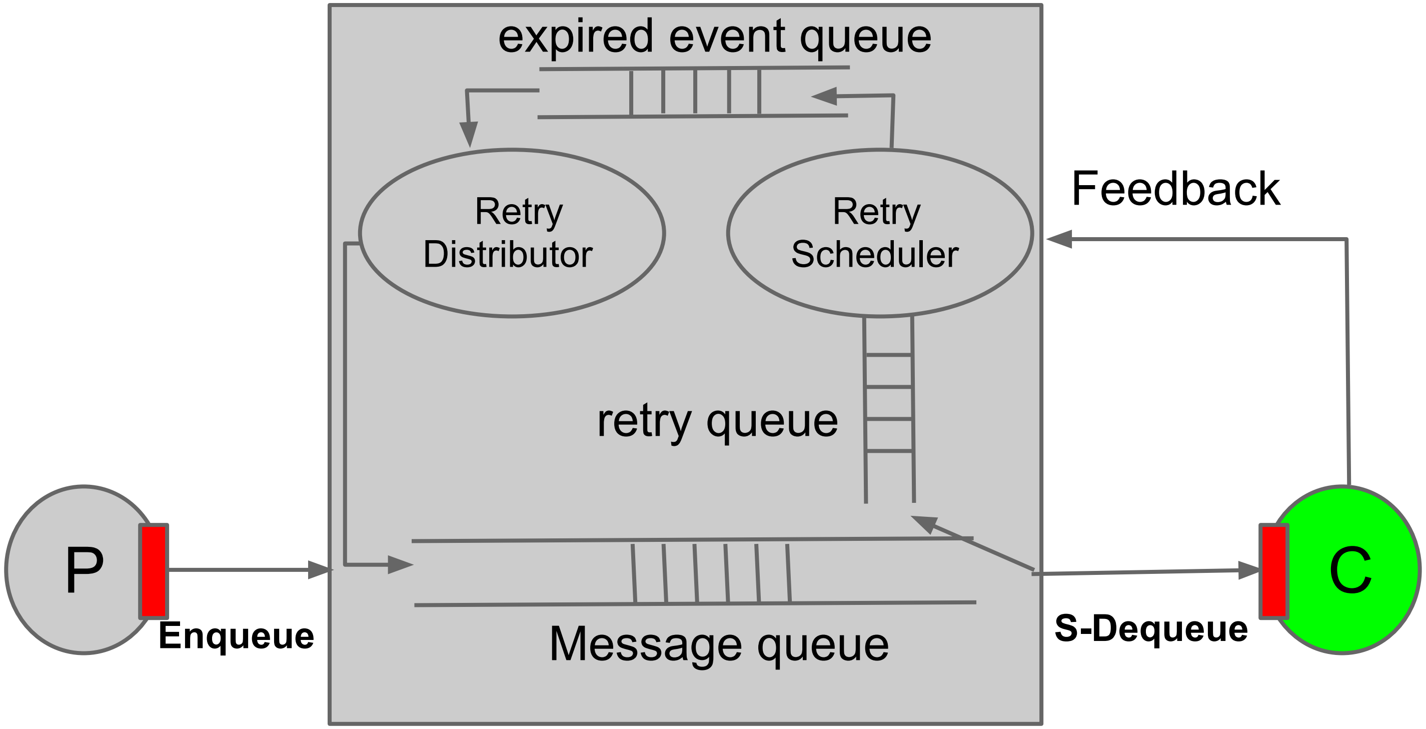
🟥 - Client lib for producer/consumer in javascript and erlang

Producer behaviour

- queue_client:enqueue(Service_Name, Meta_Data)

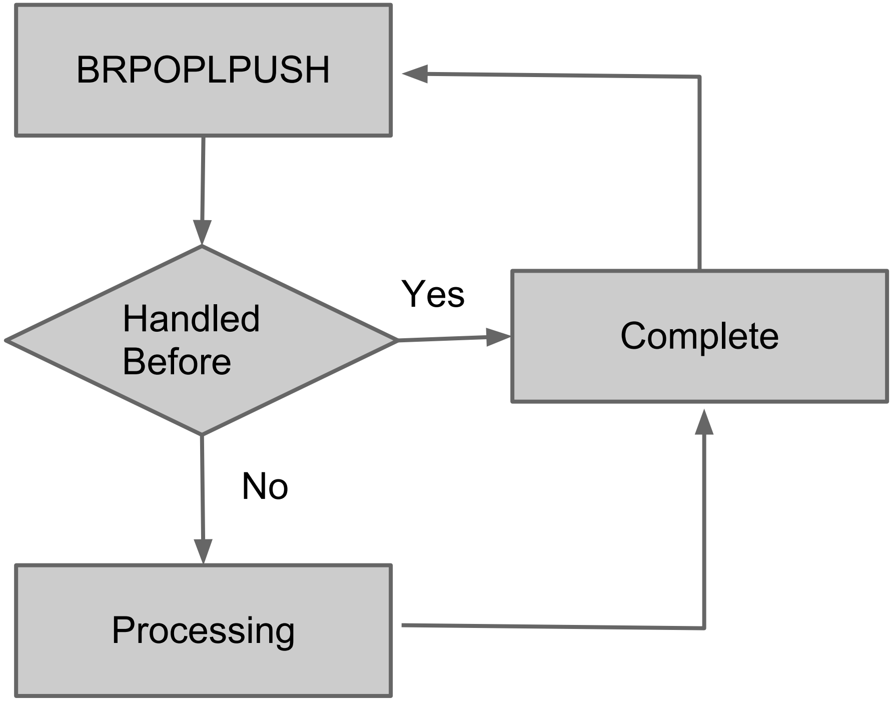
- queue_client:create_timer(Service_Name, Time_in_Future, Meta_Data)

Distributed Producer/Consumer with Guaranteed Message Delivery



🟥 - Client lib for producer/consumer in javascript and erlang

Consumer behaviour



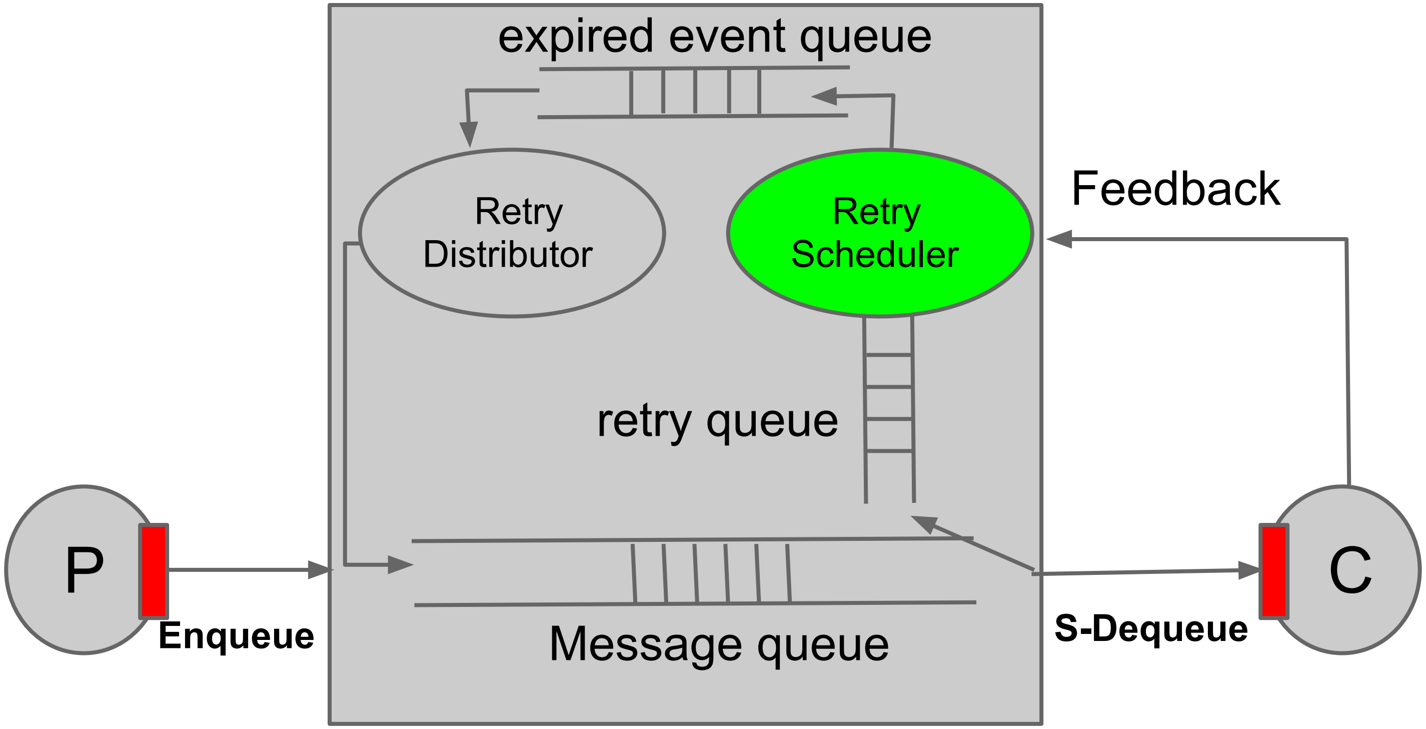
Gen_queue_consumer

```
-module(gen_queue_consumer).

-callback init() -> {ok, State ::term()}.

-callback handle_event({Id ::string(), Payload ::string()},
State ::term()) -> {ok, NewState ::term()}.
```

Consumer example

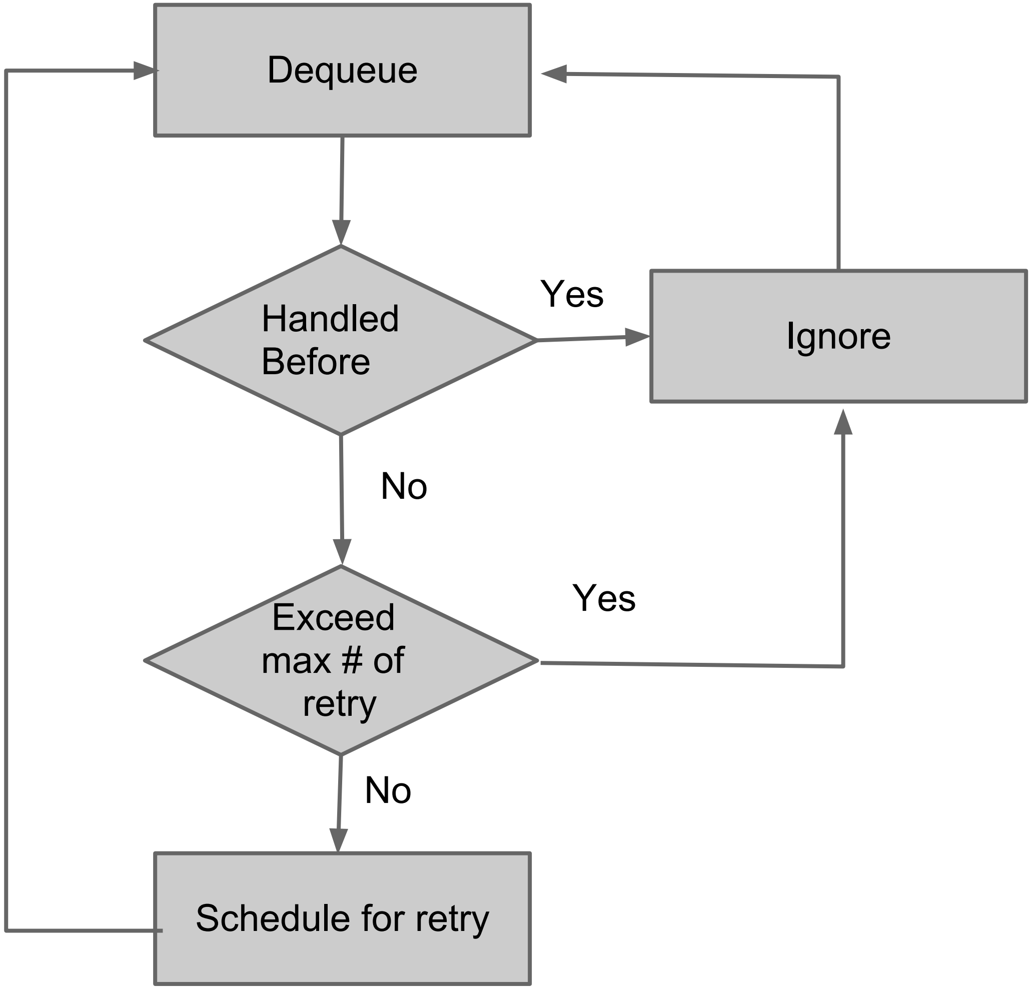
```
-module(test_service_consumer).
-author('gye@tigertext.com').
-behaviour(gen_queue_consumer).
-export([init/0, handle_event/2]).

init() -> {ok, 0}.
handle_event({Id, Payload}, State) ->
    io:format("Received event for test service, id=~p,
                payload=~p~n", [Id, Payload]),
    queue_client:complete("test_service", Id),
    {ok, State}.
```

Distributed Producer/Consumer with Guaranteed Message Delivery

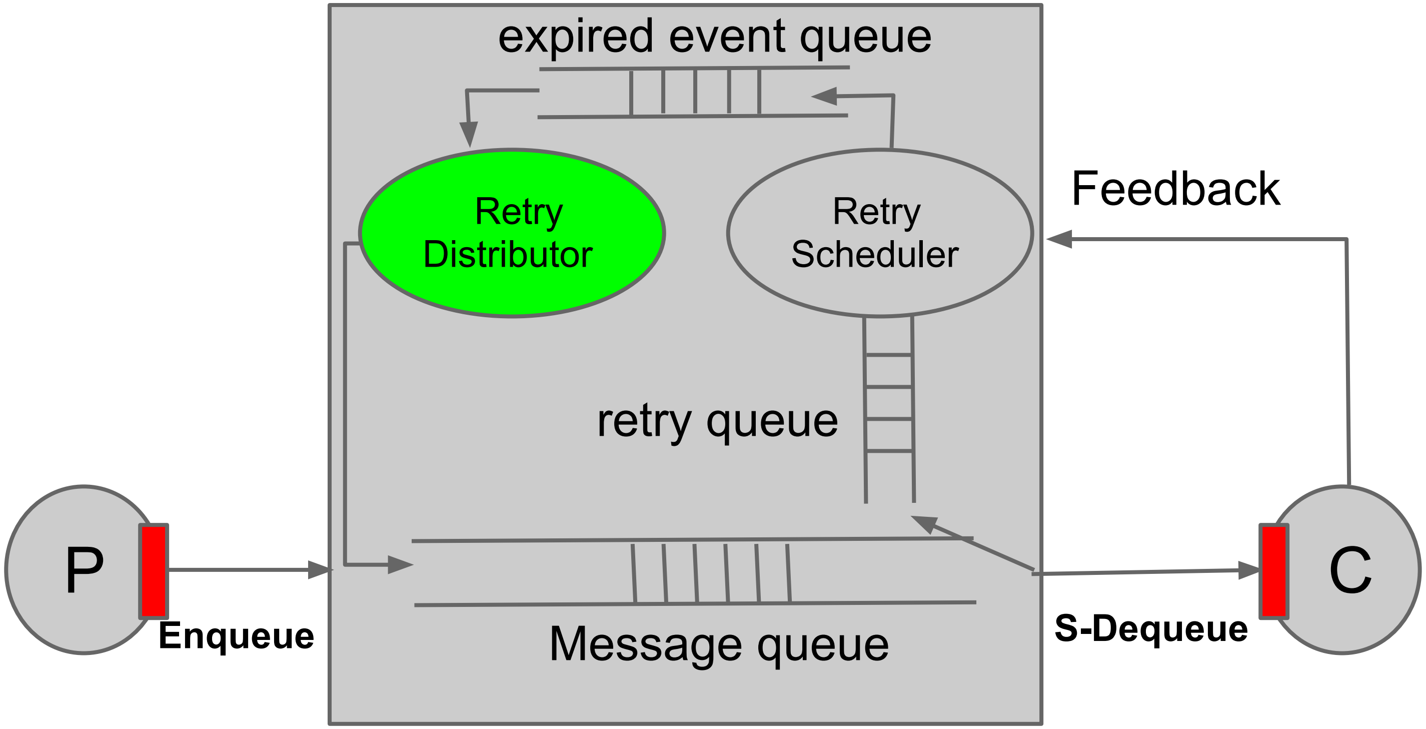


🟥 - Client lib for producer/consumer in javascript and erlang

Retry Scheduler behaviour



Distributed Producer/Consumer with Guaranteed Message Delivery



🟥 - Client lib for producer/consumer in javascript and erlang

Retry Distributor behaviour

- Dequeue expired event queue
- Get the service name from the event
- Enqueue to the right queue base on service name

What works well

- System scales with added producer/consumer
- The system does not degrade with slow consumer or stopped consumer
- It is reliable, it handles millions of events every day

Lessons learned

- redis lrem is expensive - don't use when the queue length is big
- redis expire cannot be used as real-time timer

DEMO

Weather Station

- Producer - weather man
- Consumer - A gen server that consumes weather report, and does a HTTP post to a web server
- Weather web site - Host current weather report
- End user - Whoever visits weather web site

Reference & links

redis - www.redis.io

node.js - www.nodejs.org

retry scheduler and distributor - https://github.com/georgeye/node_timer_service